

Reza Mohammadi

Data Science Foundations and Machine Learning with R: From Data to Decisions

31 July 2025

Springer

Table of contents

.....	17
Preface	17
Why This Book?.....	18
Who Should Read This Book?.....	18
Skills You Will Gain.....	19
Structure of This Book	20
How to Use This Book	22
Datasets Used in This Book	23
How to Teach with This Book	24
Acknowledgments	25
1 Getting Started with R	27
Why Choose R for Data Science?.....	28
What This Chapter Covers	28
1.1 How to Learn R	29
1.2 How to Install R	31
Keeping R Up to Date	31
1.3 How to Install RStudio	31
Installing RStudio	32
Exploring the RStudio Interface	32
Customizing RStudio	33

1.4	Getting Help and Learning More	34
1.5	Data Science and Machine Learning with R	35
1.6	How to Install R Packages	36
1.7	How to Load R Packages	38
1.8	Running Your First R Code	38
	Using Comments to Explain Your Code	40
	1.8.1 How Functions Work in R	40
1.9	Import Data into R	41
	Importing Data with RStudio's Graphical Interface	41
	Importing CSV Files with <code>read.csv()</code>	43
	Setting the Working Directory	44
	Importing Data Using Dialogs and URLs	44
	Importing Excel Files with <code>read_excel()</code>	45
	Loading Data from R Packages	45
1.10	Why Data Types Matter in R	46
1.11	Data Structures in R	47
	Vectors in R	48
	Matrices in R	49
	Data Frames in R	50
	Lists in R	53
1.12	Accessing Columns and Rows in R	54
	Using the <code>\$</code> Operator	55
	Using the <code>[]</code> Operator	55
1.13	How to Merge Data in R	56
1.14	Getting Started with Data Visualization in R	57
	Geom Functions in <code>ggplot2</code>	60
	Aesthetics in <code>ggplot2</code>	62
1.15	Formula in R	64
1.16	Simulating Data in R	66
1.17	Reporting with R Markdown	68
	R Markdown Basics	69

Table of contents	5
The Header	69
Code Chunks and Inline Code.....	70
Styling Text	73
Mastering R Markdown	74
1.18 Reporting with Quarto	74
1.19 Chapter Summary and Takeaways	75
1.20 Exercises	76
Basic Exercises.....	76
More Challenging Exercises	77
Reflect and Connect	79
2 Foundations of Data Science and Machine Learning	81
What This Chapter Covers	82
2.1 What is Data Science?	82
Key Components of Data Science	83
2.2 Why Data Science Matters	84
2.3 The Data Science Workflow	85
2.4 Problem Understanding	88
2.5 Data Preparation.....	90
2.6 Exploratory Data Analysis (EDA)	91
2.7 Data Setup for Modeling.....	92
2.8 Modeling	92
2.9 Evaluation	93
2.10 Deployment	94
2.11 Machine Learning: The Engine of Intelligent Systems	95
Supervised Learning: Learning from Labeled Data	97
Unsupervised Learning: Discovering Structure Without Labels	98
Reinforcement Learning: Learning Through Interaction	99
2.12 Chapter Summary and Takeaways	100
2.13 Exercises	101
Warm-Up Questions	101

Exploring the Data Science Workflow	101
Applied Scenarios and Case-Based Thinking	102
Applying Machine Learning Methods	102
Evaluation, Bias, and Fairness	103
Broader Reflections and Ethics	103
3 Data Preparation in Practice: Turning Raw Data into Insight	105
What This Chapter Covers	106
3.1 Data Preparation in Action: The <i>diamonds</i> Dataset.....	106
Loading and Exploring the <i>diamonds</i> Dataset	107
3.2 Identifying Feature Types	108
3.3 Key Considerations for Data Preparation	110
3.4 Outliers: What They Are and Why They Matter	111
3.5 Spotting Outliers with Visual Tools	112
Boxplots: Visualizing and Flagging Outliers	112
Histograms: Revealing Outlier Patterns.....	113
Additional Tools for Visual Outlier Detection	115
3.6 How to Handle Outliers	115
3.7 Outlier Treatment in Action	116
3.8 Missing Values: What They Are and Why They Matter	117
3.9 Imputation Techniques	118
Example: Random Sampling Imputation in R	119
Alternative Approaches	121
3.10 Feature Scaling	121
3.11 Min-Max Scaling.....	122
3.12 Z-score Scaling	123
3.13 Encoding Categorical Features for Modeling	125
3.14 Ordinal Encoding	125
3.15 One-Hot Encoding	126
3.15.1 One-Hot Encoding in R.....	127
3.16 Case Study: Preparing Data to Predict High Earners	128

Table of contents	7
3.16.1 Overview of the Dataset	129
3.16.2 Handling Missing Values	132
3.16.3 Encoding Categorical Variables.....	133
3.16.4 Handling Outliers	135
3.17 Chapter Summary and Takeaways	137
3.18 Exercises	138
Understanding Data Types.....	138
Working with the Diamonds Dataset	138
Detecting and Treating Outliers.....	138
Encoding Categorical Variables	139
Exploring the Adult Dataset.....	139
Feature Engineering and Scaling	139
Modeling and Real-World Scenarios	140
Ethics and Reflection	140
4 Exploratory Data Analysis	143
What This Chapter Covers	144
4.1 Key Objectives and Guiding Questions for EDA	144
4.2 EDA as Data Storytelling	146
4.3 EDA in Practice: The <i>Churn</i> Dataset	148
Understanding the Churn Problem	149
Getting to Know the Data	150
4.4 Examining Categorical Features and Churn Patterns	152
Relationship Between Churn and Subscription Plans	153
Relationship Between Churn and Voice Mail Plan.....	155
Summary of Categorical Findings.....	156
4.5 Examining Numerical Features and Behavioral Trends	156
Customer Service Calls and Churn	157
Daytime Minutes and Churn	159
Evening and Nighttime Minutes	160
International Calls and Churn	161

Summary of Numerical Findings	162
4.6 Identifying Redundancy and Correlated Features	163
4.7 Exploring Multivariate Relationships.....	166
4.8 Insights from Exploratory Data Analysis.....	168
4.9 Chapter Summary and Takeaways	169
4.10 Exercises	170
Conceptual Questions	171
Hands-On Practice: Exploring the Bank Dataset	172
Challenge Problems	173
5 Statistical Inference and Hypothesis Testing.....	175
What This Chapter Covers	176
5.1 Introduction to Statistical Inference	176
5.2 Estimation: Drawing Informed Conclusions from Sample Data	178
5.3 Quantifying Uncertainty: Confidence Intervals	179
5.4 Hypothesis Testing.....	182
5.4.1 Types of Hypothesis Tests	183
5.4.2 Common Tests and When to Use Them.....	184
5.5 One-sample t-test	185
5.6 Hypothesis Testing for Proportion	187
5.7 Two-sample t-test	189
5.8 Two-sample Z-test	191
5.9 Chi-square Test	193
5.10 Analysis of Variance (ANOVA) Test	195
5.11 Correlation Test.....	197
5.12 From Inference to Prediction in Data Science	199
5.13 Chapter Summary and Takeaways	200
5.14 Exercises	201
Conceptual Questions	201
Hands-On Practice: Hypothesis Testing in R	202
Reflection	207

Table of contents	9
6 Setting Up Data for Modeling	209
What This Chapter Covers	210
6.1 Why Is It Necessary to Partition the Data?	210
6.2 Partitioning Data: The Train–Test Split	212
Example: Train–Test Split in R	213
6.3 Cross-Validation for Robust Performance Estimation	213
6.4 How to Split Data for Modeling	215
Example: Train–Test Split in R	216
A Note on Cross-Validation	217
6.5 How to Validate a Train–Test Split	217
What If the Partition Is Invalid?	219
6.6 Dealing with Class Imbalance	220
6.7 Chapter Summary and Takeaways	222
6.8 Exercises	223
Conceptual Questions	223
Hands-On Practice	224
Self-Reflection	226
7 Classification Using k-Nearest Neighbors	227
What This Chapter Covers	227
7.1 Classification	228
How Classification Works	228
Classification Algorithms and the Role of kNN	229
7.2 How k-Nearest Neighbors Works	230
How Does kNN Classify a New Observation?	230
Strengths and Limitations of kNN	231
7.3 kNN in Action: A Toy Example for Drug Classification	232
7.4 How Does kNN Measure Similarity?	235
7.4.1 Euclidean Distance	236
7.5 Data Preparation for the kNN Algorithm	237
7.5.1 Encoding Categorical Variables for kNN	238

7.5.2	Scaling Features for Fair Distance Computation	240
7.5.3	Preventing Data Leakage during Scaling	241
7.6	Choosing the Right Value of k in kNN	242
7.7	Case Study: Predicting Customer Churn with kNN	244
7.7.1	Partitioning and Preprocessing the Data for kNN	246
7.7.2	Finding the Best Value for (k)	248
7.7.3	Applying the kNN Classifier	249
7.7.4	Evaluating Model Performance of the kNN Model....	250
	Summary of the kNN Case Study	251
7.8	Chapter Summary and Takeaways	251
7.9	Exercises	252
	Conceptual Questions	252
	Hands-On Practice: Applying kNN to the Bank Dataset	253
	Critical Thinking and Real-World Applications	256
	Self-Reflection	256
8	Evaluating Machine Learning Models	259
	Why Is Model Evaluation Important?	260
	What This Chapter Covers	261
8.1	Confusion Matrix	261
8.2	Sensitivity and Specificity	265
8.2.1	Sensitivity	265
8.2.2	Specificity	266
	Sensitivity vs. Specificity: A Balancing Act	267
8.3	Precision, Recall, and F1-Score	267
8.4	Taking Uncertainty into Account	269
	Tuning the Classification Threshold	271
8.5	ROC Curve	272
8.6	Area Under the Curve (AUC)	275
8.7	Metrics for Multi-Class Classification.....	276
8.8	Evaluation Metrics for Continuous Targets	278

Table of contents	11
8.9 Chapter Summary and Takeaways	279
8.10 Exercises	281
Conceptual Questions	281
Hands-On Practice: Evaluating Models with the <i>bank</i> Dataset	282
Model Training and Evaluation	284
Critical Thinking and Real-World Applications	284
Self-Reflection	285
9 Naive Bayes Classifier	287
What This Chapter Covers	288
9.1 Bayes' Theorem and Probabilistic Foundations	289
The Essence of Bayes' Theorem	290
How Does Bayes' Theorem Work?	292
From Bayes' Theorem to Naive Bayes	293
9.2 Why Is It Called "Naive"?	293
9.3 The Laplace Smoothing Technique	295
9.4 Types of Naive Bayes Classifiers	296
9.5 Case Study: Predicting Financial Risk with Naive Bayes	297
Problem Understanding	298
Data Understanding	298
Data Setup for Modeling	299
Applying the Naive Bayes Classifier	301
Prediction and Model Evaluation	304
Takeaways from the Case Study	307
Chapter Summary and Takeaways	308
9.6 Exercises	309
Conceptual Questions	309
Hands-on Implementation with the Churn Dataset	310
Training and Evaluating the Naive Bayes Classifier	311
Real-World Application and Critical Thinking	312
Self-Reflection	313

10 Regression Analysis: Foundations and Applications	315
What This Chapter Covers	316
10.1 Simple Linear Regression	317
Exploring Relationships in the Data	318
Fitting a Simple Linear Regression Model	319
Fitting the Simple Regression Model in R	321
Making Predictions with the Regression Line	323
Residuals and Model Fit	324
10.2 Hypothesis Testing in Simple Linear Regression	326
10.3 Measuring the Quality of a Regression Model	328
Residual Standard Error (RSE)	329
R-squared (R^2)	329
Adjusted R-squared	330
Interpreting Model Quality	331
10.4 Multiple Linear Regression	332
Fitting the Multiple Regression Model in R	333
Making Predictions	334
Evaluating Model Performance	335
Same Data, Different Story: What Simpson's Paradox Can Teach Us	336
Summary and Implications	338
10.5 Generalized Linear Models (GLMs)	338
10.6 Logistic Regression for Binary Classification	339
Fitting a Logistic Regression Model in R	340
10.7 Poisson Regression for Modeling Count Data	343
Fitting a Poisson Regression Model in R	344
10.8 Choosing the Right Predictors: Stepwise Regression in Action	346
How AIC Guides Model Selection	347
Stepwise Regression in Practice: Using <code>step()</code> in R	348
Strengths, Limitations, and Considerations for Stepwise Regression	352

Table of contents	13
10.9 Extending Linear Models to Capture Non-Linear Relationships	353
The Need for Non-Linear Regression	354
10.10 Polynomial Regression in Practice	356
10.11 Diagnosing and Validating Regression Models	358
10.12 Case Study: Comparing Classifiers to Predict Customer Churn	360
Partitioning and Preprocessing	361
Training the Logistic Regression Model	362
Training the Naive Bayes Model	363
Training the kNN Model	363
Model Evaluation and Comparison	364
Reflections and Takeaways	366
10.13 Chapter Summary and Takeaways	366
10.14 Exercises	367
Simple and Multiple Linear Regression (House, Insurance, and Cereal Datasets)	368
Polynomial Regression (House Dataset)	369
Logistic Regression (Bank Dataset)	370
Stepwise Regression (House Dataset).....	371
Model Diagnostics and Validation	371
Self-Reflection	371
11 Decision Trees and Random Forests	373
What This Chapter Covers	374
11.1 How Decision Trees Work.....	375
Making Predictions with a Decision Tree.....	378
Controlling Tree Complexity	379
11.2 How CART Builds Decision Trees.....	379
11.3 C5.0: More Flexible Decision Trees	381
A Simple C5.0 Example	382
Advantages and Limitations	383
11.4 Random Forests: Boosting Accuracy with an Ensemble of Trees	383

Strengths and Limitations of Random Forests	384
11.5 Case Study: Who Can Earn More Than \$50K Per Year?	385
Overview of the Dataset	385
Data Preparation.....	387
Setup Data for Modeling.....	389
Building a Decision Tree with CART	390
Building a Decision Tree with C5.0	393
Building a Random Forest Model	394
Model Evaluation and Comparison	396
Reflections and Takeaways.....	400
11.6 Chapter Summary and Takeaways	400
11.7 Exercises	401
Conceptual Understanding	401
Hands-On: Classification with the Churn Dataset	402
Regression Trees and Random Forests: The redWines Dataset	404
Conceptual Questions: Regression Trees and Random Forests	404
Hands-On: Regression with the redWines Dataset	404
12 Neural Networks: The Building Blocks of Artificial Intelligence.	407
Why Neural Networks Are Powerful.....	408
What This Chapter Covers	409
12.1 The Biological Inspiration Behind Neural Networks.....	410
12.2 How Neural Networks Work	412
Key Characteristics of Neural Networks	414
12.3 Activation Functions	415
The Threshold Activation Function	415
The Sigmoid Activation Function	416
Common Activation Functions in Deep Networks	418
Choosing the Right Activation Function	419
12.4 Network Architecture	419
12.5 How Neural Networks Learn	421

Table of contents	15
12.6 Case Study: Predicting Term Deposit Subscriptions	423
Problem Understanding	423
Overview of the Dataset	424
Setup Data for Modeling.....	425
Training a Neural Network Model in R	430
Prediction and Model Evaluation	433
Reflections and Takeaways.....	435
12.7 Chapter Summary and Takeaways	436
12.8 Exercises	437
Conceptual questions	437
Practical exercises using the <i>bank</i> dataset.....	438
Self-Reflection	441
13 Clustering for Insight: Segmenting Data Without Labels	443
What This Chapter Covers	444
13.1 What is Cluster Analysis?	444
How Do Clustering Algorithms Measure Similarity?	446
13.2 K-means Clustering	447
13.3 Selecting the Optimal Number of Clusters	451
13.4 Case Study: Segmenting Cereal Brands by Nutrition	453
13.4.1 Overview of the Dataset	453
13.4.2 Data Preprocessing	454
13.4.3 Selecting the Number of Clusters	458
13.4.4 Performing K-means Clustering	459
13.4.5 Reflections and Takeaways	461
13.5 Chapter Summary and Takeaways	462
13.6 Exercises	462
Conceptual questions	462
Practical Exercises	463
References	467

Preface

Data science is transforming how we understand the world, solve problems, and make informed decisions. From generative AI systems such as ChatGPT, DeepSeek, and Gemini to personalized recommendations on streaming platforms and fraud detection in banking, data-driven techniques are reshaping industries and everyday life. As demand grows for professionals who can analyze and model data effectively, accessible and rigorous educational resources are more essential than ever.

Data Science Foundations and Machine Learning with R: From Data to Decisions offers a hands-on introduction to this dynamic field. Designed for readers with no prior experience in analytics or programming, the book provides a clear, structured pathway into data science by emphasizing conceptual understanding, practical application, and reproducible workflows using **R**.

This book is intended for newcomers to data science and machine learning, particularly those without a background in programming or statistics. Whether you are a student, business professional, or researcher, it offers an approachable yet academically grounded learning experience. Drawing on my experience teaching data science at the university level, I emphasize an applied, example-driven approach that fosters active engagement and deep comprehension.

The motivation for this book stems from a recurring challenge I encountered in the classroom: many of my students were eager to explore data science but lacked resources that were both accessible and conceptually rigorous. I saw a clear need for materials that bridge foundational theory and meaningful application. My goal was to provide a guided, practical learning experience—lowering the barrier to entry without sacrificing depth—through real data and hands-on practice in **R**.

To support a smoother learning curve, the book adopts *active learning strategies*. Concepts are introduced progressively and reinforced through illustrative examples, guided coding exercises, and applied problem-solving. By

working directly with authentic datasets and practical scenarios, readers learn not only how data science tools work, but also when and why to use them. This experiential approach fosters lasting understanding and builds confidence in applying these skills independently.

Rather than presenting machine learning as a purely theoretical discipline, the book integrates annotated code, real datasets, and structured walk-throughs throughout. Each chapter concludes with a case study that applies the chapter’s core ideas to a realistic context, bridging the gap between theory and practice. Exercises further reinforce learning through direct implementation in **R**, helping readers develop both conceptual clarity and practical fluency.

Why This Book?

Data science is a rapidly evolving field that integrates machine learning, statistical modeling, and computational tools to extract insights from data. This book provides a structured, application-focused introduction to data analysis and machine learning using **R**—a widely adopted, open-source language known for its strengths in statistical computing, visualization, and reproducible workflows.

Unlike many textbooks that assume prior experience with programming or analytics, this book is designed to be *accessible and hands-on*. Concepts are introduced clearly and reinforced through real-world examples, guided exercises, and annotated **R** code. This approach allows readers to build theoretical understanding alongside practical fluency from the outset.

With its extensive ecosystem of packages, **R** remains a leading tool for data science across academic, industrial, and research settings. This book emphasizes its practical use in solving data-driven problems. For readers who prefer Python, a companion volume—*Data Science Foundations and Machine Learning with Python: From Data to Decisions*—is also available from the same publisher.

Who Should Read This Book?

This book is intended for anyone seeking to learn data science and machine learning, particularly those new to the field. It is well-suited for:

- Business professionals aiming to integrate data-driven decision-making into their work,
- Students and researchers applying data analysis in academic or applied contexts,
- Beginners with no prior experience in programming or analytics,
- Readers interested in learning data science and machine learning using R.

It is especially appropriate for undergraduate students in programs that emphasize quantitative reasoning—such as economics, business administration, business economics (including specializations in finance or organizational economics), communication science, psychology, and STEM fields (science, technology, engineering, and mathematics). It also supports students in Master's programs in business analytics, econometrics, and the social sciences.

Designed for both self-study and classroom use, the book offers a structured and practice-oriented path to applying data science techniques in real-world settings. It serves as the reference for courses such as *Data Analytics: Machine Learning*, *Data Wrangling*, and *Business Analytics* across several BSc and MSc programs at the University of Amsterdam.

It is equally useful for professionals pursuing continuing education in analytics, offering an accessible foundation for those looking to strengthen their skills in a rapidly evolving data landscape.

Skills You Will Gain

This book walks you through a practical and progressive journey into data science and machine learning using R, structured around the *Data Science Workflow* (Figure 0.1). Each chapter supports both conceptual mastery and applied skill development, helping you progress from understanding and applying core ideas to analyzing results and evaluating solutions.

By the end of this book, you will be able to:

- *Recognize and describe* the key stages of a data science project—from problem formulation to model evaluation;
- *Apply* core R programming concepts, including data structures, control flow, and functions, to prepare and analyze data;
- *Clean and transform* raw datasets by handling missing values, outliers, and categorical variables using best practices;

- *Explore and interpret* data using descriptive statistics and effective visualizations;
- *Build and tune* machine learning models for classification, regression, and clustering using algorithms such as k-NN, Naive Bayes, decision trees, neural networks, and K-means;
- *Assess and compare* model performance using relevant metrics tailored to each type of task;
- *Transfer and adapt* your skills to solve real-world problems in marketing, finance, operations, and beyond.

Each chapter integrates illustrative examples, annotated **R** code, and exercises that reinforce learning through practice. Chapters conclude with a case study that synthesizes the main concepts, guiding you in applying techniques to authentic scenarios. This structure ensures that by the end of the book, you are not just familiar with tools—you are equipped to use them thoughtfully and effectively.

Structure of This Book

This book is structured around the *Data Science Workflow*—an iterative framework that guides you from foundational concepts to advanced machine learning techniques through hands-on learning. Your journey begins in Chapter 1, where you will install **R**, explore its syntax, and work with essential data structures. From there, each chapter builds on the previous one, combining coding practice with real-world case studies to help you gain both understanding and experience.

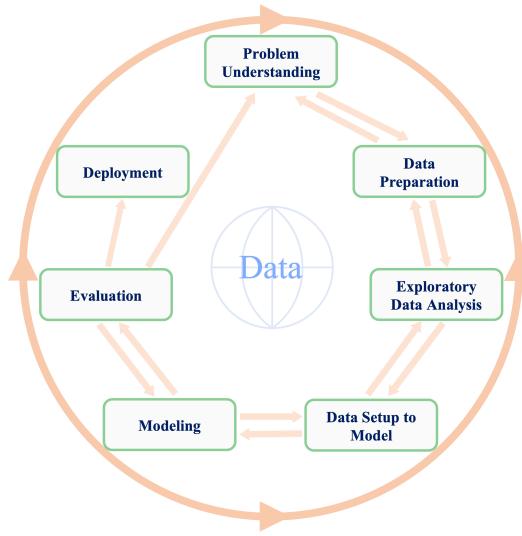


Figure 0.1: The Data Science Workflow is an iterative framework for structuring data science and machine learning projects. Inspired by the CRISP-DM model (Cross-Industry Standard Process for Data Mining), it supports systematic problem-solving and continuous refinement.

The *Data Science Workflow*, introduced in Chapter 2 and illustrated in Figure 0.1, consists of seven key stages:

1. *Problem Understanding*: Define the objective and the broader context (Chapter 2).
2. *Data Preparation*: Clean, transform, and organize raw data (Chapter 3).
3. *Exploratory Data Analysis (EDA)*: Visualize and summarize data to uncover patterns (Chapter 4).
4. *Data Setup for Modeling*: Select features, partition datasets, and scale variables (Chapter 6).
5. *Modeling*: Build and train predictive models using a range of machine learning algorithms (Chapters 7 to 13).
6. *Evaluation*: Measure model performance using appropriate metrics (Chapter 8).
7. *Deployment*: Translate models into real-world applications.

This workflow provides a practical and repeatable framework for tackling data-driven problems. The chapter sequence mirrors these phases, supporting a gradual progression from theory to implementation.

Chapter 5 also provides a concise review of key statistical ideas—such as confidence intervals and hypothesis testing—that support critical thinking and model interpretation.

To bridge theory and practice, each chapter concludes with a case study that applies its core ideas to a real-world problem. These case studies walk through the *Data Science Workflow* in action—guiding you through data preparation, model development, evaluation, and interpretation using real datasets. The datasets—listed in Table Table 0.1—are available through the `liver` package, enabling you to reproduce examples, complete exercises, and build practical skills with minimal setup.

Each chapter ends with exercises designed to consolidate learning—from conceptual questions and hands-on coding tasks to applied problem-solving challenges. These help reinforce key ideas, encourage experimentation, and build confidence in using R for data science.

How to Use This Book

This book is designed for *self-study, classroom instruction, and professional learning*. You can work through the chapters sequentially for a structured learning path or consult individual sections to focus on specific skills or concepts.

To make the most of this book:

- *Run the code* – Execute the R code examples interactively to reinforce key ideas through immediate feedback and hands-on experience.
- *Solve the exercises* – Tackle a range of questions in each chapter to deepen your understanding and strengthen analytical fluency.
- *Experiment with the code* – Modify examples, test new parameters, and experiment with different datasets to sharpen your problem-solving skills.
- *Study the case studies* – Use the end-of-chapter case studies to see the Data Science Workflow in action, from data preparation to model interpretation.
- *Use the book as a reference* – Return to chapters as needed to support your projects and refresh specific techniques.

Each chapter concludes with a case study based on a real-world dataset. These walk through the complete Data Science Workflow—preparation, modeling, evaluation, and interpretation—helping you consolidate learning and apply techniques in realistic analytical contexts.

This book also supports collaborative learning. Working through exercises and case studies in pairs or small groups can spark discussion, deepen understanding, and foster diverse perspectives—especially in classroom and workshop environments.

The book has been successfully used in data science courses at the University of Amsterdam and is well-suited for academic programs and professional training. Whether you are an independent learner, instructor, or practitioner, it offers a flexible and structured path to mastering essential tools and methods in data science and machine learning.

Datasets Used in This Book

This book incorporates real-world datasets to support its hands-on, application-focused approach to learning. These datasets are used throughout the chapters to demonstrate key data science and machine learning techniques, particularly in the end-of-chapter case studies. Table 0.1 provides an overview of the core datasets featured in the book, all of which are included in the `liver` package.

Table 0.1: Overview of datasets used for case studies in different chapters.
All datasets are included in the R package `liver`.

Name	Description	Chapter
churn	Customer churn dataset.	Chapters 4, 6, 7, 8, 10
bank	Direct marketing data from a Portuguese bank.	Chapters 7, 12
adult	US Census data for income prediction.	Chapters 3, 11
risk	Credit risk dataset.	Chapter 9
marketing	Marketing campaign performance data.	Chapter 10
house	House price prediction dataset.	Chapter 10
diamonds	Diamond pricing dataset.	Chapter 3
cereal	Nutritional information for 77 breakfast cereals.	Chapter 13

These datasets were selected to expose readers to practical challenges from a variety of domains, including marketing, finance, and predictive modeling. They are used throughout the book not only in examples and guided code but also in case studies that walk through the full Data Science Workflow.

In addition to the datasets listed in Table 0.1, the `liver` package contains over 15 datasets. Several of these appear in exercises at the end of each chapter,

providing readers with further opportunities to practice techniques, explore new data contexts, and reinforce their learning beyond the main examples.

How to Teach with This Book

This book is well-suited for introductory courses in data science and machine learning, as well as for professional training programs. Its structured progression, applied case studies, and extensive set of exercises make it a versatile resource for both instructors and learners.

To support systematic learning, the book includes over 500 exercises across three levels: conceptual questions that reinforce key ideas, applied tasks that involve real-world data, and advanced problems that deepen understanding of machine learning techniques. Together, these exercises help build a strong foundation and cultivate the analytical mindset essential for practical data science.

Each chapter also features a case study that guides students through the full Data Science Workflow—from data preparation and modeling to evaluation and interpretation—demonstrating how theoretical concepts are applied in realistic scenarios.

The book currently serves as the primary reference for courses such as *Data Analytics: Machine Learning*, *Data Wrangling*, and *Business Analytics* in BSc and MSc programs at the University of Amsterdam. It is also well suited for courses in applied statistics, econometrics, business analytics, and quantitative methods across programs in the social sciences, business, and STEM fields.

Instructors adopting this book have access to a full suite of teaching materials, including a solutions manual, presentation slides, and test banks. These resources provide a complete framework for delivering effective and engaging data science instruction.

The book is further supported by the `liver` package, which includes over 15 real-world datasets used throughout the chapters, exercises, and case studies. Combined with its emphasis on active learning through code walkthroughs, reproducible analysis, and applied problem-solving, this book offers an ideal foundation for teaching data science in both academic and professional contexts.

Acknowledgments

Writing this book has been both a challenging and deeply rewarding journey, and I would like to express my sincere gratitude to those who supported and inspired me along the way.

First and foremost, I thank my wife, Pariya, for her continuous support, patience, and encouragement. I am also grateful to my family—and especially my older brother—for their unwavering belief in me.

This book would not have taken shape without the contributions of my collaborators. I am particularly thankful to Dr. Kevin Burke for his valuable input in shaping the structure of the book. I also wish to acknowledge Dr. Jeroen van Raak and Dr. Julien Rossi, who have enthusiastically partnered with me in developing the Python edition of this book.

I am especially indebted to Eva Hiripi at Springer for her steadfast support and for encouraging me to pursue this project in the first place.

My colleagues in the Business Analytics Section at the University of Amsterdam provided thoughtful feedback and generous support during the writing process. I am particularly grateful to Prof. Ilker Birbil, Prof. Dick den Herdtog, Prof. Marc Salomon, Prof. Jeroen de Mast, Prof. Peter Kroos, Dr. Marit Schoonhoven, Dr. Stevan Rudinac, Dr. Rob Goedhart, Dr. Chintan Amrit, Dr. Inez Zwetsloot, Dr. Alex Kuiper, and Dr. Bart Lameijer. I also thank my PhD students, Lucas Vogels (soon to be Dr.) and Elias Dubbeldam, for their research insights and continued collaboration.

I would also like to acknowledge my former colleagues and co-authors, Dr. Florian Böing-Messing and Dr. Khodakaram Salimifard, for their continued academic partnership.

Finally, I am grateful to the students of the courses *Data Analytics: Machine Learning* and *Data Wrangling* at the University of Amsterdam. Their feedback has helped refine the material in meaningful ways—particularly John Gatev, whose thoughtful comments were especially valuable.

To everyone who supported this book—your encouragement, feedback, and collaboration have meant more than I can say.

Chapter 1

Getting Started with R

What do Spotify recommendations, fraud detection systems, and ChatGPT have in common? They all rely on data, and on programming languages to process, analyze, and act on it. In the world of data science, the two most widely used languages are **R** and **Python**. Both are widely adopted across academia, research, and industry, and each has distinct strengths.

This book is based on **R**, a language specifically designed for statistical computing and data analysis. By the end of this chapter, you will have installed **R**, explored its basic syntax, and loaded and visualized a real-world dataset with just a few lines of code. No prior experience with programming is required, only curiosity and a willingness to learn. If you are already familiar with **R** and comfortable using RStudio, you may wish to skim this chapter and proceed directly to Chapter 2, where we introduce the data science workflow and its core concepts.

R or Python? Students often ask whether they should choose **R** or **Python**. While **Python** is a general-purpose language widely used for application development and deep learning, **R** was designed specifically for data analysis. It excels in statistical modeling, data visualization, and reproducible reporting. In practice, many teams use both, selecting the most appropriate tool for each task. Learners with a background in **Python** often find it easier to pick up **R**, as the two languages share many foundational concepts. For those who prefer **Python**, a companion volume, *Data Science Foundations and Machine Learning with Python: From Data to Decisions*, is available from the same publisher.

To give a concrete example, imagine working with customer data to understand why users cancel their mobile service. With **R**, you can summarize call behavior, compare usage patterns across churned and retained customers, and create intuitive plots to reveal trends. For instance, you might find that customers who churn tend to spend more time on daytime calls, an insight

that could inform proactive retention strategies. This kind of analysis is explored in greater detail in Chapter 4.

Throughout this book, we follow a structured framework known as the *Data Science Workflow*, which includes seven key steps: problem understanding, data preparation, exploratory data analysis, data setup for modeling, modeling, evaluation, and communication. Each chapter builds on this framework. The foundational skills introduced here, including navigating R, importing and manipulating data, and generating basic visualizations, will support your work at every stage. A detailed overview is provided in Chapter 2 (see Figure 2.3).

Why Choose R for Data Science?

R is widely used in statistics, data analysis, and visualization, owing to its rich ecosystem of packages tailored to data science tasks. Unlike general-purpose programming languages, it was designed specifically for statistical computing, making it especially well suited for both foundational and advanced analytical workflows. With concise, readable syntax, analysts and researchers can perform everything from descriptive statistics to machine learning.

Key strengths of R include statistical modeling, high-quality graphics, reproducible reporting, and domain-specific applications in fields such as epidemiology, psychology, and economics. As a free, open-source language with cross-platform support and an active global community, R offers thousands of user-contributed packages via the Comprehensive R Archive Network (CRAN) that extend its capabilities.

To support and streamline your coding experience, RStudio—an integrated development environment (IDE)—provides an intuitive interface that includes a console, script editor, and tools for managing plots, packages, and version control.

What This Chapter Covers

This chapter is designed for true beginners, readers who may have never written a line of code before. If you are new to R, programming, or data science, you are in the right place. Drawing on years of teaching experience, this chapter answers the most common questions students ask during lectures and labs—especially when encountering real data for the first time.

You are not expected to master every detail on your first read. Feel free to skip ahead or revisit sections as needed. This chapter is meant to serve as a practical, flexible reference: one you can return to throughout the book for reminders about syntax, data structures, visualization basics, or how to import and explore data in R.

In this chapter, you will:

- Set up your environment by installing R and RStudio;
- Learn to navigate the RStudio interface and run your first commands;
- Understand core data structures such as vectors, data frames, and lists;
- Import datasets, install packages, and explore your data;
- Create basic visualizations using the `ggplot2` package;
- Document your work with reproducible reports using R Markdown and Quarto.

By the end of this chapter, you will be able to load, explore, and visualize a real-world dataset with just a few lines of code, laying the foundation for your data science journey.

1.1 How to Learn R

Learning R opens the door to a wide range of powerful tools in data analysis, statistics, and machine learning. If you are new to programming, the learning curve may appear steep at first. However, consistent practice, thoughtful exploration, and the right resources make the journey both manageable and rewarding.

There is no single best way to learn R. Some learners benefit from structured textbooks, others from interactive exercises or video tutorials. A widely recommended resource is *R for Data Science* (2017), which emphasizes practical data workflows and readable code. For those entirely new to programming, *Hands-On Programming with R* (2014) offers an accessible entry point. Learners interested in machine learning might explore *Machine Learning with R* (2019). Interactive platforms like DataCamp and Coursera offer hands-on practice, while YouTube channels such as Data School provide visual explanations of core concepts. As you grow more confident, communities like Stack Overflow and the RStudio Community become invaluable for answering specific coding questions and learning from others.

Regardless of the path you take, the most effective strategy is regular, deliberate practice. Start with small tasks, explore how example code works, and

gradually build toward your own projects. Mistakes are not only inevitable, they are an essential part of the learning process.

This mindset is well captured by James Clear's concept of *The Power of Tiny Gains*, popularized in *Atomic Habits*: a 1% improvement each day may seem small, but it compounds into remarkable progress over time. Figure 1.1, created entirely in R, illustrates this idea, and previews what you will soon be able to do: write code that explores ideas, generates plots, and communicates insights clearly and reproducibly.

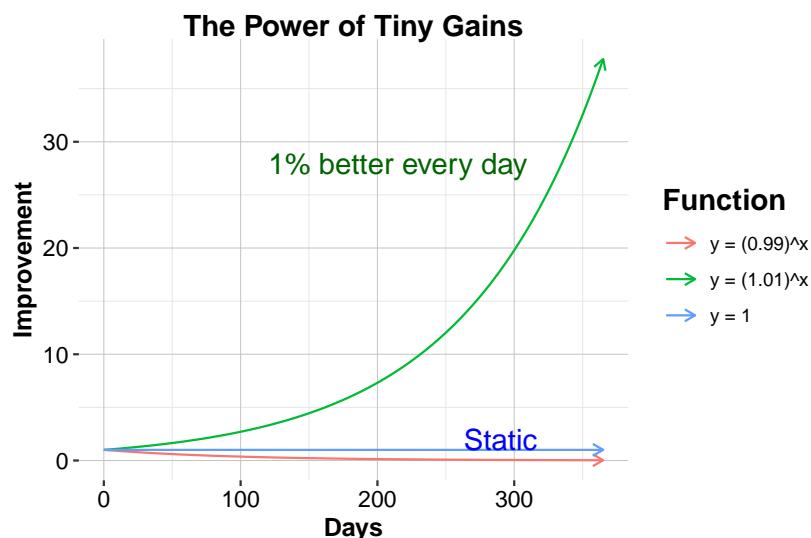


Figure 1.1: The Power of Tiny Gains: A 1% improvement every day leads to exponential growth over time. This plot was created entirely in R.

Learning R is not about mastering everything at once. It is about making steady progress, one concept at a time. With each small success—loading a dataset, creating a plot, writing a function—you are building skills that will support you throughout your data science journey. Be patient, stay curious, and trust the process. Growth in programming, as in data, is often exponential, just like the green line in Figure 1.1.

Now that you know what to expect and how to approach learning R, let us begin by setting up your environment. In the next section, you will install R and RStudio as your primary tools for writing and running R code.

1.2 How to Install R

Before you begin working with R, you need to install it on your computer. R is freely available from the Comprehensive R Archive Network (CRAN). Follow these steps:

1. Visit the [CRAN website](#).
2. Choose your operating system, Windows, macOS, or Linux.
3. Download the appropriate installer and follow the on-screen instructions to complete the installation.

Once installed, you can use the R console directly. However, most users prefer to work within an integrated development environment (IDE). In the next section, we introduce RStudio, a popular and user-friendly interface for writing and running R code.

Keeping R Up to Date

R is updated regularly, with a major release each year, typically in April, and several smaller updates in between. Staying current ensures access to new features, performance improvements, and security patches. It also helps maintain compatibility with the latest packages. That said, frequent updates are not essential for beginners. If your current setup works for your learning and analysis tasks, you can continue using it without interruption.

When upgrading to a new major version, you may need to reinstall your R packages. To simplify this process, you can save a list of your installed packages using:

```
installed.packages()[, 1]
```

Alternatively, use a package manager like **pak** or **renv** to snapshot and restore your setup. Although occasional updates may seem inconvenient, they help keep your tools stable and your workflow reliable in the long run.

1.3 How to Install RStudio

After installing the R software, it is helpful to use a dedicated set of tools that make working with R more productive and enjoyable. RStudio is a free and

open-source integrated development environment (IDE) designed specifically for R. It provides a clean interface, a powerful script editor, and integrated tools for plotting, debugging, and package management, all of which streamline the R programming experience.

Note: RStudio is an editor and does not include R itself. You must install R before using RStudio.

Installing RStudio

To install RStudio:

1. Visit the [RStudio website](#).
2. Download the latest version of RStudio Desktop (the free, open-source edition) for your operating system, Windows, macOS, or Linux.
3. Run the installer and follow the on-screen instructions.
4. Launch RStudio—you are now ready to begin coding in R.

RStudio is updated several times a year and typically notifies you when a new version is available. Keeping it up to date is recommended so that you can benefit from the latest features, stability improvements, and bug fixes.

Exploring the RStudio Interface

When you open RStudio for the first time, you will see a layout similar to Figure 1.2. This integrated interface is divided into four panels, each serving a specific purpose in your R workflow.

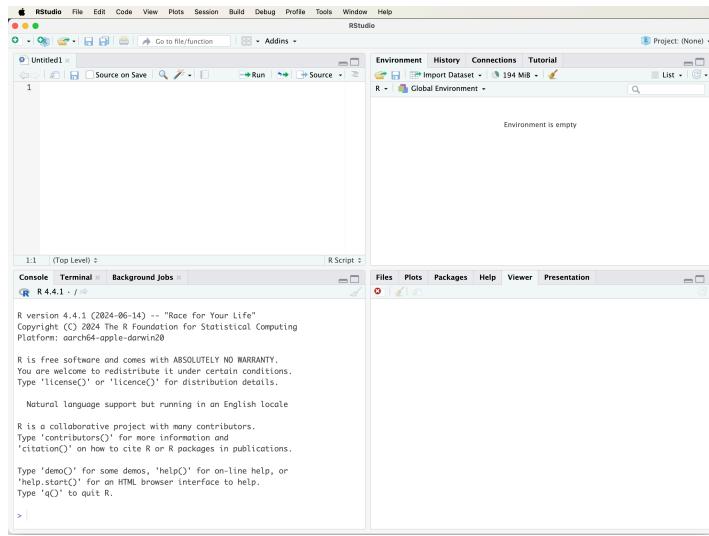


Figure 1.2: The RStudio window when you first launch the program.

If only three panels are visible initially, select *File > New File > R Script* to open the script editor. This panel enables you to write, edit, and save **R** code for future use and reproducibility.

The four main panels are:

- *Script Editor (top left)*: Write and edit **R** scripts.
- *Console (bottom left)*: Enter **R** commands and view their output.
- *Environment and History (top right)*: Inspect current variables and view previously executed commands.
- *Files, Plots, Packages, and Help (bottom right)*: Navigate files, visualize data, manage packages, and access documentation.

For now, focus on entering simple **R** commands into the console and pressing *Enter* to view the results. In later chapters, you will gradually make use of each panel as your projects grow in complexity.

Customizing RStudio

As you begin working more extensively with **R**, it is worth taking a few moments to tailor RStudio to your preferences. A well-organized and comfort-

able environment can make your coding sessions more productive and enjoyable.

To customize RStudio, go to:

Tools > Global Options

From there, you can explore several settings to personalize your workspace. For example:

- Under *Appearance*, you can change the editor theme (e.g., selecting *Tomorrow Night 80* for dark mode) to reduce eye strain. You can also adjust font size, pane layout, and syntax highlighting to suit your style.
- If installed, the *Copilot* tab allows you to enable GitHub Copilot for AI-assisted code suggestions.

These small adjustments not only improve comfort and focus but also support sustained engagement as you deepen your skills in **R**.

Now that your environment is set up, you might be wondering where to turn when you get stuck or want to deepen your understanding. The next section offers tips for getting help and continuing to learn.

1.4 Getting Help and Learning More

Learning **R** can be challenging at first—but you do not have to do it alone. Today’s learners have access to powerful support tools that can dramatically accelerate the process. Among them, AI assistants like ChatGPT offer a flexible, conversational way to learn faster, get unstuck, and deepen your understanding.

With an AI assistant, you can ask questions in natural language and receive immediate, context-aware guidance. Whether you are puzzled by an error message, unsure how a function works, or looking for help writing a block of code, tools like ChatGPT provide on-demand support. They can walk you through code step-by-step, generate examples based on your description, and explain unfamiliar concepts in clear, accessible language. This makes them especially helpful during independent study or when you need quick clarification without interrupting your workflow.

That said, AI tools are not the only resource. **R** includes robust built-in documentation, type `?function_name` or use `help()` and `example()` in the console to access function-specific help and sample usage. These features offer precise, official information and remain a fundamental resource.

Online communities also play an important role. Sites such as [Stack Overflow](#) and [RStudio Community](#) provide answers to countless common problems. Searching these forums often reveals solutions or helpful patterns. If you post a new question, remember to include a clear explanation and a reproducible example.

In the end, the best way to learn **R** is to combine resources: practice regularly, use AI tools to speed up your learning, consult the documentation for accuracy, and turn to the community for collective wisdom. With each small step, you will become more confident in your ability to explore data, write code, and solve problems with **R**.

1.5 Data Science and Machine Learning with R

Imagine building a model that predicts hospital readmissions, optimizing a marketing campaign, or uncovering fraud in financial transactions—all using code you understand and control. This book teaches you how, using **R**, one of the most powerful tools in the data scientist’s toolkit.

This book introduces the core ideas and practical tools of data science and machine learning, using **R** as the primary programming environment. You will learn how to prepare data, build models, evaluate results, and communicate insights—applying these skills to real-world datasets throughout the book.

R provides a solid foundation for statistical analysis and data visualization. Its real strength, however, lies in its extensive ecosystem of packages—modular libraries that extend **R**’s capabilities for modern analytics. To support these capabilities, **R** relies on a vast network of user-contributed packages developed by a global community of researchers and practitioners, freely available via the Comprehensive R Archive Network (CRAN).

While base **R** includes essential functions for data wrangling and basic modeling, many advanced techniques—especially in machine learning—are implemented in these contributed packages. A typical **R** package includes functions for specific tasks, sample datasets for experimentation, and documentation or vignettes to guide users.

Each machine learning method introduced in this book is paired with an appropriate package implementation. For example, in Chapter 11, we use **rpart** and **randomForest** for decision trees and ensemble learning. In Chapter 12, we use **neuralnet** to introduce the basics of neural network modeling. These tools are introduced with clear examples and hands-on exercises to help you apply them confidently.

To support the examples and exercises in this book, we developed the **liver** package. It includes real-world datasets and utility functions specifically designed for teaching data science with **R**. Several of these datasets are listed in Table 0.1. For example, in Chapter 7, we use one of these datasets to demonstrate the k-nearest neighbors (kNN) classification algorithm using functions from **liver**.

For those interested in going further, CRAN hosts thousands of additional packages covering areas such as text mining, forecasting, deep learning, and geospatial analysis. You can browse the full repository at <https://CRAN.R-project.org>.

As you progress through this book, you will become not only proficient in the **R** language but also fluent in using its rich ecosystem of packages to tackle real data science challenges. With curiosity and consistent practice, these tools will become part of your everyday workflow.

1.6 How to Install R Packages

Packages are central to working in **R**, they extend its core capabilities and allow you to perform specialized tasks such as data wrangling, modeling, and visualization. Many examples in this book use contributed packages. You will be prompted to install them as needed, starting with the **liver** package described below.

There are two common ways to install **R** packages: through RStudio's graphical interface or by using the `install.packages()` function in the **R** console.

To install a package using RStudio's interface:

1. Click on the *Tools* menu and select *Install Packages*....
2. In the dialog box, enter the name of the package (or multiple packages separated by commas).
3. Make sure the *Install dependencies* option is checked.
4. Click *Install* to begin.

See the screenshot in Figure 1.3 for a visual reference.

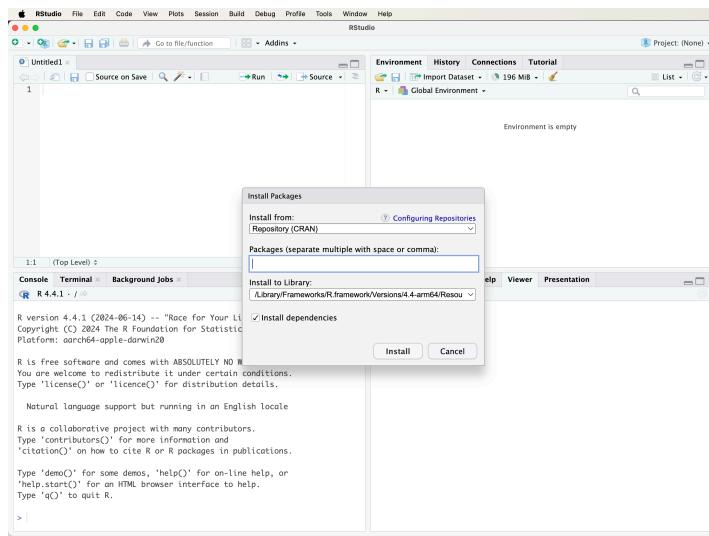


Figure 1.3: Installing packages via the graphical interface in RStudio.

The second, and more flexible, method is to install packages directly using the `install.packages()` function in the console. For example, to install the `liver` package, which provides datasets and functions used throughout this book, type:

```
install.packages("liver")
```

Press *Enter* to execute the command. R will download the package from CRAN and install it on your system. The first time you install a package, you may be prompted to select a CRAN mirror. Choose one geographically close to your location for faster downloads.

If installation fails, check your internet connection or confirm that a firewall is not blocking access to CRAN. The `install.packages()` function can also be used to install packages from a local file or a custom repository. To learn more, run:

```
?install.packages
```

Packages only need to be installed once. However, each time you open a new R session, you must load the package using the `library()` function. This will be explained in the next section.

1.7 How to Load R Packages

Once a package is installed, you need to load it into your R session before you can use its functions and datasets. R does not automatically load all installed packages; instead, it loads only those you explicitly request. This helps keep your environment organized and efficient, avoiding unnecessary memory use and potential conflicts between packages.

To load a package, use the `library()` function. For example, to load the `liver` package, enter:

```
library(liver)
```

Press *Enter* to execute the command. If you see an error such as "there is no package called 'liver'", the package has not yet been installed. In that case, return to Section 1.6 to review how to install packages using either RStudio or the `install.packages()` function.

While installing a package makes it available on your system, loading it with `library()` is necessary each time you start a new R session. Only then will its functions and datasets be accessible in your workspace.

As you progress through this book, you will use several other packages, such as `ggplot2` for visualization and `randomForest` for modeling, each introduced when needed. Occasionally, two or more packages may contain functions with the same name. When this occurs, R uses the version from the package most recently loaded.

To avoid ambiguity in such cases, use the `::` operator to explicitly call a function from a specific package. For example, to use the `partition()` function from the `liver` package (used for splitting data into training and test sets), type:

```
liver::partition()
```

This approach helps ensure that your code remains clear and reproducible, especially in larger projects where many packages are used together.

1.8 Running Your First R Code

One of the most empowering aspects of R is that it responds instantly to your commands. This immediate feedback makes it ideal for experimentation, learning, and developing intuition through trial and error. Suppose you

just made three online purchases and want to calculate the total cost. In **R**, you can do that instantly:

```
49.23 + 11.78 + 38.99  
[1] 100
```

Press *Enter*, and **R** returns the result. You can also try subtraction, multiplication, or division. Feel free to change the numbers and experiment with different operations.

You can store the result in a variable for later use:

```
total <- 49.23 + 11.78 + 38.99
```

The `<-` symbol is **R**'s assignment operator: it stores the value on the right under the name on the left. You can think of it as labeling a box and placing something inside. While the `=` operator (`total = ...`) also works, `<-` is the preferred convention, especially inside function calls.

Although `<-` is the traditional assignment operator, it is interchangeable with `=` in most situations. To keep the examples in this book beginner-friendly and consistent with other programming languages like **Python**, we will generally use `=` when assigning values. Both styles are valid, and you may choose the one that suits you best.

Once a value is stored, you can reuse it in future calculations. For example, to add 21% tax:

```
total * 1.21  
[1] 121
```

R retrieves the value of `total` and performs the calculation. Note that **R** is case-sensitive. This means `Total` and `total` refer to different objects. Always pay attention to capitalization when naming or calling variables and functions.

Try it yourself: What is the standard sales tax or VAT rate in your country? Replace `1.21` with your local multiplier (e.g., `1.07` for 7% tax) and rerun the code. You could also assign the rate to a variable like `tax_rate <- 1.21` to make your code more readable.

As you begin writing more lines of code, it becomes helpful to add comments that explain what each part is doing.

Using Comments to Explain Your Code

Comments help explain what your code is doing, making it easier to understand and maintain. In R, comments begin with a # symbol. Everything after # on the same line is ignored when the code runs.

Comments do not affect code execution but are essential for documenting your reasoning, whether for teammates, future readers, or even yourself after a few weeks. This is especially helpful in data science projects, where analyses often involve multiple steps and assumptions.

Here is an example with multiple steps and explanatory comments:

```
# Define prices of three items
prices <- c(49.23, 11.78, 38.99)

# Calculate the total cost
total <- sum(prices)

# Apply a 21% tax
total * 1.21
[1] 121
```

Clear comments turn code into a readable narrative, which helps others (and your future self) understand the logic behind your analysis.

1.8.1 How Functions Work in R

Functions are at the heart of R. They allow you to perform powerful operations with just a line or two of code—whether you are calculating a summary statistic, transforming a dataset, or creating a plot. Learning how to use functions effectively is one of the most important skills in your R journey.

A function typically takes one or more *arguments* (inputs), performs a task, and returns an *output*. For example, the `c()` function (short for “combine”) creates a vector:

```
# Define prices of three items
prices <- c(49.23, 11.78, 38.99)
```

Once you have a vector, you can use another function to compute a summary, such as the average:

```
mean(prices) # Calculate the mean of prices  
[1] 33.33333
```

The general structure of a function call in **R** looks like this:

```
function_name(argument1, argument2, ...)
```

Some functions require specific arguments, while others have optional parameters with default values. To learn more about a function and its arguments, type `? followed by the function name`:

```
?mean # or help(mean)
```

This opens the help documentation, including a description, argument list, and example usage.

You will encounter many functions throughout this book, from basic operations like `sum()` and `plot()` to specialized tools for machine learning. Functions make your code concise, modular, and expressive.

Throughout this book, you will use many built-in functions, often combining them to perform complex tasks in just a few lines of code. For now, focus on understanding how functions are structured and practicing with common examples.

1.9 Import Data into R

Before you can explore, model, or visualize anything in **R**, you first need to bring data into your session. Importing data is the starting point for any analysis—and **R** supports a wide range of formats, including text files, Excel spreadsheets, and datasets hosted on the web. Depending on your needs and the file type, you can choose from several efficient methods to load your data.

Importing Data with RStudio’s Graphical Interface

For beginners, the easiest way to import data into **R** is through RStudio’s graphical interface. In the top-right *Environment* panel, click the *Import Dataset* button (see Figure 1.4). A dialog box will appear, prompting you to choose the type of file you want to load.

You can choose from several file types depending on your data source and analysis goals. For example, text files such as CSV or tab-delimited files can be loaded using the *From Text (base)* option. Microsoft Excel files can be imported via the *From Excel* option, provided the **readxl** package is installed. Additional formats may appear depending on your installed packages and RStudio setup.

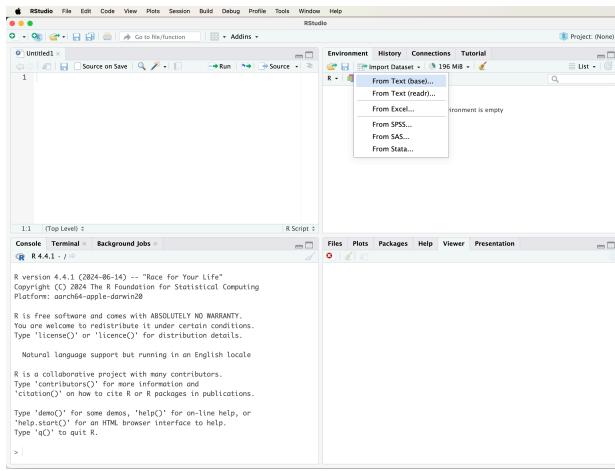


Figure 1.4: Using the ‘Import Dataset’ tab in RStudio to load data.

After selecting a file, RStudio displays a preview window (Figure 1.5) where you can review and adjust options like column names, separators, data types, and encoding. Once you confirm the settings, click *Import*. The dataset will be loaded into your environment and appear in the *Environment* panel—ready for analysis.

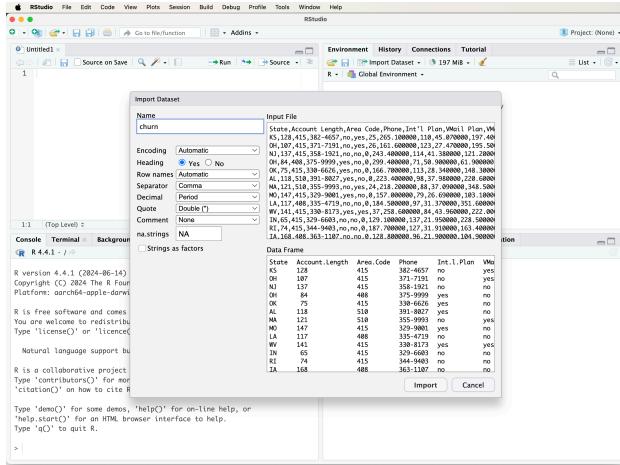


Figure 1.5: Adjusting import settings in RStudio before loading the dataset.

Importing CSV Files with read.csv()

If you prefer writing code, or want to make your analysis reproducible, you can load CSV files using the `read.csv()` function from base R. This is one of the most common ways to import data, especially for scripting or automating workflows.

To load a CSV file from your computer, use:

```
data <- read.csv("path/to/your/file.csv")
```

Replace "path/to/your/file.csv" with the actual file path. If your file does not include column names in the first row, set `header = FALSE`:

```
data <- read.csv("your_file.csv", header = FALSE)
```

If your dataset contains special characters, common in international datasets or files saved from Excel, add the `fileEncoding` argument to avoid import issues:

```
data <- read.csv("your_file.csv", fileEncoding = "UTF-8-BOM")
```

This ensures that R correctly interprets non-English characters and symbols.

Setting the Working Directory

The *working directory* is the folder on your computer where **R** looks for files to read, and where it saves any output by default. When you import data, **R** will search this folder unless you give a full path to the file.

To find out your current working directory, use:

```
getwd()
```

To change it in code:

```
setwd("~/Documents") # Adjust this path to match your system
```

You can also change it through the RStudio menu:

Session > Set Working Directory > Choose Directory...

Setting the working directory properly can save time and reduce errors when loading or saving files. If you're getting "file not found" errors, checking the working directory is a good first step.

Importing Data Using Dialogs and URLs

Sometimes you may not want to hard-code a file path or prefer to work with data hosted online. **R** offers flexible methods that make your workflow more portable and user-friendly.

To allow users to choose a file interactively, use the `file.choose()` function with `read.csv()`. This opens a file selection dialog box:

```
data <- read.csv(file = file.choose())
```

This is particularly handy when sharing code across different systems, where file locations may vary.

You can also import data directly from a URL by passing it to `read.csv()`:

```
data <- read.csv("https://example.com/data.csv")
```

This method is ideal for accessing public datasets from governments, research institutions, or data repositories. It ensures your code stays reproducible and eliminates the need for manual downloads.

Whether you are automating tasks or collaborating across platforms, these import options make it easier to load data without setup friction.

Importing Excel Files with read_excel()

Excel is one of the most common formats for storing data in business, education, and research. To import .xlsx or .xls files into **R**, use the `read_excel()` function from the `readxl` package, a lightweight, user-friendly tool in the tidyverse ecosystem.

If you have not installed the package yet, follow the guidance in Section 1.6, or use this command:

```
install.packages("readxl")
```

Then load the package and import your Excel file:

```
library(readxl)
data <- read_excel("path/to/your/file.xlsx")
```

Replace "path/to/your/file.xlsx" with the actual file path.

Unlike `read.csv()`, `read_excel()` supports multiple sheets in a single workbook. To import a specific sheet, use the `sheet` argument:

```
data <- read_excel("path/to/your/file.xlsx", sheet = 2)
```

This is especially useful when Excel files contain separate tables across different tabs. If your file includes merged cells, multi-row headers, or other formatting quirks, it is best to clean it in Excel first, or handle it programmatically in **R** after import.

Loading Data from R Packages

In addition to reading external files, **R** also provides access to datasets that come bundled with packages. These datasets are immediately usable and are ideal for practice, examples, and case studies.

In this book, we use the `liver` package, developed specifically for teaching purposes, which includes several real-world datasets. One of the main

datasets is `churn`, which contains information on customer behavior in a telecommunications context. If you have not installed the package yet, follow the guidance in Section 1.6.

To load the dataset into your environment, run:

```
library(liver) # To load the liver package  
data(churn)    # To load the churn dataset
```

Once loaded, `churn` will appear in your Environment tab and can be used like any other data frame. This dataset, along with others listed in Table 0.1, will appear throughout the book in examples related to modeling, evaluation, and visualization. In Chapter 4, you will perform exploratory data analysis (EDA) on `churn` to uncover patterns and prepare it for modeling.

Try it yourself: After loading `churn`, use `head(churn)` or `str(churn)` to explore its structure and variables.

Using datasets embedded in packages like `liver` ensures that your analysis is reproducible and portable across systems, since the data can be loaded consistently in any R session.

1.10 Why Data Types Matter in R

In R, every object—whether a number, string, or logical value—has a data type. These types play a critical role in how R stores, processes, and interprets data. Recognizing the correct type is essential for ensuring that computations behave as expected and that analyses yield valid results.

Here are the most common data types in R:

Numeric: Real numbers such as 3.14 or -5.67. Used for continuous values such as weight, temperature, or income; they support arithmetic operations.

Integer: Whole numbers like 1, 42, or -6. Integers are useful for counting, indexing rows, or representing categories with numeric codes.

Character: Text values such as "Data Science" or "Azizam". Character data is used for names, descriptions, labels, and other textual content.

Logical: Boolean values, TRUE or FALSE. Logical values are used for comparisons, filtering, and conditional statements.

Factor: Categorical variables with a defined set of levels (e.g., "Yes" and "No"). Factors are essential in modeling and grouped visualizations, where the variable should behave as a category rather than text.

To check the type of a variable, use the `class()` function:

```
class(prices)
[1] "numeric"
```

This tells you the broad data type **R** assigns to the variable. To inspect how **R** stores it internally, use `typeof()`. To explore complex structures like data frames, use `str()`:

```
typeof(prices)
[1] "double"
str(prices)
num [1:3] 49.2 11.8 39
```

Why does this matter? Treating a numeric variable as character—or vice versa—can cause functions to return incorrect results or warnings. For example:

```
income <- c("42000", "28000", "60000") # Stored as character
mean(income) # This will return NA with a warning
[1] NA
```

In this case, **R** interprets `income` as text, not numbers. You can fix the issue by converting the character vector to numeric:

```
income <- as.numeric(income)
mean(income)
[1] 43333.33
```

Later chapters—such as Exploratory Data Analysis (Chapter 4) and Statistical Inference (Chapter 5)—will show you how to apply tools specific to each variable type, whether for summarizing values, visualizing distributions, or building models.

Try it yourself: Load the `churn` dataset from the `liver` package (see Section 1.9). Then use `str(churn)` to inspect its structure. Which variables are numeric, character, or factors? Try using `class()` and `typeof()` on a few columns to explore how **R** understands them.

1.11 Data Structures in R

In **R**, data structures define how information is organized, stored, and manipulated. Choosing the right structure is essential for effective analysis—

whether you are summarizing data, creating visualizations, or building predictive models. For example, storing customer names and purchases calls for a different structure than tracking the results of a simulation.

Data structures are different from data types: data types describe *what* a value is (e.g., a number or a string), while data structures describe *how* values are arranged and grouped (e.g., in a table, matrix, or list).

The most commonly used structures in **R** include vectors, matrices, data frames, lists, and arrays. Each is suited to particular tasks and workflows. Figure 1.6 provides a visual overview of these core structures.

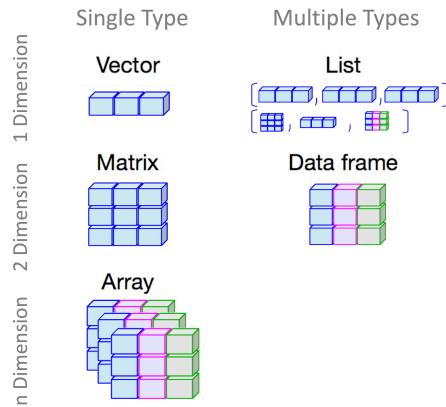


Figure 1.6: A visual guide to five common data structures in R, organized by dimensionality (1D, 2D, nD) and type uniformity (single vs. multiple types).

In this section, we explore how to create and work with the four most commonly used data structures in **R**: vectors, matrices, data frames, and lists—each with practical examples to illustrate when and how to use them.

Vectors in R

A *vector* is the most fundamental data structure in **R**. It represents a one-dimensional sequence of elements, all of the *same type*—for example, all numbers, all text strings, or all logical values (TRUE or FALSE). Vectors form the foundation of many other **R** structures, including matrices and data frames.

You can create a vector using the `c()` function (short for *combine*), which concatenates individual elements into a single sequence:

```
# Create a numeric vector representing prices of three items
prices <- c(49.23, 11.78, 38.99)

# Print the vector
prices
[1] 49.23 11.78 38.99

# Check if `prices` is a vector
is.vector(prices)
[1] TRUE

# Get the number of elements in the vector
length(prices)
[1] 3
```

In this example, `prices` is a numeric vector containing three elements. The `is.vector()` function checks whether the object is a vector, and `length(prices)` tells you how many elements it contains.

Note: All elements in a vector must be of the same type. If you mix types (e.g., numbers and characters), R will coerce them to a common type, usually character—sometimes with unintended consequences.

Matrices in R

A *matrix* is a two-dimensional data structure in R where all elements must be of the same type—numeric, character, or logical. Matrices are commonly used in mathematics, statistics, and machine learning for operations involving rows and columns.

To create a matrix, use the `matrix()` function. Here is a simple example:

```
# Create a matrix with 2 rows and 3 columns, filled row by row
my_matrix <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3, byrow = TRUE)

# Display the matrix
my_matrix
[,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6

# Check if it's a matrix
is.matrix(my_matrix)
[1] TRUE

# Check its dimensions (rows, columns)
dim(my_matrix)
[1] 2 3
```

This creates a 2×3 matrix filled *row by row* using the numbers 1 through 6. If you leave out the `byrow = TRUE` argument (or set it to `FALSE`), R fills the matrix *column by column*, which is the default behavior.

Matrices are useful in a wide range of numerical operations, such as matrix multiplication, linear transformations, or storing pairwise distances. They are the backbone of many machine learning algorithms and statistical models—most core computations in neural networks, support vector machines, and linear regression rely on matrix operations behind the scenes.

You can access specific elements using row and column indices:

```
# Access the element in row 1, column 2
my_matrix[1, 2]
[1] 2
```

This retrieves the value in the first row and second column. You can also label rows and columns using `rownames()` and `colnames()` for easier interpretation in analysis.

Try it yourself: Create a 3×3 matrix with your own numbers. Can you retrieve the value in the third row and first column?

Data Frames in R

A *data frame* is one of the most important and commonly used data structures in R. It organizes data in a two-dimensional layout—rows and columns—where each column can store a different data type: numeric, character, logical, or factor. This flexibility makes data frames ideal for tabular data, much like what you might encounter in a spreadsheet or database.

In this book, nearly all datasets—whether built-in or imported from external files—are stored and analyzed as data frames. Understanding how to work with data frames is essential for following the examples and building your own analyses.

You can create a data frame by combining vectors of equal length using the `data.frame()` function:

```
# Create vectors for student data
student_id <- c(101, 102, 103, 104)
name        <- c("Emma", "Bob", "Alice", "Noah")
age         <- c(20, 21, 19, 22)
grade       <- c("A", "B", "A", "C")
```

```
# Combine vectors into a data frame
students_df <- data.frame(student_id, name, age, grade)

# Display the data frame
students_df
  student_id  name age grade
  1          101 Emma  20    A
  2          102 Bob   21    B
  3          103 Alice 19    A
  4          104 Noah  22    C
```

This creates a data frame named `students_df` with four columns. Each row represents a student, and each column holds a different type of information. To confirm the object's structure, use:

```
class(students_df)
[1] "data.frame"
is.data.frame(students_df)
[1] TRUE
```

To explore the contents of a data frame, try:

```
head(students_df)      # View the first few rows
  student_id  name age grade
  1          101 Emma  20    A
  2          102 Bob   21    B
  3          103 Alice 19    A
  4          104 Noah  22    C
str(students_df)       # View column types and structure
'data.frame': 4 obs. of 4 variables:
 $ student_id: num 101 102 103 104
 $ name       : chr "Emma" "Bob" "Alice" "Noah"
 $ age        : num 20 21 19 22
 $ grade      : chr "A" "B" "A" "C"
summary(students_df)  # Summary statistics by column
  student_id      name         age      grade
  Min. :101.0  Length:4      Min. :19.00 Length:4
  1st Qu.:101.8 Class :character 1st Qu.:19.75 Class :character
  Median :102.5 Mode  :character  Median :20.50 Mode  :character
  Mean   :102.5                    Mean   :20.50
  3rd Qu.:103.2                    3rd Qu.:21.25
  Max.   :104.0                    Max.   :22.00
```

Accessing and Modifying Columns

You can extract a specific column from a data frame using the `$` operator or square brackets:

```
# Access the 'age' column
students_df$age
[1] 20 21 19 22
```

You can also use `students_df[["age"]]` or `students_df[, "age"]`—try each one to see how they work.

To modify a column—for example, to add 1 to each age:

```
students_df$age <- students_df$age + 1
```

You can also add a new column:

```
# Add a logical column based on age
students_df$is_adult <- students_df$age >= 21
```

This creates a new column called `is_adult` with TRUE or FALSE values.

Data frames are especially useful in real-world analysis, where datasets often mix numerical and categorical variables. For example, in this book, we frequently use the `churn` dataset from the `liver` package:

```
library(liver) # Load the liver package
data(churn) # Load the churn dataset

# Explore the structure of the data
str(churn)
'data.frame': 5000 obs. of 20 variables:
 $ state      : Factor w/ 51 levels "AK","AL","AR",...: 17 36 32 36 37 2
   ↪ 20 25 19 50 ...
 $ area.code   : Factor w/ 3 levels "area_code_408",...: 2 2 2 1 2 3 3 2
   ↪ 1 2 ...
 $ account.length: int 128 107 137 84 75 118 121 147 117 141 ...
 $ voice.plan  : Factor w/ 2 levels "yes","no": 1 1 2 2 2 2 1 2 2 1 ...
 $ voice.messages: int 25 26 0 0 0 0 24 0 0 37 ...
 $ intl.plan   : Factor w/ 2 levels "yes","no": 2 2 2 1 1 1 2 1 2 1 ...
 $ intl.mins   : num 10 13.7 12.2 6.6 10.1 6.3 7.5 7.1 8.7 11.2 ...
 $ intl.calls  : int 3 3 5 7 3 6 7 6 4 5 ...
 $ intl.charge : num 2.7 3.7 3.29 1.78 2.73 1.7 2.03 1.92 2.35 3.02 ...
 $ day.mins    : num 265 162 243 299 167 ...
 $ day.calls   : int 110 123 114 71 113 98 88 79 97 84 ...
 $ day.charge  : num 45.1 27.5 41.4 50.9 28.3 ...
 $ eve.mins   : num 197.4 195.5 121.2 61.9 148.3 ...
 $ eve.calls   : int 99 103 110 88 122 101 108 94 80 111 ...
 $ eve.charge  : num 16.78 16.62 10.3 5.26 12.61 ...
 $ night.mins  : num 245 254 163 197 187 ...
 $ night.calls : int 91 103 104 89 121 118 118 96 90 97 ...
 $ night.charge: num 11.01 11.45 7.32 8.86 8.41 ...
 $ customer.calls: int 1 1 0 2 3 0 3 0 1 0 ...
 $ churn       : Factor w/ 2 levels "yes","no": 2 2 2 2 2 2 2 2 2 2 ...
```

The `str()` function provides a concise overview of variable types and values—a key first step when working with a new dataset.

Try it yourself: Create a small data frame with three columns—one numeric, one character, and one logical. Then use `$` to extract or modify individual columns, and try adding a new column using a logical condition.

Lists in R

A *list* is a flexible and powerful data structure in R that can store a collection of elements of *different types and sizes*. Unlike vectors, matrices, or data frames—which require uniform data types across elements or columns—a list can hold a mix of objects, such as numbers, text, logical values, vectors, matrices, data frames, or even other lists.

Lists are especially useful when you want to bundle multiple results together. For example, model outputs in R often return a list that includes coefficients, residuals, summary statistics, and diagnostics—all within a single object.

To create a list, use the `list()` function:

```
# Create a list containing a vector, matrix, and data frame
my_list <- list(vector = prices, matrix = my_matrix, data_frame =
  ↵ students_df)

# Display the contents of the list
my_list
  $vector
  [1] 49.23 11.78 38.99

  $matrix
    [,1] [,2] [,3]
  [1,]    1    2    3
  [2,]    4    5    6

  $data_frame
    student_id name age grade is_adult
  1         101 Emma  21     A    TRUE
  2         102 Bob   22     B    TRUE
  3         103 Alice  20     A   FALSE
  4         104 Noah  23     C    TRUE
```

This list, `my_list`, includes three named components: a numeric vector (`prices`), a matrix (`my_matrix`), and a data frame (`students_df`). You can access individual components using the `$` operator, numeric indexing, or double square brackets:

```
# Access the matrix
my_list$matrix
  [,1] [,2] [,3]
[1,]   1   2   3
[2,]   4   5   6

# Or equivalently
my_list[[2]]
  [,1] [,2] [,3]
[1,]   1   2   3
[2,]   4   5   6
```

To inspect the structure of a list, use `str()`:

```
str(my_list)
List of 3
$ vector    : num [1:3] 49.2 11.8 39
$ matrix    : num [1:2, 1:3] 1 4 2 5 3 6
$ data.frame:'data.frame': 4 obs. of 5 variables:
..$ student_id: num [1:4] 101 102 103 104
..$ name      : chr [1:4] "Emma" "Bob" "Alice" "Noah"
..$ age       : num [1:4] 21 22 20 23
..$ grade     : chr [1:4] "A" "B" "A" "C"
..$ is_adult  : logi [1:4] TRUE TRUE FALSE TRUE
```

This gives you a compact overview of each component's type and contents, which is helpful for navigating large or nested lists.

Lists play an important role in R programming, especially in statistical modeling, simulations, and any task that involves returning or storing mixed results.

Try it yourself: Create a list that includes a character vector, a logical vector, and a small data frame. Try accessing each component using `$`, `[[]]`, and numeric indexing.

1.12 Accessing Columns and Rows in R

Once your data is loaded into R, you will often need to extract specific columns (variables) or rows (observations) for inspection, transformation, or visualization. Two common tools for this are the `$` operator and the `[]` (bracket) operator.

Using the \$ Operator

The \$ operator provides a quick and readable way to access a single column from a data frame or a named element from a list. For example, to access the name column in the students_df data frame:

```
students_df$name
[1] "Emma"  "Bob"   "Alice" "Noah"
```

This returns a vector containing the values in the name column.

You can also use \$ with lists. For instance, if you have a list named my_list:

```
my_list$vector
[1] 49.23 11.78 38.99
```

This retrieves the numeric vector stored in the vector element of the list.

Using the [] Operator

The square bracket operator [] is more flexible and powerful. For data frames, the general syntax is:

```
data[row, column]
```

To extract the first three rows of students_df:

```
students_df[1:3, ]
  student_id  name age grade is_adult
1       101 Emma  21    A    TRUE
2       102 Bob   22    B    TRUE
3       103 Alice  20    A   FALSE
```

To select specific columns, such as name and grade:

```
students_df[, c("name", "grade")]
  name grade
1  Emma     A
2  Bob      B
3 Alice     A
4 Noah     C
```

To extract a specific value—for example, the grade in the second row:

```
students_df[2, "grade"]
[1] "B"
```

Unlike \$, the bracket syntax allows you to use variable names programmatically and select multiple columns at once.

Note: The [] operator also works with lists, but its behavior differs depending on whether you use single ([]) or double ([[]]) brackets. For now, focus on using it with data frames.

Try it yourself: Load the churn dataset from the `liver` package. Extract the first seven rows using `churn[1:7,]`. Then, use subsetting to select only rows where the state is "OH" and display the churn status for those customers.

1.13 How to Merge Data in R

In real-world data analysis, information is often spread across multiple tables. Merging datasets allows you to combine related information—such as customer demographics and transaction histories—into a single structure for analysis. I have included this section early in the book because many of my students frequently ask how to combine data from different sources when working on their course projects. As soon as you begin working with real datasets, merging becomes a practical and essential skill—since the information you need rarely comes in a single file.

In R, the base function `merge()` provides a flexible way to join two data frames using one or more shared columns (known as *keys*):

```
merge(x = data_frame1, y = data_frame2, by = "column_name")
```

Here, `x` and `y` are the data frames to be merged, and `by` specifies the column(s) they have in common. If you're matching on multiple columns, you can pass a character vector to `by`.

Consider the following two example data frames:

```
df1 <- data.frame(id = c(1, 2, 3, 4),
                    name = c("Alice", "Bob", "David", "Eve"),
                    age = c(22, 28, 35, 20))

df2 <- data.frame(id = c(3, 4, 5, 6),
                    age = c(25, 30, 22, 28),
                    salary = c(50000, 60000, 70000, 80000),
                    job = c("analyst", "manager", "developer", "designer"))
```

To perform an *inner join* (keeping only rows with matching id values in both data frames):

```
merged_df <- merge(x = df1, y = df2, by = "id")
merged_df
  id  name age.x age.y salary      job
1 3  David    35    25 50000 analyst
2 4   Eve     20    30 60000 manager
```

To perform a *left join* (keeping all rows from df1 and merging matched data from df2), set all.x = TRUE:

```
merged_df_left <- merge(x = df1, y = df2, by = "id", all.x = TRUE)
merged_df_left
  id  name age.x age.y salary      job
1 1 Alice    22     NA     NA    <NA>
2 2 Bob     28     NA     NA    <NA>
3 3 David    35    25 50000 analyst
4 4 Eve     20    30 60000 manager
```

Other common options include:

- all.y = TRUE: *right join* (keep all rows from df2);
- all = TRUE: *full join* (keep all rows from both data frames).

If a row in one data frame has no match in the other, R will insert NA values in the unmatched columns.

While `merge()` is a powerful and versatile function, later in the book we will introduce the `dplyr` package, which provides more concise alternatives like `left_join()` and `full_join()`—especially helpful when working in the tidyverse style.

Tip: Always check the number of rows before and after a merge. If you see unexpected NAs or missing records, it may signal a mismatch in key values or column types.

1.14 Getting Started with Data Visualization in R

Have you ever looked at a dataset and asked, *What patterns are hiding here?* Visualization helps you answer that question—by turning raw numbers into compelling, informative graphics.

In data science, seeing is often understanding. Whether you are identifying patterns, checking assumptions, or presenting results, effective visuals

transform numbers into insight. As you will see in Chapter 4, exploratory data analysis (EDA) relies heavily on visual summaries to uncover trends, outliers, and relationships.

A key strength of R is its visualization ecosystem. From quick exploration to publication-ready figures, R makes it easy to build clear and powerful plots. The most widely used tool for this is the **ggplot2** package—celebrated for its consistency, modularity, and elegance.

There are two main systems for creating plots in R: the base graphics system and **ggplot2**. While base graphics work well for quick plotting, **ggplot2** offers a more structured and declarative approach, inspired by the *Grammar of Graphics*.

The name **ggplot2** reflects this foundation—a system for building statistical graphics by combining independent layers: data, aesthetics, geometries, and more. This modular design makes **ggplot2** intuitive to learn and flexible to use. In practice, this philosophy is implemented through the use of the + operator, which lets you construct a plot layer by layer—starting with a dataset and gradually adding visual components.

This book emphasizes **ggplot2**, and nearly all visualizations—including *Figure 1.1* introduced earlier—are built with it. Even a few lines of code can produce professional-quality output.

A typical **ggplot2** plot has three essential components:

- *Data*: the dataset to visualize.
- *Aesthetics*: how variables map to visual properties like position or color.
- *Geometries*: the visual elements: points, lines, bars, or boxes.

You add these layers using the + operator. This makes plots easy to build step by step—starting simple, and then refining with titles, colors, themes, or statistical summaries.

Figure 1.7 gives a visual summary of the seven main layers that form the backbone of any **ggplot2** visualization.

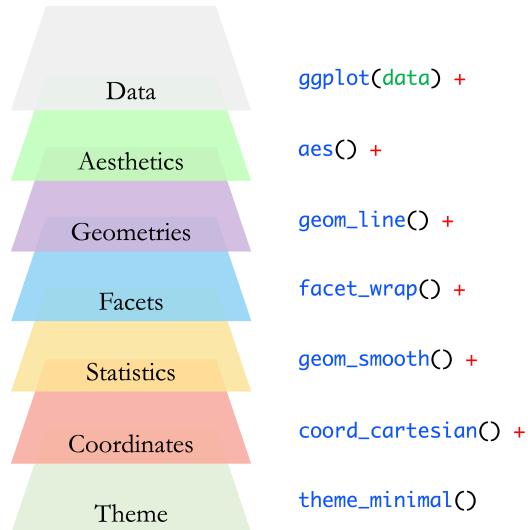


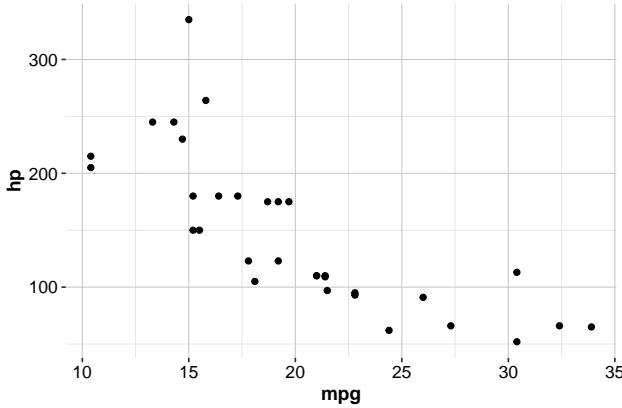
Figure 1.7: Grammar of Graphics and ggplot2 layers. The seven core layers of a ggplot: data, aesthetics, geometries, facets, statistics, coordinates, and theme.

Before using **ggplot2**, install the package as described in Section 1.6, then load it into your session:

```
library(ggplot2)
```

To see these ideas in action, consider the following simple scatter plot of miles per gallon (`mpg`) versus horsepower (`hp`) using the built-in `mtcars` dataset:

```
ggplot(data = mtcars) +
  geom_point(mapping = aes(x = mpg, y = hp))
```



In this example:

- `ggplot(data = mtcars)` initializes the plot with the dataset;
- `geom_point()` adds a layer of points;
- `aes()` defines the aesthetic mapping from variables to axes.

The general template for `ggplot2` plots is:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

Replace the angle brackets with your own inputs. This template provides a consistent framework for building plots incrementally—adding layers such as smoothing lines, facets, or custom themes. You will use this approach throughout the book to explore, analyze, and communicate insights from your data.

Geom Functions in ggplot2

In `ggplot2`, *geom functions* define what kind of plot you want to make. Each `geom_` function adds a *geometric object*—such as points, lines, or bars—as a new layer on top of your plot.

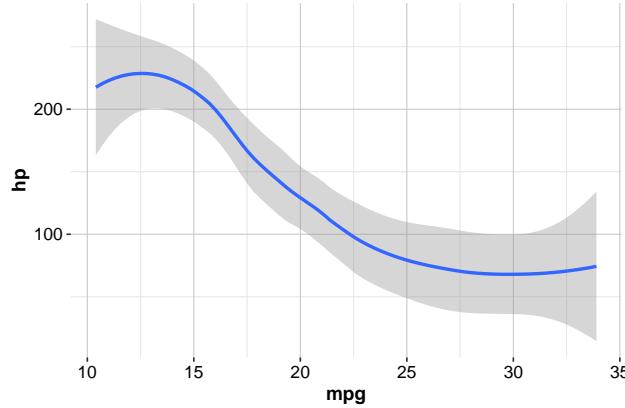
Here are some of the most commonly used geom functions:

- `geom_point()` creates a scatter plot;
- `geom_bar()` creates a bar chart;

- `geom_line()` creates a line chart;
- `geom_boxplot()` creates a box plot;
- `geom_histogram()` creates a histogram;
- `geom_density()` creates a smooth density curve;
- `geom_smooth()` adds a smoothed trend line (e.g., LOESS or linear fit).

For example, to visualize the relationship between miles per gallon (`mpg`) and horsepower (`hp`) in the `mtcars` dataset using a smoothed trend line:

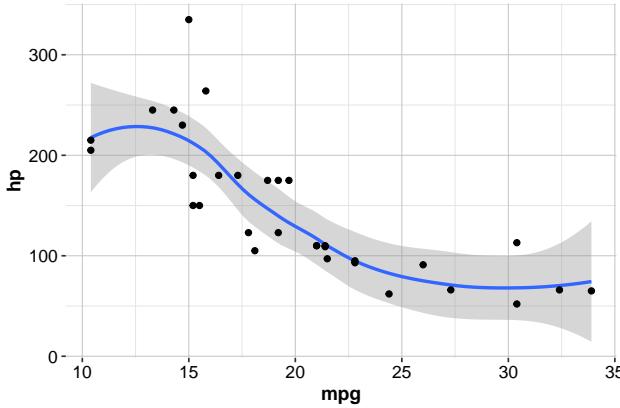
```
ggplot(data = mtcars) +  
  geom_smooth(mapping = aes(x = mpg, y = hp))
```



This plot shows the general pattern in the data, helping you see whether `mpg` tends to increase or decrease as `hp` changes.

You can also *combine multiple geoms* to enrich your plots. For instance, layering a scatter plot with a smooth trend line:

```
ggplot(data = mtcars) +  
  geom_smooth(mapping = aes(x = mpg, y = hp)) +  
  geom_point(mapping = aes(x = mpg, y = hp))
```



The order of layers matters: here, the trend line is plotted first, and the points are layered on top. This lets you preserve visibility while showing both the data and the overall pattern.

If multiple layers use the same aesthetic mappings, you can define them *once globally* inside the `ggplot()` call:

```
ggplot(data = mtcars, mapping = aes(x = mpg, y = hp)) +
  geom_smooth() +
  geom_point()
```

This avoids repeating the same mapping in each layer and keeps your code cleaner and more consistent.

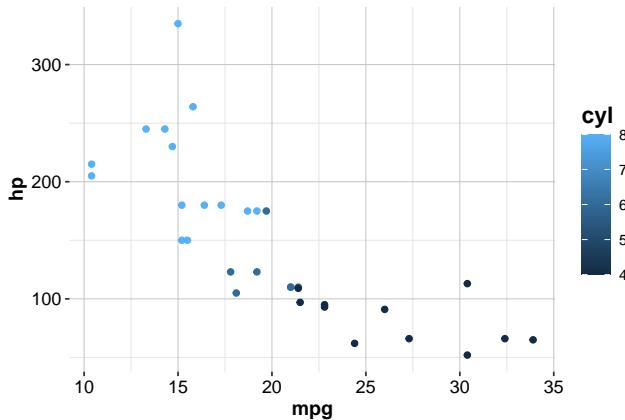
Try it yourself: Pick two numeric variables from the churn dataset—for example, `day.mins` (Day Minutes) and `eve.mins` (Evening Minutes). Use `geom_point()` to create a scatter plot. What relationship do you observe?

Aesthetics in `ggplot2`

Aesthetics in `ggplot2` define how data variables are visually mapped to features like color, size, shape, and transparency. These mappings are specified inside the `aes()` function and allow your plot to visually reflect differences in the data.

For example, to map the color of each point to the number of cylinders in a car:

```
ggplot(data = mtcars) +
  geom_point(mapping = aes(x = mpg, y = hp, color = cyl))
```

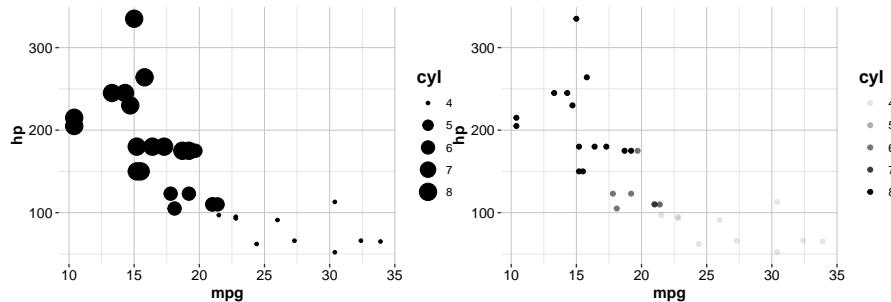


Because `color = cyl` is inside `aes()`, the color is *data-driven*. Each point's color corresponds to the number of cylinders (cyl), and `ggplot2` automatically adds a legend.

Other aesthetics you can vary include `size`, `alpha` (transparency), and `shape`:

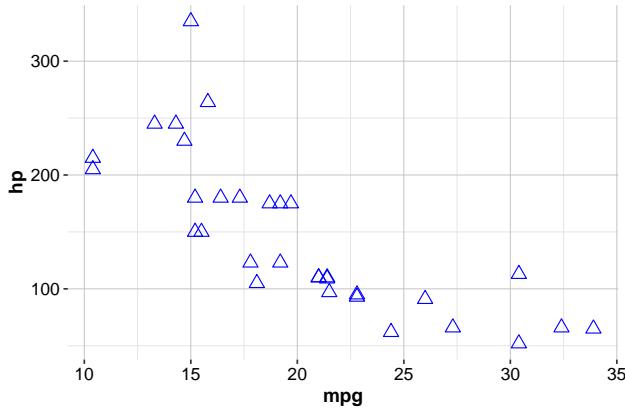
```
# Varying point size by number of cylinders
ggplot(data = mtcars) +
  geom_point(mapping = aes(x = mpg, y = hp, size = cyl))

# Varying transparency (alpha) by number of cylinders
ggplot(data = mtcars) +
  geom_point(mapping = aes(x = mpg, y = hp, alpha = cyl))
```



When aesthetics are placed *inside* `aes()`, they respond to the data. If you want to apply a constant appearance—such as making all points blue or triangles of the same size—set the attributes *outside* of `aes()`:

```
ggplot(data = mtcars) +
  geom_point(mapping = aes(x = mpg, y = hp),
             color = "blue", size = 3, shape = 2)
```



This creates a plot with fixed styling: all points are blue, triangular, and equally sized. Since the appearance is not based on the data, `ggplot2` does not generate a legend.

Try it yourself: In the churn dataset, can you map color to the churn variable and size to custserv.calls in a scatter plot of day.mins versus eve.mins? What do you observe?

With just a few core components—such as `geom_point()`, `geom_smooth()`, and `aes()`—you can construct expressive and insightful graphics. In Chapter 4, we will expand on this foundation to explore distributions, relationships, and trends as part of the EDA process.

For more details, see the [ggplot2 documentation](#). If you are curious about interactive visualizations, check out packages like [plotly](#) or [Shiny](#).

1.15 Formula in R

Formulas in R offer a simple and powerful way to describe relationships between variables. They are used extensively in statistical modeling—especially for regression, classification, and machine learning—to define how an outcome depends on one or more predictors.

A formula in R uses the tilde symbol `~` to separate the response variable (on the left) from the predictor variables (on the right). The basic structure looks like this:

```
response ~ predictor1 + predictor2 + ...
```

For example, in the diamonds dataset, you could model price as a function of carat, cut, and color using:

```
price ~ carat + cut + color
```

To include *all* other variables in the dataset as predictors, you can use a short-hand formula:

```
price ~ .
```

This is especially useful when working with large datasets where listing every predictor manually would be inefficient.

Behind the scenes, an R formula acts as a *symbolic object*. Rather than computing anything immediately, it instructs R to interpret variable names as columns in a dataset—making your modeling code both readable and flexible.

Conceptually, the left-hand side of `~` represents the *response* variable (what you want to predict), and the right-hand side lists the *predictors* (what you use to predict it). Mathematically, this can be thought of as:

$$y = f(x_1, x_2, \dots)$$

Here is a quick example using linear regression. Suppose we want to predict a diamond's price based on three features:

```
model <- lm(price ~ carat + cut + color, data = diamonds)
```

In this case, the formula defines the model structure, while the data argument tells R which dataset to use.

You will encounter formulas repeatedly throughout the book—in regression models, classification algorithms, and other statistical methods (see Chapters 7, 9, and 10). Learning this syntax early will help you build, interpret, and adjust models more effectively later on.

Try it yourself: Using the diamonds dataset, try fitting a model where price depends on carat and clarity. What happens if you use `price ~ .` instead?

1.16 Simulating Data in R

Some readers find simulation unfamiliar at first and may be tempted to skip this section. However, it is one of the most practical and empowering tools in data science. With just a few lines of code, you can generate realistic datasets to explore statistical concepts, test algorithms, or build examples when real data is unavailable.

In R, simulation is used extensively for both learning and modeling. It plays a key role in prototyping and debugging algorithms, teaching and illustrating statistical principles, generating synthetic datasets for model development, and testing methods under controlled, reproducible conditions.

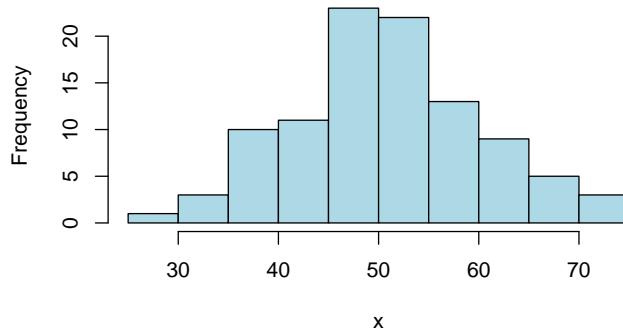
R offers many built-in functions to generate random values from common statistical distributions. Some of the most frequently used include:

- `rnorm(n, mean, sd)`: normal distribution (e.g., for heights or test scores),
- `runif(n, min, max)`: uniform distribution (e.g., for probabilities or noise),
- `rbinom(n, size, prob)`: binomial distribution (e.g., yes/no outcomes),
- `rexp(n, rate)`: exponential distribution (e.g., for time-to-event data),
- `sample(x, size, replace)`: sampling from a vector (e.g., categories or IDs).

To make your results reproducible, always use `set.seed()` before generating random values.

Example: Simulate 100 values from a normal distribution with mean 50 and standard deviation 10.

```
set.seed(123)
x <- rnorm(n = 100, mean = 50, sd = 10)
summary(x)
  Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 26.91  45.06  50.62  50.90  56.92  71.87
hist(x, main = "Simulated Normal Data", col = "lightblue")
```

Simulated Normal Data

You can also simulate multiple variables and combine them into a data frame to create small, realistic datasets.

Example: Create synthetic data for a small study on age and blood pressure.

```
set.seed(42)

age <- sample(30:80, 50, replace = TRUE)
bp  <- rnorm(50, mean = 120, sd = 15)

patients <- data.frame(age = age, blood_pressure = bp)

head(patients)
  age blood_pressure
1 78     93.55255
2 66    126.90146
3 30    110.40008
4 54    126.83175
5 39    130.57256
6 65    135.52655
```

Simulation is more than a teaching device—it is foundational in advanced techniques such as bootstrapping, Bayesian inference, and Monte Carlo methods. In this book (e.g., Chapter 7), you will use simulated data to explore and evaluate models with clarity and control.

Try it yourself: Simulate a dataset with 100 people. Assign each an age between 18 and 65, and an income drawn from a normal distribution with mean 40,000 and standard deviation 8,000. Can you visualize the income distribution using a histogram? What does a scatter plot of income vs. age reveal?

1.17 Reporting with R Markdown

Imagine you have just completed an in-depth analysis—your model predicts customer churn with 87% accuracy, the visualizations are compelling, and the insights could inform real business decisions. Now comes a crucial question: how will you present your findings?

Communicating insights is one of the most important—and often overlooked—steps in the Data Science Workflow. Analyses that are statistically sound and computationally rigorous have limited value if their conclusions are not conveyed clearly. Whether addressing technical collaborators, business stakeholders, or policymakers, you must transform complex results into a format that is both accessible and reproducible.

R Markdown is designed for this purpose. It provides a flexible and powerful environment for integrating code, output, and narrative in a single document. With R Markdown, you can weave together your analysis and explanation—producing reports, presentations, or dashboards that update automatically as your data changes.

This book was initially drafted entirely using R Markdown in combination with the [bookdown](#) package. This enabled automated output generation, version control integration, and full synchronization of code, figures, and tables—ensuring that every result remains accurate and reproducible across versions.

Unlike traditional word processors, R Markdown supports dynamic content. Reports compiled from .Rmd files are not just documents—they are living, executable records of your analysis. Supported output formats include HTML, PDF, Word, and PowerPoint, allowing you to tailor your communication to different audiences. With the addition of [Shiny](#), R Markdown can even produce interactive dashboards for real-time data exploration.

If you are new to R Markdown, two official resources provide practical and accessible starting points. The [R Markdown Cheat Sheet](#), also available in RStudio under *Help > Cheatsheets*, offers a concise overview of key syntax and features. For more advanced formatting and customization, the [R Markdown Reference Guide](#) provides detailed documentation and examples.

Try it yourself: Open RStudio, create a new R Markdown file, and type your first heading and code chunk. With just a few lines, you can generate a fully formatted HTML report that combines your analysis and your voice.

R Markdown Basics

How can you share an analysis in a way that includes your code, your reasoning, and your results—all in one place? This is the idea behind *literate programming*, a concept where code and narrative are written together in a single document. R Markdown follows this approach, allowing you to combine text, executable R code, and visual output in a fully reproducible format.

Unlike word processors that display formatting directly as you type, R Markdown separates content creation from rendering. You write in plain text and compile the document to produce the final output. During this process, R runs the code chunks, generates figures and tables, and inserts them automatically into the report. This workflow ensures that your narrative and results remain synchronized, even as the data changes.

To create a new R Markdown file in RStudio, navigate to:

File > New File > R Markdown

A dialog box will appear where you can choose the type of document you want to create. For a standard report, select “Document.” Other options include “Presentation” for slides, “Shiny” for interactive applications, and “From Template” for preconfigured formats. After entering a title and author name, choose the desired output format—HTML, PDF, or Word. HTML is typically recommended for beginners, as it compiles quickly and provides useful feedback during development.

R Markdown files use the .Rmd extension, distinguishing them from standard .R scripts. Each new file includes a built-in template that you can modify with your own text, code, and formatting. This serves as a useful starting point for learning the structure and syntax of R Markdown documents.

Try it yourself: Create a new R Markdown file and render it without changing anything. Then edit the title, add your own sentence, and run it again to see how the output updates.

The Header

At the top of every R Markdown file is a section called the *YAML header*, which serves as the control panel for your document. It contains metadata that determines how the document is rendered—such as the title, author, date, and output format. This header is enclosed between three dashes (---) at the beginning of the file.

Here is a typical example:

```
---
```

```
title: "Data Science is Cool"
author: "Your Name"
date: "June 01, 2025"
output: html_document
---
```

Each entry specifies a key element of the report:

- `title`: sets the title displayed at the top of the document.
- `author`: identifies the report's author.
- `date`: records the creation or compilation date.
- `output`: defines the output format, such as `html_document`, `pdf_document`, or `word_document`.

Additional customization options can be added to the header. For instance, to include a table of contents in an HTML report, you can modify the `output` field as follows:

```
output:
  html_document:
    toc: true
```

This is especially useful for longer documents with multiple sections, allowing readers to navigate more easily. Other options include setting figure dimensions, enabling syntax highlighting, or selecting a document theme. These settings offer precise control over both the appearance and behavior of your report.

In the next section, you will learn how to include code in your report and configure how it is displayed during rendering.

Code Chunks and Inline Code

One of the defining features of R Markdown is its ability to weave together code and narrative. This is accomplished through *code chunks* and *inline code*, which allow you to embed executable R commands directly within your report. As a result, your output—tables, plots, summaries—remains consistent with the underlying code and data.

A code chunk is a block of code enclosed in triple backticks (```) and marked with a chunk header that specifies the language (in this case, {r}). For example:

```
```{r}
2 + 3
```

```

```
[1] 5
```

When the document is rendered, R executes the code and inserts the output at the appropriate location. Code chunks are commonly used for data wrangling, statistical modeling, creating visualizations, and running simulations.

To run individual chunks interactively in RStudio, click the *Run* button at the top of the chunk or press **Ctrl + Shift + Enter**. See Figure 1.8 for a visual reference.

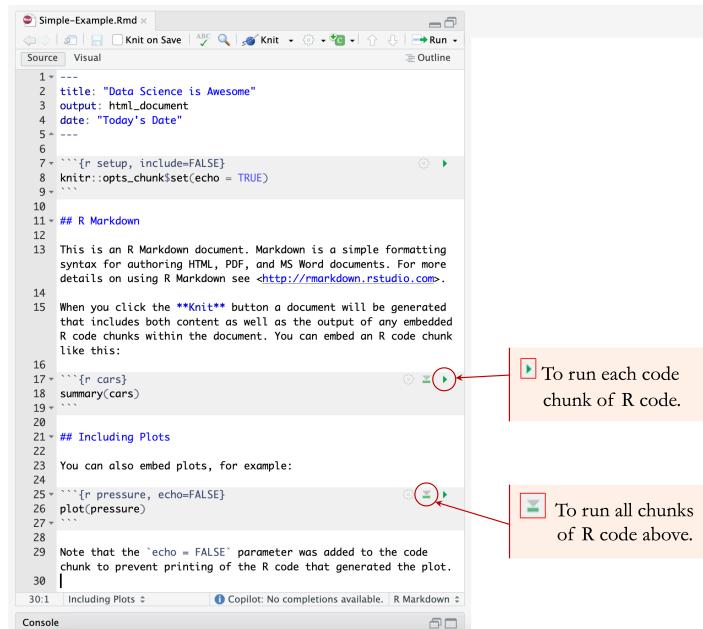


Figure 1.8: Executing a code chunk in R Markdown using the ‘Run’ button in RStudio.

Code chunks support a variety of options that control how code and output are displayed. These options are specified in the chunk header. For example:

- `echo = FALSE` hides the code but still displays the output.
- `eval = FALSE` shows the code but does not execute it.
- `message = FALSE` suppresses messages generated by functions (e.g., when loading packages).
- `warning = FALSE` hides warning messages.
- `error = FALSE` suppresses error messages.
- `include = FALSE` runs the code but omits both the code and its output.

Table 1.1 summarizes how these options affect what appears in the final report:

Table 1.1: Behavior of code chunk options in R Markdown and their impact on code execution, visibility, and output in rendered documents.

| Option | Run.Code | Show.Code | Output | Plots | Messages | Warnings | Errors |
|-------------------|----------|-----------|--------|-------|----------|----------|--------|
| 'echo = FALSE' | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 'eval = FALSE' | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 'message = FALSE' | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |
| 'warning = FALSE' | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ |
| 'error = FALSE' | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| 'include = FALSE' | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |

In addition to full chunks, you can embed small pieces of R code directly within text using *inline code*. This is done with backticks and the `r` prefix. For example:

The factorial of 5 is `r factorial(5)`.

This renders as:

The factorial of 5 is 120.

Inline code is especially useful when you want to report dynamic values—such as sample sizes, summary statistics, or dates—that update automatically whenever the document is recompiled.

Try it yourself: Create a new R Markdown file and add a code chunk that calculates the mean of a numeric vector. Then use inline code to display that mean in a sentence.

Styling Text

Clear, well-structured text is an essential part of any data report. In R Markdown, you can format your writing to emphasize key ideas, organize content, and improve readability. This section introduces a few core formatting tools that help you communicate effectively.

To create section titles and organize your document, use one or more # symbols to indicate heading levels. For example, # creates a main section, ## a subsection, and so on. Bold text is written by enclosing it in double asterisks (e.g., ****bold****), while italic text uses single asterisks (e.g., **italic**). These conventions mirror common Markdown syntax and work across all output formats.

Lists are created using * or - at the start of each line. For example:

```
* First item  
* Second item
```

To insert hyperlinks, use square brackets for the link text followed by the URL in parentheses, for example: [R Markdown website](<https://rmarkdown.rstudio.com>). You can also include images using a similar structure, with an exclamation mark at the beginning: ![Alt text](path/to/image.png).

R Markdown supports mathematical notation using LaTeX-style syntax. In-line equations are enclosed in single dollar signs, such as $\$y = \backslash\beta_0 + \backslash\beta_1 x\$$, while block equations use double dollar signs and appear centered on their own line:

```
Inline: \$y = \backslash\beta_0 + \backslash\beta_1 x$  
Block: $$ y = \backslash\beta_0 + \backslash\beta_1 x $$
```

Mathematical expressions render correctly in HTML and PDF formats; support in Word documents may be more limited.

For a full overview of Markdown formatting and additional options, see the [R Markdown Cheat Sheet](#).

Tip: Try editing a heading, adding emphasis to a word, or inserting a list in your own R Markdown document. Even small formatting improvements can make your writing easier to follow.

Mastering R Markdown

As your skills in **R** grow, R Markdown will become an increasingly powerful tool, not only for reporting results but also for building reproducible workflows that evolve with your projects. Mastery of this tool enables you to document, share, and automate your analyses with clarity and consistency.

Several resources can help you deepen your understanding. The online book [R Markdown: The Definitive Guide](#) provides a comprehensive reference, including advanced formatting, customization options, and integration with tools like **knitr** and **bookdown**. If you prefer structured lessons, the [R Markdown tutorial series](#) offers a step-by-step introduction to essential concepts and practices. For learners who enjoy interactive platforms, [DataCamp's R Markdown course](#) provides guided exercises. Finally, the [RStudio Community forum](#) is an excellent place to find answers to specific questions and engage with experienced users.

Throughout this book, you will continue using R Markdown—not just to document isolated analyses, but to support entire data science workflows. As your projects become more complex, this approach will help ensure that your code, results, and conclusions remain transparent, organized, and reproducible.

1.18 Reporting with Quarto

While R Markdown has long been a popular tool for reproducible reporting in **R**, **Quarto** represents its modern evolution. Designed to support multilingual workflows, Quarto provides a unified framework for combining narrative, code, and output across multiple programming languages, including **R**, **Python**, **Julia**, and **Observable JavaScript**.

Like R Markdown, Quarto enables dynamic document generation. But it extends these capabilities with enhanced support for cross-referencing, equation numbering, citations, and a wide range of publishing formats—including HTML, PDF, Word, slides, and websites—all from a single `.qmd` file.

This book was originally drafted using R Markdown. However, once we began developing a companion version in **Python**, we transitioned to Quarto. Its language-agnostic design allowed us to maintain a consistent workflow while supporting multiple programming languages in parallel.

If your work is primarily in **R**, R Markdown remains a mature and well-supported tool. But if you use multiple languages—or plan to create inter-

active documents, presentations, or websites—Quarto offers a more flexible and forward-looking solution.

To create a new Quarto document in RStudio:

File > New File > Quarto Document

A dialog will prompt you to select the document type and programming language. The syntax in .qmd files is similar to R Markdown, making the transition smooth for most users.

To explore the full capabilities of Quarto, visit the official documentation at quarto.org.

1.19 Chapter Summary and Takeaways

This chapter introduced the **R** programming language as the foundation for data science and machine learning. You installed **R** and RStudio, explored the RStudio interface, and executed your first commands. You also learned how **R** handles core data types and structures, how to import and examine real-world datasets, and how to visualize data effectively.

In addition, you were introduced to *reproducible reporting* using **R Markdown** and **Quarto**—two tools that integrate code, narrative, and results in a single dynamic document. These tools support transparency, collaboration, and long-term reproducibility, making them essential components of the data science toolkit.

Throughout this chapter, we emphasized foundational habits: *writing clean code*, *thinking systematically*, and *building scalable workflows*. These principles are essential at every stage of the *Data Science Workflow* and will serve as your compass throughout this book. Many of my students begin their journey with no prior programming experience. Yet by the end of the course, they confidently use **R** to explore real-world datasets, create meaningful visualizations, and draw actionable insights. If you are starting from scratch, take heart—this path is entirely achievable, and you are in good company.

Key takeaway: Mastering **R** is not about memorization—it is about developing a mindset for working with data. As illustrated in Figure 1.1, small, consistent improvements lead to substantial progress. Do not worry if everything feels unfamiliar right now. By working through the upcoming chapters step by step, you will gradually build fluency and confidence in using **R** to analyze data and communicate insights.

Now that you have installed **R**, explored its key concepts, and visualized data with **ggplot2**, it is time to apply what you have learned.

1.20 Exercises

Use the exercises below to reinforce your understanding of the tools and concepts introduced in this chapter. Begin with foundational tasks, then build toward more complex data exploration and visualization challenges.

Basic Exercises

1. Install **R** and RStudio on your computer.
2. Use `getwd()` to check your current working directory. Then use `setwd()` to change it to a location of your choice.
3. Create a numeric vector `numbers` with the values 5, 10, 15, 20, and 25. Calculate its mean and standard deviation.
4. Use the `matrix()` function to construct a 3×4 matrix filled with the numbers 1 through 12.
5. Build a data frame with the following columns:
 - `student_id`: integer.
 - `name`: character.
 - `score`: numeric.
 - `passed`: logical (set to TRUE or FALSE).

Display the first few rows using `head()`.

6. Install and load the **liver** and **ggplot2** packages. If installation fails, check your internet connection and CRAN access.
7. Load the `churn` dataset from the **liver** package. Display the first six rows with `head()`.
8. Use `str()` to inspect the structure of the `churn` dataset, and identify its variable types.
9. Use `dim()` to report the number of rows and columns in the dataset.
10. Apply `summary()` to generate descriptive statistics for all variables in `churn`.
11. Create a scatter plot of `day.mins` vs. `eve.mins` using **ggplot2**.
12. Create a histogram of the `day.calls` variable.
13. Create a boxplot of `day.mins`.

14. Create a boxplot of day.mins grouped by churn status. *Hint:* See Section 4.5.
15. Use `mean()` to compute the average number of customer service calls overall, and for customers who churned (`churn == "yes"`).
16. Create an R Markdown report that includes:
 - A title and your name,
 - A code chunk that explores the `churn` dataset,
 - At least one visualization.

Render the report to HTML.

More Challenging Exercises

17. Simulate a dataset of 200 patients using the following code. Then, use `summary()` to explore the dataset. This data will be used in Chapter 7.

```
# Simulate data for kNN
set.seed(10)

n = 200          # Number of patients
n1 = 90          # Number of patients with drug A
n2 = 60          # Number of patients with drug B
n3 = n - n1 - n2 # Number of patients with drug C

# Generate Age variable between 15 and 75
Age = sample(x = 15:75, size = n, replace = TRUE)

# Generate Drug Type variable with three levels
Type = sample(x = c("A", "B", "C"), size = n,
              replace = TRUE, prob = c(n1, n2, n3))

# Generate Sodium/Potassium Ratio based on Drug Type
Ratio = numeric(n)

Ratio[Type == "A"] = sample(x = 10:40, size = sum(Type == "A"),
                           replace = TRUE)

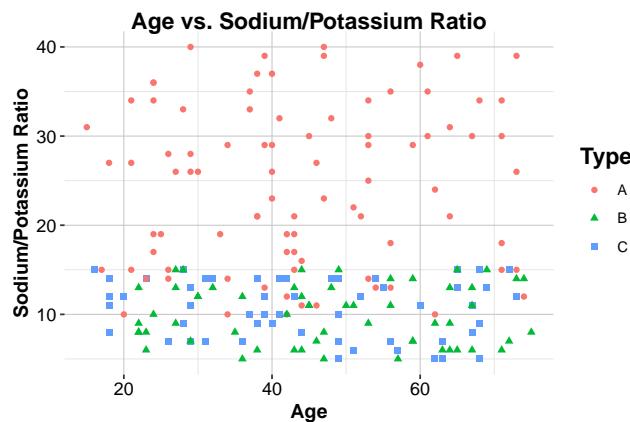
Ratio[Type == "B"] = sample(x = 5:15, size = sum(Type == "B"),
                           replace = TRUE)

Ratio[Type == "C"] = sample(x = 5:15, size = sum(Type == "C"),
                           replace = TRUE)

# Create a data frame with the generated variables
drug_data = data.frame(Age = Age, Ratio = Ratio, Type = Type)
```

18. Visualize the relationship between Age and Ratio, using color and shape to distinguish Type:

```
ggplot(drug_data, aes(x = Age, y = Ratio)) +
  geom_point(aes(color = Type, shape = Type)) +
  labs(title = "Age vs. Sodium/Potassium Ratio",
       x = "Age", y = "Sodium/Potassium Ratio")
```



19. Add an Outcome variable to drug_data where:

- Type == "A" has a high chance of "Good" outcome,
- Type == "B" or "C" have a lower chance of "Good".

20. Create a scatter plot of Age vs. Ratio, colored by Outcome.

21. Create an Age_group variable:

- "Young" if age ≤ 30 ,
- "Middle-aged" if $31 \leq \text{age} \leq 50$,
- "Senior" if age > 50 .

22. Calculate the mean Ratio for each Age_group.

23. Use **ggplot2** to create a bar chart showing average Ratio by Age_group.

24. Create a new variable Risk_factor = Ratio * Age / 10. Summarize how Risk_factor differs by Type.

25. Visualize Risk_factor in two ways:

- A histogram grouped by Type.
- A boxplot grouped by Outcome.

26. Use **R** and **ggplot2** to recreate Figure 1.1, which illustrates the compounding effect of small improvements. First, generate a data frame with three curves:

- $y = (1.01)^x$ (1 better each day),
- $y = (0.99)^x$ (1 worse each day),
- $y = 1$ (no change).

Then use `geom_line()` to plot the curves. Customize line colors and add informative labels using `annotate()`. *Hint:* Refer to the example in Section 1.14.

27. Extend the Tiny Gains plot created in Exercise 26 by:

- Changing the x-axis label to "Days of Practice",
- Applying a theme such as `theme_minimal()`,
- Adding a title: "The Power of Consistent Practice",
- Saving the plot using `ggsave()` as a PDF or PNG file.

28. For Exercise 26, change the number of days in your tiny gains plot. What do you observe if you compare 30 days to 365?

Reflect and Connect

These reflection questions encourage you to pause, assess your progress, and consider your goals as you continue.

29. Which concepts in this chapter felt most intuitive, and which ones did you find challenging?
30. How might these skills help you analyze data in your own research or field of study?
31. Looking ahead, what would you like to be able to do with **R** by the end of this book?

Chapter 2

Foundations of Data Science and Machine Learning

How does Netflix know what you want to watch, or how can a hospital predict patient risk before symptoms appear? Behind these intelligent systems lies the power of *data science* and *machine learning*. This chapter offers your entry point into that world—even if you have never written a line of code or studied statistics before.

Whether your background is in business, science, the humanities, or none of the above, this chapter is designed to be both accessible and practical. Through real-world examples, visual explanations, and hands-on exercises, you will explore how data science projects progress from raw data to meaningful insights, understand where machine learning fits in, and see why these tools are essential in today's data-driven world.

Data science is a fast-moving, interdisciplinary field that integrates computational, statistical, and analytical techniques to extract insights from data. In today's economy, data has become one of the world's most valuable assets—often referred to as the “*new oil*”, for its power to fuel innovation and transform decision-making.

Data science unleashes this potential by turning raw information into actionable knowledge. It draws on machine learning, programming, statistical modeling, and domain expertise to help organizations make evidence-based decisions, improve operations, anticipate trends, and build adaptive, intelligent systems. These capabilities are already transforming industries—from personalized healthcare and financial forecasting to targeted marketing and autonomous vehicles.

As the demand for data-driven solutions continues to grow, understanding the foundations of data science—and its deep connection to machine learning—has never been more important. This chapter introduces the core concepts, explores their societal relevance, and presents a structured workflow that guides data science projects from raw data to real impact.

While data science encompasses a wide variety of data types—including images, video, audio, and text—this book focuses on applications involving *structured, tabular data*. These are datasets commonly found in spreadsheets, relational databases, and logs. More complex forms of unstructured data analysis, such as computer vision or natural language processing, lie beyond the scope of this volume.

What This Chapter Covers

This chapter lays the groundwork for your journey into data science and machine learning. You will start by exploring what data science is, why it matters across diverse fields, and how it transforms raw data into real-world impact. You will then learn how data science projects are structured using a practical workflow that guides you from defining the problem to deploying a solution.

As the chapter progresses, you will gain insight into the role of machine learning as the modeling engine of data science. We break down its three main branches—supervised, unsupervised, and reinforcement learning—highlighting how they differ and where each is applied.

By the end of this chapter, you will have a high-level roadmap of how data science works in practice, what kinds of problems it can solve, and how the rest of this book will help you build and evaluate machine learning models with confidence. While this chapter offers a broad overview, the detailed methods, tools, and techniques will be introduced progressively in the chapters that follow.

2.1 What is Data Science?

Data science is an interdisciplinary field that combines mathematics, statistics, computer science, and domain knowledge to uncover patterns and generate actionable insights (see Figure 2.1). It integrates analytical methods and machine learning to process large, complex datasets and support informed decision-making, strategic planning, and innovation.

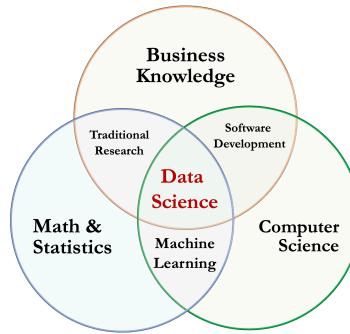


Figure 2.1: Venn diagram of data science (inspired by Drew Conway's original illustration). Data science is a multidisciplinary field that integrates computational skills, statistical reasoning, and domain knowledge to extract insights from data.

Although the term *data science* is relatively new, its foundations are rooted in long-established disciplines such as statistics, data analysis, and machine learning. What distinguishes modern data science is its scale and impact: the explosion of digital data, increasing computational resources, and the growing demand for intelligent systems have transformed it into a distinct and essential field.

At its core is *machine learning*—the study of algorithms that learn from data and improve their performance through experience. While statistical methods help summarize and infer from data, machine learning provides scalable approaches to discover complex patterns, automate predictions, and build adaptive systems. In this way, machine learning is not just a component of data science—it is one of its principal engines.

In practice, data science involves framing questions, analyzing data, developing models, and translating results into meaningful action. It draws on tools such as data visualization, predictive modeling, and domain-specific techniques to help organizations derive value from the information they generate and collect.

Key Components of Data Science

Modern data science integrates three core components that work together to transform data into insight and enable intelligent systems:

- *Data engineering* Focuses on the collection, storage, and organization of data. This includes building data pipelines and infrastructure that ensure data is reliable, scalable, and accessible. This book introduces essential techniques for data cleaning and preparation in Chapters 3 and 6. More advanced topics, such as distributed systems and real-time data processing, are beyond the scope of this volume. For further reading, see [Modern Data Science with R](#) (2017).
- *Statistical analysis and data visualization* These methods support both exploration and inference. They help uncover patterns, test hypotheses, and guide decisions throughout the modeling process. From visualizing trends to quantifying uncertainty, statistical thinking is essential for interpreting and communicating data. We focus on these skills in Chapters 4 and 5.
- *Machine learning* Enables algorithms to automatically learn patterns, make predictions, and adapt to new information. Techniques range from supervised learning to deep neural networks, and are introduced progressively in Chapters 7 through 13.

These components are developed step by step throughout the book, starting with data preparation and exploratory analysis and culminating in model building, evaluation, and deployment. Together, they provide the foundation for effective and reproducible data science practice.

2.2 Why Data Science Matters

Data is no longer just a byproduct of digital systems—it has become a core asset for innovation, strategy, and decision-making. Today’s most influential organizations—including OpenAI, Google, Apple, and Amazon—demonstrate how data, when combined with algorithms and computing power, can personalize products, optimize operations, and reshape entire industries.

In nearly every domain, data-driven decision-making is now essential. Organizations collect vast amounts of information every day—from sales transactions and web activity to clinical trials and financial records. Without the right tools and expertise, much of this data remains underutilized. Data science helps bridge this gap by identifying patterns, generating predictions, and delivering insights that support more adaptive, evidence-based decisions.

Data science influences a wide range of sectors, including finance, where it is used for credit scoring, fraud detection, algorithmic trading, and regulatory

compliance; marketing, where it enables audience segmentation and campaign optimization based on behavioral data; retail and e-commerce, where it supports inventory forecasting, dynamic pricing, and product recommendation systems; and healthcare, where it assists in early diagnosis, risk stratification, and personalized treatment planning.

Common technologies highlight these capabilities: Netflix recommends content based on viewing history, Amazon forecasts purchasing needs, and navigation systems dynamically reroute traffic. Such tools are built on structured historical data, statistical modeling, and iterative refinement.

To build systems like these, data scientists rely on a structured, repeatable approach. In the next section, we introduce the *data science workflow*—a practical framework that guides projects from initial questions to actionable insights.

2.3 The Data Science Workflow

Have you ever tried solving a complex problem—like diagnosing a patient, optimizing a delivery route, or designing a product—withouit a plan? In data science, structure is everything. Without it, even the most powerful algorithms can lead to misleading or irrelevant results.

That is why data science relies on a clear, organized workflow. The *data science workflow* provides a flexible yet disciplined approach to transforming messy data into actionable insights. It helps teams align their efforts, iterate thoughtfully, and ensure that results are accurate, reproducible, and useful.

This process is not linear—it is *iterative*. As you gain new insights, you often revisit earlier steps: redefining questions, adjusting features, or retraining models. This cyclical nature is what makes data science dynamic and adaptive.

A helpful way to visualize this transformation is through the *DIKW Pyramid*, which illustrates how raw *Data* becomes *Information*, then *Knowledge*, and finally *Wisdom* (see Figure 2.2).



Figure 2.2: The DIKW Pyramid illustrates the transformation of raw data into higher-order insights, progressing from data to information, knowledge, and ultimately wisdom.

To put this into action, we use a practical framework called *CRISP-DM* (Cross-Industry Standard Process for Data Mining). It guides data science projects through seven interconnected phases (see Figure 2.3):

1. *Problem Understanding* – Define the business or research goal and clarify what success looks like.
2. *Data Preparation* – Gather, clean, and format data for analysis.
3. *Exploratory Data Analysis (EDA)* – Use summaries and visualizations to understand distributions, spot patterns, and identify potential issues.
4. *Data Setup for Modeling* – Engineer features, normalize values, and select predictors.
5. *Modeling* – Apply machine learning or statistical models to uncover patterns and generate predictions.
6. *Evaluation* – Assess how well the model performs using appropriate metrics and validation.
7. *Deployment* – Integrate the model into real-world systems and monitor it over time.

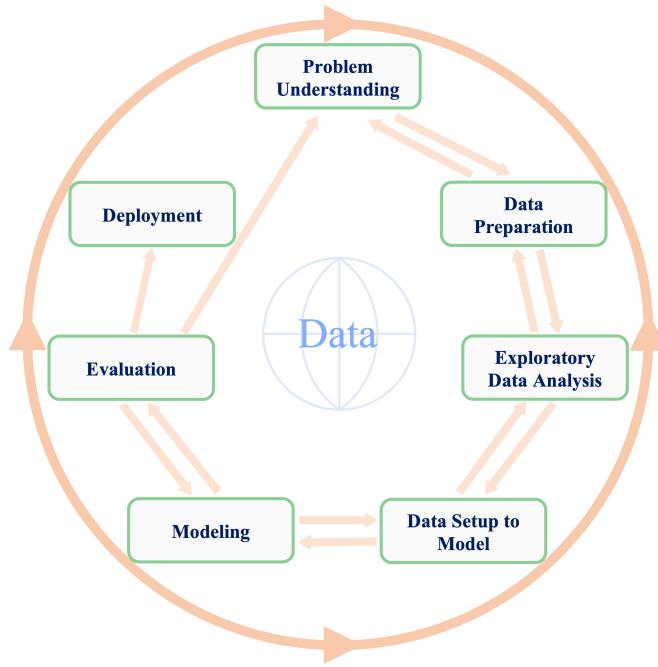


Figure 2.3: The Data Science Workflow is an iterative framework for structuring data science and machine learning projects. Inspired by the CRISP-DM model, it emphasizes reproducibility, continuous refinement, and impact-driven analysis.

Because real-world data is messy, and questions evolve, this process is rarely completed in one pass. A model that performs poorly might send you back to feature engineering. A surprising pattern during EDA might reshape your original question. Embracing this iterative process is key to success in data science—and to building systems that truly make an impact.

This book is structured around the Data Science Workflow. Each chapter corresponds to one or more stages in this process, guiding you step by step from problem definition to deployment. This is the same framework I use in my courses and when supervising BSc and MSc thesis projects in data science. By following this approach, students not only learn individual techniques, but also develop the process-oriented mindset essential for real-world practice.

In the remainder of this chapter, we walk through each stage of the Data Science Workflow—starting with problem understanding and moving through data preparation, modeling, and evaluation—illustrating how these steps connect and why each is essential for building effective, data-driven solutions.

2.4 Problem Understanding

Every data science project begins not with code or data, but with a clearly defined question. Whether the goal is to test a scientific hypothesis, improve business operations, or enhance user experience, success depends on understanding the problem clearly and aligning it with stakeholder needs. This initial stage in the Data Science Workflow ensures that projects address meaningful goals and lay the foundation for actionable outcomes.

A well-known example from World War II illustrates the importance of framing problems effectively: the case of [Abraham Wald](#) and the missing bullet holes. During the war, the U.S. military analyzed returning aircraft to determine which areas were most damaged. Bullet holes appeared mostly on the fuselage and wings, but few were found in the engines. Figure 2.4 illustrates this pattern, summarized in Table 2.1.

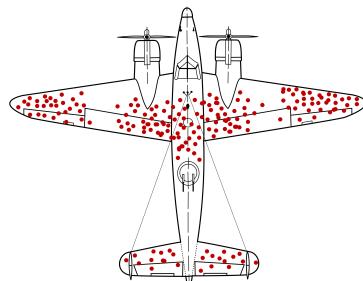


Figure 2.4: Bullet damage was recorded on planes that returned from missions—those hit in more vulnerable areas did not return. (Image: Wikipedia).

Table 2.1: Distribution of bullet holes per square foot on returned aircraft.

| Section.of.plane | Bullet.holes.per.square.foot |
|----------------------|------------------------------|
| Engine | 1.11 |
| Fuselage | 1.73 |
| Fuel system | 1.55 |
| Rest of plane | 0.31 |

Initial recommendations focused on reinforcing the most damaged areas. But statistician Abraham Wald came to a different conclusion: the data reflected only the planes that survived. The engines—where little damage was

observed—were likely the areas where hits caused planes to be lost. His counterintuitive insight was to reinforce the areas with no bullet holes.

This story highlights a central principle in data science: the most valuable insights may lie in what is missing, unseen, or misinterpreted. Without careful problem framing, even high-quality data can lead to flawed conclusions.

While platforms such as [Kaggle](#) often begin with clearly posed questions and pre-processed datasets, real-world data science rarely starts that way. In practice, analysts must begin with the first step in the Data Science Workflow: working with stakeholders to define objectives and determine how data can be used to address them. This ability to frame problems thoughtfully is one of the most important skills for a data scientist. The datasets used in this book—such as those in the [liver](#) package—are also pre-cleaned to support focused learning. But real-world projects often involve vague goals and messy data, making the problem formulation stage even more critical.

This phase involves close collaboration with business leaders, researchers, or domain experts to clarify objectives, define success criteria, and understand key constraints. Helpful questions include:

- *Why* is this question important?
- *What* outcome or impact is desired?
- *How* can data science contribute meaningfully?

As Simon Sinek emphasizes in his TED talk [“How Great Leaders Inspire Action”](#), effective efforts begin by asking *why*. In data science, understanding the deeper purpose behind a project leads to sharper analysis, better communication, and more relevant results.

For example, building a model to “predict churn” only becomes valuable when linked to a concrete goal—such as designing retention campaigns or estimating revenue loss. The way the problem is framed shapes what data is collected, which models are suitable, and how outcomes are evaluated.

Once the problem is well understood, the next step is connecting it to data. This translation is rarely straightforward and often requires both domain expertise and creative thinking. A technically correct model may still fall short if the underlying question is poorly formulated. To turn business or research goals into effective data science tasks, the following structured approach can be useful:

1. *Clearly articulate the project objectives* and requirements in terms of the overall goals of the business or research entity.
2. *Break down the objectives* to outline specific expectations and desired outcomes.

3. *Translate these objectives into a data science problem* that can be addressed using analytical techniques.
4. *Draft a preliminary strategy* for how to achieve these objectives, considering potential approaches and methodologies.

A well-scoped, data-aligned problem sets the stage for meaningful analysis. The next step is preparing the data to support that goal.

2.5 Data Preparation

With a clear understanding of the problem and its connection to data, we can now turn to preparing the dataset for analysis. This stage ensures that the data is accurate, consistent, and structured in a way that supports meaningful exploration and modeling.

In practice, raw data—whether collected from databases, spreadsheets, APIs, or web scraping—often contains issues such as missing values, outliers, duplicates, and incompatible variable types. If unaddressed, these issues can distort summaries, mislead models, or obscure important relationships.

This book focuses on structured, tabular data—the kind stored in spreadsheets, relational databases, and logs. While data science techniques also extend to unstructured sources such as images, audio, or text, those areas are beyond our scope here.

Typical tasks in data preparation include:

- Data collection and integration: Merging data from different sources, resolving schema mismatches, and aligning identifiers and time intervals.
- Handling missing values: Addressing gaps through deletion, imputation, or indicator variables that flag missingness.
- Outlier detection: Identifying unusual or extreme values that may represent input errors or highlight noteworthy patterns.
- Resolving inconsistencies: Standardizing formats, correcting typos, and consolidating duplicated categories.
- Feature engineering: Transforming or creating variables to improve interpretability or model performance.
- Data inspection and summarization: Verifying variable types, examining distributions, and checking for structural issues such as duplicate rows, mismatched keys, or inconsistent definitions.

While often labor-intensive, careful data preparation provides the foundation for accurate, interpretable, and reproducible analysis. Skimping on this phase often leads to unreliable results, regardless of the sophistication of the models that follow. In Chapter 3, we explore each of these techniques in detail, using real-world examples to illustrate their practical importance.

2.6 Exploratory Data Analysis (EDA)

Before we trust models to make predictions, we must first understand what our data is telling us. Exploratory Data Analysis (EDA) is the stage in the workflow where analysts examine data systematically—developing an informed perspective on its structure, quality, and key relationships.

EDA serves two complementary purposes. First, it plays a *diagnostic* role by revealing issues such as missing values, outliers, or inconsistent entries that may compromise later analyses. Second, it plays an *exploratory* role by uncovering patterns and associations that inform model building and feature engineering.

Several techniques are central to EDA:

- Summary statistics—including the mean, median, standard deviation, and interquartile range—provide insight into the distribution of numerical variables.
- Visualization tools such as histograms, scatter plots, and box plots help reveal patterns, anomalies, and group differences.
- Correlation analysis quantifies linear relationships between numeric variables and may suggest redundancy or predictive importance.

These tools support both data quality assessment and analytical decision-making. For example, a skewed variable may require transformation, or a highly correlated feature may suggest dimensionality reduction.

In R, EDA typically begins with functions like `summary()` and `str()` to examine structure and variable types. The `ggplot2` package provides a versatile framework for creating diagnostic and exploratory plots. We explore these techniques in greater detail in Chapter 4, using real-world datasets to demonstrate how EDA guides effective modeling and communication.

2.7 Data Setup for Modeling

After gaining a clear understanding of the data through exploratory analysis, the next step is to prepare it for modeling. This stage bridges exploration and prediction, shaping the dataset into a form that supports learning algorithms and ensures robust performance.

Several key tasks are typically involved:

- *Feature engineering* involves creating new variables or transforming existing ones to better capture the information relevant to the modeling goal. This may include combining variables, encoding categorical features numerically, or applying log transformations to reduce skewness.
- *Feature selection* focuses on identifying the most informative predictors while removing irrelevant or redundant ones. This helps reduce overfitting, enhance interpretability, and improve computational efficiency.
- *Rescaling* ensures that variables are on comparable scales. Methods such as normalization or standardization are especially important for algorithms that rely on distances or gradients, including k-nearest neighbors and support vector machines.
- *Data splitting* partitions the dataset into training, validation, and test sets. The training set is used to fit the model, the validation set supports tuning hyperparameters, and the test set provides an unbiased evaluation of model performance on unseen data.

Although often treated as a one-time step, this phase is iterative. Insights gained during modeling or evaluation may prompt revisions to feature engineering or variable selection. By the end of this stage, the dataset should be clean, informative, and structured to support effective and interpretable modeling.

We explore these techniques in depth in Chapter 6, using applied examples and reproducible R code to illustrate each concept in practice.

2.8 Modeling

Modeling is the stage where machine learning and statistical techniques are applied to the prepared data to uncover patterns, make predictions, or describe structure. The goal is to translate the insights gained during earlier stages—particularly data preparation and exploratory analysis—into formal models that can generalize to new, unseen data.

This stage requires a solid foundation in both statistical reasoning and algorithmic techniques. As such, it often benefits from prior experience and domain familiarity. It is also one of the most dynamic and rewarding phases of the workflow—where theoretical knowledge meets practical application, and patterns begin to emerge.

Key steps typically include model selection, which involves choosing an appropriate algorithm based on the nature of the task (regression, classification, or clustering), the structure of the data, and the broader analytical goals; model training, where the selected model is fitted to the training dataset to learn relationships between predictors and outcomes; and hyperparameter tuning, which adjusts model parameters (such as tree depth, number of neighbors, or learning rate) using techniques like grid search, random search, or cross-validation to improve performance.

The choice of model depends on considerations such as interpretability, computational efficiency, robustness, and predictive accuracy. In this book, we explore several widely used methods, including linear regression (Chapter 10), k-Nearest Neighbors (Chapter 7), Naïve Bayes classifiers (Chapter 9), decision trees and random forests (Chapter 11), and neural networks (Chapter 12).

Each technique has its strengths and trade-offs. In practice, multiple models are often tested and compared to identify the one that best balances predictive performance with interpretability and operational constraints.

Modeling is not the final step. It is typically followed by performance evaluation, where models are assessed for accuracy, generalizability, and relevance. In the next section, we explore how to evaluate models effectively using appropriate metrics and validation methods.

2.9 Evaluation

Once a predictive or descriptive model is built, the next step is to evaluate its performance using appropriate criteria. Evaluation plays a vital role in determining whether the model generalizes to new data, aligns with project objectives, and supports reliable decision-making. A careful assessment guards against overfitting and ensures that models are not only accurate, but also robust, interpretable, and trustworthy in practice.

The choice of evaluation metrics depends on the modeling task. For classification problems, accuracy measures the proportion of all cases correctly predicted. However, in imbalanced datasets, this metric can be misleading. Precision—the proportion of predicted positives that are truly positive—and recall—the proportion of actual positives that are correctly identified—

offer more targeted insight. Their harmonic mean, the F1-score, is especially useful when both types of error are costly. The area under the ROC curve (AUC) summarizes the trade-off between true positive and false positive rates across different thresholds.

In regression, the goal is to assess how closely predictions match observed values. The mean squared error (MSE) penalizes large deviations more heavily, while the mean absolute error (MAE) gives equal weight to all errors. The coefficient of determination (R^2) quantifies the proportion of variance in the outcome explained by the model.

To estimate performance reliably and guard against overfitting, cross-validation methods are widely used. Among them, k -fold cross-validation divides the data into multiple training and validation subsets, offering a more stable performance estimate across different segments.

Beyond numerical metrics, diagnostic tools help identify areas for improvement. In classification tasks, the confusion matrix reveals which categories are misclassified and why. In regression, residual plots can expose systematic errors that point to model misspecification or omitted features.

When performance does not meet expectations, the evaluation phase guides the next steps. Remedies may include refining the feature set, addressing class imbalance, tuning hyperparameters, or exploring alternative algorithms. In some cases, the issue lies not in the model, but in the original framing of the task—suggesting a return to the *problem understanding* stage. If, however, evaluation confirms that the model meets its objectives, the next step is deployment, where the model is integrated into production systems or decision-making pipelines. This flow—and the possibility of revisiting earlier stages—is reflected in the structure of the Data Science Workflow (Figure 2.3).

Chapter 8 explores these evaluation strategies in depth, providing practical examples and guidance on selecting metrics, interpreting diagnostics, and communicating results effectively. When the model is ready, we move to the final stage: putting it into action.

2.10 Deployment

Once a model has been rigorously evaluated and shown to meet project goals, the final step in the Data Science Workflow is deployment—integrating the model into a real-world system where it can deliver practical value. This may involve generating predictions in real time, supporting decision-making processes, or contributing to automated workflows.

Models can be deployed in a range of environments: embedded in software applications, connected to enterprise databases, or included in batch-processing pipelines. In professional settings, deployment typically involves collaboration between data scientists, software engineers, and IT personnel to ensure that the system is stable, secure, and scalable.

Importantly, deployment does not mark the end of a project. Ongoing monitoring is required to track performance and ensure reliability. As new data accumulates, the statistical properties of the input or target variables may shift—a phenomenon known as *concept drift*. For example, changes in user behavior, market conditions, or external regulations can affect the relevance of a model’s learned patterns. Without active monitoring, this can lead to performance degradation over time.

A robust deployment strategy should account for several factors:

- *Scalability* – Can the model handle increased data volume or usage?
- *Interpretability* – Can the predictions be explained to non-technical stakeholders?
- *Maintainability* – Can the model be updated, retrained, and audited efficiently?

In addition to production settings, deployment may take simpler forms—such as producing forecasts, interactive dashboards, or reproducible reports. Regardless of the form, deployment represents the point at which a model begins to inform decisions and generate impact.

While deployment is an essential component of the data science lifecycle, it is not the primary focus of this book. Our emphasis is on machine learning in practice—how to build, evaluate, and interpret models that learn from data. The next section introduces machine learning as a core engine of intelligent systems and lays the foundation for the modeling techniques explored in the remainder of this book.

2.11 Machine Learning: The Engine of Intelligent Systems

Machine learning is among the most dynamic and transformative areas of data science. It allows systems to identify patterns and make predictions from data, without relying on manually defined rules for every scenario. As data becomes increasingly abundant, machine learning offers scalable methods for turning information into actionable insights.

Whereas traditional data analysis focuses on describing what *has happened*, machine learning broadens the scope to include forecasting *what might happen next*. It powers technologies ranging from recommendation engines and fraud detection to medical diagnostics and autonomous vehicles, positioning it at the heart of modern decision-making and automation.

At its core, machine learning is a subfield of artificial intelligence (AI) dedicated to building algorithms that generalize from examples. While all machine learning is a form of AI, not all AI systems rely on learning from data—some use rule-based or logic-driven frameworks. What distinguishes machine learning is its capacity to improve through experience, making it particularly effective in complex, high-dimensional, or rapidly changing environments where static rules may fall short.

For example, in spam detection, rather than coding explicit rules to identify unwanted messages, a model can be trained on a labeled dataset of emails. It learns statistical patterns that distinguish spam from legitimate messages and generalizes this knowledge to new inputs. This ability to *learn from data* and adapt to new patterns over time is what enables intelligent systems to evolve.

As introduced in the Data Science Workflow (Figure 2.3), *modeling* is the stage where machine learning is typically applied. After problem definition, data preparation, and exploratory analysis, machine learning methods are used to construct predictive or descriptive models. This book emphasizes the practical application of these techniques: how to construct, evaluate, and interpret models that support data-informed decisions.

As illustrated in Figure 2.5, machine learning methods are commonly categorized into three types: *supervised learning*, *unsupervised learning*, and *reinforcement learning*. These categories reflect differences in how models learn from data and the types of problems they are designed to solve.

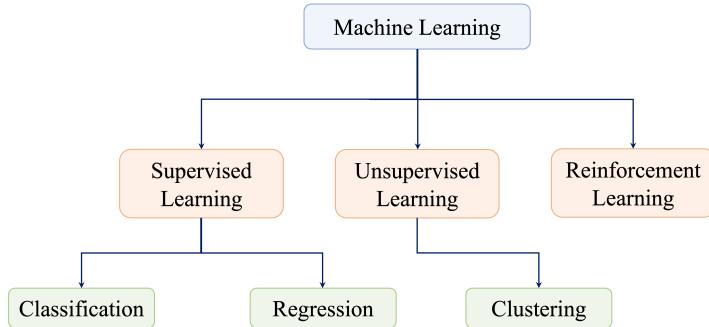


Figure 2.5: Machine learning tasks can be broadly categorized into supervised learning, unsupervised learning, and reinforcement learning, which differ in how models learn from data and what goals they pursue.

Table 2.2 summarizes how the main types of machine learning differ in terms of input data, learning objectives, and example applications.

Table 2.2: Comparison of supervised, unsupervised, and reinforcement learning tasks.

| Learning.Type | Input.Data | Goal | Example.Application |
|---------------|---------------------|--|--|
| Supervised | Labeled (X, Y) | Learn a mapping from inputs to outputs | Spam detection, disease diagnosis |
| Unsupervised | Unlabeled (X) | Discover hidden patterns or structure | Customer segmentation, anomaly detection |
| Reinforcement | Agent + Environment | Learn optimal actions through feedback | Game playing, robotic control |

In this book, we focus primarily on supervised and unsupervised learning, as they are most relevant for practical problems involving structured, tabular data. In the subsections that follow, we explore each of the three main branches of machine learning, beginning with supervised learning—the most widely used and foundational approach.

Supervised Learning: Learning from Labeled Data

Consider the task of detecting fraudulent credit card transactions. Historical data contains records labeled as either “fraud” or “legitimate,” and the goal is to build a model that learns from these examples to predict whether a fu-

ture transaction is likely to be fraudulent. This setup characterizes *supervised learning*.

Supervised learning is the most widely used approach in machine learning. It involves training a model on a dataset where each observation consists of input variables (features) and a corresponding known outcome (label). The model learns a relationship between the inputs—typically denoted as X —and the output Y , with the aim of making accurate predictions on new, unseen data. This learning process is summarized in Figure 2.6.

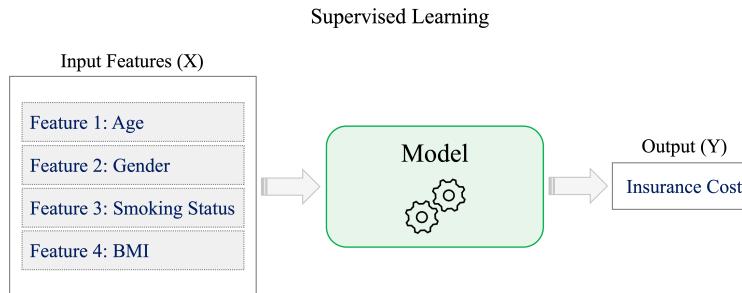


Figure 2.6: Supervised learning methods aim to predict the output variable (Y) based on input features (X).

Supervised learning problems fall into two major categories. In *classification*, the model assigns data points to discrete classes—for example, identifying spam emails or diagnosing whether a tumor is benign or malignant. In *regression*, the model predicts continuous values, such as housing prices or future product demand.

This learning paradigm powers many systems we interact with daily, from recommendation engines to credit scoring tools and automated medical diagnostics. In this book, we introduce a range of supervised learning techniques, including k-Nearest Neighbors (Chapter 7), Naïve Bayes classifiers (Chapter 9), decision trees and random forests (Chapter 11), and regression models (Chapter 10). Later chapters provide hands-on examples demonstrating how to implement these models, assess their accuracy, and interpret their results in real-world applications.

Unsupervised Learning: Discovering Structure Without Labels

How can we make sense of data when no outcomes are specified? This is the central aim of *unsupervised learning*: analyzing datasets without predefined

labels to uncover hidden patterns, natural groupings, or internal structure. In contrast to supervised learning—which relies on known outputs to guide learning—unsupervised learning is fundamentally *exploratory*. Its goal is to reveal how the data is organized, often without a specific target in mind.

Among unsupervised techniques, *clustering* is one of the most widely used and practically valuable. It groups similar observations based on shared characteristics, offering insight where no labels exist. Consider an online retailer seeking to better understand its customers. By clustering data on purchase histories and browsing behavior, the marketing team identifies three distinct segments: budget-conscious deal seekers, loyal repeat customers, and infrequent high-value buyers. This insight enables them to design targeted strategies—such as offering discounts to price-sensitive shoppers, loyalty rewards for repeat buyers, and personalized recommendations for premium customers—thereby increasing engagement and optimizing campaign effectiveness.

Clustering enables data-driven decision-making by revealing structure that may not be visible through summary statistics alone. It is particularly useful when labels are unavailable, expensive to collect, or when the objective is to explore the data's structure before applying predictive models.

We return to clustering in Chapter 13, where we examine how it can guide segmentation, anomaly detection, and pattern discovery using real-world examples.

Reinforcement Learning: Learning Through Interaction

Reinforcement learning is a distinct branch of machine learning that focuses on training agents to make a sequence of decisions by interacting with an environment. In contrast to supervised learning, which relies on labeled datasets, and unsupervised learning, which seeks patterns in unlabeled data, reinforcement learning is driven by experience. The agent receives feedback—typically in the form of numerical rewards or penalties—based on its actions, and uses this feedback to improve its behavior over time.

A central goal of reinforcement learning is to learn an optimal policy: a strategy that selects actions in each state to maximize the expected cumulative reward. This framework is especially effective in environments where decisions are interdependent and the consequences of actions may only become apparent after a delay.

This learning paradigm has enabled significant advances in fields such as robotics, where agents learn to navigate or manipulate physical environ-

ments; game-playing systems, including AlphaGo and OpenAI Five, which develop winning strategies through self-play; and dynamic decision systems used in adaptive pricing, inventory management, and personalized recommendations.

Although reinforcement learning is a powerful and rapidly evolving area of machine learning, it falls outside the scope of this book. Our focus is on supervised and unsupervised learning methods, which are more commonly used in applications involving structured data. For readers interested in reinforcement learning, the textbook *Reinforcement Learning: An Introduction* by Sutton and Barto (Sutton, Barto, et al. 1998) provides a comprehensive introduction to the topic.

2.12 Chapter Summary and Takeaways

This chapter introduced the foundational concepts that define the field of data science and its close connection to machine learning. We began by defining data science as an interdisciplinary field that transforms raw data into actionable insights, drawing from statistics, programming, and domain expertise. Through real-world examples, we illustrated its growing relevance across diverse domains—from healthcare to finance.

A central focus was the *Data Science Workflow*, a structured yet iterative framework that guides projects from problem understanding through data preparation, modeling, evaluation, and deployment. This workflow serves as the backbone for the rest of the book, helping you relate individual techniques to the broader process of building data-driven solutions.

We also explored *machine learning* as the core engine of modern modeling. You learned how supervised learning uses labeled data to predict outcomes, how unsupervised learning reveals structure in unlabeled datasets, and how reinforcement learning enables agents to learn through feedback and interaction. A comparison of these approaches clarified their inputs, objectives, and typical applications.

Key takeaways:

- *Data science is not just about data*—it is about asking meaningful questions, structuring analysis, and generating insight.
- *The workflow matters*—progress comes not from isolated steps but from thoughtful iteration across stages.
- *Machine learning enables automation and prediction*—and is most effective when applied within a structured process guided by clear goals.

In the next chapter, we turn to *data preparation*—the practical starting point for most data science projects. You will learn how to clean, structure, and transform raw datasets into a form ready for analysis and modeling.

2.13 Exercises

The exercises below are designed to reinforce the key ideas introduced in this chapter. They progress from basic definitions and conceptual understanding to applied scenarios, model evaluation, ethical considerations, and personal reflection. These questions support both individual learning and classroom discussion, and they are intended to help you build fluency with the data science workflow and the role of machine learning within it.

Warm-Up Questions

1. Define *data science* in your own words. What makes it an interdisciplinary field?
2. How does *machine learning* differ from traditional rule-based programming?
3. Why is *domain knowledge* important in a data science project?
4. What is the difference between *data* and *information*? How does the DIKW Pyramid illustrate this transformation?
5. How is machine learning related to *artificial intelligence*? In what ways do they differ?

Exploring the Data Science Workflow

6. Why is the *Problem Understanding* phase essential to the success of a data science project?
7. The Data Science Workflow is inspired by the *CRISP-DM* model. What does CRISP-DM stand for, and what are its key stages?
8. Identify two alternative methodologies to CRISP-DM that are used in the data science industry.

9. What are the primary goals of the *Data Preparation* stage, and why is it often time-consuming?
10. List common data quality issues that must be addressed before modeling can proceed effectively.

Applied Scenarios and Case-Based Thinking

11. For each of the following scenarios, identify the most relevant stage of the Data Science Workflow:
 - a. A financial institution is developing a system to detect fraudulent credit card transactions.
 - b. A city government is analyzing traffic sensor data to optimize stoplight schedules.
 - c. A university is building a model to predict which students are at risk of dropping out.
 - d. A social media platform is clustering users based on their interaction patterns.
12. Provide an example of how exploratory data analysis (EDA) can influence feature engineering or model selection.
13. What is *feature engineering*? Give two examples of engineered features from real-world datasets.
14. Why is it important to split data into training, validation, and test sets? What is the purpose of each?
15. How would you approach handling missing data in a dataset that includes both numerical and categorical variables?

Applying Machine Learning Methods

16. For each task below, classify it as *supervised* or *unsupervised* learning, and suggest an appropriate algorithm:
 - a. Predicting housing prices based on square footage and location.
 - b. Grouping customers based on purchasing behavior.
 - c. Classifying tumors as benign or malignant.
 - d. Discovering topic clusters in a large collection of news articles.

17. What is *overfitting*? How can it be detected and prevented?
18. Describe the role of *cross-validation* in model evaluation.
19. Provide an example where classification is more appropriate than regression, and another where regression is preferable.
20. What are the trade-offs between *interpretability* and *predictive performance* in machine learning models?

Evaluation, Bias, and Fairness

21. Why is *accuracy* not always a reliable metric for classification, especially in imbalanced datasets?
22. Suppose only 2% of cases in a dataset represent the positive class. Which metrics would be more informative than accuracy?
23. What is a *confusion matrix*, and how is it used to calculate *precision*, *recall*, and *F1-score*?
24. Define *bias* and *variance* in the context of machine learning. What is the *bias-variance trade-off*?
25. Provide an example of how bias in training data can lead to biased model predictions in deployment.
26. List three practices data scientists can adopt to reduce algorithmic bias and promote fairness.

Broader Reflections and Ethics

27. To what extent can data science workflows be automated? What are the potential risks of full automation?
28. Describe a real-world application in which machine learning contributed to a major positive societal impact.
29. Describe a real-world example where the use of machine learning led to controversy or harm. What could have been done differently?
30. How do *ethics*, *transparency*, and *explainability* influence public trust in machine learning systems?
31. Reflect on your own learning: Which aspect of data science or machine learning are you most interested in exploring further, and why?

Chapter 3

Data Preparation in Practice: Turning Raw Data into Insight

You have just been handed a spreadsheet with hundreds of columns and thousands of rows—missing entries, inconsistent labels, strange codes like -999, and values that make no sense. How do you begin to turn this chaos into insight?

In this chapter, we explore one of the most underestimated yet indispensable stages in the Data Science Workflow: *data preparation*. No matter how sophisticated your algorithm, the quality of your data will ultimately shape the trustworthiness of your results. It is often said that data scientists spend *up to 80% of their time* preparing data—and with good reason.

In real-world settings, data rarely arrives in a clean, analysis-ready format. Instead, it often includes missing values, outliers, inconsistent entries, and a mix of numerical and categorical variables. Preparing such data means more than just “cleaning”; it involves thoughtful transformation, encoding, and restructuring to support both exploration and modeling.

By contrast, datasets found on platforms like [Kaggle](#) are typically curated and neatly labeled, often with a clear target variable and well-posed questions. These are excellent for learning, but they can give a misleading impression of what real data science work entails. In practice, data is collected for operational—not analytical—purposes, and significant effort is required to make it useful for decision-making.

To bring data preparation techniques to life, we begin with the *diamonds* dataset from the `ggplot2` package—a structured and relatively clean dataset that allows us to practice foundational skills. Later in the chapter, we shift to the *adult* income dataset, where we apply the same techniques to a more realistic, messier context.

Although this chapter focuses on data preparation, many of the steps—such as detecting outliers, handling missing values, and transforming variables—overlap with *exploratory data analysis* (Chapter 4) and *data setup for modeling*.

(Chapter 6). These stages are often revisited in practice, reflecting the iterative nature of the Data Science Workflow.

With a clear understanding of the problem and its connection to the available data, we now move to the next step: preparing the dataset for analysis. This step builds the foundation for meaningful insights and reliable machine learning models—transforming raw information into structured knowledge.

What This Chapter Covers

This chapter introduces the essential techniques for transforming raw data into a format suitable for analysis and modeling. You will learn how to:

- Spot and manage outliers using visualization and filtering to reduce distortion,
- Detect and impute missing values to preserve data quality and completeness,
- Scale numerical features using Min-Max and Z-score methods to ensure comparability,
- Encode categorical variables through ordinal, one-hot, and frequency encoding for use in machine learning algorithms.

To ground these techniques, we begin by clarifying the analytical goals of our first dataset—the *diamonds* dataset—and frame the task of price estimation as a data science problem. The chapter concludes with a hands-on case study using the *adult* income dataset, where you will apply the full data preparation pipeline in a more complex, real-world setting. Together, these skills form a crucial step in the Data Science Workflow and prepare you to build models with clean, structured, and meaningful data.

3.1 Data Preparation in Action: The *diamonds* Dataset

How can we quantify the value of a diamond? What determines why two stones that appear nearly identical fetch vastly different prices? In this section, we bring the concepts of data preparation to life using the *diamonds* dataset—a rich, structured collection of gem attributes from the **ggplot2** package. It serves as our testing ground for exploring how to clean, transform, and organize data in ways that reveal meaningful insights.

Our central goal is to understand how features such as carat, cut, color, and clarity contribute to price variation. But before we begin processing the data, we must first frame the business problem and identify the analytical questions it raises. Data preparation is not merely technical—it is purposeful.

We focus on three guiding questions: which features have the strongest influence on diamond price; whether consistent pricing patterns emerge based on attributes like carat weight or cut quality; and whether the dataset contains anomalies—such as outliers or inconsistent entries—that must be addressed before modeling.

From a business perspective, answering these questions supports smarter pricing and inventory strategies for jewelers and online retailers. From a data science perspective, it ensures that our preprocessing choices—such as filtering outliers or encoding variables—are aligned with the real-world task at hand. This alignment between domain understanding and technical preparation is what makes data preparation both effective and impactful.

Later in the book, we will return to this dataset in Chapter 10, where we use the features prepared here to build a predictive regression model—completing the journey from raw data to actionable insight.

Loading and Exploring the diamonds Dataset

With a clear understanding of the problem and the analytical goals, we are ready to begin working directly with the data. The first step in data preparation is to load the dataset and explore its structure to identify what needs to be cleaned, transformed, or encoded.

We use the *diamonds* dataset from the **ggplot2** package—a structured collection of gem characteristics such as carat weight, cut, color, clarity, and price. This dataset serves as a clean but realistic foundation for practicing core data preparation techniques.

The dataset includes over 50,000 rows, with each observation representing a unique diamond. It provides 10 variables describing physical attributes and quality ratings. To follow along in R, make sure the **ggplot2** package is installed. If it is not yet installed, install it with `install.packages("ggplot2")`. Then load the package and access the dataset:

```
library(ggplot2)    # Load ggplot2 package  
data(diamonds)     # Load the diamonds dataset
```

To get a sense of the dataset's structure, use the `str()` function:

```
str(diamonds)
#> #> #> #> tibble [53,940 x 10] (S3: tbl_df/tbl/data.frame)
#> #> #> #> $ carat  : num [1:53940] 0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22
#> #> #> #>   ↘ 0.23 ...
#> #> #> #> $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<...
#> #> #> #>   ↘ ...
#> #> #> #> $ color   : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<...
#> #> #> #>   ↘ ...
#> #> #> #> $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<...
#> #> #> #>   ↘ ...
#> #> #> #> $ depth   : num [1:53940] 61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1
#> #> #> #>   ↘ 59.4 ...
#> #> #> #> $ table   : num [1:53940] 55 61 65 58 58 57 57 55 61 61 ...
#> #> #> #> $ price   : int [1:53940] 326 326 327 334 335 336 336 337 337 338 ...
#> #> #> #> $ x       : num [1:53940] 3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
#> #> #> #> $ y       : num [1:53940] 3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78
#> #> #> #>   ↘ 4.05 ...
#> #> #> #> $ z       : num [1:53940] 2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49
#> #> #> #>   ↘ 2.39 ...
```

This command reveals the number of observations and variables, along with data types and sample values. The dataset includes numerical features such as `carat` (weight), `price`, and physical dimensions (`x`, `y`, `z`), as well as categorical attributes like `cut` (quality of cut), `color` (ranging from D to J), and `clarity` (from I1 to IF). These features are central to the modeling task we will revisit in Chapter 10.

Here is a summary of the key variables:

- `carat`: weight of the diamond (ranging from 0.2 to 5.01),
- `cut`: quality of the cut (Fair, Good, Very Good, Premium, Ideal),
- `color`: color grade, from D (most colorless) to J (least colorless),
- `clarity`: clarity grade, from I1 (least clear) to IF (flawless),
- `depth`: total depth percentage calculated as $2 * z / (x + y)$,
- `table`: width of the top facet relative to the widest point,
- `x`, `y`, `z`: physical dimensions in millimeters,
- `price`: price in US dollars (ranging from \$326 to \$18,823).

Before we can begin cleaning or transforming the data, we need to understand the types of features we are working with. Different types of variables require different preparation strategies. In the next section, we examine how the features in the diamonds dataset are structured and classified.

3.2 Identifying Feature Types

Before detecting outliers or encoding variables, it is crucial to understand what kinds of features we are working with. Whether a variable is numerical

or categorical—and what subtype it belongs to—will shape how we clean, transform, and model the data. Feature types influence which visualizations make sense, which preprocessing steps are appropriate, and how machine learning algorithms interpret input values.

Figure 3.1 summarizes the main types of features commonly encountered in data science:

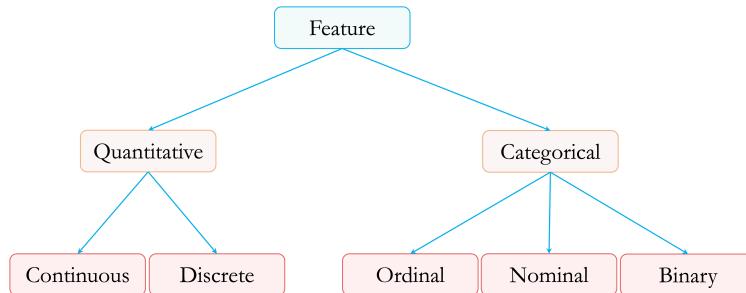


Figure 3.1: Overview of common feature types used in data analysis, including numerical (continuous and discrete) and categorical (ordinal, nominal, and binary) variables.

Features fall into two broad categories—*quantitative (numerical)* and *categorical (qualitative)*—each with important subtypes.

Quantitative (Numerical) features represent measurable quantities:

- *Continuous features* can take any value within a range. In the *diamonds* dataset, variables such as carat, price, x, y, z, and depth are continuous.
- *Discrete features* take on countable values, typically integers. Although not present in this dataset, an example might be the number of prior purchases or visits.

Categorical (Qualitative) features classify observations into groups:

- *Ordinal features* have a meaningful order, though the spacing between levels is not necessarily equal. The *diamonds* dataset includes ordinal variables like cut, color, and clarity. For example, color ranges from D to J—an alphabetical scale that reflects a progression from most to least colorless.
- *Nominal features* represent categories with no inherent order, such as product types or blood types.

- *Binary features* have exactly two categories, such as “yes”/“no” or “male”/“female”, typically encoded as 0 and 1.

Although the *diamonds* dataset does not contain discrete, nominal, or binary features, these are frequently encountered in real-world datasets and require distinct handling strategies.

Understanding feature types helps determine which preprocessing techniques are appropriate. Numerical features often benefit from normalization or standardization, typically using methods such as Min-Max Scaling or Z-score Scaling. Ordinal features are usually encoded with ordinal encoding to preserve their ranking, although one-hot encoding may be used when the rank is not relevant to the analysis. Nominal features, lacking any inherent order, are best handled using one-hot encoding to represent each category as a separate binary feature.

In **R**, how a variable is stored affects its treatment during analysis. Continuous variables are usually stored as `numeric`, while discrete ones are often `integer`. Categorical variables appear as `factor` objects, which may be either ordered or unordered. You can inspect these using `str()` for the full dataset, or use `class()` and `typeof()` to examine specific variables.

Tip: Always check how **R** interprets your variables. For example, a feature that is conceptually ordinal may be treated as a nominal factor unless explicitly declared as `ordered = TRUE`.

Now that we understand the structure of our features, we are ready to begin preparing them—starting with identifying outliers that may distort our analysis.

3.3 Key Considerations for Data Preparation

Effective data preparation serves as the bridge between raw input and meaningful statistical analysis. To prepare the *diamonds* dataset for modeling, we focus on three interrelated priorities: ensuring data quality, engineering relevant features, and transforming variables into suitable formats.

First, data quality is essential. We need to ensure that the dataset is accurate, consistent, and free from major issues. This includes identifying missing values, spotting outliers, and resolving inconsistencies that could introduce bias or reduce model performance.

Second, thoughtful feature engineering can add substantial value. For example, rather than using the `x`, `y`, and `z` dimensions separately, we might create a

new variable that captures the *volume* of each diamond. This derived feature could serve as a more interpretable and effective predictor of price.

Finally, appropriate data transformation ensures that all features are in a format suitable for modeling. Categorical variables like cut or color need to be encoded numerically, using methods that respect their structure and meaning. At the same time, numerical features may require scaling or normalization to ensure they contribute fairly in algorithms that rely on distance or gradient-based optimization.

These three pillars—data quality, feature engineering, and transformation—form the foundation of robust data preparation and help ensure that our modeling efforts are grounded in clean, well-structured, and informative data.

3.4 Outliers: What They Are and Why They Matter

Outliers are observations that stand out from the general pattern of a dataset—extreme values that differ markedly from the rest. They can arise from data entry errors, unusual measurement conditions, or genuinely rare but meaningful events. Regardless of their source, outliers can have an outsized impact on data analysis: they may distort summary statistics, skew visualizations, and mislead machine learning models.

Recognizing and handling outliers appropriately is especially important in real-world applications. In finance, an unusually large transaction could indicate fraud. In healthcare, an extreme lab result might point to a rare diagnosis—or signal a faulty instrument. In manufacturing, sensor readings that deviate from the norm might flag equipment failure or process instability.

However, not all outliers are errors to be discarded. Some reflect valuable exceptions that offer new insights. Removing them indiscriminately risks throwing away useful information; keeping them blindly can undermine model reliability. Sound judgment—grounded in both statistical reasoning and domain knowledge—is essential when deciding how to handle them.

Outlier detection can begin with visual tools such as *boxplots*, *histograms*, and *scatter plots*, which offer intuitive ways to spot anomalies. These methods are especially useful during the exploratory phase of analysis. More formal techniques—such as Z-scores and interquartile range (IQR) thresholds—provide quantitative criteria for identifying unusually high or low values.

In the next section, we apply visual tools to the *diamonds* dataset to see how outliers manifest in real data and how they might affect our understanding of key variables.

3.5 Spotting Outliers with Visual Tools

Visualization is often the most accessible and informative way to begin detecting outliers. It helps us identify values that stray far from the typical range—whether due to data entry mistakes, rare but valid events, or measurement anomalies. In this section, we apply visual methods to the *y* variable (diamond width) from the *diamonds* dataset, a feature known to contain extreme or implausible values that warrant closer inspection.

Boxplots: Visualizing and Flagging Outliers

Boxplots provide a compact visual summary of a variable’s distribution. They highlight key statistics—such as the median and interquartile range (IQR)—while flagging unusually high or low values as potential outliers. As shown in Figure 3.2, the “whiskers” extend to 1.5 times the IQR from the lower and upper quartiles, and data points beyond this range are considered outliers.

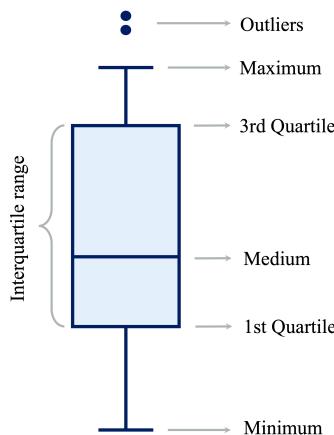
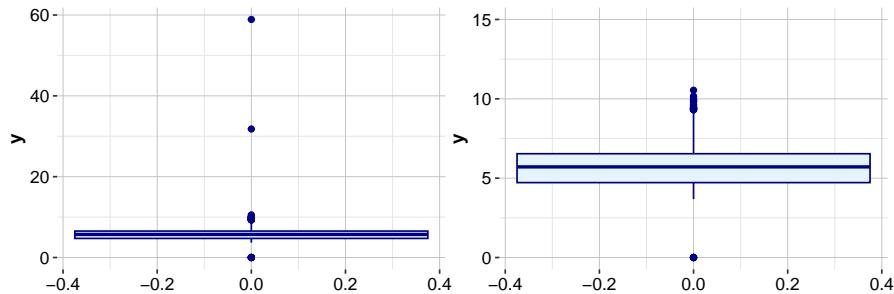


Figure 3.2: Boxplots summarize a variable’s distribution and flag extreme values. Outliers are identified as points beyond 1.5 times the interquartile range (IQR) from the quartiles.

To see this in action, let us apply a boxplot to the *y* variable (diamond width) in the *diamonds* dataset:

```
ggplot(data = diamonds) +
  geom_boxplot(mapping = aes(y = y), color = "darkblue", fill = "#e5f4fb")

ggplot(data = diamonds) +
  geom_boxplot(mapping = aes(y = y), color = "darkblue", fill = "#e5f4fb") +
  coord_cartesian(ylim = c(0, 15))
```



The left plot shows the full range of y values, where extreme observations above 15 mm compress the rest of the distribution and obscure the central pattern. The right plot zooms in on a more typical range (0–15 mm), revealing that most diamond widths fall between 2 and 6 mm, with a few clear outliers.

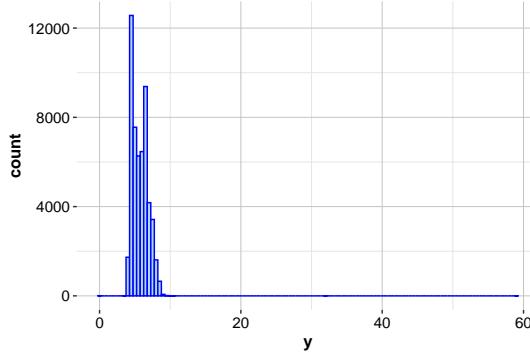
Boxplots like these allow us to quickly identify suspicious values—such as widths near 0 mm or above 15 mm—that are implausible for real diamonds. These anomalies may reflect data entry errors or rare but important cases that deserve closer inspection.

Histograms: Revealing Outlier Patterns

Histograms provide a complementary view to boxplots by showing how values are distributed across intervals. They reveal the overall shape of the distribution, highlight skewness, and help detect isolated spikes or rare values that may signal outliers.

The following histogram visualizes the distribution of the y variable (diamond width) using bins of width 0.5:

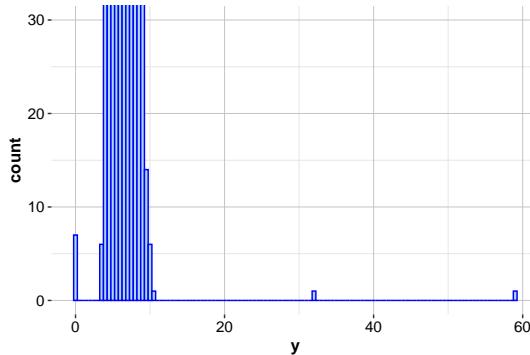
```
ggplot(data = diamonds) +
  geom_histogram(aes(x = y), binwidth = 0.5, color = 'blue', fill =
    "lightblue")
```



In this full-scale view, the concentration of values between 2 and 6 mm is clear, but less frequent values—especially those beyond 15 mm—are hard to see due to scale compression.

To focus on the rare and extreme cases, we zoom in on the lower portion of the frequency axis:

```
ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = y), binwidth = 0.5, color = 'blue', fill =
    "lightblue") +
  coord_cartesian(ylim = c(0, 30))
```



This refined view reinforces what we observed in the boxplot—namely, the presence of unusually small or large values in the `y` variable that fall outside the expected range. These may correspond to data entry errors or genuine anomalies, and they warrant closer inspection. Together with boxplots, histograms are powerful tools for identifying and interpreting potential outliers.

Additional Tools for Visual Outlier Detection

In addition to boxplots and histograms, several other visualization tools can help identify outliers in different contexts:

- *Violin plots* combine the summary statistics of a boxplot with a mirrored density curve, offering insight into both distribution shape and the presence of extreme values.
- *Density plots* provide a smoothed view of the data distribution, making it easier to detect long tails, multiple modes, or subtle irregularities.
- *Scatter plots* are particularly useful for examining relationships between two variables and identifying outliers that deviate from general trends—especially in bivariate or multivariate contexts.

These tools are valuable in the early stages of analysis, when scanning for irregular patterns or unusual cases. However, as the number of dimensions increases, visual methods alone may no longer suffice. In such cases, formal statistical techniques—covered later in the book—offer more rigorous and scalable solutions for outlier detection.

Now that we have visually identified potential outliers, the next step is to decide how to handle them: whether to remove, transform, or retain these points based on their context and potential impact on analysis and modeling.

3.6 How to Handle Outliers

Why would a diamond have a width of 0 mm—or a price twenty times higher than its peers? Outliers like these appear in nearly every real-world dataset, and deciding what to do with them is a recurring challenge in data science. When guiding students through messy datasets, I often hear the same question: “*Should I remove this outlier?*” My advice is to pause and reflect. Not all outliers are errors, and not all errors are obvious.

Once outliers have been identified—either visually or statistically—the next step is to decide how to respond. There is no universal rule. The best course of action depends on whether the outlier reflects a data entry mistake, a rare but valid observation, or a meaningful signal. Context is critical, and decisions should be shaped by the modeling goals and the domain in which the data was collected.

Below are several practical strategies for handling outliers, each with its own trade-offs:

- *Retain the outlier* if it is a valid observation that may contain valuable information. For instance, in fraud detection, statistical outliers often correspond to precisely the cases of interest. In the *adult* dataset, extreme values for `capital.gain` may reveal unique high-income individuals. Removing such observations could lead to a loss of predictive power or insight.
- *Replace the outlier with a missing value* when there is strong evidence that it is erroneous. For example, a negative carat weight or a repeated record likely stems from a data entry issue. Replacing such values with `NA` allows for flexible downstream handling, including imputation strategies introduced in the next section.
- *Flag and preserve the outlier* by creating an indicator variable (e.g., `is_outlier`) to retain the observation without distorting the analysis. This is especially useful when the outlier might be informative but requires separate treatment.
- *Apply data transformations*, such as logarithmic or square root scaling, to reduce the influence of extreme values while preserving relative patterns.
- *Apply winsorization*, which replaces extreme outlier values with less extreme percentile values (e.g., capping values at the 1st and 99th percentiles). This approach reduces the impact of outliers while retaining all observations, balancing robustness and data retention.
- *Use robust modeling techniques* that are less sensitive to outliers. Decision trees, random forests, and median-based estimators are designed to accommodate variability without being unduly influenced by extreme points.
- *Remove the outlier* only if the value is clearly invalid, cannot be reasonably corrected or imputed, and would otherwise compromise the integrity of the analysis. This should be considered a last resort, not the default action.

In general, a cautious and informed approach is best. Automatically removing all outliers is tempting, especially for beginners, but it risks discarding rare yet meaningful variation. A thoughtful strategy—grounded in domain knowledge and analytical objectives—ensures that data remains both clean and insightful.

3.7 Outlier Treatment in Action

Now that we have seen how to spot outliers, how should we handle them in practice? Let us walk through a hands-on example using the *diamonds* dataset. Take the `y` variable, which measures diamond width. As we saw earlier, some entries are clearly implausible—values equal to 0 or greater than

30 mm are unlikely for real diamonds and likely stem from data entry errors or measurement glitches.

To fix these values, we turn to the **dplyr** package, part of the **tidyverse**. It offers a powerful yet readable syntax for data manipulation. One of its core functions, `mutate()`, allows us to create or modify columns directly within a data frame. If **dplyr** is not yet installed, you can add it with `install.packages("dplyr")`. Then, load the package and apply the following transformation:

```
library(dplyr) # Load dplyr package  
diamonds_2 <- mutate(diamonds, y = ifelse(y == 0 | y > 30, NA, y))
```

This command creates a new dataset, `diamonds_2`, by replacing suspicious `y` values with `NA`. The logic is simple: if `y` is 0 or greater than 30, it is replaced; otherwise, it is left unchanged. This kind of targeted replacement keeps the rest of the data intact while removing values likely to distort analysis.

To verify the update, summarize the variable:

```
summary(diamonds_2$y)  
Min. 1st Qu. Median Mean 3rd Qu. Max. NA's  
3.680 4.720 5.710 5.734 6.540 10.540 9
```

This summary confirms how many values were flagged and how the range has shifted. The extreme outliers no longer dominate the distribution, and we can now proceed with a cleaner variable.

Try it yourself: Repeat this process for the `x` and `z` variables, which represent length and depth. Do they contain similar implausible values? Replace any unusual values with `NA`, and use `summary()` to assess the results. Practicing this on multiple variables helps reinforce the principle that outlier treatment is part of an ongoing diagnostic mindset—not just a one-time fix.

In the next section, we will move on to the next step in the data preparation pipeline: how to impute missing values in a statistically informed way.

3.8 Missing Values: What They Are and Why They Matter

Missing values are not just blank entries—they are clues to a larger story in your dataset. Like a puzzle with missing pieces, incomplete data can hide important patterns, distort results, and mislead your models. Before drawing conclusions or fitting algorithms, it is essential to detect and handle missing data with care.

In **R**, missing values usually appear as `NA`, but in real-world datasets, they often hide behind special codes like `-1`, `999`, or `99.9`. These placeholder values are easy to miss—and if ignored, can quietly undermine your analysis. For instance, in the `cereal` dataset from the `liver` package (explored in Section 13.4), the `calories` variable uses `-1` to indicate missing data. Similarly, in the `bank` marketing dataset (Section 12.6), the `pday` variable uses `-1` to signal that a client was not contacted. Recognizing and recoding these special cases is a crucial first step.

As highlighted in the story of Abraham Wald (Section 2.4), missing data is not always random. Wald’s insight came from what he *could not* see—damage on planes that never returned. In data science, the absence of information can be just as telling as its presence. Overlooking this subtlety can lead to flawed assumptions and inaccurate models.

Students often ask, “How should we deal with missing values?” My advice is to avoid deleting data unless there is no alternative. While dropping incomplete rows is fast, it is rarely the best choice. In fact, if just 5% of values are missing across 30 variables, removing all rows with missing entries could eliminate up to 80% of your dataset. That is a steep price to pay. A more thoughtful approach—such as imputing missing values—preserves valuable information while supporting the goals of your analysis.

Broadly, there are two strategies for handling missing data:

- *Imputation* involves estimating missing values using observed data. This method retains all records and helps maintain analytical completeness.
- *Removal* excludes rows or columns with missing entries. While sometimes necessary, this approach can significantly shrink your dataset and introduce bias—especially if the missingness is not random.

In the following sections, we will explore how to identify missing values, understand their impact, and apply appropriate imputation techniques to create a more complete and trustworthy dataset.

3.9 Imputation Techniques

Once missing values have been identified, the next step is to choose an appropriate strategy for estimating them. The best method depends on the structure of the data, the analysis objectives, and how much complexity is acceptable. Below are several commonly used approaches:

- *Mean, median, or mode imputation* replaces missing entries with the average (for numerical variables), median (for skewed distributions), or the most

frequent category (for categorical variables). These simple methods are best suited for low levels of missingness and relatively uniform data.

- *Random sampling* fills in missing values by drawing randomly from the observed values of the same variable. This technique preserves the variable's distribution better than mean imputation but introduces randomness into the analysis.
- *Predictive imputation* uses other variables in the dataset to estimate missing values through models such as linear regression, decision trees, or k -nearest neighbors. This approach is more accurate when strong relationships exist between variables.
- *Multiple imputation* generates several completed datasets by repeating the imputation process multiple times and then combines results across them. This method accounts for uncertainty in the missing values and is especially useful for inference or reporting confidence intervals.

Selecting the right imputation method involves balancing simplicity and precision. For numerical variables with few missing entries, mean or median imputation often suffices. For categorical variables, mode imputation provides a straightforward alternative. In cases with substantial missingness or complex dependencies, predictive methods—such as regression or k -nearest neighbors—offer more reliable results. If a variable is missing too frequently, it may be better to exclude it or reconsider its role in the analysis.

In the coming example, we demonstrate the random sampling approach for its simplicity and illustrative value. However, in real-world applications, more advanced techniques are often preferred. Later in this chapter and in Chapter 13 (Section 13.4), we revisit this topic using Random Forest algorithm for imputation. This transition from basic to sophisticated methods reflects the increasing demands of applied data science.

Example: Random Sampling Imputation in R

Let us now put imputation into practice using the `y` variable (diamond width) in the `diamonds` dataset. Earlier, we flagged implausible values—such as 0 or values above 30 mm—as missing (`NA`). Our goal here is to replace those missing values using *random sampling*, a simple technique that draws replacements from the observed, non-missing values of the same variable.

We use the `impute()` function from the **Hmisc** package, which supports several basic imputation strategies. If the package is not installed on your system, you can add it using `install.packages("Hmisc")`. Then, load the package and apply the imputation:

```
library(Hmisc) # Load Hmisc package

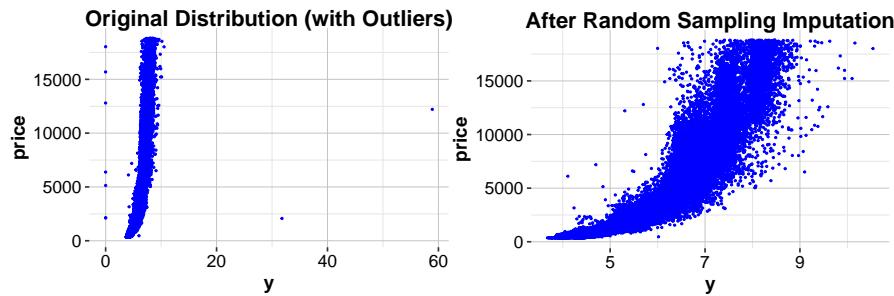
diamonds_2$y <- impute(diamonds_2$y, "random") # Randomly fill in missing
# values in y
```

This command replaces each NA in the y column with a randomly selected value from the observed values, preserving the variable's distribution. Because this technique introduces no systematic bias, it works well in exploratory settings where preserving the shape of the data is more important than prediction accuracy.

To assess the effect of imputation visually, the scatter plots below show the relationship between diamond width (y) and price before and after imputation:

```
# Before imputation (with NAs)
ggplot(data = diamonds, aes(x = y, y = price)) +
  geom_point(color = "blue", size = 0.3) +
  ggtitle("Original Distribution (with Outliers)")

# After random sampling imputation
ggplot(data = diamonds_2, aes(x = y, y = price)) +
  geom_point(color = "blue", size = 0.3) +
  ggtitle("After Random Sampling Imputation")
```



These plots illustrate that after removing implausible values and imputing missing entries, the data retains its overall structure while discarding extremes that could distort model training. Since price is our eventual target variable (see Chapter 10), this visualization also helps assess whether the cleaned y values still support meaningful patterns.

Try it yourself: Apply the same technique to the x and z variables, which represent diamond length and depth. First identify implausible values, recode them as NA, and then impute them using random sampling. Reflect on how these changes affect the variables' relationships with price.

Alternative Approaches

The `impute()` function also supports statistical imputation methods such as mean, median, and mode. By default, it performs *median imputation*. For more advanced tasks, `aregImpute()` (also from **Hmisc**) enables predictive imputation using techniques like additive regression and bootstrapping.

While removing missing records via `na.omit()` is a simple option, it is usually discouraged unless the missingness is widespread or systematically biased. Properly addressing missing data helps maintain data integrity and prepares the dataset for subsequent steps—such as encoding categorical variables and aggregating country-level categories.

In later sections—including the case study in Chapter 13.4—we will revisit this topic using more advanced methods such as *k*-nearest neighbors imputation, which estimates missing values based on the similarity across multiple variables.

Now that missing values have been identified and imputed, the dataset is more complete and ready for further preparation. The next key step is feature scaling, which ensures that numerical variables are on comparable scales.

3.10 Feature Scaling

What happens when one variable—such as price in dollars—spans tens of thousands, while another, like carat weight, ranges only from 0 to 5? Without scaling, machine learning models that rely on distances or gradients may give disproportionate weight to features with larger numerical ranges, regardless of their actual importance.

Feature scaling addresses this imbalance by adjusting the range or distribution of numerical features to make them comparable. It is particularly important for algorithms such as *k*-nearest neighbors (Chapter 7), support vector machines, and neural networks. Scaling can also improve optimization stability in models like logistic regression and enhance the interpretability of coefficients.

In the `diamonds` dataset, for example, carat ranges from 0.2 to 5, while price spans from 326 to 18823. Without scaling, features like price may dominate the learning process—not because they are more predictive, but simply because of their magnitude.

This section introduces two widely used scaling techniques:

- *Min-Max Scaling* rescales values to a fixed range, typically [0, 1].

- *Z-score Scaling* centers values at zero with a standard deviation of one.

Choosing between these methods depends on the model and the structure of the data. Min-Max Scaling is preferred when a fixed input range is required (as in neural networks), whereas Z-score Scaling is better suited for algorithms that assume standardized input distributions or rely on variance-sensitive optimization.

Note that scaling is not always necessary. Tree-based models, such as decision trees and random forests, are scale-invariant and do not require rescaled inputs. But for many other algorithms, scaling improves model performance, convergence speed, and fairness across features.

One caution: scaling can obscure real-world units or exaggerate the effect of outliers—especially when using Min-Max Scaling. As always, the choice of technique should reflect your modeling objectives and the nature of your dataset. In the following sections, we demonstrate how to apply each technique using the *diamonds* dataset.

3.11 Min-Max Scaling

When one feature ranges from 0 to 1 and another spans thousands, models that rely on distances—like *k*-nearest neighbors—may be skewed toward features with larger scales. *Min-Max Scaling* addresses this by rescaling each feature to a common range, typically [0, 1], so that no single variable dominates due to its units.

The transformation is defined by the formula:

$$x_{\text{scaled}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}},$$

where x is the original value, and x_{\min} and x_{\max} are the minimum and maximum values of the feature. This method ensures that the minimum value becomes 0 and the maximum becomes 1.

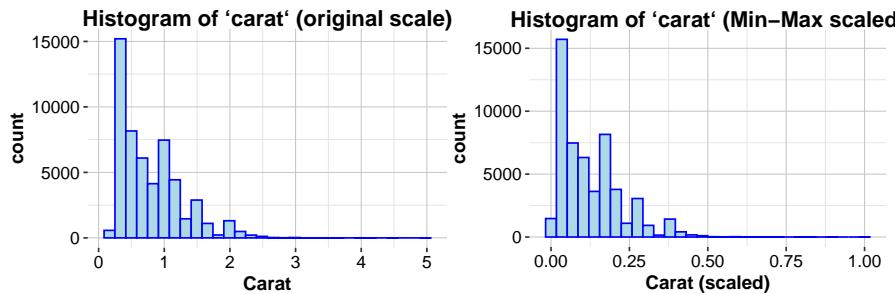
Min-Max Scaling is especially useful for algorithms that use distance or gradient information, including neural networks and support vector machines. However, this technique is sensitive to outliers: extreme values can stretch the scale, compressing most of the data into a narrow band and reducing resolution for typical values.

To see Min-Max Scaling in action, we apply it to the `carat` variable in the *diamonds* dataset. This variable ranges from approximately 0.2 to 5. We use

the `minmax()` function from the `liver` package to rescale the values to the $[0, 1]$ interval:

```
ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = carat), bins = 30,
                 color = 'blue', fill = "lightblue") +
  ggtitle("Histogram of 'carat' (original scale)") +
  xlab("Carat")

ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = minmax(carat)), bins = 30,
                 color = 'blue', fill = "lightblue") +
  ggtitle("Histogram of 'carat' (Min-Max scaled)") +
  xlab("Carat (scaled)")
```



The left panel shows the raw distribution of carat values, while the right panel displays the scaled version. After transformation, all values fall within the $[0, 1]$ range, making this feature numerically comparable to others. This is particularly important when modeling techniques depend on distance or gradient magnitude.

3.12 Z-score Scaling

While Min-Max Scaling compresses values into a fixed range, *Z-score Scaling*—also known as standardization—centers each numerical feature around zero and scales it to have unit variance. This technique is especially useful for algorithms that assume normally distributed input or rely on gradient-based optimization, such as linear regression, logistic regression, and support vector machines.

The formula for Z-score Scaling is:

$$x_{\text{scaled}} = \frac{x - \text{mean}(x)}{\text{sd}(x)},$$

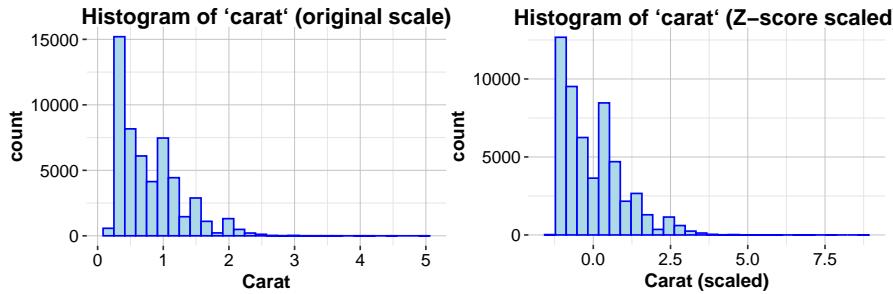
where x is the original feature value, $\text{mean}(x)$ is the mean of the feature, and $\text{sd}(x)$ is its standard deviation. The result, x_{scaled} , tells us how many standard deviations the value is from the mean.

Z-score Scaling helps place features with different units or magnitudes on a common footing. However, it is still sensitive to outliers, since both the mean and standard deviation can be influenced by extreme values.

To illustrate, let us apply Z-score Scaling to the `carat` variable in the *diamonds* dataset. The mean and standard deviation of `carat` are approximately 0.8 and 0.47, respectively. We use the `zscore()` function from the `liver` package:

```
ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = carat), bins = 30,
                 color = 'blue', fill = "lightblue") +
  ggtitle("Histogram of 'carat' (original scale)") +
  xlab("Carat")

ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = zscore(carat)), bins = 30,
                 color = 'blue', fill = "lightblue") +
  ggtitle("Histogram of 'carat' (Z-score scaled)") +
  xlab("Carat (scaled)")
```



The left panel shows the original distribution of `carat` values. The right panel displays the Z-score scaled version, where the values are centered around 0 and spread in units of standard deviation. While the location and scale are adjusted, the shape of the distribution—including any skewness—remains unchanged.

Note that Z-score Scaling does not make a variable normally distributed. It standardizes the location and spread but preserves the shape. If the original variable is skewed, it will remain skewed after transformation.

With numerical features now on a comparable scale, the next step is to turn our attention to *categorical variables*. Unlike numerical features, these cannot be used directly in most models—they must be encoded numerically. In the next section, we explore encoding strategies for different types of categorical data, including ordinal and nominal variables.

3.13 Encoding Categorical Features for Modeling

Categorical features often need to be transformed into numeric format before they can be used in machine learning models. Algorithms such as k -Nearest Neighbors and neural networks require numerical inputs, and failure to encode categorical data properly can lead to misleading results or even errors during model training.

Encoding categorical variables is a critical step in data preparation. It allows us to incorporate qualitative information—such as quality ratings, group memberships, or types—into models that operate on numerical representations. In this section, we explore several encoding techniques and illustrate their use with examples from the *diamonds* dataset, which includes the categorical variables `cut`, `color`, and `clarity`.

The appropriate encoding strategy depends on the nature of the categorical variable. For ordinal variables—those with an inherent ranking—ordinal encoding preserves the order of categories using numerical values. For example, the `cut` variable in the *diamonds* dataset ranges from “Fair” to “Ideal” and benefits from ordinal encoding.

In contrast, nominal variables—categories without intrinsic order—are better served by one-hot encoding. This approach creates binary indicators for each category and is particularly effective for features such as marital status in the *bank* dataset introduced in Chapter 12.6.

In the sections that follow, we demonstrate these encoding techniques in practice—starting with ordinal encoding and one-hot—using the *diamonds* and *bank* datasets as guiding examples.

3.14 Ordinal Encoding

When a categorical variable has a meaningful ranking, *ordinal encoding* provides a simple and effective transformation. Instead of treating each category as unrelated, we assign numerical values that reflect the natural order—allowing models to capture rank-based patterns.

Take the cut variable in the *diamonds* dataset, which reflects increasing quality:

- “Fair” \mapsto 1;
- “Good” \mapsto 2;
- “Very Good” \mapsto 3;
- “Premium” \mapsto 4;
- “Ideal” \mapsto 5.

This numeric representation preserves the ordinal structure and can enhance the performance of models that recognize order, such as linear regression or decision trees.

However, ordinal encoding should only be used when the order of categories is *truly meaningful*. Applying this method to *nominal* variables—like “red”, “green”, and “blue”—can be misleading. Encoding those as 1, 2, and 3 suggests a false hierarchy and may bias the model.

Always match the encoding method to the nature of the variable. For unordered categories, consider one-hot or frequency encoding instead. In the next section, we explore one-hot encoding, which treats each category as distinct—ideal for variables without any natural ordering.

3.15 One-Hot Encoding

How can we represent unordered categories—like marital status or race—so that machine learning algorithms can work with them? *One-hot encoding* is a widely used solution. It transforms each unique category into a separate binary column, allowing algorithms to process categorical data without introducing artificial order.

This method is especially useful for *nominal variables*—categorical features with no inherent ranking. For example, a variable like `marital.status` in the *adult* dataset includes categories such as Divorced, Married, and Never-married. One-hot encoding creates binary variables like:

- `marital.status_Divorced`;
- `marital.status_Married`;
- `marital.status_Never-married`;
- `marital.status_Separated`.

Each column indicates the presence (1) or absence (0) of a specific category. If there are k levels, only $k - 1$ binary columns are needed to avoid

multicollinearity—the omitted category is implicitly captured when all others are zero.

Let us take a quick look at the `marital.status` variable in the *adult* dataset:

```
data(adult)

table(adult$marital.status)

  Divorced      Married Never-married     Separated      Widowed
       6613        22847         16096        1526        1516
```

The output shows the distribution of values across the five categories. We will use one-hot encoding to transform these into model-ready binary features. This approach ensures that all categories are represented without assuming any order or relationship among them.

Note that one-hot encoding is essential for models that rely on distance metrics (e.g., k-nearest neighbors, neural networks) or linear models where numeric input is required.

3.15.1 One-Hot Encoding in R

To apply one-hot encoding in practice, we use the `one.hot()` function from the `liver` package. This function automatically detects categorical variables and generates a new column for each unique level, converting them into binary indicators.

Let us apply it to the `marital.status` variable in the *adult* dataset:

```
# One-hot encode the 'marital.status' variable
adult_encoded <- one.hot(adult, cols = c("marital.status"), dropCols = FALSE)

# Examine the structure of the resulting dataset
str(adult_encoded)
'data.frame': 48598 obs. of 20 variables:
 $ age                  : int  25 38 28 44 18 34 29 63 24 55 ...
 $ workclass             : Factor w/ 6 levels
   ↘ "?", "Gov", "Never-worked",...: 4 4 2 4 1 4 1 5 4 4 ...
 $ demogweight           : int  226802 89814 336951 160323 103497
   ↘ 198693 227026 104626 369667 104996 ...
 $ education              : Factor w/ 16 levels "10th", "11th", ...: 2 12
   ↘ 8 16 16 1 12 15 16 6 ...
 $ education.num          : int  7 9 12 10 10 6 9 15 10 4 ...
 $ marital.status          : Factor w/ 5 levels
   ↘ "Divorced", "Married", ...: 3 2 2 2 3 3 3 2 3 2 ...
 $ marital.status_Divorced : int  0 0 0 0 0 0 0 0 0 0 ...
```

```
$ marital.status_Married      : int  0 1 1 1 0 0 0 1 0 1 ...
$ marital.status_Never-married: int  1 0 0 0 1 1 1 0 1 0 ...
$ marital.status_Separated    : int  0 0 0 0 0 0 0 0 0 0 ...
$ marital.status_Widowed     : int  0 0 0 0 0 0 0 0 0 0 ...
$ occupation                  : Factor w/ 15 levels
  ↵ "?","Adm-clerical",...: 8 6 12 8 1 9 1 11 9 4 ...
$ relationship                 : Factor w/ 6 levels
  ↵ "Husband","Not-in-family",...: 4 1 1 1 4 2 5 1 5 1 ...
$ race                         : Factor w/ 5 levels
  ↵ "Amer-Indian-Eskimo",...: 3 5 5 3 5 5 3 5 5 5 ...
$ gender                        : Factor w/ 2 levels "Female","Male": 2 2 2
  ↵ 2 1 2 2 2 1 2 ...
$ capital.gain                 : int  0 0 0 7688 0 0 0 3103 0 0 ...
$ capital.loss                  : int  0 0 0 0 0 0 0 0 0 0 ...
$ hours.per.week                : int  40 50 40 40 30 30 40 32 40 10 ...
$ native.country                 : Factor w/ 42 levels "?","Cambodia",...: 40
  ↵ 40 40 40 40 40 40 40 40 40 ...
$ income                         : Factor w/ 2 levels "<=50K",">50K": 1 1 2 2
  ↵ 1 1 1 2 1 1 ...
```

The `cols` argument specifies which variable(s) to encode. Setting `dropCols = FALSE` keeps the original variable alongside the new binary columns; use `TRUE` if you prefer to remove it after encoding.

This transformation adds new columns such as `marital.status_Divorced`, `marital.status_Married`, and so on—each indicating whether a given observation belongs to that category.

Try it yourself: What happens if you encode multiple variables at once? Try applying `one_hot()` to both `marital.status` and `race` and inspect the output.

While one-hot encoding is simple and effective, it can quickly increase the number of features, especially when applied to high-cardinality variables (e.g., zip codes or product names). Before encoding, consider whether the added dimensionality is manageable for your model and whether all categories are meaningful for the analysis.

3.16 Case Study: Preparing Data to Predict High Earners

How can we predict whether a person earns more than \$50,000 per year based on their background? This question arises in many domains—from economic research to policy evaluation and algorithmic hiring tools. In this case study, we use the `adult` dataset, originally compiled from the [US Census Bureau](#), to explore how structured demographic data can inform such predictions.

The *adult* dataset includes information on age, education, marital status, occupation, and income, and is available via the `liver` package. For documentation, see [this link](#).

Our objective is to predict whether an individual earns more than \$50,000 annually. In Chapter 11, we will return to this dataset to build predictive models using decision trees and random forests (see Section 11.5). Here, our focus is on preparing the data for modeling: identifying and addressing missing values, encoding categorical variables, detecting outliers, and scaling numerical features.

3.16.1 Overview of the Dataset

The *adult* dataset is a classic benchmark in machine learning, widely used for exploring income prediction based on demographic features. It reflects many of the data preparation challenges that analysts encounter in real-world applications.

To begin, let us load the *adult* dataset from the `liver` package. If you do not have the package installed, use the following command: `install.packages("liver")`. Then load the package and dataset:

```
library(liver) # Load the liver package
data(adult)     # Load the adult dataset
```

To explore the dataset structure and data types, use the `str()` function:

```
str(adult)
'data.frame': 48598 obs. of 15 variables:
 $ age          : int  25 38 28 44 18 34 29 63 24 55 ...
 $ workclass    : Factor w/ 6 levels "?","Gov","Never-worked",...: 4 4 2 4
   ↪ 1 4 1 5 4 4 ...
 $ demogweight  : int 226802 89814 336951 160323 103497 198693 227026
   ↪ 104626 369667 104996 ...
 $ education    : Factor w/ 16 levels "10th","11th",...: 2 12 8 16 16 1 12
   ↪ 15 16 6 ...
 $ education.num: int 7 9 12 10 10 6 9 15 10 4 ...
 $ marital.status: Factor w/ 5 levels "Divorced","Married",...: 3 2 2 2 3 3
   ↪ 3 2 3 2 ...
 $ occupation   : Factor w/ 15 levels "?","Adm-clerical",...: 8 6 12 8 1 9
   ↪ 1 11 9 4 ...
 $ relationship  : Factor w/ 6 levels "Husband","Not-in-family",...: 4 1 1
   ↪ 1 4 2 5 1 5 1 ...
 $ race         : Factor w/ 5 levels "Amer-Indian-Eskimo",...: 3 5 5 3 5 5
   ↪ 3 5 5 5 ...
```

```
$ gender      : Factor w/ 2 levels "Female","Male": 2 2 2 2 1 2 2 2 1 2
  ↪ ...
$ capital.gain : int 0 0 0 7688 0 0 0 3103 0 0 ...
$ capital.loss : int 0 0 0 0 0 0 0 0 0 0 ...
$ hours.per.week: int 40 50 40 40 30 30 40 32 40 10 ...
$ native.country: Factor w/ 42 levels "?","Cambodia",...: 40 40 40 40 40
  ↪ 40 40 40 40 40 ...
$ income      : Factor w/ 2 levels "<=50K",">50K": 1 1 2 2 1 1 1 2 1 1
  ↪ ...
```

The dataset contains 48598 observations and 15 variables. Most are predictors, while the target variable, `income`, indicates whether an individual earns more than \$50,000 per year (`>50K`) or not (`<=50K`). The features include a mix of numerical and categorical variables, reflecting a range of demographic and economic attributes.

Below is a summary of the main variables:

- `age`: Age in years (numerical);
- `workclass`: Employment type (categorical; 6 levels);
- `demogweight`: Census weighting factor (numerical);
- `education`: Highest educational attainment (categorical; 16 levels);
- `education.num`: Years of education (numerical);
- `marital.status`: Marital status (categorical; 5 levels);
- `occupation`: Job type (categorical; 15 levels);
- `relationship`: Household role (categorical; 6 levels);
- `race`: Racial background (categorical; 5 levels);
- `gender`: Gender identity (categorical; 2 levels);
- `capital.gain`: Annual capital gains (numerical);
- `capital.loss`: Annual capital losses (numerical);
- `hours.per.week`: Weekly working hours (numerical);
- `native.country`: Country of origin (categorical; 42 levels);
- `income`: Income bracket (`<=50K` or `>50K`).

The dataset includes both numeric and categorical features, which we group as follows:

- Numerical variables: `age`, `demogweight`, `education.num`, `capital.gain`, `capital.loss`, `hours.per.week`.
- Binary variables: `gender`, `income`.
- Ordinal variable: `education` (ordered from “Preschool” to “Doctorate”).
- Nominal variables: `workclass`, `marital.status`, `occupation`, `relationship`, `race`, `native.country`.

To better understand the dataset, use `summary()` to inspect distributions, detect possible anomalies, and identify missing values:

```
summary(adult)
      age          workclass      demogweight      education
Min. :17.0    ?       : 2794  Min.   : 12285  HS-grad   :15750
1st Qu.:28.0   Gov     : 6536  1st Qu.: 117550 Some-college:10860
Median :37.0   Never-worked:  10   Median : 178215 Bachelors  : 7962
Mean   :38.6   Private   :33780  Mean   : 189685 Masters   : 2627
3rd Qu.:48.0   Self-emp  : 5457  3rd Qu.: 237713 Assoc-voc  : 2058
Max.   :90.0   Without-pay:  21   Max.   :1490400 11th     : 1812
                                         (Other)   : 7529
      education.num      marital.status      occupation
Min.   : 1.00  Divorced       : 6613  Craft-repair  : 6096
1st Qu.: 9.00  Married       :22847  Prof-specialty: 6071
Median :10.00  Never-married:16096  Exec-managerial: 6019
Mean   :10.06  Separated     : 1526  Adm-clerical  : 5603
3rd Qu.:12.00  Widowed      : 1516  Sales        : 5470
Max.   :16.00                                         Other-service : 4920
                                         (Other)       :14419
      relationship      race         gender
Husband       :19537  Amer-Indian-Eskimo: 470  Female:16156
Not-in-family :12546  Asian-Pac-Islander: 1504 Male  :32442
Other-relative: 1506  Black          : 4675
Own-child     : 7577  Other          : 403
Unmarried     : 5118  White          :41546
Wife          : 2314
      capital.gain      capital.loss      hours.per.week      native.country
Min.   : 0.0    Min.   : 0.00  Min.   : 1.00  United-States:43613
1st Qu.: 0.0    1st Qu.: 0.00  1st Qu.:40.00  Mexico       : 949
Median : 0.0    Median : 0.00  Median :40.00   ?           : 847
Mean   : 582.4   Mean  : 87.94  Mean   :40.37  Philippines  : 292
3rd Qu.: 0.0    3rd Qu.: 0.00  3rd Qu.:45.00  Germany     : 206
Max.   :41310.0   Max.  :4356.00  Max.   :99.00  Puerto-Rico : 184
                                         (Other)       : 2507
      income
<=50K:37155
>50K :11443
```

This summary provides a foundation for the next stages of data preparation: identifying missing values, detecting outliers, and encoding categorical features for modeling.

With a clear understanding of the dataset's structure and variable types, we now begin the data preparation process. The first step is to detect and handle missing values—an essential task for ensuring the completeness and reliability of our analysis.

3.16.2 Handling Missing Values

A crucial early step in data preparation is identifying and resolving missing values. If left untreated, missing data can distort statistical summaries, reduce model accuracy, and bias conclusions.

The `summary()` function shows that three variables—`workclass`, `occupation`, and `native.country`—contain missing entries. In this dataset, however, missing values are not coded as `NA`, but as the string `"?"`, a placeholder commonly used in public datasets such as those from the UCI Machine Learning Repository. Because R does not automatically recognize `"?"` as missing, we must recode it manually:

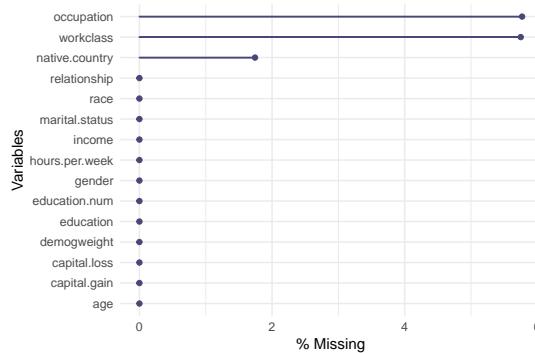
```
adult[adult == "?"] <- NA
```

We also apply `droplevels()` to remove unused categories from factor variables. This step helps prevent issues in later steps such as encoding:

```
adult <- droplevels(adult)
```

To visualize the extent of missingness, we use the `gg_miss_var()` function from the `naniar` package. This function generates a bar chart showing the number and percentage of missing values per variable:

```
library(naniar)
gg_miss_var(adult, show_pct = TRUE)
```



The resulting plot confirms that only three variables contain missing values: `workclass` with 2794 entries, `occupation` with 2804 entries, and `native.country` with 847 entries.

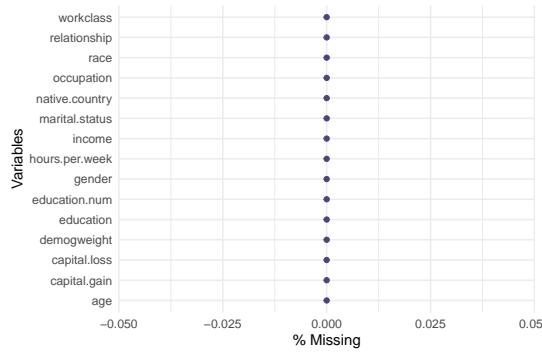
Since the proportion of missing values is small—less than 0.06% in each case—we choose to impute the missing values rather than remove rows, which could lead to information loss. To preserve each variable’s distribution, we use *random imputation*, which replaces missing entries by randomly sampling from observed (non-missing) values:

```
library(Hmisc)

adult$workclass      <- impute(adult$workclass,      "random")
adult$native.country <- impute(adult$native.country, "random")
adult$occupation     <- impute(adult$occupation,    "random")
```

Finally, we recheck for missingness to confirm that the imputation succeeded:

```
gg_miss_var(adult, show_pct = TRUE)
```



The updated plot should confirm that all missing values have been addressed. With this step complete, we are ready to encode categorical variables and scale numerical features—two more critical tasks in preparing the dataset for modeling.

3.16.3 Encoding Categorical Variables

Categorical variables often contain many unique values, which can complicate modeling and lead to overfitting if not handled carefully. In the *adult* dataset, two features—*native.country* and *workclass*—have a relatively high number of categories. To simplify modeling while preserving interpretability, we group related categories into broader, more informative classes.

Grouping native.country by Region

The `native.country` variable includes 41 distinct countries. Modeling each one as a separate category could dilute predictive power and inflate the feature space. A more structured approach is to group countries by geographic region:

- Europe: England, France, Germany, Greece, Netherlands, Hungary, Ireland, Italy, Poland, Portugal, Scotland, Yugoslavia.
- Asia: China, Hong Kong, India, Iran, Cambodia, Japan, Laos, Philippines, Vietnam, Taiwan, Thailand.
- North America: Canada, United States, Puerto Rico.
- South America: Colombia, Cuba, Dominican Republic, Ecuador, El Salvador, Guatemala, Haiti, Honduras, Mexico, Nicaragua, Outlying US territories, Peru, Jamaica, Trinidad and Tobago.
- Other: Includes the ambiguous category "South", which lacks clear documentation in the dataset. We treat it as a separate group.

We use the `fct_collapse()` function from the **forcats** package to perform the reclassification:

```
library(forcats)

Europe <- c("England", "France", "Germany", "Greece", "Holand-Netherlands",
           ↪ "Hungary", "Ireland", "Italy", "Poland", "Portugal", "Scotland",
           ↪ "Yugoslavia")
Asia <- c("China", "Hong", "India", "Iran", "Cambodia", "Japan", "Laos",
           ↪ "Philippines", "Vietnam", "Taiwan", "Thailand")
N.America <- c("Canada", "United-States", "Puerto-Rico")
S.America <- c("Columbia", "Cuba", "Dominican-Republic", "Ecuador",
           ↪ "El-Salvador", "Guatemala", "Haiti", "Honduras", "Mexico", "Nicaragua",
           ↪ "Outlying-US(Guam-USVI-etc)", "Peru", "Jamaica", "Trinidad&Tobago")

adult$native.country <- fct_collapse(adult$native.country,
                                       "Europe"      = Europe,
                                       "Asia"        = Asia,
                                       "N.America"   = N.America,
                                       "S.America"   = S.America,
                                       "Other"       = c("South"))
)
```

To verify the transformation:

| | Asia | N.America | S.America | Europe | Other |
|--|------|-----------|-----------|--------|-------|
| | 993 | 44747 | 1946 | 797 | 115 |

This grouping reduces sparsity, enhances interpretability, and avoids overfitting caused by excessive fragmentation of categories.

Simplifying workclass

The `workclass` variable categorizes employment types. Two of its levels—“Never-worked” and “Without-pay”—are rare and similar in nature, representing individuals outside formal employment. We consolidate these under a single label: Unemployed.

```
adult$workclass <- fct_collapse(adult$workclass, "Unemployed" =
  c("Never-worked", "Without-pay"))
```

To confirm the update:

| | Gov | Unemployed | Private | Self-emp |
|------|-----|------------|---------|----------|
| 6919 | 32 | 35851 | 5796 | |

By reducing the number of categories in `workclass` and `native.country`, we streamline the dataset while retaining interpretability. This prepares the data for modeling algorithms that are sensitive to high-cardinality categorical inputs.

3.16.4 Handling Outliers

Identifying and addressing outliers is an important part of data preparation. Extreme values can distort summary statistics and influence the performance of models—particularly those sensitive to the range of numeric features. In this section, we focus on the `capital.loss` variable from the `adult` dataset to examine whether outliers are present and how they should be handled.

We begin with a basic summary:

```
summary(adult$capital.loss)
  Min. 1st Qu. Median Mean 3rd Qu. Max.
  0.00  0.00  0.00  87.94  0.00 4356.00
```

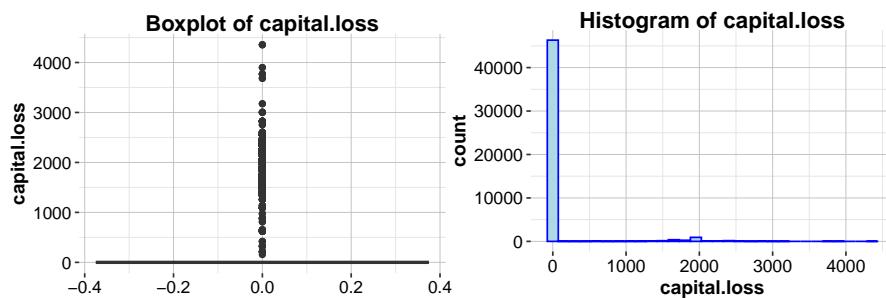
From this output, we observe that the minimum value is 0, and the maximum is 4356. A majority of observations—over 75%—have a value of 0. The me-

dian, 0, is substantially lower than the mean, 87.94, indicating a right-skewed distribution influenced by high values.

To explore this further, we visualize the distribution using both a boxplot and a histogram:

```
ggplot(data = adult, aes(y = capital.loss)) +
  geom_boxplot() +
  ggtitle("Boxplot of capital.loss")

ggplot(data = adult, aes(x = capital.loss)) +
  geom_histogram(bins = 30, color = "blue", fill = "lightblue") +
  ggtitle("Histogram of capital.loss")
```



The boxplot shows a strong positive skew, with several high values extending beyond the upper whisker. The histogram confirms that most individuals report zero capital loss, with a few concentrated peaks near 2,000 and 4,000.

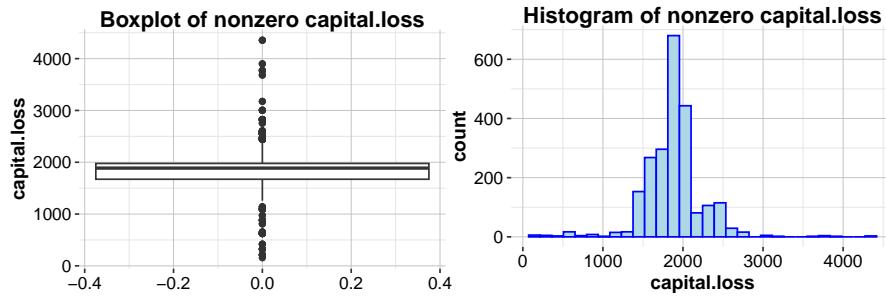
To focus more closely on the nonzero values, we restrict our analysis to rows where `capital.loss > 0`:

```
subset_adult <- subset(adult, capital.loss > 0)

ggplot(data = subset_adult, aes(y = capital.loss)) +
  geom_boxplot() +
  ggtitle("Boxplot of nonzero capital.loss")

ggplot(data = subset_adult, aes(x = capital.loss)) +
  geom_histogram(bins = 30, color = "blue", fill = "lightblue") +
  ggtitle("Histogram of nonzero capital.loss")
```

Among those with a nonzero capital loss, the majority of values are concentrated below 500. However, a small number of observations exceed 4,000. Despite their rarity, these high values appear to follow a relatively smooth and symmetric distribution, suggesting they reflect genuine variation in the data rather than data entry errors.



Given this context, we choose to retain the extreme values in `capital.loss`. They are likely to reflect meaningful differences across individuals and should not be removed without strong justification. If we later find that these values introduce problems for modeling, we may consider one of several strategies: applying a log or square-root transformation to reduce skewness, creating a binary variable to indicate whether a capital loss occurred, or using winsorization to cap extreme values at a defined threshold.

Understanding the structure of `capital.loss` also prepares us for handling the related variable `capital.gain`, which we explore next. For a hands-on example, see the guided exercise at the end of this chapter.

3.17 Chapter Summary and Takeaways

This chapter covered the essential steps of data preparation, showing how to transform raw data into a format suitable for analysis. Using the *diamonds* and *adult* datasets, we practiced identifying outliers, handling missing values, scaling numerical features, and encoding categorical variables.

We emphasized that data preparation is not just technical—it is driven by the structure and meaning of each feature. Outliers can distort results, missing values must be addressed carefully, and categorical variables require appropriate encoding. Scaling ensures fairness across features for many algorithms.

These techniques, though often overlooked, are crucial for building reliable and interpretable models. Clean, well-structured data forms the foundation of every successful data science project.

Next chapter, we move to exploratory data analysis, where we begin uncovering patterns and relationships that inform our modeling choices.

3.18 Exercises

This section includes exercises designed to deepen your understanding of data preparation. They cover conceptual questions, hands-on practice with the *diamonds* and *adult* datasets, and reflection on ethical and real-world issues. These exercises can be completed in R and build on the examples discussed in this chapter.

Understanding Data Types

1. Explain the difference between continuous and discrete numerical variables, and give a real-world example of each.
2. How do ordinal and nominal categorical variables differ? Provide one example for each type.
3. Use the `typeof()` and `class()` functions in R to investigate how numerical and categorical variables are represented internally.
4. Why is it important to identify the correct data types before modeling?

Working with the Diamonds Dataset

5. Use `summary()` to inspect the *diamonds* dataset. What patterns or irregularities do you notice?
6. Classify the variables in the *diamonds* dataset as numerical, ordinal, or nominal.
7. Create histograms of `carat` and `price`. Describe their distributions and note any skewness or gaps.

Detecting and Treating Outliers

8. Identify outliers in the `x` variable using boxplots and histograms. If outliers are found, apply a method to handle them as done for `y` in Section 3.4.
9. Repeat the process for the `z` variable and comment on the results.

10. Examine the depth variable. What method would you use to detect and treat outliers in this case?
11. Discuss the pros and cons of removing outliers versus applying a transformation. Use examples from this chapter to support your answer.

Encoding Categorical Variables

12. Show how to apply ordinal encoding to the cut variable in the *diamonds* dataset.
13. Demonstrate one-hot encoding for the color variable using appropriate R functions.
14. What problems can arise when applying ordinal encoding to a nominal variable?
15. Under what conditions is frequency encoding more appropriate than one-hot encoding?

Exploring the Adult Dataset

16. Load the *adult* dataset from the **liver** package and classify its categorical variables as nominal or ordinal.
17. Compute the proportion of individuals earning more than \$50K. What does this tell you about income distribution?
18. Generate a boxplot and histogram of *capital.gain*. What insights or anomalies can you identify?
19. Are there any outliers in *capital.gain*? If so, suggest how you would handle them.
20. Create a correlation matrix for the numerical variables. What do the correlations reveal about the relationships between features?

Feature Engineering and Scaling

21. Use the `cut()` function to group age into three categories: Young (≤ 30), Middle-aged (31–50), and Senior (> 50). Name the new variable `Age_Group`.

22. Calculate the mean `capital.gain` for each `Age_Group`. What does this reveal?
23. Create a binary variable indicating whether an individual has nonzero `capital.gain`, and use it to create an exploratory plot.
24. Group the 16 education levels into broader categories using `fct_collapse()`. Propose at least three meaningful groupings.
25. Define a new variable `net.capital` as the difference between `capital.gain` and `capital.loss`. Analyze its distribution using plots.
26. Apply Min-Max scaling to `age`, `capital.gain`, `capital.loss`, and `hours.per.week` using `mutate()` and summarize the results.
27. Apply Z-score normalization to the same variables. How do the results compare to Min-Max scaling?

Modeling and Real-World Scenarios

28. Build a logistic regression model predicting income (>50K) using selected numerical predictors. Preprocess the data and interpret key coefficients.
29. If you were designing a credit scoring system, how would you encode features like `employment.type` or `loan.purpose`?
30. In a dataset where 25% of income values are missing, which imputation strategy would you use, and why?
31. A customer satisfaction variable includes five levels from “Very satisfied” to “Very dissatisfied.” Which encoding method is most appropriate, and why?
32. Would you handle outliers in patient temperature data differently than outliers in income data? Explain your reasoning.

Ethics and Reflection

33. What ethical concerns might arise from dropping rows with missing data, particularly for underrepresented groups?
34. How can careless encoding of categorical variables lead to biased or unfair model outcomes? Provide an example.

35. A model performs well in training but poorly in production. How might decisions made during data preparation explain this discrepancy?
36. Reflect on a dataset you have worked with (or the *housePrice* dataset from the **liver** package). Which data preparation steps would you revise based on the techniques from this chapter?

Chapter 4

Exploratory Data Analysis

Exploratory Data Analysis (EDA) is the essential first step before building models or conducting statistical inference. It involves examining data carefully, thoroughly, and creatively to uncover insights. EDA helps reveal unexpected patterns, spot anomalies, and highlight relationships that shape the direction of further analysis.

EDA plays a pivotal role in the Data Science Workflow (see Figure 2.3), functioning as the bridge between data preparation (Chapter 3) and data setup for modeling (Chapter 6). This phase deepens our understanding of the data's structure, quality, and potential—ensuring that downstream decisions rest on a solid foundation.

Unlike formal hypothesis testing, EDA is not rigid or sequential. It is an iterative, open-ended process that invites curiosity. Different datasets pose different questions. Some exploratory paths reveal meaningful trends, while others point to data issues or dead ends. Through iterative exploration, analysts develop intuition and refine their focus, ultimately identifying the most informative features for modeling.

The primary aim of EDA is not to confirm theories, but to generate insight. Summary statistics, exploratory visualizations, and correlation measures offer an initial map of the data landscape. Such exploratory findings must be interpreted with caution, as they are preliminary and may not reflect causal relationships. Later chapters—particularly Chapter 5—will introduce formal tools for inference and prediction.

EDA also emphasizes the importance of *practical relevance*. In large datasets, even weak patterns can be statistically significant but lack real-world utility. For example, a weak correlation between customer engagement and churn may appear significant due to sample size, yet offer little actionable value. Integrating domain knowledge is essential for interpreting such findings.

Finally, EDA is closely tied to data quality. Outliers, missing values, inconsistent formats, and redundant variables often come to light during exploration. Addressing these issues early is critical for building effective and trustworthy models. The choice of EDA techniques depends on both the characteristics of the data and the analytical questions at hand. Histograms and box plots reveal distributions; scatter plots and correlation matrices highlight relationships. The next sections introduce these tools in context and explain how to apply them effectively.

What This Chapter Covers

This chapter introduces Exploratory Data Analysis (EDA) as a critical phase in the data science workflow. You will learn how to apply summary statistics and visual techniques to examine variable distributions, detect anomalies, and uncover relationships that inform downstream modeling. In addition, you will explore how correlation analysis can identify redundancy and how multivariate patterns contribute to predictive insight.

We begin with two foundational sections: *Key Objectives and Guiding Questions for EDA*, which frame the exploratory mindset, and *EDA as Data Storytelling*, which emphasizes the importance of communicating insights clearly and effectively.

Following these conceptual foundations, we walk you through a guided EDA of the *churn* dataset. You will explore how real-world patterns emerge from the data, how visualizations reveal customer behavior, and how these insights prepare the ground for classification modeling using k-nearest neighbors in Chapter 7.

The chapter concludes with hands-on exercises and a guided project using the *bank* dataset. This project offers further opportunities to practice EDA techniques and lays the foundation for the neural network case study presented in Chapter 12.

4.1 Key Objectives and Guiding Questions for EDA

Exploratory Data Analysis (EDA) marks the first substantive interaction between analyst and dataset—the moment when raw data begins to reveal its structure, surprises, and potential narratives. Rather than rushing into modeling, experienced data scientists pause to ask: *What is in the data? What patterns stand out? What problems need attention?*

Before applying specific techniques, it is helpful to clarify what EDA is designed to accomplish. At its core, EDA aims to:

- *Identify the structure of the data* – Detect variable types, value ranges, missing entries, and potential anomalies.
- *Examine variable distributions* – Assess spread, skewness, and central tendency for numerical and categorical features.
- *Investigate relationships between variables* – Uncover associations, dependencies, or interactions that may inform predictions.
- *Detect patterns and outliers* – Spot unusual values or subgroups that could indicate errors—or hidden signals.

Together, these objectives provide a foundation for effective modeling. They help analysts refine feature choices, anticipate modeling challenges, and surface early insights worth pursuing.

EDA becomes even more powerful when guided by focused questions. These typically fall into two broad categories: *univariate analysis*, which examines variables individually, and *multivariate analysis*, which explores their relationships.

Univariate analysis asks: What does each variable reveal on its own? This includes inspecting distributions, central tendencies, and variability, as well as checking for missing values or irregularities. Common questions include:

- What is the distribution of the target variable?
- How are numerical features—such as income or age—distributed?
- Are there missing values, and do they follow a specific pattern?

Histograms, box plots, and summary statistics (mean, median, quartiles, standard deviation) are essential tools at this stage.

Multivariate analysis shifts the focus to interactions among variables. It uncovers dependencies, correlations, or redundancies that may affect modeling. Key questions include:

- How does the target variable relate to its predictors?
- Are any predictors highly correlated, raising concerns about multicollinearity?
- How do categorical and numerical features interact?

Scatter plots, grouped visualizations, and correlation matrices help reveal these patterns and guide thoughtful feature selection.

One recurring challenge in EDA—especially for students—is deciding which plots or techniques are best suited to different types of data and questions. Table 4.1 offers a concise mapping of exploratory objectives to effective tools. It serves as a practical reference to support informed, purposeful analysis.

Table 4.1: Overview of Recommended Tools for Common EDA Objectives.

| Exploratory.Objective | Applicable.Data.Type | Recommended.Techniques |
|--|---------------------------|---|
| Examine a variable's distribution | Numerical | Histogram, box plot, density plot, summary statistics |
| Summarize a categorical variable | Categorical | Bar chart, frequency table |
| Identify outliers | Numerical | Box plot, histogram |
| Detect missing data patterns | Any | Summary statistics, missingness maps |
| Explore the relationship between two numerical variables | Numerical & Numerical | Scatter plot, correlation coefficient |
| Compare a numerical variable across groups | Numerical & Categorical | Box plot, grouped bar chart, violin plot |
| Analyze interactions between two categorical variables | Categorical & Categorical | Stacked bar chart, mosaic plot, contingency table |
| Assess correlation among multiple numerical variables | Multiple Numerical | Correlation matrix, scatterplot matrix |

By aligning key objectives with guiding questions and appropriate methods, EDA becomes more than an exploratory routine—it becomes a strategic component of the data science workflow. It improves data quality, informs feature design, and lays the foundation for effective modeling.

In the next section, we turn to the role of storytelling in EDA and explore how visual and narrative techniques can make exploratory insights more accessible and impactful.

4.2 EDA as Data Storytelling

Exploratory Data Analysis is not only a technical process for uncovering patterns—it is also a means of communicating insights clearly and persuasively. While EDA reveals structure, anomalies, and relationships, these

findings become valuable only when they are communicated with clarity and purpose. This is where data storytelling becomes essential: it transforms exploration into insight.

Effective storytelling in data science weaves together analytical evidence, contextual knowledge, and visual clarity. Rather than presenting statistics or charts in isolation, strong EDA links each observation to a broader narrative. Whether the audience includes fellow analysts, business stakeholders, or policymakers, the goal remains the same: to convey insights in ways that are meaningful and relevant.

Consider a typical finding: customers with high daytime usage are more likely to churn. Stating this fact is informative—but incomplete. A compelling narrative connects the pattern to its implication:

“Customers with extensive daytime usage are significantly more likely to churn, possibly due to pricing concerns or dissatisfaction with service quality. Targeted retention strategies, such as customized discounts or flexible pricing plans, may help mitigate this risk.”

This shift from raw description to interpretation is at the heart of data storytelling. It invites action and supports informed decision-making.

Visualizations play a central role in this process. While summary statistics provide a structural overview, visual displays make patterns tangible. Scatter plots and correlation matrices reveal relationships among numerical features; histograms and box plots clarify distributions and skewness; bar charts and stacked visuals enable comparisons across categories. Selecting the right visual tool enhances both understanding and communication.

Data storytelling is now common across domains—from business and journalism to public policy and scientific research. A well-known example appears in Hans Rosling’s TED Talk “[New insights on poverty](#)”. Figure Figure 4.1, adapted from his presentation, illustrates how GDP per capita and life expectancy have changed across global regions from 1962 to 2011. It presents decades of demographic and health data into an engaging and comprehensible format. Although this example is drawn from global development, the same principles apply when exploring customer behavior, financial trends, or healthcare outcomes.

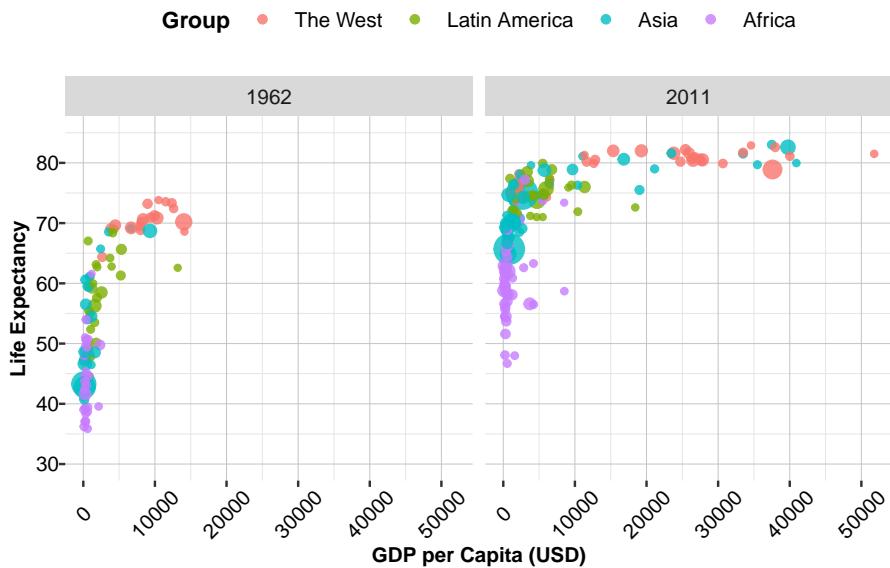


Figure 4.1: Changes in GDP per capita and life expectancy by region from 1962 to 2011. Dot size is proportional to population.

As you conduct EDA, it is worth asking not only *what* the data shows, but also *why* those patterns matter. What story is emerging? How can that story inform a decision, challenge an assumption, or inspire further analysis? Framing EDA as storytelling ensures that exploratory work is not merely descriptive but purposeful—anchored in the real-world questions that motivated the analysis in the first place.

To illustrate these principles in practice, the next section walks through a detailed EDA of customer churn—demonstrating how statistical summaries, visual tools, and domain reasoning can uncover meaningful patterns that guide predictive modeling.

4.3 EDA in Practice: The *Churn* Dataset

Exploratory Data Analysis (EDA) is most effective when applied to real data with real questions. In this section, we demonstrate how to conduct EDA using the *churn* dataset, which contains behavioral and demographic information about customers—along with a binary outcome indicating whether each customer has churned (i.e., discontinued the service).

This walkthrough follows the structure of the Data Science Workflow introduced in Chapter 2. We begin by briefly revisiting the first two steps—*Problem Understanding* and *Data Understanding*—to set the business context and examine the structure of the dataset. The main emphasis here is on *Step 3: Exploratory Data Analysis*, where we use summary statistics, visualizations, and guiding questions to uncover meaningful patterns related to customer churn.

The insights developed in this section serve as a springboard for the next phases of analysis: preparing the data for modeling in Chapter 6, building predictive models using k-Nearest Neighbors in Chapter 7, and evaluating model performance in Chapter 8. By working through these steps in sequence, you will see how a well-executed EDA not only enhances understanding but also improves the quality of decisions made through modeling.

Understanding the Churn Problem

Customer churn (the loss of existing customers) is a central concern in subscription-based industries such as telecommunications, finance, and streaming services. Because retaining customers is typically more cost-effective than acquiring new ones, understanding the drivers of churn is a priority for both analysts and decision-makers.

From a business standpoint, this challenge prompts three key questions:

- *Why* are customers deciding to leave?
- *What* behavioral or demographic patterns are associated with higher churn risk?
- *How* can these insights inform strategies to improve customer retention?

Exploratory Data Analysis (EDA) provides the foundation for addressing these questions. By identifying relevant patterns in the data, EDA uncovers potential signals that can inform targeted interventions. It also helps frame the predictive modeling task that follows.

In Chapter 7, we will develop a k-nearest neighbors (kNN) model to predict customer churn. Before building that model, however, it is essential to understand the structure of the data, the types of variables available, and the relationships they reveal.

Getting to Know the Data

Before diving into visualizations and summary statistics, it is important to understand the dataset we will explore throughout this chapter. The *churn* dataset, included in the **liver** package, provides a realistic example for practicing Exploratory Data Analysis (EDA). It includes 5,000 customer records across 20 variables, combining demographic information, service usage, account attributes, and customer support interactions.

The core variable of interest is *churn*, which indicates whether a customer has left the service (yes) or remained (no). This binary outcome will eventually serve as the target for classification modeling in Chapter 7, but our first task is to understand the data that surrounds it.

To load and inspect the dataset, run the following code in R:

```
library(liver)
data(churn)
str(churn)
'data.frame': 5000 obs. of 20 variables:
 $ state      : Factor w/ 51 levels "AK","AL","AR",...: 17 36 32 36 37 2
   ↪ 20 25 19 50 ...
 $ area.code   : Factor w/ 3 levels "area_code_408",...: 2 2 2 1 2 3 3 2
   ↪ 1 2 ...
 $ account.length: int 128 107 137 84 75 118 121 147 117 141 ...
 $ voice.plan  : Factor w/ 2 levels "yes","no": 1 1 2 2 2 2 1 2 2 1 ...
 $ voice.messages: int 25 26 0 0 0 0 24 0 0 37 ...
 $ intl.plan   : Factor w/ 2 levels "yes","no": 2 2 2 1 1 1 2 1 2 1 ...
 $ intl.mins   : num 10 13.7 12.2 6.6 10.1 6.3 7.5 7.1 8.7 11.2 ...
 $ intl.calls  : int 3 3 5 7 3 6 7 6 4 5 ...
 $ intl.charge : num 2.7 3.7 3.29 1.78 2.73 1.7 2.03 1.92 2.35 3.02 ...
 $ day.mins    : num 265 162 243 299 167 ...
 $ day.calls   : int 110 123 114 71 113 98 88 79 97 84 ...
 $ day.charge  : num 45.1 27.5 41.4 50.9 28.3 ...
 $ eve.mins   : num 197.4 195.5 121.2 61.9 148.3 ...
 $ eve.calls   : int 99 103 110 88 122 101 108 94 80 111 ...
 $ eve.charge  : num 16.78 16.62 10.3 5.26 12.61 ...
 $ night.mins  : num 245 254 163 197 187 ...
 $ night.calls : int 91 103 104 89 121 118 118 96 90 97 ...
 $ night.charge: num 11.01 11.45 7.32 8.86 8.41 ...
 $ customer.calls: int 1 1 0 2 3 0 3 0 1 0 ...
 $ churn       : Factor w/ 2 levels "yes","no": 2 2 2 2 2 2 2 2 2 2 ...
```

This reveals that the dataset is stored as a `data.frame` with 5000 observations and 20 variables. Most of the predictors are numerical or categorical features that describe how customers use the service and interact with the provider.

A structured overview of the variables is provided below:

- `state`: Categorical — U.S. state of the customer (51 levels).

- `area.code`: Categorical — Area code assigned to the customer.
- `account.length`: Discrete numerical — Number of days the account has been active.
- `voice.plan`: Binary categorical — Whether the customer subscribes to a voice mail plan.
- `voice.messages`: Discrete numerical — Number of voice mail messages received.
- `intl.plan`: Binary categorical — Whether the customer has an international calling plan.
- `intl.mins`: Continuous numerical — Total international call minutes.
- `intl.calls`: Discrete numerical — Number of international calls made.
- `intl.charge`: Continuous numerical — Charges for international calls.
- `day.mins`, `day.calls`, `day.charge`: Metrics for daytime usage.
- `eve.mins`, `eve.calls`, `eve.charge`: Metrics for evening usage.
- `night.mins`, `night.calls`, `night.charge`: Metrics for nighttime usage.
- `customer.calls`: Discrete numerical — Number of calls to customer service.
- `churn`: Binary categorical — Indicates whether the customer churned (yes/no).

To get a quick sense of each variable's distribution and spot potential issues, use the `summary()` function:

```
summary(churn)
  state           area.code   account.length  voice.plan
  WV      : 158  area_code_408:1259    Min.   : 1.0  yes:1323
  MN      : 125  area_code_415:2495  1st Qu.: 73.0   no :3677
  AL      : 124  area_code_510:1246  Median :100.0
  ID      : 119                           Mean   :100.3
  VA      : 118                           3rd Qu.:127.0
  OH      : 116                           Max.   :243.0
  (Other):4240
  voice.messages  intl.plan   intl.mins      intl.calls      intl.charge
  Min.   : 0.000  yes: 473  Min.   : 0.00  Min.   : 0.000  Min.   : 0.000
  1st Qu.: 0.000  no :4527   1st Qu.: 8.50  1st Qu.: 3.000  1st Qu.:2.300
  Median : 0.000                           Median :10.30  Median : 4.000  Median :2.780
  Mean   : 7.755                           Mean   :10.26  Mean   : 4.435  Mean   :2.771
  3rd Qu.:17.000                           3rd Qu.:12.00  3rd Qu.: 6.000  3rd Qu.:3.240
  Max.   :52.000                           Max.   :20.00  Max.   :20.000  Max.   :5.400
  day.mins       day.calls   day.charge     eve.mins      eve.calls
  Min.   : 0.0  Min.   : 0   Min.   : 0.00  Min.   : 0.0  Min.   : 0.0
  1st Qu.:143.7 1st Qu.: 87  1st Qu.:24.43  1st Qu.:166.4  1st Qu.:
  ↪ 87.0
  Median :180.1  Median :100  Median :30.62  Median :201.0  Median
  ↪ :100.0
  Mean   :180.3  Mean   :100  Mean   :30.65  Mean   :200.6  Mean   :100.2
  3rd Qu.:216.2  3rd Qu.:113  3rd Qu.:36.75  3rd Qu.:234.1  3rd
  ↪ Qu.:114.0
```

```

Max.    :351.5   Max.    :165   Max.    :59.76  Max.    :363.7   Max.    :170.0
Min.    :0.00    Min.    :0.0   Min.    :0.00   Min.    :0.000
1st Qu.:14.14   1st Qu.:166.9 1st Qu.:87.00 1st Qu.:7.510
Median  :17.09   Median :200.4  Median :100.00 Median  :9.020
Mean    :17.05   Mean   :200.4  Mean   :99.92  Mean   :9.018
3rd Qu.:19.90   3rd Qu.:234.7 3rd Qu.:113.00 3rd Qu.:10.560
Max.    :30.91   Max.   :395.0  Max.   :175.00 Max.   :17.770

customer.calls churn
Min.    :0.00    yes: 707
1st Qu.:1.00    no :4293
Median  :1.00
Mean    :1.57
3rd Qu.:2.00
Max.    :9.00

```

This high-level snapshot helps identify variables with unusual values, large spreads, or potential outliers. It also confirms that the dataset is clean—there are no missing values, and variable names are well-labeled and interpretable.

One observation worth highlighting: although there are 51 unique states, there are only 3 unique area codes. This suggests that area codes are not uniquely tied to states, which may influence how we interpret geographic information or decide which variable to use in modeling.

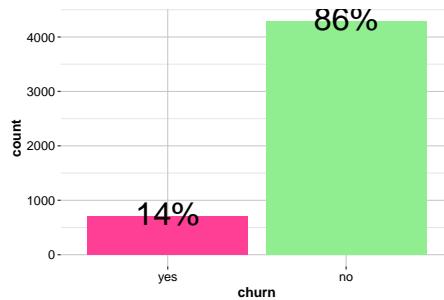
In the next sections, we begin our exploratory analysis—starting with the categorical variables. By examining how these variables relate to churn outcomes, we lay the groundwork for uncovering the patterns and insights that EDA is designed to reveal.

4.4 Examining Categorical Features and Churn Patterns

Categorical variables represent discrete groupings—such as labels, names, or binary flags—that encode structural information about customers and their interactions with a service. In the *churn* dataset, several such features stand out: `state`, `area.code`, `voice.plan`, and `intl.plan`. Analyzing their distributions—and how they relate to the outcome variable `churn`—can reveal early indicators of customer dissatisfaction or loyalty.

To begin, we examine the distribution of the target variable `churn`, which records whether a customer has left the service. Knowing this distribution is essential for assessing class balance—an important factor in both model performance and interpretability.

```
ggplot(data = churn, aes(x = churn, label =
  scales::percent(prop.table(stat(count)))) +
  geom_bar(fill = c("violetred1", "lightgreen")) +
  geom_text(stat = 'count', vjust = 0.4, size = 10)
```



The bar plot reveals a clear class imbalance: the majority of customers remain active (`churn = "no"`), while a smaller proportion have left the service (`churn = "yes"`). Approximately 14.1% of customers have churned.

This visualization builds on foundational plotting techniques introduced in Section 1.14. A simpler version—omitting color and percentage labels—can be generated with:

```
ggplot(data = churn) +
  geom_bar(aes(x = churn))
```

While quick to produce, the simplified plot may be less informative in presentation settings. Enhancements such as color coding and annotated percentages improve clarity and are especially helpful when communicating results to stakeholders without a technical background.

Class imbalance is more than a descriptive statistic—it can distort predictive modeling by encouraging algorithms to favor the majority class. We will return to this issue in Chapter 6, Section 6.6, where methods for addressing imbalance are introduced.

With this context in place, consider the next question: Are some service features more closely associated with churn than others? To explore this, we begin with the international calling plan.

Relationship Between Churn and Subscription Plans

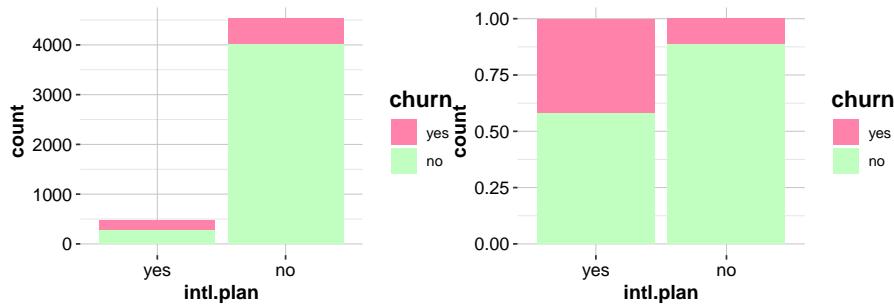
Among the features available in the `churn` dataset, the variable `intl.plan` is especially relevant from a business and customer experience standpoint. It

indicates whether a customer subscribes to an international calling plan—an option that may reflect unique communication needs and cost sensitivities.

As a binary categorical variable, `intl.plan` allows for a straightforward comparison of churn behavior between customers who do and do not use international services.

```
ggplot(data = churn) +
  geom_bar(aes(x = intl.plan, fill = churn)) +
  scale_fill_manual(values = c("palevioletred1", "darkseagreen1"))

ggplot(data = churn) +
  geom_bar(aes(x = intl.plan, fill = churn), position = "fill") +
  scale_fill_manual(values = c("palevioletred1", "darkseagreen1"))
```



The first plot shows the absolute number of churners and non-churners in each group. The second, normalized by group size, reveals a sharper insight: customers subscribed to an international plan are disproportionately more likely to churn.

This observation can be formally examined using a contingency table:

```
addmargins(table(churn$churn, churn$intl.plan,
                 dnn = c("Churn", "International Plan")))
International Plan
Churn   yes   no  Sum
  yes    199  508 707
  no     274 4019 4293
  Sum    473 4527 5000
```

The table confirms the visual pattern. Among those with an international plan, the rate of churn is noticeably elevated. This may signal customer dissatisfaction—possibly with pricing, call quality, or perceived value.

Such findings have practical implications. Identifying segments with elevated churn risk supports more informed retention strategies. For example,

international plan users might benefit from targeted offers, feedback surveys, or enhanced service guarantees.

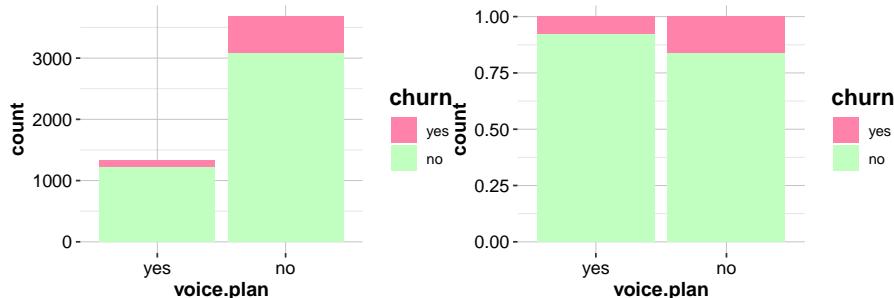
In the next subsection, we examine whether voice mail subscription shows a similar or contrasting association with churn behavior.

Relationship Between Churn and Voice Mail Plan

We now turn to another service-related feature: the voice mail plan. The variable `voice.plan` indicates whether a customer subscribes to this additional feature. Like the international plan, it may reflect aspects of customer engagement, satisfaction, or perceived value.

```
ggplot(data = churn) +
  geom_bar(aes(x = voice.plan, fill = churn)) +
  scale_fill_manual(values = c("palevioletred1", "darkseagreen1"))

ggplot(data = churn) +
  geom_bar(aes(x = voice.plan, fill = churn), position = "fill") +
  scale_fill_manual(values = c("palevioletred1", "darkseagreen1"))
```



The left panel displays the absolute number of customers who churned or remained, segmented by voice mail plan subscription. The right panel presents the same information in proportional form, highlighting differences in churn rates between groups. While the difference is not dramatic, customers with a voice mail plan appear to churn at slightly lower rates.

To support this visual impression, we generate a contingency table:

```
addmargins(table(churn$churn, churn$voice.plan, dnn = c("Churn", "Voice Mail
  ↵ Plan")))
  Voice Mail Plan
```

| Churn | yes | no | Sum |
|-------|------|------|------|
| yes | 102 | 605 | 707 |
| no | 1221 | 3072 | 4293 |
| Sum | 1323 | 3677 | 5000 |

The results confirm a modest association between voice mail usage and customer retention. While less striking than the pattern observed for the international calling plan, the trend may suggest that customers who subscribe to add-on services are more engaged—or perhaps more satisfied with their experience.

Even subtle effects can be meaningful in practice. In industries where churn reduction drives profitability, small gains in retention can yield significant impact. These early observations offer a foundation for later predictive modeling and underscore the value of analyzing categorical features in detail.

Summary of Categorical Findings

The exploratory analysis of categorical variables reveals several early signals that help frame the churn problem:

- Customers subscribed to an international calling plan are noticeably more likely to churn. This group may warrant closer attention, as it could reflect unmet service expectations or dissatisfaction with pricing.
- Those who subscribe to a voice mail plan tend to churn less frequently, hinting at a possible link between service engagement and customer loyalty.
- Even modest differences in churn rates across service features can help identify vulnerable segments and guide retention efforts.

These findings offer more than surface-level description—they begin to connect customer characteristics to behavioral outcomes. As we move on to numerical variables, we continue this investigative process, looking for patterns that deepen our understanding of churn dynamics.

4.5 Examining Numerical Features and Behavioral Trends

Now that we have examined categorical predictors, we shift our focus to numerical features—variables that often capture the subtleties of customer

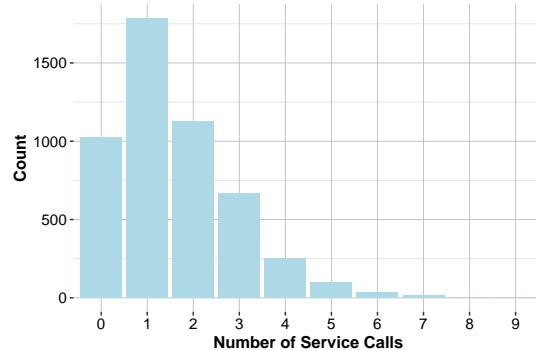
behavior. In the *churn* dataset, these include patterns of service usage and support interactions that may not be obvious at first glance. While summary statistics provide a high-level overview, it is through visual tools (e.g. histograms, box plots, and density plots) that meaningful trends, outliers, and predictive signals begin to emerge. By visualizing these variables, we can uncover behavioral patterns that may help explain why some customers stay and others leave.

Customer Service Calls and Churn

What can frequent support calls tell us about customer satisfaction? The variable `customer.calls` records how often each customer has contacted customer service—a potentially revealing signal. High contact frequency may indicate dissatisfaction, unresolved problems, or greater service dependency, all of which can raise the risk of churn.

We begin by visualizing the distribution of this variable:

```
ggplot(data = churn) +
  geom_bar(aes(x = factor(customer.calls)), fill = "lightblue") +
  labs(x = "Number of Service Calls", y = "Count")
```



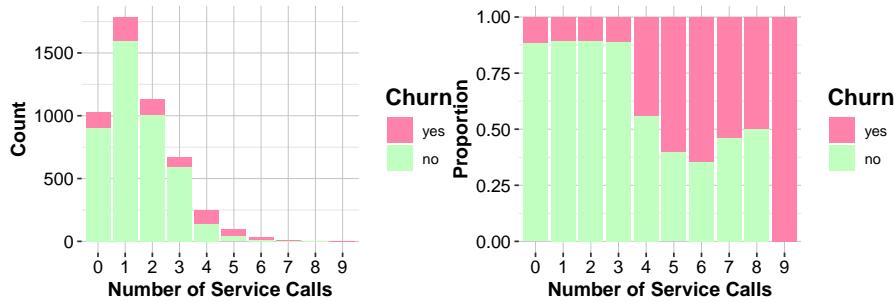
Because `customer.calls` is a count variable with a small number of discrete values (ranging from 0 to 9), we convert it to a factor and use a bar plot to highlight its distribution. While a histogram could be used, it offers little advantage here due to the limited number of distinct values. In contrast, density plots or box plots are less suitable and may obscure meaningful patterns.

The bar plot reveals a right-skewed distribution: most customers contact support rarely, while a smaller group makes frequent calls. To explore how

this behavior relates to churn, we compare service call frequency across churn outcomes:

```
ggplot(data = churn) +
  geom_bar(aes(x = factor(customer.calls), fill = churn), position =
    "stack") +
  scale_fill_manual(values = c("palevioletred1", "darkseagreen1")) +
  labs(x = "Number of Service Calls", y = "Count", fill = "Churn")

ggplot(data = churn) +
  geom_bar(aes(x = factor(customer.calls), fill = churn), position = "fill")
  + scale_fill_manual(values = c("palevioletred1", "darkseagreen1")) +
  labs(x = "Number of Service Calls", y = "Proportion", fill = "Churn")
```



The left plot shows raw counts of churners and non-churners at each call level. The right plot—normalized by call frequency—reveals a clear trend: customers who make *four or more service calls* are significantly more likely to churn.

This pattern suggests that frequent contact with customer service may signal dissatisfaction or unmet expectations. From a business perspective, this insight could inform retention strategies—for example, flagging at-risk customers after their third support call. From a modeling perspective, creating a binary feature such as “frequent caller” may better capture this non-linear trend than using the raw count alone.

These kinds of behavioral signals are precisely what EDA is designed to uncover. In the next subsection, we shift our focus to usage patterns—starting with daytime call minutes.

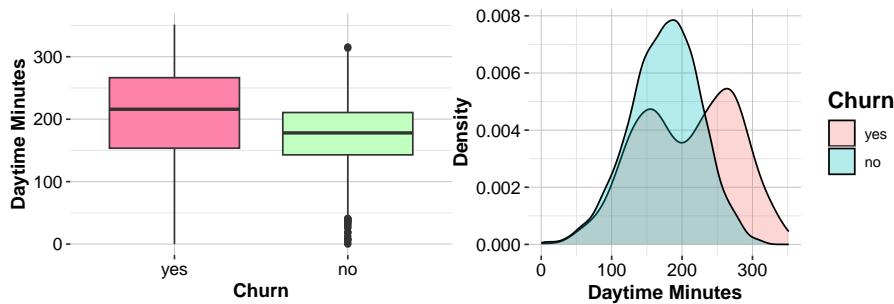
Daytime Minutes and Churn

Do customers who use their phone service more frequently during the day also churn more often? One variable that can help answer this is `day.mins`, which records the number of minutes each customer spends on daytime calls. High daytime usage may indicate stronger dependency on the service—along with heightened sensitivity to pricing, call quality, or reliability.

Because `day.mins` is a continuous numerical variable, we use a box plot to compare distributions across churn outcomes and a density plot to visualize the shape of each distribution. These plots help detect shifts, spread, and overlap between groups. You could also explore this variable with a histogram, which offers another view of distribution—try generating one using `geom_histogram()` as an exercise.

```
ggplot(data = churn) +
  geom_boxplot(aes(x = churn, y = day.mins),
               fill = c("palevioletred1", "darkseagreen1")) +
  labs(x = "Churn", y = "Daytime Minutes")

ggplot(data = churn) +
  geom_density(aes(x = day.mins, fill = churn), alpha = 0.3) +
  labs(x = "Daytime Minutes", y = "Density", fill = "Churn")
```



The box plot shows that customers who churn tend to have higher daytime usage, and the density plot confirms this shift. While the difference is not dramatic, it is consistent—suggesting that high-usage customers may be at elevated churn risk.

Key Insights and Business Implications: Customers with high `day.mins` usage are more likely to churn. This trend may reflect dissatisfaction driven by unmet service expectations or cost concerns. From a business perspective, proactive retention strategies—such as personalized plans or loyalty

benefits—could help retain high-usage customers. From a modeling perspective, `day.mins` is a valuable predictor and may also interact with other features such as `day.charge` or `customer.calls`.

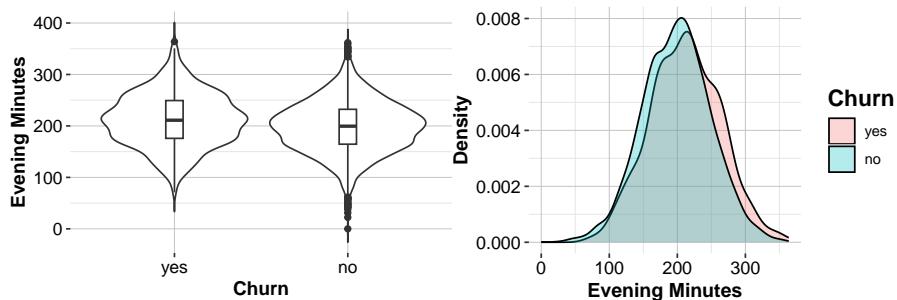
Evening and Nighttime Minutes

Do customers who make more evening or nighttime calls tend to churn more often? To investigate this, we examine two numerical features: `eve.mins`, which measures total minutes of evening calls, and `night.mins`, which records nighttime usage. These time periods may reflect different customer routines, pricing structures, or service expectations compared to daytime activity.

Since both variables are continuous, we begin by visualizing `eve.mins` using a violin plot layered with a box plot to show distributional shape and spread, and a density plot to examine the overlap between churners and non-churners:

```
ggplot(data = churn, aes(x = churn, y = eve.mins)) +
  geom_violin(trim = FALSE) +
  geom_boxplot(width = 0.1) +
  labs(x = "Churn", y = "Evening Minutes")

ggplot(data = churn) +
  geom_density(aes(x = eve.mins, fill = churn), alpha = 0.3) +
  labs(x = "Evening Minutes", y = "Density", fill = "Churn")
```

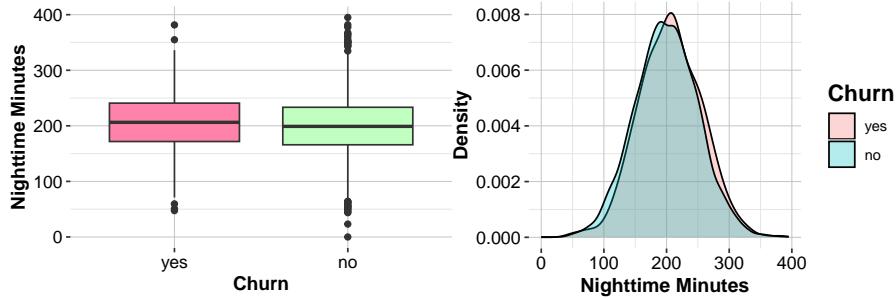


The visualizations suggest a slight shift toward higher evening call usage among churners, but the overlap between groups remains substantial. Compared to `day.mins`, the association here is clearly weaker.

We now apply the same approach to `night.mins`:

```
ggplot(data = churn) +
  geom_boxplot(aes(x = churn, y = night.mins), fill = c("palevioletred1",
  "darkseagreen1")) +
  labs(x = "Churn", y = "Nighttime Minutes")

ggplot(data = churn) +
  geom_density(aes(x = night.mins, fill = churn), alpha = 0.3) +
  labs(x = "Nighttime Minutes", y = "Density", fill = "Churn")
```



The box and density plots for nighttime usage reveal no meaningful difference between churners and non-churners, suggesting little association with churn behavior.

Key Insights and Business Implications: Evening and nighttime usage show limited individual predictive power. While eve.mins may be slightly elevated for churners, the effect is modest and inconsistent. These variables are unlikely to drive churn predictions on their own but may contribute in combination with other features. For practical applications, they are best interpreted as part of a broader usage pattern rather than in isolation.

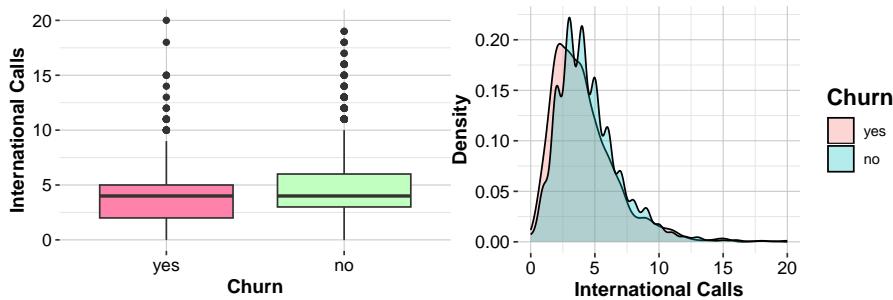
International Calls and Churn

In a previous section, we observed that customers subscribed to an international calling plan are more likely to churn. But does actual calling behavior support this pattern? The variable intl.calls records the total number of international calls placed by each customer, providing a behavioral counterpart to the plan subscription flag.

Because intl.calls is a discrete numeric variable with moderate range, we examine its distribution by churn status using a box plot and a density plot:

```
ggplot(data = churn) +
  geom_boxplot(aes(x = churn, y = intl.calls),
               fill = c("palevioletred1", "darkseagreen1")) +
  labs(x = "Churn", y = "International Calls")

ggplot(data = churn) +
  geom_density(aes(x = intl.calls, fill = churn), alpha = 0.3) +
  labs(x = "International Calls", y = "Density", fill = "Churn")
```



The visualizations suggest that churners make slightly fewer international calls than non-churners. However, the difference is small and the distributions overlap considerably.

Key Insights and Business Implications: The `intl.calls` variable shows only a weak association with churn. This finding suggests that subscribing to an international plan (a binary indicator) may matter more than how much it is actually used. Formal statistical testing—such as a two-sample t-test—can help assess whether the observed difference is significant. We will revisit this question in Chapter 5 when discussing hypothesis testing.

Summary of Numerical Findings

The exploratory analysis of numerical features in the `churn` dataset reveals the following key patterns:

- `customer.calls` and `day.mins` are most strongly associated with churn. Customers making four or more service calls, and those with high daytime call usage, are more likely to leave the service.
- These features are likely to play an important role in predictive modeling and may serve as effective early indicators for targeted retention strategies.

- Other variables, such as `eve.mins`, `night.mins`, and `intl.calls`, show weaker or inconsistent associations with churn when analyzed individually. Their value may become clearer in multivariate contexts or when interacting with other predictors.

These findings will inform both feature selection and model development in later chapters. In the next section, we examine relationships between variables to detect potential redundancy and explore more complex patterns relevant to classification tasks.

4.6 Identifying Redundancy and Correlated Features

Before analyzing more complex interactions among variables, it is helpful to assess how individual features relate to one another. Correlation analysis is a key tool in this process, revealing variables that may carry overlapping information or exhibit redundancy. Identifying such relationships early not only simplifies the modeling process but also enhances interpretability and reduces the risk of multicollinearity.

Correlation quantifies the degree to which two variables move together. A positive correlation indicates that as one variable increases, the other tends to increase as well; a negative correlation suggests that one tends to decrease as the other increases. If there is little or no correlation, changes in one variable do not help predict changes in the other. This relationship is typically summarized using the Pearson correlation coefficient, denoted by r , which ranges from -1 to 1. A value of $r = 1$ implies a perfect positive linear relationship, $r = -1$ implies a perfect negative one, and $r = 0$ suggests no linear association.

Figure 4.2 illustrates scatterplots with various degrees of correlation.

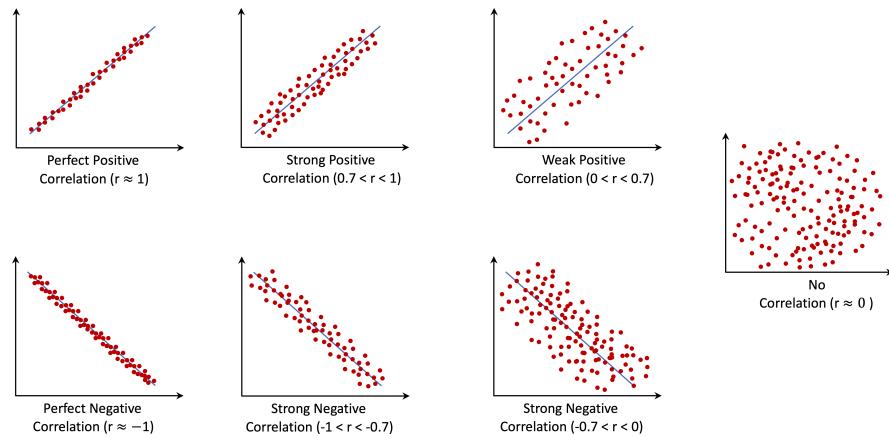


Figure 4.2: Example scatterplots showing different correlation coefficients.

Note: Correlation does not imply causation. For example, a strong positive correlation between customer service calls and churn does not mean that making service calls *causes* customers to leave. Rather, both behaviors may stem from an underlying factor—such as dissatisfaction with service.

To underscore this caution, Figure 4.3 presents a memorable example from Messerli (2012). It shows a surprisingly strong correlation between per capita chocolate consumption and Nobel Prize wins across countries. While entertaining, the example highlights a key message: correlations may arise from coincidence, reverse causality, or the influence of a third variable.

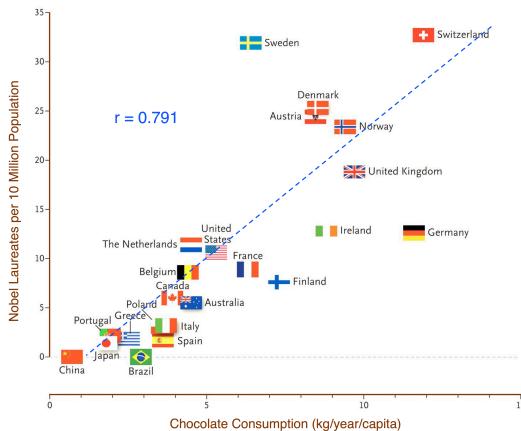


Figure 4.3: Scatterplot illustrating the correlation between Nobel Prize wins and chocolate consumption (per 10 million population) across countries. Adapted from Messerli (2012).

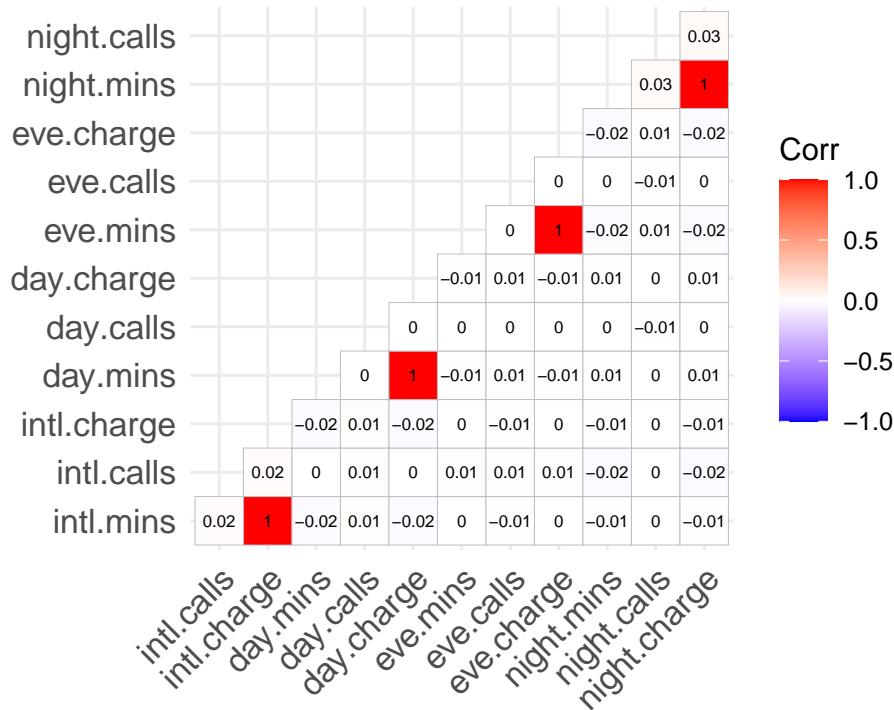
Returning to our dataset, we now compute and visualize the correlation matrix for key numerical variables using a heatmap. This is done using the `ggcorrplot()` function from the `ggcorrplot` package, an extension of `ggplot2` that produces clean, annotated correlation plots. It allows users to customize triangle display, label coefficients, and control aesthetics for better interpretation.

```
library(ggcorrplot)

variable_list = c("intl.mins", "intl.calls", "intl.charge",
                 "day.mins", "day.calls", "day.charge",
                 "eve.mins", "eve.calls", "eve.charge",
                 "night.mins", "night.calls", "night.charge")

cor_matrix = cor(churn[, variable_list])

ggcorrplot(cor_matrix, type = "lower", lab = TRUE, lab_size = 2)
```



The resulting heatmap highlights two main patterns. First, the *charge variables* (such as `day.charge`, `eve.charge`, and `intl.charge`) are nearly perfectly correlated with their corresponding *minutes variables*. This is expected, as charges are calculated directly from call duration. Including both in a model would add redundancy without new information.

Second, the *call count variables* (e.g., `day.calls`, `night.calls`) show only weak correlations with their corresponding *minutes variables*. This suggests that call frequency and call duration capture distinct aspects of user behavior and should be considered complementary rather than interchangeable.

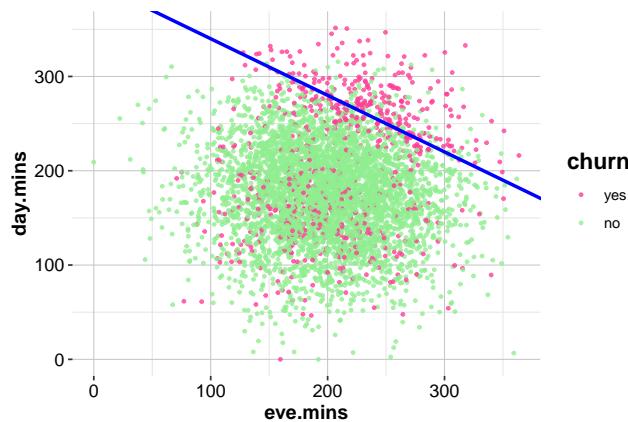
Based on these findings, it is advisable to remove the charge variables when preparing the dataset for modeling, while retaining both frequency and duration measures. Doing so streamlines the feature set, avoids multicollinearity, and contributes to more interpretable and stable predictive models.

4.7 Exploring Multivariate Relationships

While correlation analysis helps identify linear associations between pairs of variables, it often misses important patterns that emerge only when multiple features are considered together. Multivariate analysis enables us to explore such interactions and conditional relationships—insights that are crucial for understanding complex behaviors like customer churn.

Let us begin by jointly analyzing `day.mins` (daytime call duration) and `eve.mins` (evening call duration). Are customers who use the service heavily during both periods more likely to leave?

```
ggplot(data = churn) +
  geom_point(aes(x = eve.mins, y = day.mins, color = churn), size = 0.7,
             alpha = 0.8) +
  scale_color_manual(values = c("violetred1", "lightgreen")) +
  geom_abline(intercept = 400, slope = -0.6, color = "blue", size = 1)
```



To help identify behavioral segments, we add a diagonal line using `geom_abline()`, which overlays a reference line with a given intercept and slope. This line represents the boundary:

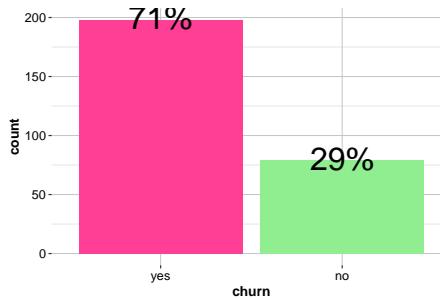
$$\text{day.mins} = 400 - 0.6 \times \text{eve.mins}$$

Customers above this line—those with *simultaneously high day and evening usage*—show a noticeably greater tendency to churn. This insight would not be visible from analyzing either variable in isolation, highlighting the importance of considering variable interactions.

To quantify this finding, we isolate this high-usage group using the `subset()` function:

```
sub_churn = subset(churn, (day.mins > 400 - 0.6 * eve.mins))

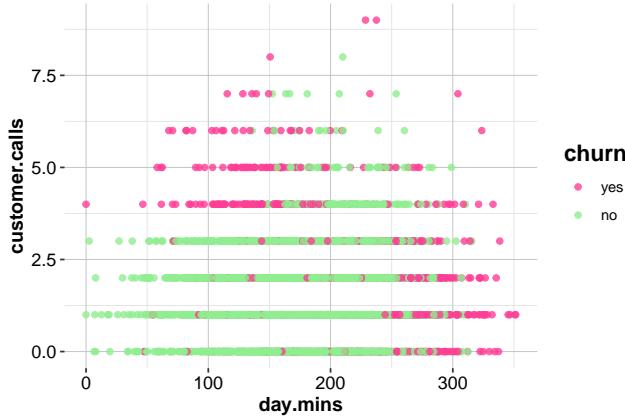
ggplot(data = sub_churn, aes(x = churn, label =
  scales::percent(prop.table(stat(count)))) +
  geom_bar(fill = c("violetred1", "lightgreen")) +
  geom_text(stat = 'count', vjust = 0.4, size = 10)
```



The churn rate within this group is approximately 71.5%, compared to an overall churn rate of 14.1%. This difference suggests that intensive usage across multiple periods may be a signal of dissatisfaction—possibly related to billing complexity or service expectations.

Next, we examine another potential interaction: the relationship between `customer.calls` and `day.mins`.

```
ggplot(data = churn) +
  geom_point(aes(x = day.mins, y = customer.calls, color = churn), alpha =
  0.8) +
  scale_color_manual(values = c("violetred1", "lightgreen"))
```



This scatter plot reveals an interesting segment: customers who contact support frequently but have low usage tend to churn at higher rates. These may be frustrated users who experience issues despite using the service sparingly. In contrast, customers with high usage and support contact show more varied churn outcomes—possibly because they value the service more, despite occasional problems.

Together, these examples illustrate how multivariate analysis can expose interaction effects and behavioral subgroups that are not apparent in simpler analyses. From a modeling perspective, such findings suggest opportunities to engineer features that capture joint effects—such as interaction terms or binary flags for high-risk combinations. From a business standpoint, these insights can guide proactive interventions aimed at specific customer profiles.

In Chapter 5, we return to these findings using formal statistical methods to assess their significance and evaluate their role in predictive modeling.

4.8 Insights from Exploratory Data Analysis

The exploratory analysis of the *churn* dataset has revealed several behavioral patterns and feature relationships that are relevant to both modeling and business strategy. By systematically examining individual variables and their interactions, we identified predictors of customer attrition and gained a deeper understanding of usage and support dynamics.

One of the most salient findings is the association between customer service interactions and churn. Customers who contact support four or more times are substantially more likely to leave the service, pointing to dissatisfaction

or unresolved problems. This pattern suggests a need for early intervention based on support behavior.

High combined usage of daytime and evening minutes also emerged as a strong churn signal. Customers in this segment exhibit churn rates several times higher than the average, likely reflecting unmet expectations related to pricing, billing, or service quality. Similarly, frequent service calls paired with low daytime usage define another at-risk segment—one that would not be evident in univariate summaries alone.

The presence of an international calling plan is another key predictor: customers subscribed to this plan churn at significantly higher rates. In contrast, customers with a voice mail plan are less likely to churn, suggesting that certain service features may promote retention.

The analysis also highlighted opportunities to streamline the feature set. Charge variables, which are perfectly correlated with call durations, should be removed to reduce redundancy and avoid multicollinearity. In addition, variables such as state and area.code appear to offer limited predictive value and may be excluded from modeling unless interaction effects are hypothesized.

These insights inform several strategic directions. Customers making their third support call could be flagged for proactive retention efforts—such as tailored offers or escalated service handling. High-usage customers may benefit from personalized rate plans or loyalty incentives. The international plan should be reevaluated for its pricing structure and value proposition. And voice mail features could be more prominently marketed or bundled as part of retention strategies.

Even variables with weak individual associations—such as nighttime minutes—may still contribute predictive value in combination with other features. Their joint effects will be assessed in subsequent chapters through statistical testing and machine learning.

By uncovering these behavioral patterns through exploratory analysis, we have laid the groundwork for the next phase of the data science workflow: building predictive models that quantify risk and support decision-making. In the next chapter, we introduce statistical tools to formalize the relationships observed here, setting the stage for classification models such as logistic regression and k-nearest neighbors.

4.9 Chapter Summary and Takeaways

This chapter introduced Exploratory Data Analysis (EDA) as a foundational step in the data science workflow—an approach that emphasizes curiosity-

driven inquiry guided by patterns within the data. Unlike confirmatory analysis, EDA does not begin with fixed hypotheses; rather, it involves iterative examination through descriptive statistics, visualizations, and correlation measures to uncover structure, relationships, and irregularities.

Drawing on the *churn* dataset, we investigated behavioral signals associated with customer attrition. The analysis revealed that frequent contact with customer service, heavy usage during both daytime and evening hours, and enrollment in international calling plans were linked to a heightened risk of churn. Conversely, customers subscribed to a voice mail plan were more likely to stay, suggesting that certain service features may enhance satisfaction. We also detected redundancy among variables—for instance, call charges being deterministically derived from call duration—and used multivariate plots to uncover interaction effects invisible in isolation.

Several principles emerged through this exploratory process. EDA is iterative and context-sensitive, helping to surface relevant questions rather than answer predefined ones. Identifying class imbalance, outliers, and redundant predictors early supports more effective model design. Visualizations serve not only as tools for discovery but also as instruments for clear communication. Interactions between features—such as the combination of high usage and frequent support calls—can point to distinct behavioral profiles and suggest avenues for intervention. Throughout, domain knowledge played a key role in interpreting the practical meaning of patterns that might otherwise be mistaken for noise or spurious correlation.

These findings lay the groundwork for the next stages of analysis. Equipped with the ability to uncover structure, spot data quality issues, and extract early insights, you are now ready to formalize these observations using statistical inference. In the following chapter, we introduce statistical tools to evaluate the significance of relationships observed here and begin building predictive models—starting with logistic regression and k-nearest neighbors—to estimate churn risk and support data-driven decisions.

4.10 Exercises

The following exercises are designed to reinforce core principles of exploratory data analysis introduced in this chapter. They span conceptual understanding, hands-on implementation, and critical thinking. The section begins with interpretive questions, followed by applied analysis using the *bank* dataset, and concludes with advanced challenge problems for synthesis and reflection.

Conceptual Questions

1. Why is exploratory data analysis essential before building predictive models? What risks might arise if this step is skipped?
2. If a variable does not show a clear relationship with the target during EDA, should it be excluded from modeling? Consider potential interactions, hidden effects, and the role of feature selection in your response.
3. What does it mean for two variables to be correlated? Explain direction and strength of correlation and contrast correlation with causation using an example.
4. How can you detect and address correlated predictors during EDA? Describe how this helps improve modeling performance and interpretability.
5. What are the consequences of including highly correlated variables in a predictive model? Discuss the effects on performance, interpretability, and model stability.
6. Is it always advisable to remove one of two correlated predictors? When might keeping both be justified?
7. For each of the following methods—histograms, box plots, density plots, scatter plots, summary statistics, correlation analysis, contingency tables, bar plots, and heatmaps—state whether it applies to categorical data, numerical data, or both. Briefly describe its use in EDA.
8. A telecommunications firm finds that customers with high day and evening minutes tend to churn more. What actions could the company take in response, and how might this guide customer retention?
9. Suppose several pairs of variables in a dataset have high correlation (e.g., $r > 0.9$). How would you handle this to ensure robust and interpretable modeling?
10. Why is it important to consider both statistical and practical relevance in evaluating correlations? Give an example of a statistically strong but practically weak correlation.
11. Why is it important to investigate multivariate relationships in EDA? Describe a scenario where an interaction between two variables reveals a pattern that univariate analysis would miss.
12. How does data visualization support EDA? Provide two specific examples where visual tools reveal insights that summary statistics might obscure.

13. Suppose you discover that customers with both high daytime usage and frequent service calls are more likely to churn. What business strategies might be informed by this finding?
14. What are some causes of outliers in data? How would you determine whether to retain, modify, or exclude an outlier?
15. Why are missing values important to address during EDA? Discuss strategies for handling missing data and when each might be appropriate.

Hands-On Practice: Exploring the Bank Dataset

The *bank* dataset from the R package **liver** contains data on direct marketing campaigns of a Portuguese bank. The objective is to predict whether a client subscribes to a term deposit. This dataset will also be used for classification in Chapters 7 and 12. More details: <https://rdrr.io/cran/liver/man/bank.html>

To load and inspect the dataset:

```
library(liver)
data(bank)
str(bank)
```

16. *Dataset Overview*: Summarize the structure and variable types. What does this reveal about the dataset?
17. *Target Variable Analysis*: Plot the target variable `deposit`. What proportion of clients subscribed to a term deposit?
18. *Binary Variables*: Explore `default`, `housing`, and `loan`. Use bar plots and contingency tables. What patterns emerge?
19. *Numerical Variables*: Visualize the distributions of numerical features using histograms and box plots. Note any skewness or unusual observations.
20. *Outlier Detection*: Identify outliers in numerical variables. What strategies would you consider for managing them?
21. *Correlation Matrix*: Compute and visualize the correlation matrix. Which variables are highly correlated, and how might you address this?
22. *Key Findings*: Summarize your main EDA findings. How would you present these in a report?
23. *Business Insights*: What actionable conclusions could the bank draw from these patterns?

24. *Multivariate Patterns*: Explore whether higher values of campaign (number of contacts) relate to greater subscription rates. Visualize and interpret.
25. *Feature Engineering Insight*: Based on your findings, propose one new variable that could help improve model performance.
26. *Seasonality Effects*: Investigate subscription rates by month. Are some months more successful than others?
27. *Employment Type*: How does job relate to deposit? Which categories have higher success?
28. *Interaction Effects*: Analyze the joint impact of education and job on subscription. What do you observe?
29. *Effect of Call Duration*: Does the duration of last contact influence deposit outcome?
30. *Campaign Comparison*: How do success rates vary across campaigns? What strategies might this suggest?

Challenge Problems

31. *Data Storytelling from EDA*: Create a concise 1–2 plot visual summary of an EDA finding from the *bank* dataset. Focus on making the insight clear to a non-technical audience, using clean annotations and brief explanation.
32. *Segment-Based Risk Profiling*: Use the *adult* dataset to identify a subgroup likely to earn over \$50K. Describe their profile and how you uncovered it through EDA.
33. *EDA vs. Predictive Modeling*: A variable appears weakly related to the target in univariate plots. Under what conditions could it still improve model accuracy?
34. *Exploring Class Imbalance by Subgroup*: Does the proportion of deposit outcomes differ by marital or job? What insights or hypotheses emerge?
35. *Feature Pruning with EDA*: Using the *adult* dataset, identify predictors that may not add value for modeling. Justify your choices with EDA.

Chapter 5

Statistical Inference and Hypothesis Testing

Imagine a company observes that customers who make frequent service calls are more likely to churn. Is this pattern the result of a genuine relationship, or is it simply the result of random variation? This is the kind of question that *statistical inference* is designed to answer.

Statistical inference uses sample data to make conclusions about a larger population, enabling us to move beyond simple descriptions and toward informed, evidence-based decisions. In this chapter, we explore how inference helps answer questions such as: *What proportion of customers are likely to churn?* and *How many service calls do churners typically make?*

In Chapter 4, we explored the *churn* dataset and identified potential patterns, such as increased churn among customers with high daytime minutes or frequent service calls. But how confident can we be that these differences reflect genuine effects rather than sampling variability? Statistical inference offers tools to rigorously assess such questions.

Statistical inference requires not only computation, but also *critical thinking*. This chapter highlights how to identify common misuses of statistical reasoning and avoid pitfalls in your own analyses, enabling you to draw conclusions that are both rigorous and defensible. For a deeper look at how statistical reasoning can be misused, Darrell Huff's classic book *How to Lie with Statistics* remains a relevant resource. Developing a strong foundation in inference will not only enhance your confidence in results but also strengthen your ability to interpret evidence and reason analytically.

What This Chapter Covers

In this chapter, we introduce statistical inference, a set of tools that allow us to draw conclusions about populations based on samples. Building on your exploratory work, you will learn how to transition from spotting patterns to validating insights—an essential shift in the data science workflow. Specifically, you will explore:

- *Point estimation*: Using sample statistics to estimate population parameters.
- *Confidence intervals*: Quantifying uncertainty and interpreting the range of plausible values.
- *Hypothesis testing*: Determining whether observed effects reflect genuine patterns or chance.
- *Practical tools*: Applying common statistical tests including the one- and two-sample *t*-tests, proportion tests, Z-tests, ANOVA, Chi-square tests, and correlation tests.

You will apply these tools to real-world datasets in **R**, gaining hands-on experience with inference techniques that support sound, data-driven decisions. Along the way, you will sharpen your ability to distinguish signal from noise, interpret *p*-values and confidence intervals responsibly, and recognize both the power and limitations of statistical evidence.

By the end of the chapter, you will also revisit where statistical inference fits in the broader data science workflow, especially in steps like validating data partitions and selecting meaningful features for modeling, as discussed in Chapter 6.

5.1 Introduction to Statistical Inference

Statistical inference bridges the gap between *what we observe in a sample* and *what we aim to understand about the population*, as illustrated in Figure 5.1. It plays a central role in the Data Science Workflow (see Figure 2.3), particularly after exploratory analysis and before modeling. While exploratory data analysis (EDA) helps uncover potential patterns—such as higher churn among customers with frequent service calls—*inference* provides a formal framework for testing whether those patterns are statistically meaningful or likely due to random chance.

In fact, hypothesis testing is often used in the next step of the workflow—for example, to verify that training and test sets retain the key characteristics of the full dataset, as discussed in Chapter 6.

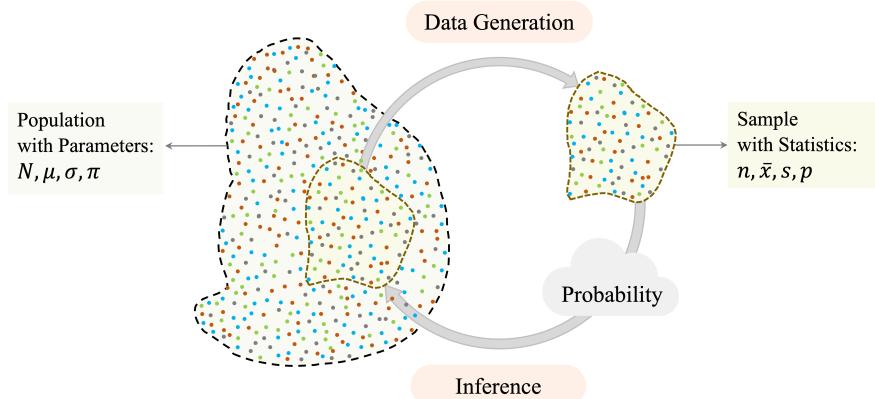


Figure 5.1: A conceptual overview of statistical inference. We generate data from a population to infer properties such as the mean, using probability to quantify uncertainty.

As summarized in Figure 5.2, statistical inference relies on three core components:

- *Point estimation*: Estimating population parameters (e.g., the mean or proportion) using sample data.
- *Confidence intervals*: Quantifying uncertainty around these estimates.
- *Hypothesis testing*: Assessing whether observed effects are statistically significant or likely due to chance.

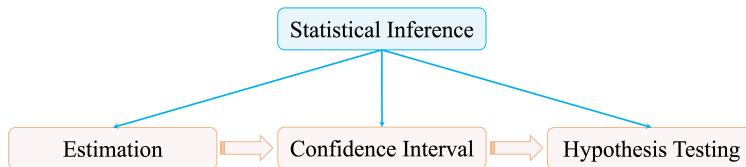


Figure 5.2: The three core goals of statistical inference: point estimation, confidence intervals, and hypothesis testing. Together, these components allow analysts to make principled inferences from data, moving beyond description toward reliable generalization.

These components build upon one another, starting with estimation, incorporating uncertainty, and culminating in formal testing. Together, they enable *data-driven decision-making* by helping distinguish real signals from statistical noise. The next sections explore each component in turn, starting with point estimation, through concrete examples and applications in R.

5.2 Estimation: Drawing Informed Conclusions from Sample Data

Suppose a company is planning its customer retention strategy and wants to know: what proportion of users are likely to churn? Or, how many times does a typical chunner contact customer service before leaving? These are not just descriptive questions—they are inferential ones. To answer them reliably, we turn to *estimation*, a fundamental task in statistical inference.

Estimation allows us to draw informed conclusions about population characteristics using sample data—especially when examining the entire population is impractical or impossible. In the *churn* dataset, for example, we may want to estimate the average number of customer service calls among chunners or determine the proportion of customers subscribed to the International Plan.

As outlined in Figure 5.2, estimation typically takes two forms. **Point estimation** provides a single best guess for a population parameter—for instance, using the sample mean as an estimate of the population mean. **Interval estimation**, by contrast, accounts for sampling uncertainty by offering a range of plausible values (a confidence interval) within which the true parameter is likely to lie.

Let us explore both types of estimation using concrete examples from the *churn* dataset.

Example: Estimating the Proportion of Churners

To estimate the *proportion of chunners* in the population, we use the sample proportion as a point estimate:

```
library(liver)
data(churn)

# Compute the sample proportion of chunners
prop.table(table(churn$churn))["yes"]
  yes
0.1414
```

The estimated proportion of churners is 0.14. This value provides a reasonable estimate of the true proportion in the broader customer population.

Example: Estimating the Mean Number of Customer Service Calls

To estimate the *average number of customer service calls among churners*, we calculate the sample mean:

```
# Filter churners
churned_customers = subset(churn, churn == "yes")

# Calculate the sample mean
(mean_calls <- mean(churned_customers$customer.calls))
[1] 2.254597
```

The average number of service calls among churners is 2.25. This sample mean serves as a point estimate for the population average.

While point estimates are informative, they do not communicate how *precise* those estimates are. Without accounting for uncertainty, we risk mistaking random variation for meaningful insight—an especially common pitfall when interpreting small or noisy datasets.

This is where confidence intervals play a crucial role. They provide a principled way to express uncertainty and assess the reliability of our estimates. In the next section, we learn how to construct and interpret confidence intervals, addressing questions such as: *How close is our estimate likely to be to the true value?* and *What range of values is supported by the data?*

5.3 Quantifying Uncertainty: Confidence Intervals

Suppose a company reports that the average number of customer service calls made by churned customers is 4. But how much trust should we place in that number? Could the true average be slightly higher, or significantly lower? A single number rarely tells the whole story. This is where *confidence intervals* become essential.

Confidence intervals help quantify the uncertainty associated with estimates of population parameters. Rather than reporting only a point estimate, such as “the average is 4,” a confidence interval might say, “we are 95% confident that the true average lies between 3.8 and 4.2.” This interval accounts for the natural variability that arises when working with sample data.

Formally, a confidence interval combines a point estimate, such as the sample mean or proportion, with a margin of error that reflects expected sampling variability. The general form is:

Point Estimate \pm Margin of Error

For a population mean, the confidence interval is calculated using the following formula:

$$\bar{x} \pm z_{\frac{\alpha}{2}} \times \left(\frac{s}{\sqrt{n}} \right),$$

where \bar{x} is the sample mean, s is the sample standard deviation, n is the sample size, and $z_{\frac{\alpha}{2}}$ is the critical value from the standard normal distribution (for example, 1.96 for a 95% confidence level).

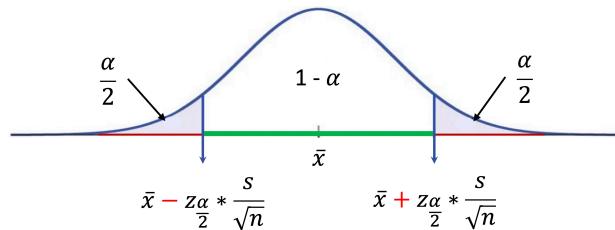


Figure 5.3: Confidence interval for the population mean. The interval is centered around the point estimate, with the width determined by the margin of error. The confidence level specifies the probability that the interval contains the true population parameter.

Several factors influence the width of a confidence interval. If the sample size is large, the interval tends to be narrower, indicating more precision. On the other hand, greater variability in the data leads to a wider interval. The choice of confidence level also matters. For instance, a 99% confidence level produces a wider interval than a 90% level because it must include a broader range of plausible values.

To illustrate this, imagine estimating the average height of all students in a university. If the sample includes only 10 students, the resulting interval will likely be wide due to limited data. If the sample includes 1,000 students, the estimate becomes more precise and the confidence interval narrower. This example highlights how increasing the amount of data reduces uncertainty and enhances the reliability of our estimates.

Let us apply this concept to the *churn* dataset by constructing a 95% confidence interval for the average number of customer service calls among churned customers:

```
# Calculate mean and standard error
mean_calls <- mean(churned_customers$customer.calls)
se_calls <- sd(churned_customers$customer.calls) /
  sqrt(nrow(churned_customers))

# Confidence Interval
z_score <- 1.96 # For 95% confidence
ci_lower <- mean_calls - z_score * se_calls
ci_upper <- mean_calls + z_score * se_calls

cat("95% Confidence Interval: [", ci_lower, ", ", ci_upper, "]")
95% Confidence Interval: [ 2.120509 , 2.388685 ]
```

If the computed interval is [2.12, 2.39], we can state that we are 95% confident the true average number of customer service calls for churned customers falls within this range. Technically, this means that if we were to repeat the process many times using different random samples, about 95% of those intervals would contain the true population mean.

When working with smaller samples, it is preferable to use the *t*-distribution instead of the normal distribution. The *t*-distribution accounts for the added uncertainty introduced when estimating the population standard deviation from a small sample. This adjustment is applied automatically in R when using the `t.test()` function:

```
t.test(churned_customers$customer.calls, conf.level = 0.95)$conf.int
[1] 2.120509 2.388685
attr("conf.level")
[1] 0.95
```

Although `t.test()` is primarily used for hypothesis testing, it also calculates confidence intervals using the appropriate distribution based on the sample size and variability.

Confidence intervals are particularly useful when comparing groups. For example, if the confidence intervals for churners and non-churners do not substantially overlap, this suggests a meaningful difference in customer behavior. By providing a range of plausible values instead of a single point estimate, confidence intervals enhance the transparency and credibility of data-driven conclusions.

In the next section, we extend this framework by introducing *hypothesis testing*, a formal approach for evaluating whether observed effects in a sample are likely to reflect true differences in the population or could have occurred by random chance.

5.4 Hypothesis Testing

Suppose you want to evaluate whether a new pricing strategy improves customer retention. You observe a difference in churn rates between two groups: customers exposed to the strategy and those who were not. But is this difference meaningful, or could it simply be due to chance? Hypothesis testing provides a systematic way to evaluate such questions using statistical evidence.

In the broader data science workflow, hypothesis testing helps move from exploratory observations to conclusions that are supported by formal statistical reasoning. It allows us to test whether the patterns we see in data are likely to reflect genuine population-level effects rather than the randomness of sampling.

At its core, hypothesis testing evaluates claims about population characteristics using sample data. While confidence intervals provide a range of plausible values for an estimate, hypothesis testing focuses on whether the observed evidence is strong enough to support a specific claim about a parameter. The overall process—from stating hypotheses to drawing conclusions—is summarized visually in Figure 5.4.

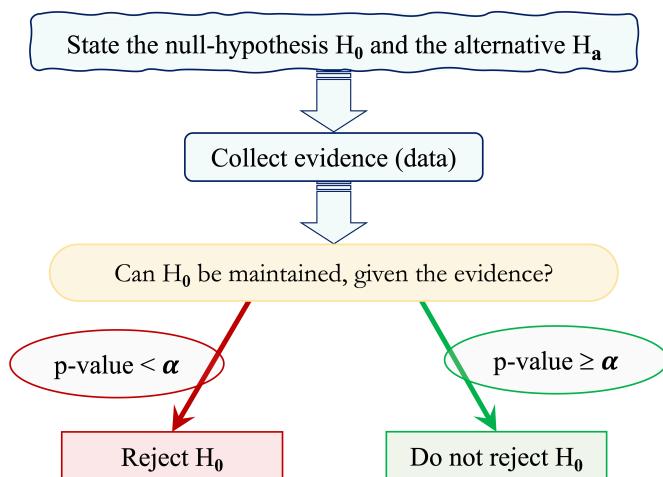


Figure 5.4: Visual summary of hypothesis testing, showing how sample evidence informs the decision to reject or fail to reject the null hypothesis (H_0).

The hypothesis testing framework revolves around two competing statements:

- The *null hypothesis* (H_0) represents the default assumption, typically stating that there is no effect, no difference, or no association.
- The *alternative hypothesis* (H_a) contradicts H_0 , asserting that a difference, effect, or relationship does exist.

We assess the strength of evidence by calculating a *p-value*: the probability of obtaining results as extreme as those observed, assuming H_0 is true. Smaller *p*-values indicate stronger evidence against H_0 . By convention, we reject H_0 when the *p*-value falls below a predetermined threshold, called the *significance level* (α), often set at 0.05.

Reject H_0 if the *p*-value $< \alpha$.

For example, if $p = 0.03$ and $\alpha = 0.05$, the evidence is considered sufficient to reject H_0 . If $p = 0.12$, we retain H_0 because the evidence is not strong enough to support H_a . It is important to remember that a *p*-value does not reflect the probability that H_0 is true, but rather the likelihood of observing such data if H_0 were true.

The choice of α determines the risk of making a *Type I error*, which means rejecting a true null hypothesis. In high-stakes applications, such as clinical trials or aerospace systems, researchers may choose a stricter threshold (e.g., $\alpha = 0.01$) to minimize this risk.

Although *p*-values offer a standardized way to evaluate hypotheses, they have limitations. In large datasets, even minor differences can appear statistically significant. In small samples, meaningful effects may go undetected due to limited power. Furthermore, reducing outcomes to a binary decision—reject or retain H_0 —can oversimplify complex findings.

5.4.1 Types of Hypothesis Tests

The structure of a hypothesis test depends on the nature of the question and the direction of the effect being tested. Depending on how the alternative hypothesis is formulated, tests can take one of the following forms:

- *Left-tailed test*: The alternative hypothesis proposes that the parameter is *less than* a specified value ($H_a : \theta < \theta_0$). For instance, testing whether the average number of customer service calls is less than 3.
- *Right-tailed test*: The alternative hypothesis asserts that the parameter is *greater than* a specified value ($H_a : \theta > \theta_0$). For example, testing whether the churn rate exceeds 30%.

- *Two-tailed test:* The alternative hypothesis states that the parameter is *not equal* to a specified value ($H_a : \theta \neq \theta_0$), capturing deviations in either direction. An example would be testing whether the mean monthly charges differ from \$50.

To better understand the logic behind these decisions, consider the analogy of a criminal trial. The *null hypothesis* represents the presumption of innocence. The *alternative hypothesis* corresponds to a claim of guilt. The jury weighs the evidence and must either convict (reject H_0) or acquit (fail to reject H_0 due to insufficient evidence). Just as legal verdicts can result in mistakes, hypothesis testing is subject to two types of error, summarized in Table 5.1.

Table 5.1: Possible outcomes of hypothesis testing, with two correct decisions and two types of error.

| <i>Decision</i> | <i>Reality: H_0 is True</i> | <i>Reality: H_0 is False</i> |
|----------------------|--|--|
| Fail to Reject H_0 | <i>Correct Decision: Acquit an innocent person.</i> | <i>Type II Error (β): Acquit a guilty person.</i> |
| Reject H_0 | <i>Type I Error (α): Convict an innocent person.</i> | <i>Correct Decision: Convict a guilty person.</i> |

A *Type I error* (α) occurs when we reject H_0 even though it is actually true—this is analogous to convicting an innocent person. A *Type II error* (β) occurs when we fail to reject H_0 even though it is false—similar to acquitting someone who is guilty.

The probability of a Type I error is set by the significance level α , which we choose before conducting the test. In contrast, the probability of a Type II error depends on multiple factors, including the sample size, variability in the data, and the magnitude of the effect. A key concept here is *power*, which refers to the test's ability to detect a true effect when it exists. Higher power reduces the likelihood of making a Type II error and is often achieved by increasing the sample size.

5.4.2 Common Tests and When to Use Them

How can you determine whether a new marketing campaign increases conversion rates? Or whether customer churn differs significantly across segments? Such questions are central to statistical analysis and require choosing the appropriate hypothesis test based on the structure of the data.

Many learners find it challenging to identify which test suits their specific question, particularly when dealing with different types of variables. To support this decision process, Table Table 5.2 summarizes commonly used hypothesis tests, their null hypotheses, and the data types they apply to. This reference is introduced in lectures and included here as a guide throughout the book.

Table 5.2: Common hypothesis tests, their null hypotheses, and the types of variables they apply to.

| Test | Null Hypothesis
(H_0) | Applied To |
|------------------------------|------------------------------|---|
| One-sample t-test | $\mu = \mu_0$ | Single numerical variable |
| Test for Proportion | $\pi = \pi_0$ | Single categorical variable |
| Two-sample t-test | $\mu_1 = \mu_2$ | Numerical outcome by binary group |
| Two-proportion Z-test | $\pi_1 = \pi_2$ | Two binary categorical variables |
| Chi-square Test | $\pi_1 = \pi_2 = \pi_3$ | Two categorical variables with > 2 categories |
| Analysis of Variance (ANOVA) | $\mu_1 = \mu_2 = \mu_3$ | Numerical outcome by multi-level group |
| Correlation Test | $\rho = 0$ | Two numerical variables |

Each test serves a distinct purpose. The t-test compares group means, the Z-test evaluates differences in proportions, and the Chi-square test assesses associations between categorical variables. ANOVA determines whether means differ across multiple groups, and the correlation test measures the strength and direction of linear relationships between numerical variables.

These tests form a core part of the data analyst's toolkit. In the sections that follow, we demonstrate how to apply them using real-world data and **R**, guiding you through both interpretation and implementation.

5.5 One-sample t-test

Suppose a company believes that customers who churn typically make two service calls before leaving. Is this assumption still valid? Have recent service trends altered customer behavior? The one-sample t-test offers a principled way to test whether a sample mean differs from a specified benchmark, making it ideal for such questions.

The one-sample t-test evaluates whether the mean of a numerical variable in a population is equal to a given value. It is typically applied when we want to compare a sample mean to a theoretical expectation or business assumption. The test statistic follows a t-distribution, which accounts for added uncertainty when the population standard deviation is unknown and must be estimated from the sample.

The hypotheses depend on the research question:

- *Two-tailed test:*

$$\begin{cases} H_0 : \mu = \mu_0 \\ H_a : \mu \neq \mu_0 \end{cases}$$

- *Left-tailed test:*

$$\begin{cases} H_0 : \mu \geq \mu_0 \\ H_a : \mu < \mu_0 \end{cases}$$

- *Right-tailed test:*

$$\begin{cases} H_0 : \mu \leq \mu_0 \\ H_a : \mu > \mu_0 \end{cases}$$

Example: Testing the Average Number of Customer Service Calls

Suppose we want to test whether the average number of customer service calls among churners differs from the assumed value of 2. The hypotheses are:

$$\begin{cases} H_0 : \mu = 2 \\ H_a : \mu \neq 2 \end{cases}$$

We use the *churn* dataset from the **liver** package and filter for customers who have churned:

```
library(liver)
data(churn)

# Filter churners
churned_customers <- subset(churn, churn == "yes")
```

We use the *subset()* function to isolate the churned customers. The relevant variable is *customer.calls*, which records how many times each customer contacted service before churning.

We then apply the *t.test()* function in **R** to compare the sample mean to the benchmark value of 2:

```
t_test <- t.test(churned_customers$customer.calls, mu = 2)
t_test

One Sample t-test

data: churned_customers$customer.calls
t = 3.7278, df = 706, p-value = 0.0002086
alternative hypothesis: true mean is not equal to 2
95 percent confidence interval:
2.120509 2.388685
sample estimates:
mean of x
2.254597
```

The function returns several key results: a test statistic, p-value, confidence interval, and degrees of freedom. The p -value is 0, which is smaller than $\alpha = 0.05$. We therefore reject the null hypothesis and conclude that the average number of calls differs from 2.

The 95% *confidence interval* is [2.12, 2.39]. Since this interval does not include 2, the result reinforces our conclusion. The sample mean is 2.25, which serves as our best estimate of the true average in the population.

Because the population standard deviation is unknown, the test statistic follows a t-distribution with $n - 1$ degrees of freedom.

The one-sample t-test is a powerful method for comparing a sample mean to a fixed benchmark. While statistical significance provides useful evidence, practical relevance must also be considered. A difference of 0.1 calls may be negligible, whereas a difference of two calls might signal a need to revisit service policies. By combining statistical reasoning with business context, the one-sample t-test helps translate data into informed action.

5.6 Hypothesis Testing for Proportion

Suppose a telecom company believes that 15% of its customers churn each year. Has that rate changed in the current quarter? Are recent retention strategies making a measurable difference? These are typical questions in business analytics. When we want to assess whether the observed proportion of a binary outcome—such as churned versus not churned—differs from a historical or expected benchmark, we use a *test for proportion*. This approach is particularly useful for binary categorical variables and helps determine whether a sample proportion deviates meaningfully from a specified value.

The test evaluates whether the proportion (π) of a specific category in a population differs significantly from a hypothesized value (π_0). In the example

below, we apply this test using the `prop.test()` function in **R**, which can be used to evaluate either a single proportion or the difference between two proportions.

Example: Testing Whether the Churn Rate Differs from 15%

A company assumes that 15% of its customers churn. To test whether the actual churn rate in the *churn* dataset differs from this expectation, we set up the following hypotheses:

$$\begin{cases} H_0 : \pi = 0.15 \\ H_a : \pi \neq 0.15 \end{cases}$$

We conduct a two-tailed proportion test using **R**:

```
prop_test <- prop.test(x = sum(churn$churn == "yes"),
                       n = nrow(churn),
                       p = 0.15)
prop_test

 1-sample proportions test with continuity correction

data: sum(churn$churn == "yes") out of nrow(churn), null probability 0.15
X-squared = 2.8333, df = 1, p-value = 0.09233
alternative hypothesis: true p is not equal to 0.15
95 percent confidence interval:
 0.1319201 0.1514362
sample estimates:
 p
0.1414
```

In this code, `x` gives the number of churners, `n` specifies the total sample size, and `p = 0.15` defines the hypothesized proportion. The test evaluates whether the observed proportion significantly differs from this value, using a chi-square approximation.

The output includes three key results: the *p*-value, the confidence interval, and the observed sample proportion. The *p*-value is 0.0923. Because it exceeds the standard significance level of $\alpha = 0.05$, we do not reject the null hypothesis. We conclude that there is no statistically significant evidence to suggest that the population proportion of churners differs from 15%.

The 95% confidence interval for the true proportion is 0.13, 0.15. Since this interval contains 0.15, the result is consistent with the decision not to reject H_0 . The observed sample proportion is 0.14, which serves as our best estimate of the true churn rate in the population.

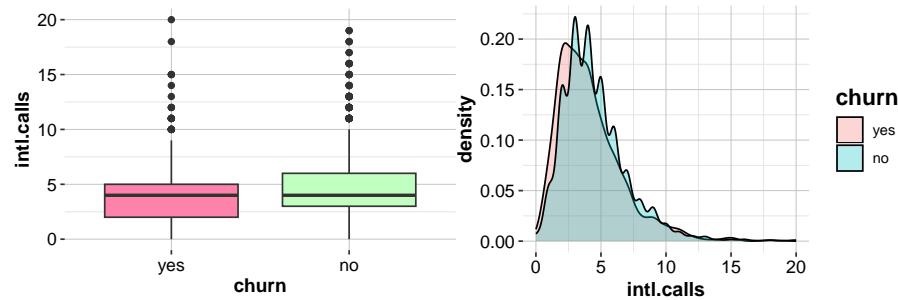
This example illustrates how a proportion test can be used to validate assumptions about categorical outcomes. The *p*-value determines statistical

significance, while the confidence interval and sample proportion provide important context. When used alongside domain knowledge, this method supports evidence-based decisions in evaluating changes in customer behavior.

5.7 Two-sample t-test

Do customers who churn use international calling services less than those who stay? If so, can international call behavior help predict churn risk? The two-sample t-test provides a statistical approach to answer such questions by comparing the means of a numerical variable between two independent groups. Also known as *Student's t-test*, this method evaluates whether an observed difference in group means is statistically significant or likely due to random variation. It is named after [William Sealy Gosset](#), who published under the pseudonym "Student" while working at Guinness Brewery.

In Chapter 4, Section 4.5, we explored the relationship between international calling behavior (`intl.calls`) and churn status using box plots and density plots. These visualizations suggested that churners might make fewer international calls than non-churners. But are these differences statistically meaningful?



The boxplot (left) shows that churners tend to make slightly fewer international calls, and the density plot (right) suggests their distribution is modestly left-shifted. To evaluate whether these differences are statistically significant, we apply the two-sample t-test.

We start by formulating the hypotheses:

$$\begin{cases} H_0 : \mu_1 = \mu_2 \\ H_a : \mu_1 \neq \mu_2 \end{cases}$$

Here, μ_1 and μ_2 represent the average number of international calls for churners and non-churners, respectively. The null hypothesis states that the group means are equal, while the alternative suggests a difference.

To perform the test, we use the `t.test()` function in R. The formula syntax `intl.calls ~ churn` tells R to compare the `intl.calls` variable across the two levels of the `churn` group:

```
t_test_calls <- t.test(intl.calls ~ churn, data = churn)
t_test_calls

Welch Two Sample t-test

data: intl.calls by churn
t = -3.2138, df = 931.13, p-value = 0.001355
alternative hypothesis: true difference in means between group yes and
→ group no is not equal to 0
95 percent confidence interval:
-0.5324872 -0.1287201
sample estimates:
mean in group yes mean in group no
4.151344        4.481947
```

This function outputs the test statistic, p -value, degrees of freedom, confidence interval, and estimated group means. The p -value is 0.0014, which is smaller than the standard significance level of $\alpha = 0.05$. We therefore reject the null hypothesis and conclude that the mean number of international calls differs between churners and non-churners.

The 95% confidence interval for the difference in means is [-0.53, -0.13]. Because zero is not in this range, the result supports our conclusion that the group means are different. The estimated group means are 4.15 for churners and 4.48 for non-churners. These values suggest that churners tend to make fewer international calls on average—an insight that may have business implications.

The two-sample t-test assumes that the two groups are independent and that the numerical variable is approximately normally distributed within each group. When sample sizes are large, the Central Limit Theorem provides robustness to non-normality. If the data are heavily skewed or contain outliers, a nonparametric alternative such as the Mann–Whitney U test may be more appropriate.

From a business perspective, this result suggests that international call frequency may be associated with customer churn. If lower usage of international calls reflects dissatisfaction or cost sensitivity, the company might consider offering promotional plans or targeted outreach to low-usage customers.

It is also worth considering the size of the effect. A statistically significant difference does not always imply a practically meaningful one. For example, a difference of 0.1 calls may be negligible, while a difference of two calls could justify a change in pricing or support strategies.

Although we used a two-tailed test here, a one-tailed version may be preferable if the research question is directional, for example, if we expect churners to make *fewer* international calls.

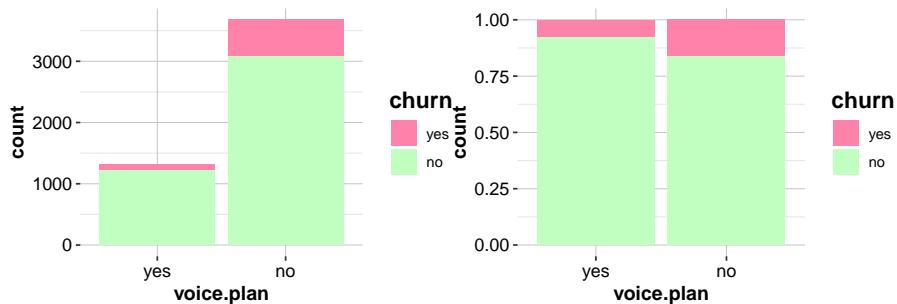
The two-sample t-test is a powerful tool for confirming patterns seen in exploratory analysis. It connects statistical reasoning with real-world decisions, allowing analysts to move from visual insights to data-driven conclusions.

5.8 Two-sample Z-test

Suppose your company notices that customers without a Voice Mail Plan seem to churn more often. Is this pattern statistically significant, or just a coincidence? The two-sample Z-test helps answer questions like this by comparing proportions between two independent groups.

While the two-sample t-test is used for comparing means of numerical variables, the Z-test is designed for binary outcomes such as churn status or plan enrollment. It evaluates whether the observed difference in proportions between two groups is statistically significant or likely due to chance.

In Chapter 4.4, we explored the relationship between the *Voice Mail Plan* (*voice.plan*) and churn status using bar plots. While the visualizations suggested that churn rates might differ between customers with and without a Voice Mail Plan, hypothesis testing provides a formal framework for evaluating whether this difference is statistically meaningful.



The left plot shows raw churn counts across Voice Mail Plan categories, while the right plot shows proportional differences. Customers without a Voice Mail Plan appear more likely to churn. To test whether this visual pattern reflects a statistically significant difference, we define the hypotheses:

$$\begin{cases} H_0 : \pi_1 = \pi_2 \\ H_a : \pi_1 \neq \pi_2 \end{cases}$$

Here, π_1 and π_2 are the proportions of Voice Mail Plan users among churners and non-churners, respectively.

We begin by summarizing the data in a contingency table:

```
table_plan = table(churn$churn, churn$voice.plan, dnn = c("churn",
  ↵ "voice.plan"))
table_plan
  voice.plan
churn   yes   no
  yes   102   605
  no   1221  3072
```

Next, we apply the `prop.test()` function to compare the two proportions:

```
z_test = prop.test(table_plan)
z_test

 2-sample test for equality of proportions with continuity correction

data: table_plan
X-squared = 60.552, df = 1, p-value = 7.165e-15
alternative hypothesis: two.sided
95 percent confidence interval:
-0.1701734 -0.1101165
sample estimates:
prop 1    prop 2
0.1442716 0.2844165
```

The `prop.test()` function performs a test for equality of two proportions. It returns a p -value, confidence interval for the difference in proportions, and the estimated group proportions.

The p -value is 0, which is smaller than the significance level of 0.05. This leads us to reject the null hypothesis and conclude that the proportion of Voice Mail Plan users differs significantly between churners and non-churners.

The 95% confidence interval for the difference in proportions is [-0.1702, -0.1101]. Since this interval does not include zero, it reinforces our conclusion. The observed sample proportions are 0.1443 for churners and 0.2844

for non-churners, suggesting that customers without a Voice Mail Plan are more likely to churn.

This result highlights a potential link between Voice Mail Plan usage and customer retention. Businesses might explore whether the plan provides value to customers or if it serves as a marker of engagement. They may also consider promoting the plan to at-risk customers. While the statistical result is clear, practical significance requires further evaluation—would promoting plan enrollment lead to better retention outcomes? That is a question best answered through follow-up testing and experimentation.

The two-sample Z-test provides a rigorous way to compare proportions and complements visual exploration. By combining statistical inference with domain knowledge, companies can make informed decisions to improve retention strategies.

5.9 Chi-square Test

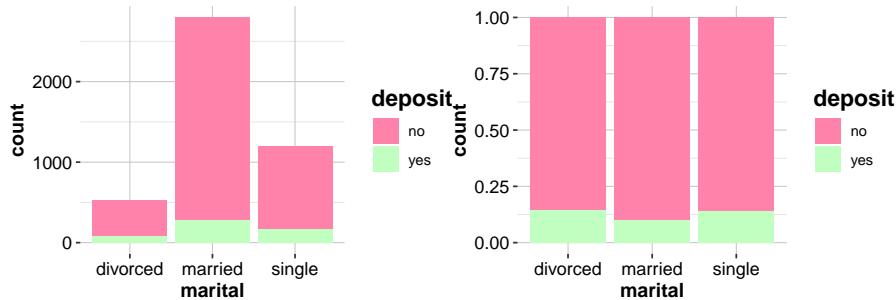
Do deposit subscription rates vary by marital status? And if so, can banks use this information to better target their campaigns? These are typical questions in marketing analytics, where categorical variables often have more than two levels. The *Chi-square test* is a statistical method designed to answer such questions. It evaluates whether two categorical variables are associated, making it especially useful in business contexts such as customer segmentation and marketing strategy.

While earlier tests focused on comparing means or proportions between two groups, the Chi-square test examines whether the distribution of outcomes across categories deviates from what we would expect under the assumption of independence. It quantifies whether observed discrepancies in frequencies are too large to be attributed to random variation.

To illustrate, we analyze whether marital status is associated with deposit purchases in the *bank* dataset (from the **liver** package). The variable `marital` includes “divorced,” “married,” and “single” categories. The outcome variable `deposit` indicates whether a customer purchased a term deposit (“yes” or “no”). This dataset will also appear in later chapters for classification modeling.

We begin by visualizing the relationship between marital status and deposit subscription using bar plots:

The first plot shows the raw counts of deposit outcomes across marital categories, while the second plot presents relative proportions. The visuals suggest possible differences in behavior among marital groups, but we need statistical evidence to determine whether these differences are significant.



To perform a Chi-square test, we first summarize the data in a **contingency table**. This table tallies the number of observations for each combination of marital status and deposit outcome:

```
table_marital <- table(bank$deposit, bank$marital, dnn = c("deposit",
  "marital"))
table_marital
  marital
  deposit divorced married single
    no      451     2520   1029
    yes      77      277    167
```

This table serves as the input to the `chisq.test()` function, which is designed to assess whether two categorical variables are independent. The hypotheses are:

$$\begin{cases} H_0 : \text{marital and deposit are independent} \\ H_a : \text{marital and deposit are not independent} \end{cases}$$

We conduct the test as follows:

```
chisq_test <- chisq.test(table_marital)
chisq_test

Pearson's Chi-squared test

data: table_marital
X-squared = 19.03, df = 2, p-value = 7.374e-05
```

The `chisq.test()` function compares the observed counts in the table with expected counts under independence. It returns a p -value, Chi-square statistic, degrees of freedom, and expected frequencies. If the p -value is 7.3735354×10^{-5} and less than the significance level ($\alpha = 0.05$), we reject the null hypothesis and conclude that marital status and deposit purchase are not independent.

The expected counts are especially useful. They show how many deposit purchases would be expected in each marital category if no association existed. By comparing these to the observed counts, we can identify which group differences are driving the result. For instance, if married customers subscribe to deposits more frequently than expected, they may be a key contributor to the observed association.

From a business standpoint, this finding supports the idea that marketing strategies can be tailored based on marital status. Banks might promote family-oriented savings plans for married clients, while designing more flexible, goal-oriented offers for single individuals.

The Chi-square test is a tool for uncovering associations between categorical variables. It complements visualizations and provides a rigorous foundation for exploring customer behavior patterns and designing more targeted outreach strategies.

5.10 Analysis of Variance (ANOVA) Test

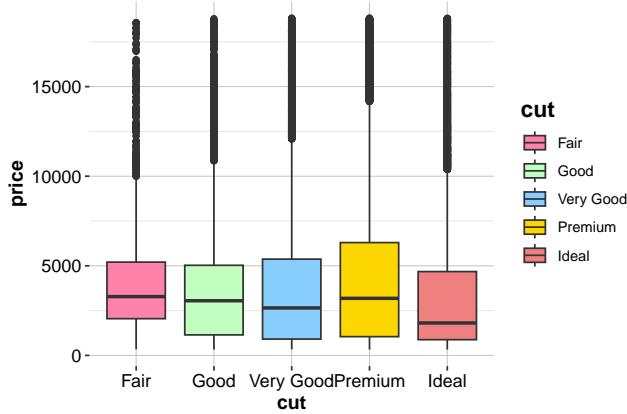
So far, we have examined hypothesis tests that compare two groups, such as the *two-sample t-test* and the *Z-test*. But what if we want to compare more than two groups? For example, does the average price of diamonds vary across different quality ratings? When dealing with a categorical variable that has multiple levels, the *Analysis of Variance (ANOVA)* provides a principled way to test whether at least one group mean differs significantly from the others.

ANOVA is especially useful for evaluating how a categorical factor with more than two levels affects a numerical outcome. It assesses whether the variability between group means is greater than what would be expected due to random sampling alone. The test statistic follows an F-distribution, which compares variance across and within groups.

To illustrate, consider the *diamonds* dataset from the **ggplot2** package. We analyze whether the mean price (`price`) differs by cut quality (`cut`), which has five levels: “Fair,” “Good,” “Very Good,” “Premium,” and “Ideal.” What can we learn from the distribution of prices across cut categories?

```
data(diamonds)

ggplot(data = diamonds) +
  geom_boxplot(aes(x = cut, y = price, fill = cut)) +
  scale_fill_manual(values = c("palevioletred1", "darkseagreen1",
    "skyblue1", "gold1", "lightcoral"))
```



The box plot shows the spread and median prices for each cut category. While the visual suggests that better cuts tend to command higher prices, we turn to statistical testing to determine whether these differences are meaningful.

To evaluate whether cut quality affects diamond price, we compare the mean price across all five categories. Our hypotheses are:

$$\begin{cases} H_0 : \mu_1 = \mu_2 = \mu_3 = \mu_4 = \mu_5 & \text{(All group means are equal)} \\ H_a : \text{At least one group mean differs} & \end{cases}$$

We apply the `aov()` function in R, which fits a linear model and returns an ANOVA table that summarizes variation between and within groups:

```
anova_test <- aov(price ~ cut, data = diamonds)
summary(anova_test)
Df      Sum Sq  Mean Sq F value Pr(>F)
cut      4 1.104e+10 2.760e+09   175.7 <2e-16 ***
Residuals 53935 8.474e+11 1.571e+07
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The output provides the F-statistic, degrees of freedom, and *p*-value. If the *p*-value is less than the significance level ($\alpha = 0.05$), we reject the null hypothesis and conclude that cut quality has a statistically significant effect on diamond price.

However, rejecting H_0 only tells us that at least one group differs. It does not tell us which cuts differ from each other. To identify the specific differences, we can use post-hoc tests such as Tukey's Honest Significant Difference (HSD) test, which controls for multiple comparisons while revealing which group pairs have statistically different means.

From a business standpoint, these insights can support pricing strategy. If higher-quality cuts yield higher prices, retailers might highlight them in marketing campaigns. On the other hand, if prices between mid-tier cuts are statistically similar, pricing policies might be adjusted to better reflect perceived value.

In summary, ANOVA is a valuable method for testing whether group means differ when working with categorical variables that have multiple levels. By combining graphical summaries with formal inference, it supports rigorous and interpretable conclusions that inform data-driven decisions.

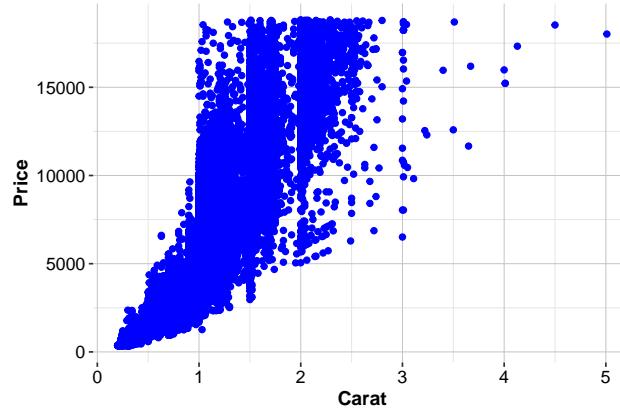
5.11 Correlation Test

Suppose you are analyzing sales data and notice that as advertising spend increases, so do product sales. Is this trend real, or just a coincidence? In exploratory data analysis (see Section 4.6), we used scatter plots and heatmaps to visually assess such relationships. Now, we take a step further. The *correlation test* offers a formal way to determine whether an observed linear association between two numerical variables is statistically significant—or merely a product of random chance.

The *correlation test* evaluates both the strength and direction of a relationship by testing the null hypothesis that the population correlation coefficient (ρ) is equal to zero. It is particularly useful when investigating how two continuous variables co-vary—insights that can inform business strategy, pricing models, or predictive analysis.

To illustrate, we examine whether a significant relationship exists between carat (diamond weight) and price in the *diamonds* dataset from the **ggplot2** package. A positive correlation is anticipated between these variables, as larger diamonds typically command higher prices. We begin with a scatter plot to visually assess the relationship:

```
ggplot(data = diamonds) +  
  geom_point(aes(x = carat, y = price), colour = "blue") +  
  labs(x = "Carat", y = "Price")
```



The scatter plot reveals a clear upward trend, suggesting that as carat increases, price tends to rise as well. However, visualization alone does not provide formal statistical evidence. To test the relationship more rigorously, we define the following hypotheses:

$$\begin{cases} H_0 : \rho = 0 & \text{(No linear correlation)} \\ H_a : \rho \neq 0 & \text{(A significant linear correlation exists)} \end{cases}$$

We conduct the test using the `cor.test()` function in **R**, which performs a Pearson correlation test and returns the correlation coefficient, *p*-value, and a confidence interval for ρ :

```
cor_test <- cor.test(diamonds$carat, diamonds$price)
cor_test

Pearson's product-moment correlation

data: diamonds$carat and diamonds$price
t = 551.41, df = 53938, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9203098 0.9228530
sample estimates:
      cor
0.9215913
```

The output includes three key results. First, the *p*-value is 0, which is smaller than the significance level ($\alpha = 0.05$). This leads us to reject the null hypothesis, concluding that the correlation between carat and price is statistically significant. Second, the correlation coefficient is 0.92, indicating a strong positive linear association. Finally, the 95% confidence interval for the true population correlation is 0.92, 0.92, which does not include zero—further reinforcing the statistical significance of the relationship.

This result has both statistical and practical implications. From a business perspective, the strong correlation between carat and price suggests that carat is a key determinant in diamond pricing. However, while statistically significant, the correlation alone may not capture all relevant factors. For instance, variation in price at similar carat levels may be explained by other attributes such as clarity, cut, or certification. Further analysis, such as multivariate regression, can help disentangle these effects.

The correlation test provides a rigorous framework for assessing linear relationships between numerical variables. By integrating visual exploration, hypothesis testing, and confidence estimation, it supports evidence-based modeling and business decision-making.

As we have seen across different types of hypothesis tests—including tests for means, proportions, and correlations—statistical inference provides a structured way to move from observed data to generalizable conclusions. In the next section, we take a step back to view these tools in a broader context: understanding how they support the goals of data science and guide decision-making under uncertainty.

5.12 From Inference to Prediction in Data Science

You have identified a statistically significant relationship between churn and service calls. But will this insight help predict which customers are likely to churn next month? This question captures a key transition in the data science workflow: moving from understanding relationships to predicting outcomes.

While the core principles of estimation, confidence intervals, and hypothesis testing remain foundational, their role evolves as we shift from traditional statistical inference to building predictive models. In classical statistics, the goal is to draw conclusions about populations based on sample data. In contrast, data science emphasizes predictive performance and generalization to new, unseen data.

In large datasets, even small effects can achieve statistical significance. However, statistical significance does not always imply practical importance. For instance, discovering that churners make 0.1 fewer calls on average may be statistically significant but offer little value for decision-making. When building models, the focus is less on whether a variable is significant in isolation and more on whether it contributes meaningfully to prediction accuracy.

Traditional inference begins with a specific hypothesis—such as whether a promotion improves sales. Data science often starts from exploration. Ana-

lists examine many features, iterate on transformations, and refine models based on validation performance. The emphasis shifts from confirming hypotheses to uncovering patterns that support generalization.

Even so, inference remains a valuable tool in the modeling process, particularly during data preparation and diagnostics. When splitting a dataset into training and test sets, for example, t-tests or chi-square tests can verify that the subsets are comparable, helping ensure that model evaluation is fair and unbiased. In feature selection, inference-based reasoning helps identify variables that show meaningful associations with the outcome. Later, in the modeling phase, residual analysis and other diagnostic checks are grounded in statistical concepts to detect overfitting or systematic bias.

Understanding how the role of inference shifts in predictive contexts allows us to apply these methods more effectively. In the next chapter, we begin building models. The principles introduced here—rigorous thinking about uncertainty, structure, and significance—remain essential for developing models that are robust, interpretable, and grounded in evidence.

5.13 Chapter Summary and Takeaways

This chapter equipped you with the essential tools of statistical inference. You learned how to use point estimates and confidence intervals to quantify uncertainty and how to apply hypothesis testing to evaluate evidence for or against specific claims about populations.

We applied a range of hypothesis tests through real-world examples: t-tests for comparing group means, proportion tests for binary outcomes, ANOVA for examining differences across multiple groups, the Chi-square test for assessing associations between categorical variables, and correlation tests for measuring linear relationships between numerical variables.

Together, these methods form a framework for drawing rigorous, data-driven conclusions. In the context of data science, they support not only analysis but also model diagnostics, variable selection, and interpretability. While p -values help assess statistical significance, they should be considered alongside effect size, assumptions, and domain relevance to ensure results are meaningful and actionable.

Statistical inference remains a vital part of the modeling pipeline—guiding how we validate data partitions, assess residual patterns, and select features that contribute to robust and explainable models. For readers seeking a deeper dive into statistical inference beyond its role in data science workflows, we recommend *Intuitive Introductory Statistics* by Wolfe and Schneider (Wolfe and Schneider 2017).

In the next chapter, we transition from inference to modeling—beginning with one of the most critical steps in any supervised learning task: dividing data into training and test sets. This step ensures that model evaluation is fair, transparent, and reliable, setting the stage for building predictive systems that generalize to new data.

5.14 Exercises

This set of exercises is designed to help you consolidate and apply what you have learned about statistical inference. They are organized into three parts: conceptual questions to deepen your theoretical grasp, hands-on tasks to practice applying inference methods in R, and reflection prompts to encourage thoughtful integration of statistical thinking into your broader data science workflow.

Conceptual Questions

1. Why is hypothesis testing important in data science? Explain its role in making data-driven decisions and how it complements exploratory data analysis.
2. What is the difference between a confidence interval and a hypothesis test? How do they provide different ways of drawing conclusions about population parameters?
3. The p -value represents the probability of observing the sample data, or something more extreme, assuming the null hypothesis is true. How should p -values be interpreted, and why is a p -value of 0.001 in a two-sample t-test not necessarily evidence of practical significance?
4. Explain the concepts of *Type I* and *Type II* errors in hypothesis testing. Why is it important to balance the risks of these errors when designing statistical tests?
5. In a hypothesis test, failing to reject the null hypothesis does not imply that the null hypothesis is true. Explain why this is the case and discuss the implications of this result in practice.
6. When working with small sample sizes, why is the t-distribution used instead of the normal distribution? How does the shape of the t-distribution change as the sample size increases?

7. One-tailed and two-tailed hypothesis tests serve different purposes. When would a one-tailed test be more appropriate than a two-tailed test? Provide an example where each type of test would be applicable.
8. Both the two-sample Z-test and the Chi-square test analyze categorical data but serve different purposes. How do they differ, and when would one be preferred over the other?
9. The *Analysis of Variance* (ANOVA) test is designed to compare means across multiple groups. Why can't multiple t-tests be used instead? What is the advantage of using ANOVA in this context?

Hands-On Practice: Hypothesis Testing in R

For the following exercises, use the *churn*, *bank*, *marketing*, and *diamonds* datasets available in the **liver** and **ggplot2** packages. We have previously used the *churn*, *bank*, and *diamonds* datasets in this and earlier chapters. In Chapter 10, we will introduce the *marketing* dataset for regression analysis.

To load the datasets, use the following commands:

```
library(liver)

library(ggplot2)

# To import the datasets
data(churn)
data(bank)
data(marketing, package = "liver")
data(diamonds)
```

10. We are interested in knowing the 90% confidence interval for the population mean of the variable “night.calls” in the *churn* dataset. In R, we can obtain a confidence interval for the population mean using the `t.test()` function as follows:

```
t.test(x = churn$night.calls, conf.level = 0.90)$"conf.int"
[1] 99.45484 100.38356
attr(,"conf.level")
[1] 0.9
```

Interpret the confidence interval in the context of customer service calls made at night. Report the 99% confidence interval for the population mean of “night.calls” and compare it with the 90% confidence interval. Which interval is wider, and what does this indicate about the precision of the esti-

mates? Why does increasing the confidence level result in a wider interval, and how does this impact decision-making in a business context?

11. Subgroup analyses help identify behavioral patterns in specific customer segments. In the *churn* dataset, we focus on customers with both an *International Plan* and a *Voice Mail Plan* who make more than 220 daytime minutes of calls. To create this subset, we use:

```
sub_churn = subset(churn, (intl.plan == "yes") & (voice.plan == "yes") &
                   (day.mins > 220))
```

Next, we compute the 95% confidence interval for the proportion of churners in this subset using `prop.test()`:

```
prop.test(table(sub_churn$churn), conf.level = 0.95)$"conf.int"
[1] 0.2595701 0.5911490
attr("conf.level")
[1] 0.95
```

Compare this confidence interval with the overall churn rate in the dataset (see Section 5.3). What insights can be drawn about this customer segment, and how might they inform retention strategies?

12. In the *churn* dataset, we test whether the mean number of customer service calls (`customer.calls`) is greater than 1.5 at a significance level of 0.01. The right-tailed test is formulated as:

$$\begin{cases} H_0 : \mu \leq 1.5 \\ H_a : \mu > 1.5 \end{cases}$$

Since the level of significance is $\alpha = 0.01$, the confidence level is $1 - \alpha = 0.99$. We perform the test using:

```
t.test(x = churn$customer.calls,
       mu = 1.5,
       alternative = "greater",
       conf.level = 0.99)

One Sample t-test

data: churn$customer.calls
t = 3.8106, df = 4999, p-value = 7.015e-05
alternative hypothesis: true mean is greater than 1.5
99 percent confidence interval:
 1.527407      Inf
sample estimates:
```

```
mean of x
1.5704
```

Report the p -value and determine whether to reject the null hypothesis at $\alpha = 0.01$. Explain your decision and discuss its implications in the context of customer service interactions.

13. In the *churn* dataset, we test whether the proportion of churners (π) is less than 0.14 at a significance level of $\alpha = 0.01$. The confidence level is 99%, corresponding to $1 - \alpha = 0.99$. The test is conducted in R using:

```
prop.test(table(churn$churn),
  p = 0.14,
  alternative = "less",
  conf.level = 0.99)

 1-sample proportions test with continuity correction

data: table(churn$churn), null probability 0.14
X-squared = 0.070183, df = 1, p-value = 0.6045
alternative hypothesis: true p is less than 0.14
99 percent confidence interval:
 0.0000000 0.1533547
sample estimates:
  p
0.1414
```

State the null and alternative hypotheses. Report the p -value and determine whether to reject the null hypothesis at $\alpha = 0.01$. Explain your conclusion and its potential impact on customer retention strategies.

14. In the *churn* dataset, we examine whether the number of customer service calls (*customer.calls*) differs between churners and non-churners. To test this, we perform a two-sample t-test:

```
t.test(customer.calls ~ churn, data = churn)

  Welch Two Sample t-test

data: customer.calls by churn
t = 11.292, df = 804.21, p-value < 2.2e-16
alternative hypothesis: true difference in means between group yes and
↪ group no is not equal to 0
95 percent confidence interval:
 0.6583525 0.9353976
sample estimates:
mean in group yes mean in group no
 2.254597      1.457722
```

State the null and alternative hypotheses. Determine whether to reject the null hypothesis at a significance level of $\alpha = 0.05$. Report the p -value and interpret the results, explaining whether there is evidence of a relationship between churn status and customer service call frequency.

15. In the *marketing* dataset, we test whether there is a *positive* relationship between revenue and spend at a significance level of $\alpha = 0.025$. We perform a one-tailed correlation test using:

```
cor.test(x = marketing$spend,
          y = marketing$revenue,
          alternative = "greater",
          conf.level = 0.975)

Pearson's product-moment correlation

data: marketing$spend and marketing$revenue
t = 7.9284, df = 38, p-value = 7.075e-10
alternative hypothesis: true correlation is greater than 0
97.5 percent confidence interval:
 0.6338152 1.0000000
sample estimates:
cor
0.789455
```

State the null and alternative hypotheses. Report the p -value and determine whether to reject the null hypothesis. Explain your decision and discuss its implications for understanding the relationship between marketing spend and revenue.

16. In the *churn* dataset, for the variable “*day.mins*”, test whether the mean number of “Day Minutes” is greater than 180. Set the level of significance to be 0.05.
17. In the *churn* dataset, for the variable “*intl.plan*” test at $\alpha = 0.05$ whether the proportion of customers who have international plan is less than 0.15.
18. In the *churn* dataset, test whether there is a relationship between the target variable “*churn*” and the variable “*intl.charge*” with $\alpha = 0.05$.
19. In the *bank* dataset, test whether there is a relationship between the target variable “*deposit*” and the variable “*education*” with $\alpha = 0.05$.
20. Compute the proportion of customers in the *churn* dataset who have an International Plan (*intl.plan*). Construct a 95% confidence interval for this proportion using R, and interpret the confidence interval in the context of customer subscriptions.

21. Using the *churn* dataset, test whether the average number of daytime minutes (`day.mins`) for churners differs significantly from 200 minutes. Conduct a one-sample t-test in **R** and interpret the results in relation to customer behavior.
22. Compare the average number of international calls (`intl.calls`) between churners and non-churners. Perform a two-sample t-test and evaluate whether the observed differences in means are statistically significant.
23. Test whether the proportion of customers with a Voice Mail Plan (`voice.plan`) differs between churners and non-churners. Use a two-sample Z-test in **R** and interpret the results, considering the implications for customer retention strategies.
24. Investigate whether marital status (`marital`) is associated with deposit subscription (`deposit`) in the *bank* dataset. Construct a contingency table and perform a Chi-square test to assess whether marital status has a significant impact on deposit purchasing behavior.
25. Using the *diamonds* dataset, test whether the mean price of diamonds differs across different diamond cuts (`cut`). Conduct an ANOVA test and interpret the results. If the test finds significant differences, discuss how post-hoc tests could be used to further explore the findings.
26. Assess the correlation between `carat` and `price` in the *diamonds* dataset. Perform a correlation test in **R** and visualize the relationship using a scatter plot. Interpret the results in the context of diamond pricing.
27. Construct a 95% confidence interval for the mean number of customer service calls (`customer.calls`) among churners. Explain how the confidence interval helps quantify uncertainty and how it might inform business decisions regarding customer support.
28. Take a random sample of 100 observations from the *churn* dataset and test whether the average `eve.mins` differs from 200. Repeat the test using a sample of 1000 observations. Compare the results and discuss how sample size affects hypothesis testing and statistical power.
29. Suppose a hypothesis test indicates that customers with a Voice Mail Plan are significantly less likely to churn ($p < 0.01$). What are some potential business strategies a company could implement based on this finding? Beyond statistical significance, what additional factors should be considered before making marketing decisions?

Reflection

30. How do confidence intervals and hypothesis tests complement each other when assessing the reliability of results in data science?
31. In your work or studies, can you think of a situation where failing to reject the null hypothesis was an important finding? What did it help clarify?
32. Describe a time when statistical significance and practical significance diverged in a real-world example. What lesson did you learn?
33. How might understanding Type I and Type II errors influence how you interpret results from automated reports, dashboards, or A/B tests?
34. When designing a data analysis for your own project, how would you decide which statistical test to use? What questions would guide your choice?
35. How can confidence intervals help communicate uncertainty to non-technical stakeholders? Can you think of a better way to present this information visually?
36. Which statistical test from this chapter do you feel most comfortable with, and which would you like to practice more? Why?

Chapter 6

Setting Up Data for Modeling

Suppose a churn prediction model reports 95% accuracy, yet consistently fails to identify customers who actually churn. What went wrong? In many cases, the issue lies not in the algorithm itself but in how the data was prepared for modeling. Before reliable machine learning models can be built, the dataset must be not only clean but also properly structured to support learning, validation, and generalization.

This chapter focuses on the final preparatory step in the Data Science Workflow introduced in Figure 2.3: *Step 4: Setting Up Data for Modeling*. This step involves structuring the dataset in a way that enables fair training, reliable testing, and robust model evaluation.

To accomplish this, we complete three essential tasks:

1. *Partitioning*: Splitting the dataset into training and testing subsets.
2. *Validating*: Verifying that the subsets are representative of the original data.
3. *Balancing*: Addressing class imbalance when one class dominates in classification tasks.

The work in previous chapters lays the foundation for this step. In Section 2.4, you defined the modeling objective. In Chapter 3, you cleaned the data and transformed key features. Chapter 4 guided your exploratory analysis, while Chapter 5 introduced tools to test whether datasets are statistically comparable.

Now, we move to the *setup phase*, a critical, yet often overlooked step. It ensures that the data is not only clean but also statistically sound and properly balanced for modeling. These preparations help prevent common issues such as overfitting, biased evaluation, and data leakage.

This process, particularly for newcomers, raises important questions, such as, *Why is it necessary to partition the data?*, *How can we verify that training and test sets are truly comparable?*, and *What can we do if one class is severely underrepresented?*

These are not just technical details. They reflect essential principles in modern data science: *fairness*, *reproducibility*, and *trust*. By walking through partitioning, validating, and balancing, we lay the groundwork for building models that perform well, and do so credibly, in real-world settings.

What This Chapter Covers

This chapter completes *Step 4 of the Data Science Workflow, Setting Up Data for Modeling*. You will learn how to:

- *Partition* a dataset into training and testing subsets to simulate deployment scenarios.
- *Validate* that your split is statistically representative and free from data leakage.
- *Address class imbalance* using oversampling, undersampling, or class weighting techniques.

By mastering these tasks, you ensure that your data is not only clean but also structured for training machine learning models that are robust, fair, and generalizable.

6.1 Why Is It Necessary to Partition the Data?

The first step in preparing data for supervised learning is partitioning the dataset into training and testing subsets, a practice often misunderstood by newcomers to data science. A common question is: *Why split the data before modeling?* The answer lies in *generalization*, the model’s ability to make accurate predictions on new, unseen data. This section explains why this step is essential for building models that not only perform well during training but also hold up in real-world applications.

As part of Step 4 in the Data Science Workflow, partitioning precedes validation and class balancing. Dividing the data into a *training set* for model development and a *test set* for evaluation simulates real-world deployment.

This practice guards against two key modeling pitfalls: *overfitting* and *underfitting*. Their trade-off is illustrated in Figure 6.1.

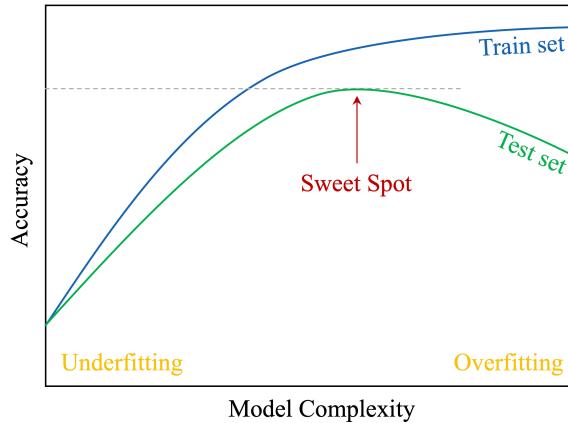


Figure 6.1: The trade-off between model complexity and accuracy on the training and test sets. Optimal performance is achieved at the point where test set accuracy is highest, before overfitting begins to dominate.

Overfitting occurs when a model captures noise and specific patterns in the training data rather than general trends. Such models perform well on training data but poorly on new observations. For instance, a churn model might rely on customer IDs rather than behavior, resulting in poor generalization.

Underfitting arises when the model is too simplistic to capture meaningful structure, often due to limited complexity or overly aggressive preprocessing. An underfitted model may assign nearly identical predictions across all customers, failing to reflect relevant differences.

Evaluating performance on a separate test set helps detect both issues. A large gap between high training accuracy and low test accuracy suggests overfitting, while low accuracy on both may indicate underfitting. In either case, model adjustments are needed to improve generalization.

Another critical reason for partitioning is to prevent *data leakage*, the inadvertent use of information from the test set during training. Leakage can produce overly optimistic performance estimates and undermine trust in the model. Strict separation of the training and test sets ensures that evaluation reflects a model's true predictive capability on unseen data.

Figure 6.2 summarizes the typical modeling process in supervised learning:

1. *Partition* the dataset and validate the split.

2. *Train* models on the training data.
3. *Evaluate* model performance on the test data.

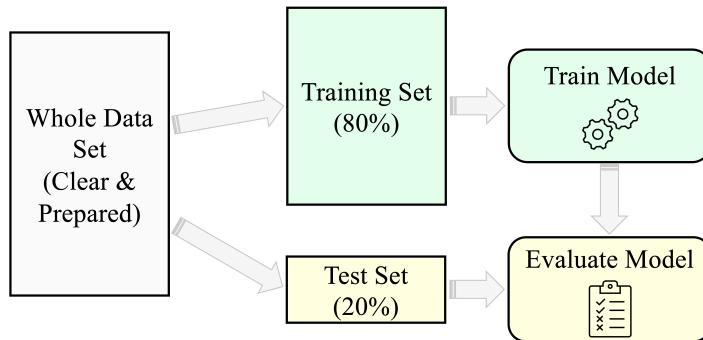


Figure 6.2: A general supervised learning process for building and evaluating predictive models. The 80–20 split ratio is a common default but may be adjusted based on the problem and dataset size.

By following this structure, we develop models that are both accurate and reliable. The remainder of this chapter addresses how to carry out each step in practice, beginning with partitioning strategies, followed by validation techniques and class balancing methods.

6.2 Partitioning Data: The Train–Test Split

Having established why partitioning is essential, we now turn to how it is implemented in practice. The most common method is the *train–test split*, also known as the *holdout method*. In this approach, the dataset is divided into two subsets: a *training set* used to develop the model and a *test set* reserved for evaluating the model’s ability to generalize to new, unseen data. This separation is essential for assessing out-of-sample performance.

Typical split ratios include 70–30, 80–20, or 90–10, depending on the dataset’s size and the modeling objectives. Both subsets include the same predictor variables and the outcome of interest, but only the training set’s outcome values are used during model fitting. The test set remains untouched during training to avoid data leakage and provides a realistic benchmark for evaluating the model’s predictive performance.

Example: Train–Test Split in R

We illustrate the train–test split using R and the `liver` package. We return to the `churn` dataset introduced in Chapter 4.3, where the goal is to predict customer churn using machine learning models (discussed in the next chapter). First, we load the data:

```
library(liver)  
data(churn)
```

The `partition()` function in the `liver` package provides a straightforward method to split a dataset based on a specified ratio. Below, we divide the dataset into 80% training and 20% test data:

```
set.seed(42)  
  
data_sets = partition(data = churn, ratio = c(0.8, 0.2))  
  
train_set = data_sets$part1  
test_set = data_sets$part2  
  
test_labels = test_set$churn
```

The use of `set.seed(42)` ensures *reproducibility*, meaning the same split will occur each time the code is run—a vital practice for ensuring reproducibility in model development and evaluation. The `test_labels` vector stores the actual target values from the test set and is used for evaluating model predictions. These labels must remain hidden during model training to avoid data leakage.

Splitting data into training and test sets allows us to assess a model’s generalization performance—that is, how well it predicts new, unseen data. While the train–test split is widely used, it can yield variable results depending on how the data is divided. A more robust and reliable alternative is *cross-validation*, introduced in the next section.

6.3 Cross-Validation for Robust Performance Estimation

While the train–test split is widely used for its simplicity, the resulting performance estimates can vary substantially depending on how the data is divided—especially when working with smaller datasets. To obtain more stable and reliable estimates of a model’s generalization performance, *cross-validation* provides a valuable alternative.

Cross-validation is a resampling method that offers a more comprehensive evaluation than a single train–test split. In k -fold cross-validation, the dataset is randomly partitioned into k non-overlapping subsets (folds) of approximately equal size. The model is trained on $k-1$ folds and evaluated on the remaining fold. This process is repeated k times, with each fold serving once as the validation set. The overall performance is then estimated by averaging the metrics across all k iterations. Common choices for k include 5 or 10, as illustrated in Figure 6.3.

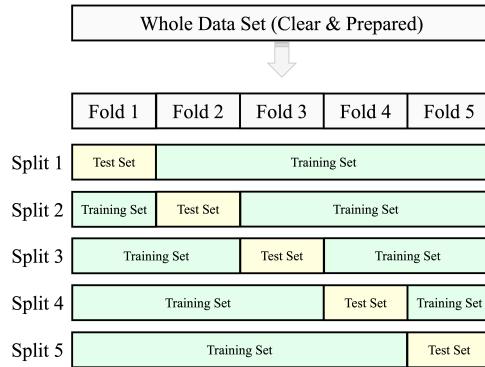


Figure 6.3: Illustration of k -fold cross-validation. The dataset is randomly split into k non-overlapping folds ($k = 5$ shown). In each iteration, the model is trained on $k-1$ folds (shown in green) and evaluated on the remaining fold (shown in yellow).

Cross-validation is especially useful for comparing models or tuning hyperparameters. However, repeated use of the test set during model development can lead to *information leakage*, resulting in overly optimistic performance estimates. To avoid this, it is best practice to hold out a separate *test set* for final evaluation, using cross-validation exclusively within the *training set*. In this setup, model selection and tuning rely on the cross-validated results from the training data, while the final model is evaluated once on the untouched test set.

This approach is depicted in Figure 6.4. It eliminates the need for a fixed validation subset and makes more efficient use of the training data, while preserving an unbiased test set for final performance reporting.

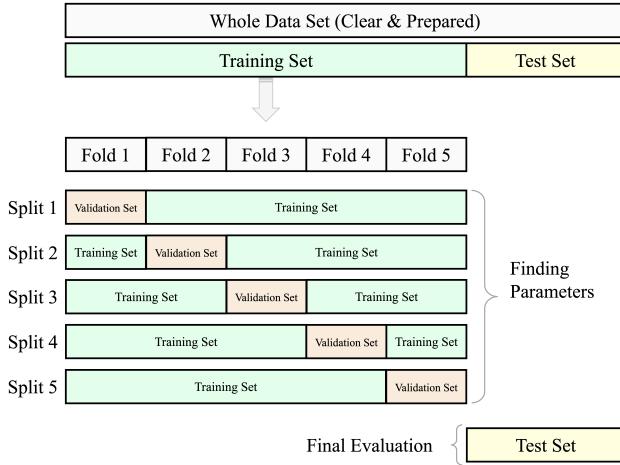


Figure 6.4: Cross-validation applied within the training set. The test set is held out for final evaluation only. This strategy eliminates the need for a separate validation set and maximizes the use of available data for both training and validation.

Although more computationally intensive, k-fold cross-validation helps reduce the variance of performance estimates and is particularly advantageous when data is limited. It ensures that evaluation reflects a model's ability to generalize, rather than its performance on a specific data split. For further details and implementation examples, see Chapter 5 of *An Introduction to Statistical Learning* (James et al. 2013).

Partitioning data is a foundational step in predictive modeling. Yet even with a carefully designed split, it is important to verify whether the resulting subsets are representative of the original data. The next section addresses how to validate the quality of the partition before training begins.

6.4 How to Split Data for Modeling

Now that we understand *why* partitioning matters, let us see *how* to do it in practice. Splitting the dataset is a key step in preparing for machine learning, and the most common approach is the *train-test split*, also called the holdout method. In this method, you divide the data into two parts: a *training set* to build the model and a *testing set* to evaluate how well it performs on new, unseen cases. This separation ensures a fair and realistic assessment of the model's ability to generalize.

Common split ratios include 70-30, 80-20, or 90-10, depending on the dataset size and modeling goals. Both sets contain the same structure—predictor features and the target outcome—but during training, the model only sees the target values from the training set. After training, the model predicts outcomes for the test set, and you compare these predictions to the actual values to measure accuracy and robustness.

Example: Train-Test Split in R

To demonstrate a typical train-test split, we use **R** and the **liver** package. We will revisit the *churn* dataset from Section 4.3, where the goal is to predict customer churn using a machine learning algorithm (introduced in the next chapter). Since the dataset is bundled with the **liver** package, we begin by loading it:

```
library(liver)  
data(churn)
```

There are several ways to partition a dataset in **R**. An easy and efficient approach is to use the `partition()` function from the **liver** package, which randomly splits a dataset according to a specified ratio. The example below divides the *churn* dataset into an 80% training set and a 20% test set:

```
set.seed(42)  
  
data_sets = partition(data = churn, ratio = c(0.8, 0.2))  
  
train_set = data_sets$part1  
test_set = data_sets$part2  
  
test_labels = test_set$churn
```

The `set.seed(42)` command ensures *reproducibility*, producing the same random split across sessions—a vital practice for consistent model evaluation and collaboration. The `partition()` function randomly assigns 80% of the data to `train_set`, which is used for model training, and 20% to `test_set`, reserved for performance evaluation. The `ratio` argument controls the split and can be adjusted as needed. We also extract `test_labels`—the actual class labels in the test set—for use in model evaluation. These labels must remain hidden during training to avoid data leakage.

Reproducibility is a cornerstone of reliable modeling. Setting a random seed ensures consistent results across runs and supports transparency in code sharing, debugging, and publication.

A Note on Cross-Validation

While the train-test split is simple and widely used, it can yield results that are sensitive to how the data is divided—particularly in small datasets. To obtain a more stable and reliable estimate of model performance, *cross-validation* offers a valuable alternative.

In k -fold cross-validation, the dataset is divided into k equally sized parts, or folds. The model is trained on $k-1$ folds and evaluated on the remaining fold. This process is repeated k times, with each fold serving once as the test set. Performance metrics are then averaged across the k runs, reducing variability and offering a more robust assessment of generalization.

Cross-validation is especially useful when comparing models or tuning hyperparameters, where a single train-test split may not provide enough confidence. Although not used in this chapter’s case study, it plays a central role in many advanced modeling workflows. For further explanation and implementation examples, see Chapter 5 of *An Introduction to Statistical Learning* (James et al. 2013).

Partitioning is the starting point for building trustworthy machine learning models. However, a single split does not ensure that the training and test sets are representative of the full dataset. In the next section, we will evaluate whether our partition is statistically sound, critical check before training predictive models.

6.5 How to Validate a Train-Test Split

How can we be sure that our train-test split truly represents the original dataset? After splitting the data, we must validate that the partition is statistically sound. A reliable split ensures that the training set reflects the broader population and that the test set mimics real-world deployment. Without this step, we risk building models that learn from biased data or fail to generalize.

Validation involves comparing the distributions of key variables—especially the target and influential predictors—across the training and testing sets. Since most datasets include many features, we usually focus on a subset that plays a central role in modeling. The statistical test we choose depends on the type of variable, as summarized in Table 6.1.

Table 6.1: Suggested hypothesis tests for validating partitions, based on the type of target feature.

| Type of Features | Suggested Test (from Chapter 5) |
|---|---------------------------------|
| Numerical feature | Two-sample t-test |
| Binary / Flag feature | Two-sample Z-test |
| Categorical feature (with > 2 categories) | Chi-square test |

Each test has specific assumptions. Parametric tests like the t-test and Z-test are most appropriate when sample sizes are large and distributions are approximately normal. For categorical features with more than two levels, the Chi-square test is the standard choice.

Let us illustrate this with the *churn* dataset by checking whether the proportion of churners is consistent across the training and testing sets. The target variable, *churn* (indicating whether a customer has churned), is binary. To determine whether the training and testing sets have similar churn rates, we conduct a two-sample Z-test. Thus, the hypotheses are defined as follows:

$$\begin{cases} H_0 : \pi_{\text{churn, train}} = \pi_{\text{churn, test}} \\ H_a : \pi_{\text{churn, train}} \neq \pi_{\text{churn, test}} \end{cases}$$

The R code below performs the test:

```
x1 <- sum(train_set$churn == "yes")
x2 <- sum(test_set$churn == "yes")

n1 <- nrow(train_set)
n2 <- nrow(test_set)

test_churn <- prop.test(x = c(x1, x2), n = c(n1, n2))
test_churn

 2-sample test for equality of proportions with continuity correction

data: c(x1, x2) out of c(n1, n2)
X-squared = 0.098945, df = 1, p-value = 0.7531
alternative hypothesis: two.sided
95 percent confidence interval:
-0.02946053 0.02046053
sample estimates:
prop 1 prop 2
0.1405 0.1450
```

Here, x_1 and x_2 denote the number of churners in the training and testing sets, respectively; n_1 and n_2 are the corresponding sample sizes. The func-

tion `prop.test()` performs the two-sample Z-test and returns a p -value indicating whether the difference in proportions is statistically significant.

Suppose the p -value is 0.75. Since this value exceeds the typical significance threshold ($\alpha = 0.05$), we fail to reject H_0 . This suggests that the observed difference in churn rates is not statistically significant, and the split appears valid with respect to the target variable.

Beyond the target variable, checking the distribution of key predictors helps detect imbalances that could bias the model. Unequal distributions in important features can lead the model to learn misleading or unrepresentative patterns. For instance, apply a *two-sample t-test* to compare means for numerical predictors such as `customer.calls` or `day.mins`, and use a *Chi-square test* for categorical variables like `area.code`. If `day.mins` is notably higher in the test set, a model trained on lower values may underpredict in deployment. Although it is rarely feasible to check every variable in high-dimensional datasets, focusing on known or selected predictors helps ensure a balanced and representative partition.

What If the Partition Is Invalid?

What should you do if the training and testing sets turn out to be significantly different? If validation reveals statistical imbalances, it is essential to take corrective steps to ensure that both subsets more accurately reflect the original dataset:

- *Revisit the random split:* Even a random partition can result in imbalance due to chance. Try adjusting the random seed or modifying the split ratio to improve representativeness.
- *Use stratified sampling:* This approach preserves the proportions of key categorical features—especially the target variable—across both training and test sets.
- *Apply cross-validation:* Particularly valuable for small or imbalanced datasets, cross-validation reduces reliance on a single split and yields more stable performance estimates.

Even with careful attention, some imbalance may persist, especially in small or high-dimensional datasets. In such cases, additional techniques like bootstrapping or repeated sampling can improve stability and provide more reliable evaluations.

Remember, validation is more than a procedural checkpoint—it is a safeguard for the integrity of your modeling workflow. By ensuring that the

training and test sets are representative, you enable models that learn honestly, perform reliably, and yield trustworthy insights. In the next section, we tackle another common issue: imbalanced classes in the training set.

6.6 Dealing with Class Imbalance

Imagine training a fraud detection model that labels every transaction as legitimate. It might boast 99% accuracy—yet fail completely at catching fraud. This scenario highlights the risk of *class imbalance*, where one class dominates the dataset and overshadows the rare but critical outcomes we aim to detect.

In many real-world classification tasks, one class is far less common than the other—a challenge known as *class imbalance*. This can lead to models that perform well on paper—often reporting high overall accuracy—while failing to identify the minority class. For example, in fraud detection, fraudulent cases are rare, and in churn prediction, most customers stay. If the model always predicts the majority class, it may appear accurate but will miss the cases that matter most.

Most machine learning algorithms optimize for *overall accuracy*, which can be misleading when the rare class is the true focus. A churn model trained on imbalanced data might predict nearly every customer as a non-churner, yielding high accuracy but missing actual churners—the very cases we care about. Addressing class imbalance is therefore an important step in setting up data for modeling, particularly when the minority class carries high business or scientific value.

Several strategies are commonly used to balance the training dataset and ensure that both classes are adequately represented during learning. *Oversampling* increases the number of minority class examples by duplicating existing cases or generating synthetic data. The popular SMOTE (Synthetic Minority Over-sampling Technique) method creates realistic synthetic examples instead of simple copies. *Undersampling* reduces the number of majority class examples by randomly removing observations and is useful when the dataset is large and contains redundant examples. *Hybrid methods* combine both approaches to achieve a balanced representation. Another powerful technique is *class weighting*, which adjusts the algorithm to penalize misclassification of the minority class more heavily. Many models—including logistic regression, decision trees, and support vector machines—support this approach natively.

These techniques must be applied *only to the training set* to avoid data leakage. The best choice depends on factors such as dataset size, the degree of imbalance, and the algorithm being used.

Let us walk through a concrete example using the *churn* dataset. The goal is to predict whether a customer has churned. First, we examine the distribution of the target variable in the training dataset:

```
# Check the class distribution
table(train_set$churn)

yes   no
562 3438

prop.table(table(train_set$churn))

yes     no
0.1405 0.8595
```

Suppose the output shows that churners (`churn = "yes"`) make up only a small proportion—say, around 0.14, compared to non-churners. This imbalance may lead to a model that overlooks the very class we aim to predict.

To address this in **R**, we can use the `ovun.sample()` function from the **ROSE** package to oversample the minority class so that it makes up 30% of the training set. This target ratio is illustrative; the optimal value depends on the use case and modeling goals.

If the **ROSE** package is not yet installed, use `install.packages("ROSE")`.

```
# Load the ROSE package
library(ROSE)

# Oversample the training set to balance the classes with 30% churners
balanced_train_set <- ovun.sample(churn ~ ., data = train_set, method =
  "over", p = 0.3)$data

# Check the new class distribution
table(balanced_train_set$churn)

no   yes
3438 1446

prop.table(table(balanced_train_set$churn))

no      yes
0.7039312 0.2960688
```

The `ovun.sample()` function generates a new training set in which the minority class is oversampled to represent 30% of the data. The formula `churn ~ .` tells **R** to balance based on the target variable while keeping all predictors.

Always apply balancing after the data has been partitioned and *only* to the training set. Modifying the test set would introduce bias and make the

model's performance appear artificially better than it would be in deployment. This safeguard prevents *data leakage* and ensures honest evaluation.

Balancing is not always necessary. Many modern algorithms incorporate internal strategies for handling class imbalance, such as class weighting or ensemble techniques. These adjust the model to account for rare events without requiring explicit data manipulation. Furthermore, rather than relying solely on overall accuracy, evaluation metrics such as *precision*, *recall*, *F1-score*, and *AUC-ROC* offer more meaningful insights into model performance on imbalanced data. We will explore these evaluation metrics in more depth in Chapter 8, where we assess model performance under class imbalance.

In summary, dealing with class imbalance helps the model focus on the right outcomes and make more equitable predictions. It is a crucial preparatory step in classification workflows, particularly when the minority class holds the greatest value.

6.7 Chapter Summary and Takeaways

This chapter finalized *Step 4: Data Setup for Modeling* in the Data Science Workflow by preparing the dataset for valid and generalizable model development.

- The partitioning of data into training and testing sets was established as a safeguard against overfitting and a way to simulate real-world prediction tasks.
- The validation of train-test splits ensured that both subsets were statistically representative, supporting reliable model evaluation.
- The handling of class imbalance through methods such as oversampling, undersampling, and class weighting improved model sensitivity to underrepresented outcomes.

Some modeling algorithms, such as k-Nearest Neighbors introduced in the next chapter, require additional preprocessing steps like *feature rescaling*. These techniques were presented earlier in Chapter 3 and will be applied when appropriate in the modeling chapters that follow.

Unlike other chapters in this book, this chapter does not include a dedicated case study. This is because the procedures introduced here—partitioning, validating, and balancing—are integrated into the case studies in the remainder of the book. For example, the churn classification example in Section 7.7 illustrates how these steps are applied in practice.

Together, these preparatory steps mitigate common risks such as biased evaluation and data leakage, providing a sound foundation for predictive modeling. The next chapter builds on this foundation by introducing and evaluating classification models, beginning with logistic regression.

6.8 Exercises

This section includes both *conceptual questions* and *applied programming exercises* that reinforce key ideas from the chapter. The goal is to consolidate essential preparatory steps for predictive modeling, with a focus on partitioning, validating, and, when necessary, balancing datasets to support fair and generalizable learning.

Conceptual Questions

1. Why is partitioning the dataset crucial before training a machine learning model? Explain its role in ensuring generalization.
2. What is the main risk of training a model without separating the dataset into training and testing subsets? Provide an example where this could lead to misleading results.
3. Explain the difference between *overfitting* and *underfitting*. How does proper partitioning help address these issues?
4. Describe the role of the *training set* and the *testing set* in machine learning. Why should the test set remain unseen during model training?
5. What is *data leakage*, and how can it occur during data partitioning? Provide an example of a scenario where data leakage could lead to overly optimistic model performance.
6. Compare and contrast *random partitioning* and *stratified partitioning*. When would stratified partitioning be preferred?
7. Why is it necessary to validate the partition after splitting the dataset? What could go wrong if the training and test sets are significantly different?
8. How would you test whether numerical features, such as `customer.calls` in the *churn* dataset, have similar distributions in both the training and testing sets?

9. If a dataset is highly imbalanced, why might a model trained on it fail to generalize well? Provide an example from a real-world domain where class imbalance is a serious issue.
10. Compare *oversampling*, *undersampling*, and *hybrid methods* for handling imbalanced datasets. What are the advantages and disadvantages of each?
11. Why should balancing techniques be applied *only* to the training dataset and *not* to the test dataset?
12. Some machine learning algorithms are robust to class imbalance, while others require explicit handling of imbalance. Which types of models typically require class balancing, and which can handle imbalance naturally?
13. When dealing with class imbalance, why is *accuracy* not always the best metric to evaluate model performance? Which alternative metrics should be considered?
14. Suppose a dataset has a rare but critical class (e.g., fraud detection). What steps should be taken in the *data partitioning and balancing phase* to ensure an effective model?
15. After completing this chapter, which preparatory step—partitioning, validating, or balancing—do you find most critical for building trustworthy models, and why?

Hands-On Practice

The following exercises use the *churn*, *bank*, and *risk* datasets from the **liver** package. The *churn* and *bank* datasets have been introduced previously; *risk* will be used in Chapter 9.

```
library(liver)  
  
data(churn)  
data(bank)  
data(risk)
```

Partitioning the Data

16. Partition the *churn* dataset into 75 percent training and 25 percent testing. Set a reproducible seed.
17. Perform a 90–10 split on the *bank* dataset. Report the number of observations in each subset.

18. Use stratified sampling to ensure that the churn rate is consistent across both subsets of the *churn* dataset.
19. Apply a 60–40 split on the *risk* dataset. Save the outputs as `train_risk` and `test_risk`.
20. Generate density plots to compare the distribution of income between training and test sets in the *bank* dataset.

Validating the Partition

21. Use a two-sample Z-test to assess whether the churn proportion differs significantly between training and test sets.
22. Apply a two-sample t-test to evaluate whether average age differs across subsets in the *bank* dataset.
23. Conduct a Chi-square test to assess whether the distribution of marital status differs between subsets in the *bank* dataset.
24. Suppose the churn proportion is 30 percent in training and 15 percent in testing. Identify an appropriate statistical test and explain a correction strategy.
25. Select three numerical variables in the *risk* dataset and assess whether their distributions differ between the two subsets.

Balancing the Training Dataset

26. Examine the class distribution of churn in the training set. Report the proportion of churners.
27. Apply random oversampling to increase the chunner class to 40 percent of the training data using the **ROSE** package.
28. Use undersampling to equalize the `deposit = "yes"` and `deposit = "no"` classes in the training set of the *bank* dataset.
29. Create bar plots to compare the class distribution in the *churn* dataset before and after balancing.

Self-Reflection

30. Which of the three preparation steps—partitioning, validation, balancing—currently feels most intuitive, and which would benefit from additional practice? Justify the response.
31. How does a deeper understanding of data preparation influence perceptions of model evaluation and fairness?

Chapter 7

Classification Using k-Nearest Neighbors

Classification is a foundational task in machine learning that enables algorithms to assign observations to specific categories based on patterns learned from labeled data. Whether filtering spam emails, detecting fraudulent transactions, or predicting customer churn, classification plays a vital role in many real-world decision systems. This chapter introduces classification as a form of supervised learning, emphasizing accessible and practical methods for those beginning their journey into predictive modeling.

This chapter also marks the start of *Step 5: Modeling* in the Data Science Workflow (Figure Figure 2.3). Building on earlier chapters—where we cleaned and explored data, developed statistical reasoning, and prepared datasets for modeling—we now turn to the exciting stage of applying machine learning techniques.

What This Chapter Covers

We begin by defining classification and contrasting it with regression, then introduce common applications and categories of classification algorithms. The focus then shifts to one of the most intuitive and interpretable methods: *k-Nearest Neighbors (kNN)*, a distance-based algorithm that predicts the class of a new observation by examining its closest neighbors in the training set.

To demonstrate the method in action, we apply kNN to the *churn* dataset, where the goal is to predict whether a customer will discontinue a service. The chapter walks through the full modeling workflow—data preparation, selecting an appropriate value of k , implementing the model in R, and evaluating its predictive performance—offering a step-by-step blueprint for real-world classification problems.

By the end of this chapter, readers will have a clear understanding of how classification models operate, how kNN translates similarity into prediction, and how to apply this method effectively to real-world data.

7.1 Classification

How do email applications filter spam, streaming services recommend the next show, or banks detect fraudulent transactions in real time? These intelligent systems rely on *classification*, a core task in supervised machine learning that assigns input data to one of several predefined categories.

In classification, models learn from labeled data to predict categorical outcomes. For example, given customer attributes, a model might predict whether a customer is likely to churn. This contrasts with regression, which predicts continuous quantities such as income or house price.

The target variable—often called the *class* or *label*—can take different forms. In *binary classification*, the outcome has two possible categories, such as spam versus not spam. In *multiclass classification*, the outcome includes more than two categories, such as distinguishing between a pedestrian, a car, or a bicycle in an object recognition task.

Classification underpins a wide array of applications. Email clients detect spam based on message features and sender behavior. Financial systems flag anomalous transactions to prevent fraud. Businesses use churn models to identify customers at risk of leaving. In healthcare, models assist in diagnosing diseases from clinical data. Autonomous vehicles rely on object recognition to navigate safely. Recommendation systems apply classification logic to tailor content to users.

These examples illustrate how classification enables intelligent systems to translate structured inputs into meaningful, actionable predictions. As digital data becomes more pervasive, classification remains a foundational technique for building effective and reliable predictive models.

How Classification Works

Classification typically involves two main phases:

1. *Training phase:* The model learns patterns from a labeled dataset, where each observation contains input features along with a known class label.

For example, a fraud detection system might learn that high-value transactions originating from unfamiliar locations are often fraudulent.

2. *Prediction phase:* Once trained, the model is used to classify new, unseen observations. Given the features of a new transaction, the model predicts whether it is fraudulent.

A well-performing classification model captures meaningful patterns in the data rather than simply memorizing the training set. Its value lies in the ability to generalize—that is, to make accurate predictions on new data not encountered during training. This ability to generalize is a defining characteristic of all supervised learning methods.

Classification Algorithms and the Role of kNN

A wide range of algorithms can be used for classification, each with its own strengths depending on the nature of the data and the modeling goals. Some commonly used methods include:

- *k-Nearest Neighbors (kNN):* A simple, distance-based algorithm that assigns labels based on the nearest neighbors. It is the focus of this chapter.
- *Naive Bayes:* A probabilistic method well-suited to text classification tasks such as spam detection (see Chapter 9).
- *Logistic Regression:* A widely used model for binary outcomes, known for its interpretability (see Chapter 10).
- *Decision Trees and Random Forests:* Flexible models that can capture complex, nonlinear relationships (see Chapter 11).
- *Neural Networks:* High-capacity algorithms effective for high-dimensional or unstructured data, including images and text (see Chapter 12).

Choosing an appropriate algorithm depends on several factors, including dataset size, the types of features, the need for interpretability, and computational constraints. For small to medium-sized datasets or when transparency is a priority, simpler models such as kNN or Decision Trees may be suitable. For more complex tasks involving large datasets or unstructured inputs, Neural Networks may offer better predictive performance.

To illustrate, consider the *bank* dataset, where the task is to predict whether a customer will subscribe to a term deposit (*deposit = yes*). Predictor variables such as age, education, and marital status can be used to build a classification model. Such a model can support targeted marketing by identifying customers more likely to respond positively.

Among these algorithms, *k*-Nearest Neighbors (kNN) stands out for its ease of use and intuitive decision-making process. Because it makes minimal assumptions about the underlying data, kNN is often used as a baseline model—helping to gauge how challenging a classification problem is before considering more complex approaches. In the sections that follow, we explore how the kNN algorithm works, how to implement it in R, and how to apply it to a real-world classification task using the *churn* dataset.

7.2 How k-Nearest Neighbors Works

Imagine making a decision by consulting a few trusted peers who have faced similar situations. The *k*-Nearest Neighbors (kNN) algorithm works in much the same way: it predicts outcomes based on the most similar observations from previously seen data. This intuitive, experience-based approach makes kNN one of the most accessible methods in classification.

Unlike many algorithms that involve an explicit training phase, kNN follows a *lazy learning* strategy. It stores the entire training dataset and postpones computation until a prediction is needed. When a new observation arrives, the algorithm calculates its distance from all training points, identifies the *k* closest neighbors, and assigns the most common class among them. The choice of *k*—the number of neighbors used—is crucial: small values make the model sensitive to local patterns, while larger values promote broader generalization. Because kNN defers all computation until prediction, it avoids upfront model fitting but shifts the computational burden to the prediction phase.

How Does kNN Classify a New Observation?

When classifying a new observation, the kNN algorithm first computes its *distance* to all data points in the training set, typically using the *Euclidean distance*. It then identifies the *k* nearest neighbors and assigns the most frequent class label among them as the predicted outcome.

Figure 7.1 illustrates this idea using a toy dataset with two classes: Class A (red circles) and Class B (blue squares). A new data point, shown as a *dark star*, must be assigned to one of the two classes. The classification result depends on the chosen value of *k*:

- When $k = 3$, the three closest neighbors include two blue squares and one red circle. Since the majority class is *Class B*, the new point is labeled accordingly.
- When $k = 6$, the nearest neighbors include four red circles and two blue squares, resulting in a prediction of *Class A*.

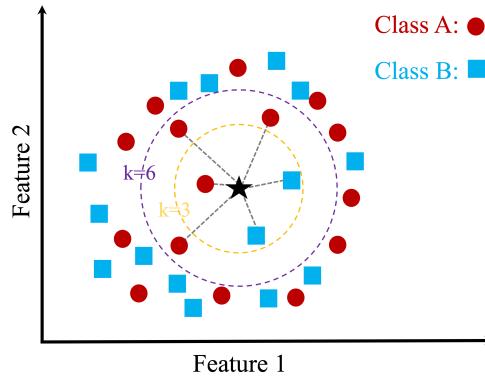


Figure 7.1: A two-dimensional toy dataset with two classes (Class A and Class B) and a new data point (dark star), illustrating the k-Nearest Neighbors algorithm with $k = 3$ and $k = 6$.

These examples demonstrate how the choice of k directly affects the classification result. A smaller k makes the model more sensitive to local variation and potentially noisy observations—leading to overfitting. In contrast, a larger k smooths the decision boundaries by incorporating more neighbors but may overlook meaningful local structure. Choosing the right value of k is therefore essential for balancing variance and bias, a topic we revisit later in this chapter.

Strengths and Limitations of kNN

The *k*-Nearest Neighbors (kNN) algorithm is valued for its simplicity and transparent decision-making process, making it a common starting point in classification tasks. It requires no explicit model training; instead, it stores the training data and performs computations only at prediction time. This approach makes kNN easy to implement and interpret—particularly effective for small datasets with well-separated class boundaries.

However, this simplicity comes with important trade-offs. The algorithm is sensitive to irrelevant or noisy features, which can distort distance calculations and degrade predictive performance. Moreover, since kNN calculates distances to all training examples at prediction time, it can become computationally expensive as the dataset grows.

Another crucial consideration is the choice of k , which directly affects model behavior. A small k may lead to overfitting and heightened sensitivity to noise, whereas a large k may oversmooth the decision boundary, obscuring meaningful patterns. As we discuss later in the chapter, selecting an appropriate value of k is key to balancing variance and bias.

Finally, the effectiveness of kNN often hinges on proper data preprocessing. Feature selection, scaling, and outlier handling all play a significant role in ensuring that distance computations reflect meaningful structure in the data—topics we address in the next sections.

7.3 kNN in Action: A Toy Example for Drug Classification

To further illustrate how kNN works in practice, consider a simplified classification scenario involving drug prescriptions. A synthetically generated dataset of 200 patients includes their *age*, *sodium-to-potassium (Na/K) ratio*, and the prescribed drug type. Although simulated, the dataset reflects patterns that one might encounter in real-world clinical data. For details on how the dataset was generated, see Section 1.20.

Figure 7.2 visualizes the distribution of patient records. Each point represents a patient, with different shapes and colors used to indicate drug type. Red circles correspond to Drug A, green triangles to Drug B, and blue squares to Drug C.

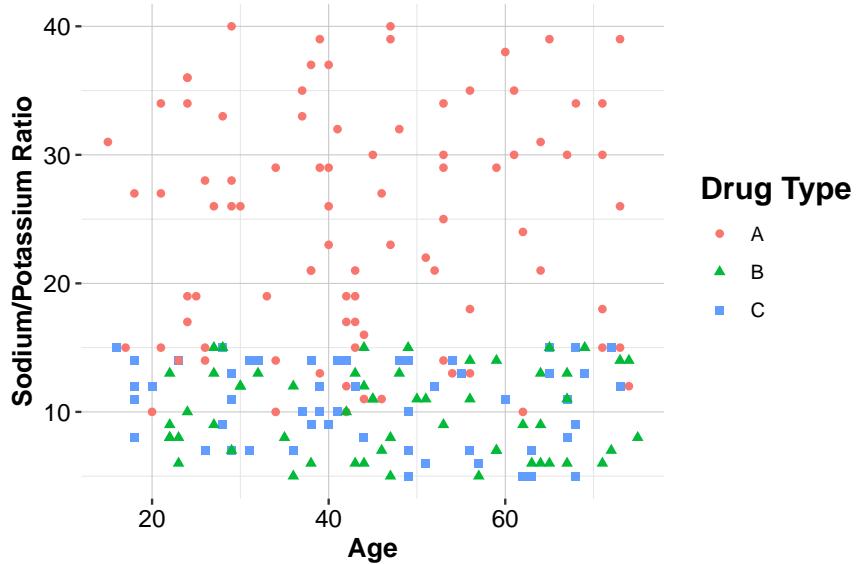


Figure 7.2: Scatter plot of Age vs. Sodium/Potassium Ratio for 200 patients, with drug type indicated by color and shape.

Suppose three new patients arrive at the clinic, and we need to determine which drug is most suitable for them based on their *age* and *sodium-to-potassium ratio*. Patient 1 is a 40-year-old with a Na/K ratio of 30.5. Patient 2 is 28 years old with a Na/K ratio of 9.6. Patient 3 is 61 years old with a Na/K ratio of 10.5.

These patients are represented by large orange circles with cross markers in Figure 7.3. The following analysis explores how kNN assigns drug classifications to these individuals based on proximity to labeled cases in the training data.

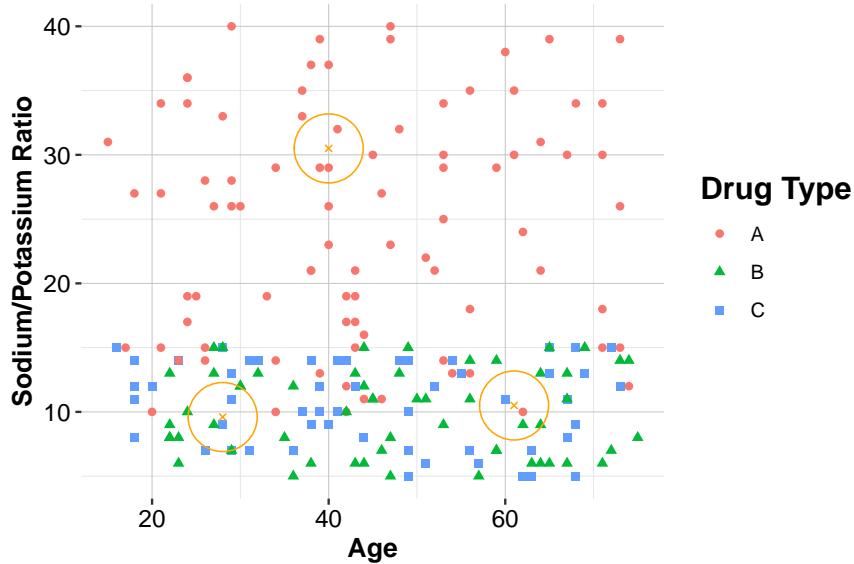


Figure 7.3: Scatter plot of Age vs. Sodium/Potassium Ratio for 200 patients, with drug type indicated by color and shape. The three new patients are represented by large orange circles.

For Patient 1, who is located deep within a cluster of red-circle points (Drug A), the classification is straightforward. All the nearest neighbors belong to Drug A, making it a confident and unambiguous decision.

For Patient 2, the classification outcome depends on the value of k . When $k = 1$, the nearest neighbor is a blue square, and the predicted class is Drug C. When $k = 2$, there is a tie between Drug B and Drug C, and no majority class can be determined. When $k = 3$, two of the three nearest neighbors are blue squares, so the prediction remains Drug C.

For Patient 3, the classification becomes more uncertain. When $k = 1$, the closest neighbor is a blue square, and the classification is Drug C. However, when $k = 2$ or $k = 3$, the nearest neighbors belong to different classes, leading to ambiguity in the predicted label.

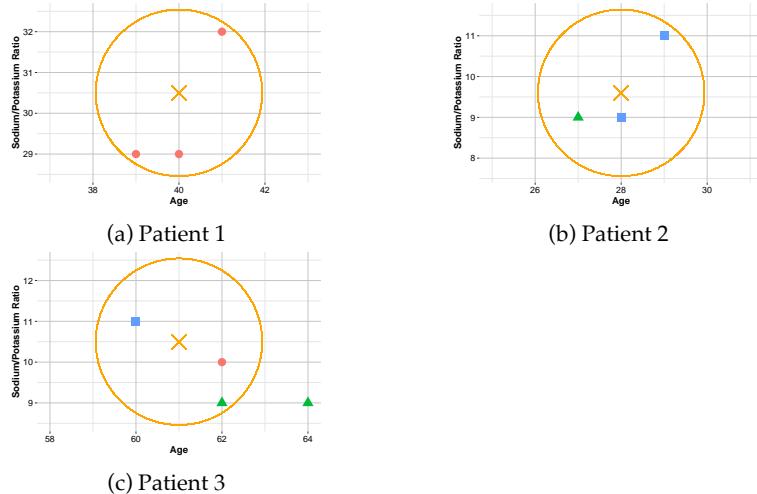


Figure 7.4: Zoom-in plots for the three new patients and their nearest neighbors. The left plot is for Patient 1, the middle plot is for Patient 2, and the right plot is for Patient 3.

This example highlights key considerations for using kNN effectively. The choice of k strongly influences the decision boundary, with smaller values emphasizing local variation and larger values yielding smoother classifications. The distance metric shapes how similarity is assessed, and appropriate feature scaling ensures that all features contribute meaningfully. Together, these design decisions play a critical role in the success of kNN in practice.

7.4 How Does kNN Measure Similarity?

Suppose you are a physician comparing two patients based on age and sodium-to-potassium (Na/K) ratio. One patient is 40 years old with a Na/K ratio of 30.5, and the other is 28 years old with a ratio of 9.6. Which of these patients is more similar to a new case you are evaluating?

In the k-Nearest Neighbors (kNN) algorithm, classifying a new observation depends on identifying the most *similar* records in the training set. While similarity may seem intuitive, machine learning requires a precise definition. Specifically, similarity is quantified using a *distance metric*, which determines how close two observations are in a multidimensional feature space. These distances govern which records are chosen as neighbors and, ultimately, how a new observation is classified.

In this medical scenario, similarity is measured by comparing numerical features such as age and lab values. The smaller the computed distance between two patients, the more similar they are assumed to be, and the more influence they have on classification. Since kNN relies on the assumption that nearby points tend to share the same class label, choosing an appropriate distance metric is essential for accurate predictions.

7.4.1 Euclidean Distance

A widely used measure of similarity in kNN is *Euclidean distance*, which corresponds to the straight-line, or “as-the-crow-flies,” distance between two points. It is intuitive, easy to compute, and well-suited to numerical data with comparable scales.

Mathematically, the Euclidean distance between two points x and y in n -dimensional space is given by:

$$\text{dist}(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2},$$

where $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_n)$ are the feature vectors.

For example, suppose we want to compute the Euclidean distance between two new patients from the previous section, using their *age* and *sodium-to-potassium (Na/K) ratio*. Patient 1 is 40 years old with a Na/K ratio of 30.5, and Patient 2 is 28 years old with a Na/K ratio of 9.6. The Euclidean distance between these two patients is visualized in Figure 7.5 in a two-dimensional feature space, where each axis represents one of the features (age and Na/K ratio). The line connecting Patient 1 (40, 30.5) and Patient 2 (28, 9.6) represents their Euclidean distance:

$$\text{dist}(x, y) = \sqrt{(40 - 28)^2 + (30.5 - 9.6)^2} = \sqrt{144 + 436.81} = 24.11$$

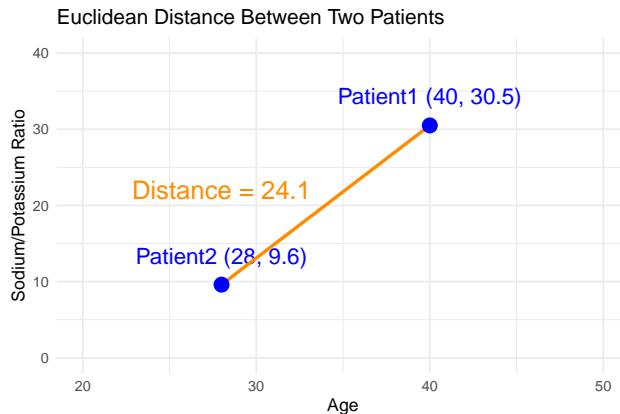


Figure 7.5: Visual representation of Euclidean distance between two patients in 2D space.

This value quantifies how dissimilar the patients are in the two-dimensional feature space, and it plays a key role in determining how the new patient would be classified by kNN.

Although other distance metrics exist, such as Manhattan distance, Hamming distance, or cosine similarity, Euclidean distance is the most commonly used in practice, especially when working with numerical features. Its geometric interpretation is intuitive and it works well when variables are measured on similar scales. In more specialized contexts, other distance metrics may be more appropriate depending on the structure of the data or the application domain. Readers interested in alternative metrics can explore resources such as the `proxy` package in R or consult advanced machine learning texts.

In the next section, we will examine how preprocessing steps like feature scaling ensure that Euclidean distance yields meaningful and balanced comparisons across features.

7.5 Data Preparation for the kNN Algorithm

The performance of the *k*-Nearest Neighbors (kNN) algorithm is highly sensitive to how the data is preprocessed. Because kNN relies on distance calculations to assess similarity between observations, careful preparation of the feature space is essential. Two key steps—*encoding categorical variables* and *feature scaling*—ensure that both categorical and numerical features are properly represented in these computations.

These tasks fall under the *Data Preparation* phase of the Data Science Workflow introduced in Chapter 3 (see Figure 2.3), and they also connect to the broader *Data Setup for Modeling* stage discussed in Chapter 6. Regardless of where they appear in the workflow, these steps are crucial for ensuring that kNN’s similarity measures are both consistent and reliable.

To make this concrete, imagine you are working with patient data that includes age, sodium-to-potassium (Na/K) ratio, marital status, and education level. While age and Na/K ratio are numeric, marital status and education are categorical. To prepare these features for use in a distance-based model, we must convert them into numerical form in a way that preserves their original meaning.

In most tabular datasets (such as the *bank* and *churn* datasets introduced earlier), features include a mix of categorical and numerical variables. A common and recommended approach is to begin by *encoding* the categorical features into numeric format and then *scaling* all numeric features. This sequence ensures that distance calculations are performed on a unified numerical scale, without introducing artificial distortions. In the next subsection, we begin with the challenge of encoding categorical variables.

7.5.1 Encoding Categorical Variables for kNN

Categorical features—such as marital status or education level—are common in real-world datasets. But before they can be used in kNN, they must be transformed into numerical form in a way that aligns with how distances are computed. Since kNN cannot interpret text labels or non-numeric values directly, all categorical variables must be encoded. The appropriate encoding strategy depends on whether the variable is binary, nominal, or ordinal.

These techniques were introduced in Chapter 3, with general guidance in Section 3.13, ordinal handling in Section 3.14, and one-hot encoding in Section 3.15.

Binary and Nominal Variables: One-Hot Encoding

For *binary* variables (such as yes / no) and *nominal* variables (such as marital status with categories single, married, divorced), the recommended approach is *one-hot encoding*. This technique creates one binary column per category, omitting one to avoid redundancy—a practice that avoids the so-called dummy variable trap, where perfectly collinear columns interfere with computation.

For example, the `marital` variable in the `bank` dataset can be one-hot encoded using the `one.hot()` function from the `liver` package:

```
data(bank)

# One-hot encode the "marital" variable from the bank dataset
bank_encoded <- one.hot(bank, cols = c("marital"), dropCols = FALSE)
str(bank_encoded)
'data.frame': 4521 obs. of 20 variables:
 $ age           : int 30 33 35 30 59 35 36 39 41 43 ...
 $ job           : Factor w/ 12 levels "admin.", "blue-collar", ...: 11 8 5
   ↵ 5 2 5 7 10 3 8 ...
 $ marital        : Factor w/ 3 levels "divorced", "married", ...: 2 2 3 2 2
   ↵ 3 2 2 2 2 ...
 $ marital_divorced: int 0 0 0 0 0 0 0 0 0 ...
 $ marital_married : int 1 1 0 1 1 0 1 1 1 1 ...
 $ marital_single  : int 0 0 1 0 0 1 0 0 0 0 ...
 $ education       : Factor w/ 4 levels "primary", "secondary", ...: 1 2 3 3
   ↵ 2 3 3 2 3 1 ...
 $ default         : Factor w/ 2 levels "no", "yes": 1 1 1 1 1 1 1 1 1 ...
 $ balance         : int 1787 4789 1350 1476 0 747 307 147 221 -88 ...
 $ housing          : Factor w/ 2 levels "no", "yes": 1 2 2 2 2 1 2 2 2 2 ...
 $ loan            : Factor w/ 2 levels "no", "yes": 1 2 1 2 1 1 1 1 1 2 ...
 $ contact          : Factor w/ 3 levels "cellular", "telephone", ...: 1 1 1 3
   ↵ 3 1 1 1 3 1 ...
 $ day              : int 19 11 16 3 5 23 14 6 14 17 ...
 $ month             : Factor w/ 12 levels "apr", "aug", "dec", ...: 11 9 1 7 9
   ↵ 4 9 9 9 1 ...
 $ duration         : int 79 220 185 199 226 141 341 151 57 313 ...
 $ campaign          : int 1 1 1 4 1 2 1 2 2 1 ...
 $ pdays            : int -1 339 330 -1 -1 176 330 -1 -1 147 ...
 $ previous          : int 0 4 1 0 0 3 2 0 0 2 ...
 $ poutcome          : Factor w/ 4 levels "failure", "other", ...: 4 1 1 4 4 1
   ↵ 2 4 4 1 ...
 $ deposit           : Factor w/ 2 levels "no", "yes": 1 1 1 1 1 1 1 1 1 ...
```

This transformation results in three new binary columns: `marital_single`, `marital_married`, and `marital_divorced`. Each column indicates whether the corresponding category is present (1) or absent (0) for each observation. This approach ensures that categories are treated as equidistant in the feature space, which is appropriate for variables with no inherent order.

Ordinal Variables: Numeric Encoding

For *ordinal* variables with a meaningful ranking (such as low, medium, high), it is preferable to assign numeric values that reflect their order (e.g., low = 1, medium = 2, high = 3). This preserves the ordinal relationship in the distance calculations, which would otherwise be ignored by one-hot encoding.

For instance, consider the *education* variable in the *bank* dataset, which has levels primary, secondary, and tertiary. We can convert this variable to numeric scores as follows:

```
# Convert an ordinal variable to numeric scores
bank$education_level <- factor(bank$education,
                                levels = c("primary", "secondary", "tertiary"),
                                labels = c(1, 2, 3))
```

This numeric encoding preserves the order of education levels and allows the kNN model to treat higher education levels as more distant from lower ones, reflecting real-world meaning.

In summary, binary and nominal variables should be transformed using one-hot encoding to ensure that each category contributes fairly and independently to the distance computation. For ordinal variables, assigning numeric values that reflect their inherent order allows kNN to capture meaningful relationships. By contrast, treating unordered categories as numeric—such as assigning red = 1, blue = 2, green = 3—can mislead the model by introducing artificial structure. Careful attention to encoding choices helps preserve the integrity of similarity calculations and improves classification accuracy.

7.5.2 Scaling Features for Fair Distance Computation

Once categorical variables have been encoded, all numeric features—both original and derived—should be scaled to ensure they contribute fairly to similarity calculations. Even after proper encoding, features may vary substantially in scale. For example, *age* might range from 20 to 70, while *income* could vary from 20,000 to 150,000. Without appropriate scaling, variables with larger ranges may disproportionately influence the distance calculation, resulting in biased neighbor selection.

Two widely used methods address this issue: *min-max scaling* and *z-score standardization*. These approaches were introduced in Section 3.10.

Min-max scaling transforms each feature to a fixed range, typically [0, 1], using the formula:

$$x_{\text{scaled}} = \frac{x - \min(x)}{\max(x) - \min(x)},$$

where $\min(x)$ and $\max(x)$ are calculated from the training data. This method ensures that all features contribute on the same numerical scale.

Z-score standardization rescales features so that they have a mean of 0 and a standard deviation of 1:

$$x_{\text{scaled}} = \frac{x - \text{mean}(x)}{\text{sd}(x)}.$$

This method is more appropriate when features contain outliers or have differing units or distributions.

Min-max scaling is generally appropriate when feature values are bounded and preserving relative distances is important. Z-score standardization is preferable when features are measured on different units or contain outliers, as it reduces the influence of extreme values.

7.5.3 Preventing Data Leakage during Scaling

Scaling should be performed *after* splitting the dataset into training and test sets. This prevents *data leakage*, a common pitfall in predictive modeling where information from the test set inadvertently influences the model during training. Specifically, parameters such as the mean, standard deviation, minimum, and maximum must be computed *only* from the training data, and then applied to scale both the training and test sets.

The comparison in Figure 7.6 visualizes the importance of applying scaling correctly. The middle panel shows proper scaling using training-derived parameters; the right panel shows the distortion caused by scaling the test data independently.

To illustrate, consider the drug classification task from earlier. Suppose age and Na/K ratio are the two predictors. The following code demonstrates both correct and incorrect approaches to scaling, using the `minmax()` function from the `liver` package:

```
library(liver)

# Correct scaling: Apply train-derived parameters to test data
train_scaled = minmax(train_data, col = c("Age", "Ratio"))

test_scaled = minmax(
  test_data,
  col = c("Age", "Ratio"),
  min = c(min(train_data$Age), min(train_data$Ratio)),
  max = c(max(train_data$Age), max(train_data$Ratio))
)
```

```
# Incorrect scaling: Apply separate scaling to test set
train_scaled_wrongly = minmax(train_data, col = c("Age", "Ratio"))
test_scaled_wrongly = minmax(test_data , col = c("Age", "Ratio"))
```

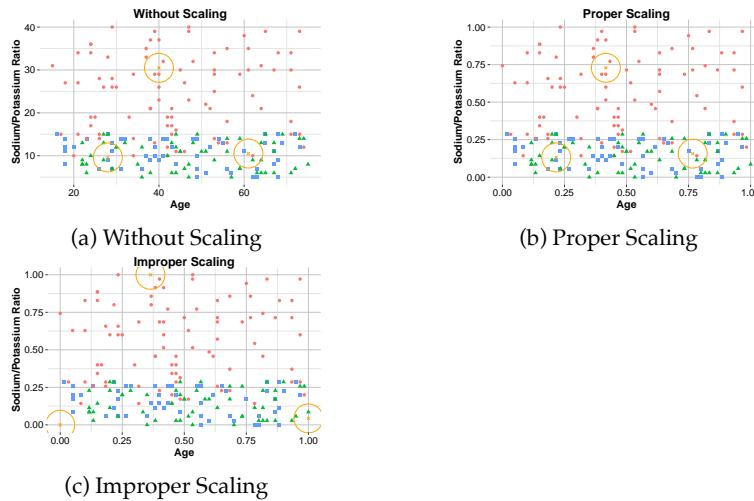


Figure 7.6: Visualization illustrating the difference between proper scaling and improper scaling. The left panel shows the original data without scaling. The middle panel shows the results of proper scaling. The right panel shows the results of improper scaling.

Note. Scaling parameters should always be derived from the training data and then applied consistently to both the training and test sets. Failing to do so can result in incompatible feature spaces, leading the kNN algorithm to identify misleading neighbors and produce unreliable predictions.

With similarity measurement and data preparation steps now complete, the next task is to determine an appropriate value of k . The following section examines how this crucial hyperparameter influences the behavior and performance of the kNN algorithm.

7.6 Choosing the Right Value of k in kNN

Imagine you are new to a city and looking for a good coffee shop. If you ask just one person, you might get a recommendation based on their personal taste, which may differ from yours. If you ask too many people, you could

be overwhelmed by conflicting opinions or suggestions that average out to a generic option. The sweet spot is asking a few individuals whose preferences align with your own. Similarly, in the k -Nearest Neighbors (kNN) algorithm, selecting an appropriate number of neighbors (k) requires balancing specificity and generalization.

The parameter k , which determines how many nearest neighbors are considered during classification, plays a central role in shaping model performance. There is no universally optimal value for k ; the best choice depends on the structure of the dataset and the nature of the classification task. Selecting k involves navigating the trade-off between overfitting and underfitting.

When k is too small, such as $k = 1$, the model becomes overly sensitive to individual training points. Each new observation is classified based solely on its nearest neighbor, making the model highly reactive to noise and outliers. This often leads to *overfitting*, where the model performs well on the training data but generalizes poorly to new cases. A small cluster of mislabeled examples, for instance, could disproportionately influence the results.

As k increases, the algorithm includes more neighbors in its classification decisions, smoothing the decision boundary and reducing the influence of noisy observations. However, when k becomes too large, the model may begin to overlook meaningful patterns, leading to *underfitting*. If k approaches the size of the training set, predictions may default to the majority class label.

To determine a suitable value of k , it is common to evaluate a range of options using a validation set or cross-validation. Performance metrics such as accuracy, precision, recall, and F1-score help guide the selection process.

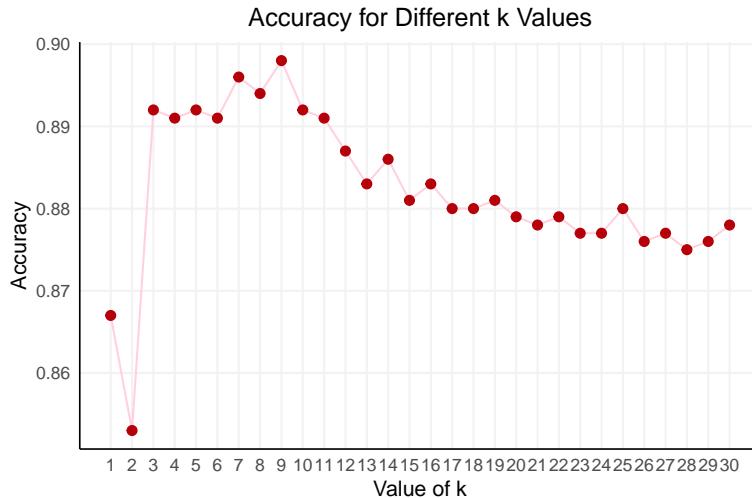


Figure 7.7: Accuracy of the k-Nearest Neighbors algorithm for different values of k in the range from 1 to 30.

As an example, we revisit the *churn* dataset and assess the accuracy of the kNN classifier for k values ranging from 1 to 30. Figure 7.7 presents the results, generated using the `kNN.plot()` function from the `liver` package in R. Accuracy fluctuates as k increases, and the best performance is achieved when $k = 9$. At this value, the model reaches an accuracy of 0.898 and an error rate of 0.102, balancing local sensitivity with broader generalization.

Choosing k is ultimately an empirical process informed by validation and domain knowledge. There is no universal rule, but careful experimentation helps identify a value that generalizes well for the problem at hand. A detailed case study in the following section revisits this example and walks through the complete modeling process, including the generation of Figure 7.7.

7.7 Case Study: Predicting Customer Churn with kNN

In this case study, we demonstrate how to apply the k -Nearest Neighbors (kNN) algorithm to a real-world classification problem. Using the *churn* dataset from the `liver` package in R, we follow the complete modeling workflow—from data preparation to model training and evaluation. This provides a practical context to reinforce the concepts introduced earlier in the chapter.

The *churn* dataset captures customer behavior and service usage across multiple dimensions, including account length, service plans, call metrics, and customer service interactions. The modeling task is to predict whether a customer has churned (yes) or not (no) based on these features.

Readers unfamiliar with the dataset are encouraged to review the exploratory analysis presented in Section 4.3, which provides important context and preliminary findings. We begin here by inspecting the dataset structure:

```
str(churn)
'data.frame': 5000 obs. of 20 variables:
 $ state      : Factor w/ 51 levels "AK","AL","AR",...: 17 36 32 36 37 2
   ↪ 20 25 19 50 ...
 $ area.code   : Factor w/ 3 levels "area_code_408",...: 2 2 2 1 2 3 3 2
   ↪ 1 2 ...
 $ account.length: int 128 107 137 84 75 118 121 147 117 141 ...
 $ voice.plan   : Factor w/ 2 levels "yes","no": 1 1 2 2 2 2 1 2 2 1 ...
 $ voice.messages: int 25 26 0 0 0 0 24 0 0 37 ...
 $ intl.plan    : Factor w/ 2 levels "yes","no": 2 2 2 1 1 1 2 1 2 1 ...
 $ intl.mins    : num 10 13.7 12.2 6.6 10.1 6.3 7.5 7.1 8.7 11.2 ...
 $ intl.calls   : int 3 3 5 7 3 6 7 6 4 5 ...
 $ intl.charge  : num 2.7 3.7 3.29 1.78 2.73 1.7 2.03 1.92 2.35 3.02 ...
 $ day.mins     : num 265 162 243 299 167 ...
 $ day.calls    : int 110 123 114 71 113 98 88 79 97 84 ...
 $ day.charge   : num 45.1 27.5 41.4 50.9 28.3 ...
 $ eve.mins     : num 197.4 195.5 121.2 61.9 148.3 ...
 $ eve.calls    : int 99 103 110 88 122 101 108 94 80 111 ...
 $ eve.charge   : num 16.78 16.62 10.3 5.26 12.61 ...
 $ night.mins   : num 245 254 163 197 187 ...
 $ night.calls   : int 91 103 104 89 121 118 118 96 90 97 ...
 $ night.charge  : num 11.01 11.45 7.32 8.86 8.41 ...
 $ customer.calls: int 1 1 0 2 3 0 3 0 1 0 ...
 $ churn        : Factor w/ 2 levels "yes","no": 2 2 2 2 2 2 2 2 2 2 ...
```

The dataset is an R data frame containing 5000 observations and 19 predictor variables, along with a binary outcome variable, *churn*.

Based on the earlier exploratory analysis and domain relevance, we focus on the following features for building the kNN model:

account.length, *voice.plan*, *voice.messages*, *intl.plan*, *intl.mins*, *intl.calls*, *day.mins*, *day.calls*, *eve.mins*, *eve.calls*, *night.mins*, *night.calls*, and *customer.calls*.

In the remainder of this section, we proceed step by step: partitioning and preprocessing the data, selecting an appropriate value of *k*, fitting the kNN model, making predictions, and evaluating classification performance.

7.7.1 Partitioning and Preprocessing the Data for kNN

To assess how well the kNN model generalizes to new observations, we begin by splitting the dataset into training and test sets. This separation provides an unbiased estimate of predictive accuracy by evaluating model performance on previously unseen data.

Because the *churn* dataset is already cleaned and free of missing values (see Chapter 3), we proceed directly to data partitioning using the `partition()` function from the `liver` package. This function divides the data into an 80% training set and a 20% test set:

```
set.seed(42)

data_sets = partition(data = churn, ratio = c(0.8, 0.2))

train_set = data_sets$part1
test_set = data_sets$part2

test_labels = test_set$churn
```

The `partition()` function preserves the class distribution of the target variable (`churn`) across both sets, ensuring that the test set remains representative of the population. This stratified sampling approach is especially important for classification tasks with imbalanced outcomes. For additional background on data partitioning and validation strategies, refer to Section 6.5.

Encoding Categorical Features for kNN

Because the kNN algorithm requires numerical inputs and relies on distance calculations, categorical features must be converted into numeric format. In the *churn* dataset, `voice.plan` and `intl.plan` are binary categorical variables that require transformation.

The `one.hot()` function from the `liver` package automates this process by generating binary indicator features:

```
categorical_vars = c("voice.plan", "intl.plan")

train_onehot = one.hot(train_set, cols = categorical_vars)
test_onehot = one.hot(test_set, cols = categorical_vars)

str(test_onehot)
'data.frame': 1000 obs. of 22 variables:
 $ state : Factor w/ 51 levels "AK", "AL", "AR", ...: 37 16 10 6 41 27
   .. 1 32 21 16 ...
```

```
$ area.code      : Factor w/ 3 levels "area_code_408",...: 2 2 2 1 2 1 1 1
  ↵ 1 1 ...
$ account.length: int 75 65 147 77 111 54 36 149 135 60 ...
$ voice.plan_yes: int 0 0 0 0 0 1 0 1 0 ...
$ voice.plan_no : int 1 1 1 1 1 0 1 0 1 ...
$ voice.messages: int 0 0 0 0 0 30 0 41 0 ...
$ intl.plan_yes : int 1 0 0 0 0 0 0 1 0 ...
$ intl.plan_no  : int 0 1 1 1 1 1 1 1 0 1 ...
$ intl.mins     : num 10.1 12.7 10.6 5.7 7.7 14.7 14.5 11.1 14.6 6.8 ...
$ intl.calls    : int 3 6 4 6 6 4 6 9 15 3 ...
$ intl.charge   : num 2.73 3.43 2.86 1.54 2.08 3.97 3.92 3 3.94 1.84 ...
$ day.mins     : num 166.7 129.1 155.1 62.4 110.4 ...
$ day.calls    : int 113 137 117 89 103 73 128 94 85 57 ...
$ day.charge   : num 28.3 21.9 26.4 10.6 18.8 ...
$ eve.mins     : num 148 228 240 170 137 ...
$ eve.calls    : int 122 83 93 121 102 100 80 92 107 115 ...
$ eve.charge   : num 12.6 19.4 20.4 14.4 11.7 ...
$ night.mins   : num 187 209 209 210 190 ...
$ night.calls  : int 121 111 133 64 105 68 109 108 78 129 ...
$ night.charge : num 8.41 9.4 9.4 9.43 8.53 ...
$ customer.calls: int 3 4 0 5 2 3 0 1 0 1 ...
$ churn        : Factor w/ 2 levels "yes","no": 2 1 2 1 2 2 2 2 1 2 ...
```

For each binary variable, the function creates two columns (e.g., `voice.plan_yes` and `voice.plan_no`). Since the presence of one category implies the absence of the other, only one indicator variable (e.g., `voice.plan_yes`) is retained. This avoids redundancy and maintains interpretability while ensuring compatibility with distance-based modeling.

Feature Scaling for kNN

To ensure that all numerical features contribute equally to distance calculations, we apply *min-max scaling*. This technique rescales each feature to the $[0, 1]$ range using the minimum and maximum values calculated from the *training set*. The same scaling parameters are then applied to the test set to prevent data leakage:

```
numeric_vars = c("account.length", "voice.messages", "intl.mins",
                 "intl.calls", "day.mins", "day.calls", "eve.mins",
                 "eve.calls", "night.mins", "night.calls",
                 "customer.calls")

min_train = sapply(train_set[, numeric_vars], min) # Compute column-wise
  ↵ minimums
max_train = sapply(train_set[, numeric_vars], max) # Compute column-wise
  ↵ maximums

train_scaled = minmax(train_onehot, col = numeric_vars, min = min_train, max
  ↵ = max_train)
```

```
test_scaled = minmax(test_onehot, col = numeric_vars, min = min_train, max
← = max_train)
```

Here, the `sapply()` function is used to compute the column-wise minimum and maximum values across the selected numeric variables in the training set. These values define the scaling range. The `minmax()` function from the `liver` package then applies min-max normalization to both the training and test sets, using the training-set values as reference.

This step ensures that all features are on a comparable scale, preventing those with larger ranges from disproportionately influencing the kNN distance calculations. For a more detailed discussion of scaling methods and best practices, see Section 3.10 and the preparation overview in Section 7.5. Now that the data are properly encoded and scaled, we are ready to choose the optimal number of neighbors (k) for the kNN algorithm.

7.7.2 Finding the Best Value for (k)

The number of neighbors, k , is a key hyperparameter in the kNN algorithm. Choosing too small a k can make the model overly sensitive to noise in the data, while a very large k can oversmooth the decision boundary, potentially missing important local patterns.

In R, there are several ways to choose the optimal value of k . A common approach is to evaluate the classification accuracy of the kNN algorithm across a range of values (e.g., from 1 to 30), and then select the k that yields the highest accuracy. This can be implemented manually using a for loop and tracking performance for each k .

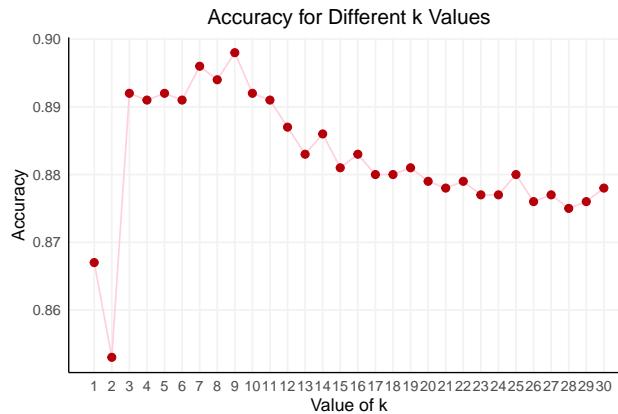
To simplify this process, the `liver` package provides the `knn.plot()` function, which automates this task. It computes accuracy across a specified range of k values and produces a visual summary of the results, making it easier to identify the best-performing k .

Before running the function, we define a `formula` object that specifies the relationship between the target variable (`churn`) and the predictor variables. The predictors include all scaled numerical features as well as the binary indicators generated through one-hot encoding—namely `intl.plan_yes` and `voice.plan_yes`:

```
formula = churn ~ voice.plan_yes + intl.plan_yes + account.length +
  voice.messages + intl.mins + intl.calls +
  day.mins + day.calls + eve.mins + eve.calls +
  night.mins + night.calls + customer.calls
```

We now apply the `kNN.plot()` function:

```
kNN.plot(formula = formula, train = train_scaled, test = test_scaled,
         k.max = 30, reference = "yes", set.seed = 42)
```



The arguments to `kNN.plot()` play distinct roles in shaping the evaluation. The `train` and `test` inputs specify the scaled datasets used for training and testing, respectively, ensuring that distance calculations are made on comparable feature scales. The `k.max = 30` argument defines the maximum number of neighbors to evaluate, allowing us to observe model behavior across a range of values. Setting `reference = "yes"` indicates that the "yes" class represents the positive outcome of interest—i.e., customer churn. Finally, `set.seed = 42` ensures reproducibility by fixing the random seed for any internal random processes.

The resulting plot shows how model accuracy varies with k . In this case, the highest accuracy is achieved when $k = 9$, suggesting that this value offers a good balance between capturing local structure and maintaining generalization. With the optimal value of k identified, we are now ready to apply the kNN algorithm to classify new customer records in the test set.

7.7.3 Applying the kNN Classifier

With the optimal value $k = 9$ identified, we now apply the k -Nearest Neighbors (kNN) algorithm to classify customer churn in the test set. This step brings together the work from the previous sections—data preparation, feature encoding, scaling, and hyperparameter tuning.

Unlike many machine learning algorithms, kNN does not build an explicit predictive model during training. Instead, it retains the training data and

performs classification *on demand* by computing distances to determine the closest training examples.

In R, we use the `kNN()` function from the **liver** package to apply the *k*-Nearest Neighbors algorithm. This function provides a formula-based interface, consistent with other modeling functions in R, making the syntax more readable and the modeling process more transparent. An alternative is the `knn()` function from the **class** package, which requires manually specifying input matrices and class labels. While effective, this approach is less intuitive for beginners and is not used in this book:

```
kNN_predict = kNN(formula = formula, train = train_scaled, test =
  ↵ test_scaled, k = 9)
```

In this command, `formula` defines the relationship between the response variable (`churn`) and the predictors. The `train` and `test` arguments specify the scaled datasets prepared in earlier steps. The parameter `k = 9` sets the number of nearest neighbors to use, as determined in the tuning step.

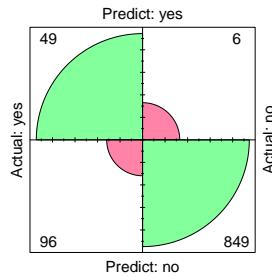
The `kNN()` function classifies each test observation by computing its distance to all records in the training set and assigning the majority class among the nine nearest neighbors.

7.7.4 Evaluating Model Performance of the kNN Model

With predictions in hand, the final step is to assess how well the kNN model performs. A fundamental and intuitive evaluation tool is the *confusion matrix*, which summarizes the correspondence between predicted and actual class labels in the test set.

We use the `conf.mat.plot()` function from the **liver** package to compute and visualize this matrix. The argument `reference = "yes"` specifies that the positive class refers to customers who have churned:

```
conf.mat.plot(kNN_predict, test_labels, reference = "yes")
```



The resulting matrix displays the number of true positives, true negatives, false positives, and false negatives. In this example, the model correctly classified 898 observations and misclassified 102.

While the confusion matrix provides a useful snapshot of model performance, it does not capture all aspects of classification quality. In Chapter 8, we introduce additional evaluation metrics—including accuracy, precision, recall, and F1-score—that offer a more nuanced assessment.

Summary of the kNN Case Study

This case study has demonstrated the complete modeling pipeline for applying *k*-Nearest Neighbors (kNN): starting with data partitioning, followed by preprocessing (including encoding and scaling), tuning the hyperparameter k , applying the classifier, and evaluating the results. Each stage plays a critical role in ensuring that the final predictions are both accurate and interpretable.

While the confusion matrix provides an initial evaluation of model performance, a more comprehensive assessment requires additional metrics such as accuracy, precision, recall, and F1-score. These will be explored in the next chapter (Chapter 8), which introduces tools and techniques for evaluating and comparing machine learning models more rigorously.

7.8 Chapter Summary and Takeaways

This chapter introduced the *k*-Nearest Neighbors (kNN) algorithm, a simple yet effective method for classification. We began by revisiting the concept of classification and its practical applications, distinguishing between binary and multi-class problems. We then examined how kNN classifies observations by identifying their nearest neighbors using distance metrics.

To ensure meaningful distance comparisons, we discussed essential preprocessing steps such as one-hot encoding of categorical variables and feature scaling. We also explored how to select the optimal number of neighbors (k), emphasizing the trade-off between overfitting and underfitting. These concepts were demonstrated through a complete case study using the `liver` package in R and the `churn` dataset, highlighting the importance of thoughtful data preparation and parameter tuning.

The simplicity and interpretability of kNN make it a valuable introductory model. However, its limitations—including sensitivity to noise, reliance on

proper scaling, and inefficiency with large datasets—can reduce its practicality for large-scale applications. Despite these drawbacks, kNN remains a strong baseline for classification tasks and a useful reference point for model comparison.

While our focus has been on *classification*, the kNN algorithm also supports *regression*. In *kNN regression*, the target variable is numeric, and predictions are based on averaging the outcomes of the k nearest neighbors. This variant follows the same core principles and offers a non-parametric alternative to traditional regression models.

Another important use case is *imputation of missing values*, where kNN fills in missing entries by identifying similar observations and using their values (via majority vote or averaging). This method preserves local structure in the data and often outperforms basic imputation techniques such as mean substitution, especially when the extent of missingness is moderate.

In the chapters that follow, we turn to more advanced classification methods. We begin with Naive Bayes (Chapter 9), followed by Logistic Regression (Chapter 10), and Decision Trees (Chapter 11). These models address many of kNN’s limitations and provide more scalable and robust tools for real-world predictive tasks.

7.9 Exercises

The following exercises reinforce key ideas introduced in this chapter. Begin with conceptual questions to test your understanding, continue with hands-on modeling tasks using the *bank* dataset, and conclude with reflective prompts and real-world considerations for applying kNN.

Conceptual Questions

1. Explain the fundamental difference between classification and regression. Provide an example of each.
2. What are the key steps in applying the kNN algorithm?
3. Why is the choice of k important in kNN, and what happens when k is too small or too large?
4. Describe the role of distance metrics in kNN classification. Why is Euclidean distance commonly used?

5. What are the limitations of kNN compared to other classification algorithms?
6. How does feature scaling impact the performance of kNN? Why is it necessary?
7. How is one-hot encoding used in kNN, and why is it necessary for categorical variables?
8. How does kNN handle missing values? What strategies can be used to deal with missing data?
9. Explain the difference between *lazy learning* (such as kNN) and *eager learning* (such as decision trees or logistic regression). Give one advantage of each.
10. Why is kNN considered a non-parametric algorithm? What advantages and disadvantages does this bring?

Hands-On Practice: Applying kNN to the Bank Dataset

The following tasks apply the kNN algorithm to the *bank* dataset from the **liver** package. This dataset includes customer demographics and banking history, with the goal of predicting whether a customer subscribed to a term deposit. These exercises follow the same modeling steps as the churn case study and offer opportunities to deepen your practical understanding.

To begin, load the necessary package and dataset:

```
library(liver)

# Load the dataset
data(bank)

# View the structure of the dataset
str(bank)
'data.frame': 4521 obs. of 17 variables:
 $ age      : int 30 33 35 30 59 35 36 39 41 43 ...
 $ job      : Factor w/ 12 levels "admin.", "blue-collar", ...: 11 8 5 5 2 5
   ↵ 7 10 3 8 ...
 $ marital  : Factor w/ 3 levels "divorced", "married", ...: 2 2 3 2 2 3 2 2
   ↵ 2 2 ...
 $ education: Factor w/ 4 levels "primary", "secondary", ...: 1 2 3 3 2 3 3 2
   ↵ 3 1 ...
 $ default  : Factor w/ 2 levels "no", "yes": 1 1 1 1 1 1 1 1 1 ...
 $ balance  : int 1787 4789 1350 1476 0 747 307 147 221 -88 ...
 $ housing  : Factor w/ 2 levels "no", "yes": 1 2 2 2 2 1 2 2 2 2 ...
 $ loan     : Factor w/ 2 levels "no", "yes": 1 2 1 2 1 1 1 1 1 2 ...
 $ contact  : Factor w/ 3 levels "cellular", "telephone", ...: 1 1 1 3 3 1 1
   ↵ 1 3 1 ...
```

```
$ day      : int  19 11 16 3 5 23 14 6 14 17 ...
$ month    : Factor w/ 12 levels "apr","aug","dec",...
  ↵ 1 ...
$ duration : int  79 220 185 199 226 141 341 151 57 313 ...
$ campaign : int  1 1 1 4 1 2 1 2 2 1 ...
$ pdays    : int  -1 339 330 -1 -1 176 330 -1 -1 147 ...
$ previous : int  0 4 1 0 0 3 2 0 0 2 ...
$ poutcome : Factor w/ 4 levels "failure","other",...
  ↵ ...
$ deposit  : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...

```

Data Exploration and Preparation

11. Load the *bank* dataset and display its structure. Identify the target variable and the predictor variables.
12. Perform an initial EDA:
 - What are the distributions of key numeric variables like age, balance, and duration?
 - Are there any unusually high or low values that might influence distance calculations in kNN?
13. Explore potential associations:
 - Are there noticeable differences in numeric features (e.g., balance, duration) between customers who subscribed to a deposit versus those who did not?
 - Are there categorical features (e.g., job, marital) that seem associated with the outcome?
14. Count the number of instances where a customer subscribed to a term deposit (*deposit* = “yes”) versus those who did not (*deposit* = “no”). What does this tell you about class imbalance?
15. Identify nominal variables in the dataset. Apply one-hot encoding using the `one.hot()` function. Retain only one dummy variable per categorical feature to avoid redundancy and multicollinearity.
16. Partition the dataset into 80% training and 20% testing sets using the `partition()` function. Ensure the target variable remains proportionally distributed in both sets.
17. Validate the partitioning by comparing the class distribution of the target variable in the training and test sets.

18. Apply min-max scaling to numerical variables in both training and test sets. Ensure that the scaling parameters are derived from the training set only.

Diagnosing the Impact of Preprocessing

19. What happens if you skip feature scaling before applying kNN? Train a model without scaling and compare its accuracy to the scaled version.
20. What happens if you leave categorical variables as strings without applying one-hot encoding? Does the model return an error, or does performance decline? Explain why.

Choosing the Optimal k

21. Use the `kNN.plot()` function to determine the optimal k value for classifying `deposit` in the `bank` dataset.
22. What is the best k value based on accuracy? How does accuracy change as k increases?
23. Interpret the meaning of the accuracy curve generated by `kNN.plot()`. What patterns do you observe?

Building and Evaluating the kNN Model

24. Train a kNN model using the optimal k and make predictions on the test set.
25. Generate a confusion matrix for the kNN model predictions using the `conf.mat()` function. Interpret the results.
26. Calculate the accuracy of the kNN model. How well does it perform in predicting `deposit`?
27. Compare the performance of kNN with different values of k (e.g., $k = 1, 5, 15, 25$). How does changing k affect the classification results?
28. Train a kNN model using only a subset of features: `age`, `balance`, `duration`, and `campaign`. Compare its accuracy with the full-feature model. What does this tell you about feature selection?

29. Compare the accuracy of kNN when using min-max scaling versus z-score standardization. How does the choice of scaling method impact model performance?

Critical Thinking and Real-World Applications

30. Suppose you are building a fraud detection system for a bank. Would kNN be a suitable algorithm? What are its advantages and limitations in this context?
31. How would you handle imbalanced classes in the *bank* dataset? What strategies could improve classification performance?
32. In a high-dimensional dataset with hundreds of features, would kNN still be an effective approach? Why or why not?
33. Imagine you are working with a dataset where new observations are collected continuously. What challenges would kNN face, and how could they be addressed?
34. If a financial institution wants to classify customers into different risk categories for loan approval, what preprocessing steps would be essential before applying kNN?
35. In a dataset where some features are irrelevant or redundant, how could you improve kNN's performance? What feature selection methods would you use?
36. If computation time is a concern, what strategies could you apply to make kNN more efficient for large datasets?
37. Suppose kNN is performing poorly on the *bank* dataset. What possible reasons could explain this, and how would you troubleshoot the issue?

Self-Reflection

38. What did you find most intuitive about the kNN algorithm? What aspects required more effort to understand?
39. How did the visualizations (e.g., scatter plots, accuracy curves, and confusion matrices) help you understand the behavior of the model?
40. If you were to explain how kNN works to a colleague or friend, how would you describe it in your own words?

41. How would you decide whether kNN is a good choice for a new dataset or project you are working on?
42. Which data preprocessing steps—such as encoding or scaling—felt most important in improving kNN’s performance?

Chapter 8

Evaluating Machine Learning Models

How do we know whether a machine learning model is actually good? Is 95% accuracy always impressive—or can it be misleading? How do we balance catching true cases while minimizing false alarms? These are the kinds of questions we begin to answer in this chapter.

Consider this: if you give the same dataset and research question to ten different data science teams, you are likely to receive ten different results—often with dramatically different conclusions. Why does this happen? These differences often arise not from the data or models themselves, but from how each team *evaluates* their results. One team’s “successful” model might be another team’s failure, depending on the metrics they prioritize, how they set thresholds, and what trade-offs they are willing to make. As [George Box](#) famously said, “*All models are wrong, but some are useful.*” Evaluation helps us determine which models are useful enough to inform decisions and guide action. This chapter gives you the tools to understand those differences—and to evaluate models with clarity and confidence.

In the previous chapter, we introduced our first machine learning algorithm: kNN. We learned how to build a classifier using the *churn* dataset, carefully scaling features and tuning the number of neighbors to improve predictions. This raises a critical question: *How can we determine whether the model truly performs well?*

This brings us to the *Model Evaluation* phase, a pivotal step in the Data Science Workflow first introduced in Chapter 2 and illustrated in Figure 2.3. By this point, we have already completed the first five phases:

1. *Problem Understanding*: defining the question we aim to solve;
2. *Data Preparation*: cleaning, transforming, and organizing the data;
3. *Exploratory Data Analysis (EDA)*: identifying patterns and relationships;
4. *Data Setup for Modeling*: scaling, encoding, and partitioning the data;

5. *Modeling*: training algorithms to make predictions or uncover structure.

Now, we arrive at *Model Evaluation*, where we assess the performance and reliability of our models using quantitative metrics. This phase answers the essential question: *How well does our model generalize to new, unseen data?*

A model may perform well during development but fail in deployment, where data distributions can shift and the cost of errors is often higher. Model evaluation ensures that our predictions are trustworthy and that the model captures genuine patterns rather than memorizing noise.

Why Is Model Evaluation Important?

Developing a model is only the beginning. Its real value lies in its ability to generalize to *new, unseen data*. A model may perform well during development but fail in real-world deployment, where data distributions often shift and the cost of errors can be much higher.

Consider a model built to detect fraudulent credit card transactions. Suppose it achieves 95% accuracy. While this might sound impressive, it could be misleading if only 1% of the transactions are actually fraudulent. In such an imbalanced dataset, a model might simply label all transactions as legitimate to attain high accuracy—yet fail entirely to detect fraud. This example illustrates a crucial point: *accuracy alone does not always tell the full story*, especially in class-imbalanced settings.

Effective model evaluation offers a more nuanced view of performance by revealing both strengths and limitations. It helps clarify what the model does well—such as correctly identifying fraud—and where it falls short, such as missing fraudulent cases or producing too many false alarms. It also surfaces trade-offs between competing priorities, such as sensitivity versus specificity, or precision versus recall.

In this sense, evaluation is not just about metrics—it is about trust. A well-evaluated model informs responsible decision-making by aligning performance with the needs and risks of the application. Key questions include:

- How well does the model handle class imbalance?
- Can it reliably detect true positives, such as diagnosing cancer in medical data?
- Does it minimize false positives, such as incorrectly classifying a legitimate email as spam?

These considerations show why model evaluation is a critical step in the data science workflow. Choosing appropriate metrics—and interpreting them in context—enables us to move beyond surface-level performance and toward robust, reliable solutions.

What This Chapter Covers

This chapter introduces essential tools for evaluating machine learning models. We begin with binary classification, focusing on how to interpret confusion matrices and compute key performance metrics such as accuracy, precision, recall, and the F1-score. We also examine how adjusting classification thresholds can influence predictions and introduce ROC curves and the Area Under the Curve (AUC) as effective tools for visualizing and comparing classifier performance.

We then introduce evaluation metrics for regression models, including Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and the coefficient of determination (R^2), with guidance on interpreting these values in applied contexts.

These evaluation metrics are among the most widely used in machine learning. They will appear frequently in the chapters that follow as we explore a variety of classification and regression algorithms. Mastering them now will equip you with the tools to assess and compare models throughout the rest of the book.

To build your intuition, this chapter includes visualizations and practical examples for each evaluation method. By the end, you will be able to choose appropriate metrics for different tasks, interpret model performance critically, and evaluate models effectively in both classification and regression settings.

We begin with one of the most foundational tools in model evaluation: the *confusion matrix*, which provides a structured summary of prediction outcomes.

8.1 Confusion Matrix

How can we tell exactly where a model performs well—and where it fails? The *confusion matrix* provides a structured answer. It is one of the most fundamental tools for evaluating classification models, breaking down predic-

tions into four outcomes based on the comparison of predicted and actual class labels.

In binary classification, one class is typically designated as the *positive class*—the class of primary interest—while the other is considered the *negative class*. For instance, in fraud detection, fraudulent transactions are usually labeled as positive, and legitimate ones as negative.

Table 8.1: Confusion matrix for binary classification, summarizing correct and incorrect predictions based on whether the actual class is positive or negative.

| <i>Predicted</i> | Positive | Negative |
|------------------------|---------------------------|---------------------|
| <i>Actual Positive</i> | True Positive (TP) | False Negative (FN) |
| <i>Actual Negative</i> | False Positive (FP) | True Negative (TN) |

Each cell in the matrix represents one of four possible outcomes. *True Positives (TP)* occur when the model correctly predicts the positive class (e.g., fraud detected as fraud). *False Positives (FP)* arise when the model incorrectly flags the positive class (e.g., legitimate transactions incorrectly flagged as fraud). *True Negatives (TN)* are correct predictions of the negative class, while *False Negatives (FN)* represent missed detections of the positive class (e.g., fraudulent transactions classified as legitimate).

This structure may feel familiar—it mirrors the concept of *Type I and Type II errors* introduced in Chapter 5 on hypothesis testing. The diagonal elements (TP and TN) represent correct predictions, while the off-diagonal elements (FP and FN) represent errors.

With the confusion matrix in hand, we can compute basic performance metrics. Two of the most general are *accuracy* and *error rate*.

Accuracy (also called success rate) measures the proportion of correct predictions:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{Total Predictions}} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}}$$

Error rate is the proportion of incorrect predictions:

$$\text{Error Rate} = 1 - \text{Accuracy} = \frac{\text{FP} + \text{FN}}{\text{Total Predictions}}$$

These metrics offer a broad summary of model performance. However, they can be misleading. Imagine a dataset in which only 5% of transactions are

fraudulent. A model that labels all transactions as legitimate would still achieve 95% accuracy—yet completely fail to detect fraud. This illustrates a classic case where accuracy hides deeper weaknesses, especially in imbalanced datasets.

To better understand a model’s strengths and weaknesses—such as how well it identifies positive cases or avoids false alarms—we need more detailed metrics. The next section introduces sensitivity, specificity, precision, and recall.

Let us revisit the kNN model from Chapter 7, which was used to predict customer churn using the churn dataset. We now evaluate its performance using a confusion matrix. We begin by applying the kNN model to the test set:

```
library(liver)

data(churn)
set.seed(42)

data_sets = partition(data = churn, ratio = c(0.8, 0.2))
train_set = data_sets$part1
test_set = data_sets$part2
test_labels = test_set$churn

formula = churn ~ account.length + voice.plan + voice.messages +
           intl.plan + intl.mins + intl.calls +
           day.mins + day.calls + eve.mins + eve.calls +
           night.mins + night.calls + customer.calls

kNN_predict = kNN(formula = formula, train = train_set,
                  test = test_set, k = 9, scaler = "minmax")
```

For details on how this model was built, see Section 7.7. Here we use `scaler = "minmax"` to scale numeric features to the [0, 1] range and one-hot encode binary variables.

In R, one way to compute a confusion matrix is by using the `conf.mat()` function from the `liver` package, which provides a consistent interface for classification evaluation. The same package also includes `conf.mat.plot()` for visualizing confusion matrices, making interpretation more intuitive.

We compute the confusion matrix as follows:

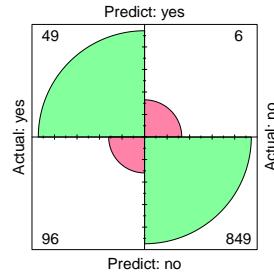
```
conf.mat(pred = kNN_predict, actual = test_labels, reference = "yes")
          Actual
Predict yes no
  yes   49   6
  no   96 849
```

The `conf.mat()` function accepts several key arguments. The `pred` argument specifies the predicted class labels or probabilities, while `actual` contains the true labels from the test set. The `reference` argument designates which class is treated as the *positive class*. The optional `cutoff` argument is used when probabilities are predicted (defaulting to 0.5), but it is not needed here because `kNN_predict` returns class labels. In this example, "yes" denotes the positive class—indicating customers who churned.

The confusion matrix shows that the model correctly identified 49 churners (*true positives*) and 849 non-churners (*true negatives*). However, it also incorrectly predicted that 96 non-churners would churn (*false positives*), and failed to identify 6 actual churners (*false negatives*).

We can also visualize the confusion matrix:

```
conf.mat.plot(pred = kNN_predict, actual = test_labels)
Setting levels: reference = "yes", case = "no"
```



This plot provides a clear visual summary of prediction outcomes.

Next, we compute the accuracy and error rate:

$$\text{Accuracy} = \frac{49 + 849}{1000} = 0.898$$

$$\text{Error Rate} = \frac{96 + 6}{1000} = 0.102$$

Thus, the model correctly classified 89.8% of cases, while 10.2% were misclassified.

Having reviewed accuracy and error rate, we next turn to additional evaluation metrics that provide more insight into a model's strengths and limitations—particularly in imbalanced or high-stakes classification settings. The next section introduces *sensitivity*, *specificity*, *precision*, and *recall*.

8.2 Sensitivity and Specificity

Suppose your model achieves 98% accuracy in detecting credit card fraud. Sounds impressive—but what if only 2% of transactions are actually fraudulent? Could a model that simply labels every transaction as not fraud still reach that 98% accuracy? This is where accuracy falls short—and where *sensitivity* and *specificity* become essential.

In classification, it is not enough to know how many predictions are correct overall. We also need to understand *how well the model identifies each class*. Sensitivity and specificity are two complementary metrics that help answer this question, especially in situations with *imbalanced data*—when one class appears much more frequently than the other.

These metrics allow us to examine the model’s strengths and weaknesses more critically by asking whether it can detect rare but important cases, such as fraud or disease, and whether it avoids misclassifying too many negative cases. By separating performance across the positive and negative classes, sensitivity and specificity help us go beyond accuracy and build more trustworthy models.

8.2.1 Sensitivity

How good is your model at *catching the cases that matter most*? That is the question sensitivity helps answer. *Sensitivity*—also known as *recall* in fields like information retrieval—measures a model’s ability to correctly identify *positive cases*. In other words, it answers: *Out of all the actual positives, how many did the model correctly predict?*

This matters most in situations where missing a positive case has serious consequences, such as failing to detect fraud or disease. The formula for sensitivity is:

$$\text{Sensitivity} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

Let us apply this to the kNN model from Chapter 7, where we predicted customer churn (churn = yes). Sensitivity tells us the percentage of customers who actually left—and were correctly flagged by the model. Using the confusion matrix from the previous section, we have:

$$\text{Sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{49}{49 + 6} = 0.891$$

This means the model correctly identified 89.1% of actual churners.

Note that a model with 100% *sensitivity* flags *every single* positive case. But here's the catch: even a model that labels *everyone* as positive would score perfectly on sensitivity. That is why this metric must always be interpreted in context—with help from other measures like *specificity* and *precision*.

8.2.2 Specificity

If sensitivity tells us how well a model *catches the positive cases*, specificity tells us how well it *avoids false alarms*. *Specificity* measures the model's ability to correctly identify *negative cases*. It answers the question: *Out of all the actual negatives, how many did the model correctly predict?*

This becomes crucial when false positives are costly. Think of spam filters: marking a legitimate email as spam (a false positive) might cause users to miss important messages. In such cases, high specificity is essential. The formula for specificity is:

$$\text{Specificity} = \frac{\text{True Negatives (TN)}}{\text{True Negatives (TN)} + \text{False Positives (FP)}}$$

Let us return to the kNN model from Chapter 7, where we predicted customer churn. Specificity in this case tells us how well the model identified customers who *did not churn*.

Using the confusion matrix from the previous section, we can compute specificity as follows:

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}} = \frac{849}{849 + 96} = 0.898$$

This means the model correctly identified 89.8% of the customers who stayed.

While each metric is valuable on its own, in practice, we often want *high sensitivity and high specificity*. But boosting one can sometimes lower the other. The right balance depends on your application. For the kNN model in the previous section, sensitivity is 0.891, while specificity is 0.898. This trade-off may be acceptable in this context, as identifying churners (sensitivity) is often more important than minimizing false positives (specificity).

Sensitivity vs. Specificity: A Balancing Act

The trade-off between sensitivity and specificity is a central consideration in the evaluation of classification models. In many practical applications, improving one of these metrics tends to reduce the other: increasing sensitivity typically results in more false positives, thereby decreasing specificity, while increasing specificity may lower the number of false positives but can lead to more false negatives, thus reducing sensitivity.

The appropriate balance depends on the context and the relative costs of different types of errors. For instance, in medical diagnostics, failing to identify a disease case (a false negative) may have severe consequences. In such situations, models with high sensitivity are preferred, even at the expense of a higher false positive rate. Conversely, in applications such as email spam detection, misclassifying a legitimate message as spam (a false positive) can be more problematic than missing a few spam messages. In these cases, higher specificity is more desirable.

Understanding the interplay between these two metrics allows practitioners to tailor model evaluation to the priorities and risks associated with a specific application. In the next section, we introduce two additional metrics—precision and recall—that provide further insight into the model’s effectiveness in detecting and correctly classifying the positive class.

8.3 Precision, Recall, and F1-Score

Imagine you are designing a system to detect fraudulent credit card transactions. If your model predicts fraud and it is wrong, an annoyed customer gets a call and may lose trust in your service. But if it fails to detect actual fraud, money is lost. Which mistake is worse?

This kind of dilemma reveals why metrics beyond accuracy are essential. Accuracy tells us how often the model is right—but not *how* it goes wrong. Sensitivity tells us how many positives we catch—but not how many false alarms we raise. Precision and recall close this gap.

When your model flags a customer as likely to churn, how confident can you be that it is right? And how often does it miss actual churners? These are the kinds of questions precision and recall help us answer.

In addition to sensitivity and specificity, the metrics of *Precision*, *recall*, and the *F1-score* provide a more detailed view of a classification model’s performance. These metrics are especially useful in imbalanced datasets, where accuracy can be misleading.

Precision, also referred to as the *positive predictive value*, measures how many of the predicted positives are actually positive. It answers the question: *When the model predicts a positive case, how often is it correct?*

This is formally defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Precision becomes particularly important in applications where false positives are costly. In fraud detection, for example, incorrectly flagging legitimate transactions can inconvenience customers and require unnecessary investigation.

Recall, which is equivalent to sensitivity, measures the model's ability to identify all actual positive cases. It addresses the question: *Out of all the actual positives, how many did the model correctly identify?*

The formula for recall is:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Recall is crucial in settings where missing a positive case has serious consequences, such as medical diagnosis or fraud detection. Recall is synonymous with sensitivity; both measure how many actual positives are correctly identified. While the term sensitivity is common in biomedical contexts, recall is often used in fields like information retrieval and text classification.

There is typically a trade-off between precision and recall. Increasing precision makes the model more conservative in predicting positives, which reduces false positives but may also miss true positives, resulting in lower recall. Conversely, increasing recall ensures more positive cases are captured, but often at the cost of a higher false positive rate, thus lowering precision. For instance, in cancer screening, maximizing recall ensures no cases are missed, even if some healthy patients are falsely flagged. In contrast, in email spam detection, a high precision is desirable to avoid misclassifying legitimate emails as spam.

To quantify this trade-off, the *F1-score* combines precision and recall into a single metric. It is the harmonic mean of the two:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \times \text{TP}}{2 \times \text{TP} + \text{FP} + \text{FN}}$$

The F1-score is particularly valuable when dealing with imbalanced datasets. Unlike accuracy, it accounts for both false positives and false negatives, offering a more balanced evaluation.

Let us now compute these metrics using the kNN model in Section 8.1, which predicts whether a customer will churn (churn = yes).

Precision measures how often the model's churn predictions are correct:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{49}{49 + 96} = 0.338$$

This indicates that the model's predictions of churn are correct in 33.8% of cases.

recall (or sensitivity) reflects how many actual churners were correctly identified:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{49}{49 + 6} = 0.891$$

The model thus successfully identifies 89.1% of churners.

F1-score combines these into a single measure:

$$F1 = \frac{2 \times 49}{2 \times 49 + 96 + 6} = 0.49$$

This score summarizes the model's ability to correctly identify churners while balancing the cost of false predictions.

The F1-score is a valuable metric when precision and recall are both important. However, in practice, their relative importance depends on the context. In healthcare, recall might be prioritized to avoid missing true cases. In contrast, in filtering systems like spam detection, precision may be more important to avoid misclassifying valid items.

In the next section, we shift our focus to metrics that evaluate classification models across a range of thresholds, rather than at a fixed cutoff. This leads us to the Receiver Operating Characteristic (ROC) curve and Area Under the Curve (AUC), which offer a broader view of classification performance.

8.4 Taking Uncertainty into Account

Imagine a doctor evaluating whether a patient has a rare disease. The model outputs a 0.72 probability. Should the doctor act on this prediction? This scenario illustrates a critical point in classification: many models can return probability estimates, not just hard labels. These probabilities express the

model's confidence in its predictions, offering a richer basis for decision-making than binary outputs alone.

Most of the evaluation metrics we have discussed, such as precision, recall, and the F1-score, are based on fixed, binary predictions. But this perspective omits an important aspect of many classification models: *uncertainty*.

Many models, including *k*-Nearest Neighbors (*kNN*), can return predicted probabilities. These scores reflect the likelihood that an instance belongs to the positive class. To convert probabilities into class labels, a *classification threshold* must be chosen. A value of 0.5 is commonly used by default: if the predicted probability exceeds 50%, the instance is labeled as positive. However, this threshold is not fixed in stone. Adjusting it can significantly change a model's behavior and allows it to better reflect the priorities of a specific application.

For instance, in fraud detection, missing a fraud case (a false negative) may be more costly than flagging a legitimate transaction (a false positive). In such cases, lowering the threshold can increase sensitivity, capturing more fraud cases even at the expense of additional false alarms. Conversely, in spam filtering, false positives may be more problematic, and a higher threshold may be preferable to protect legitimate emails.

Let us return to the *kNN* model in Section 8.1, which predicts customer churn (churn = yes). This time, we extract predicted probabilities by setting type = "prob" in the *knn()* function:

```
kNN_prob = kNN(formula = formula, train = train_set,
                 test = test_set, k = 9, scaler = "minmax",
                 type = "prob")

kNN_prob[1:6, ]
      yes       no
 5  0.4444444 0.5555556
 11 0.2222222 0.7777778
 21 0.1111111 0.8888889
 22 0.5555556 0.4444444
 24 0.0000000 1.0000000
 28 0.1111111 0.8888889
```

The matrix *kNN_prob* has one column per class, so column 1 represents the model's estimated probability that each observation belongs to the positive class (churn = yes) and the second column represents the probability for the negative class (churn = no). For example, the first entry of the first column is 0.44, which means the model is 44% confident the customer will churn (churn = yes).

The type = "prob" option is available for all classification models introduced in this book, making probability-based evaluation consistent across methods.

To convert these probabilities to class predictions, we use the `cutoff` argument in the `conf.mat()` function. Here, we compare two different thresholds:

```
conf.mat(kNN_prob[, 1], test_labels, reference = "yes", cutoff = 0.5)
  Actual
Predict yes no
  yes 49 6
  no 96 849
conf.mat(kNN_prob[, 1], test_labels, reference = "yes", cutoff = 0.7)
  Actual
Predict yes no
  yes 11 1
  no 134 854
```

Note that `kNN_prob[, 1]` extracts the predicted probabilities for `churn = yes` from the first column of the `kNN_prob` matrix. A threshold of 0.5 casts a wider net, predicting churn for anyone with at least 50% probability. This increases sensitivity but may allow more false positives. A threshold of 0.7 is stricter, requiring stronger evidence (70% certainty) before labeling a customer as likely to churn. This generally reduces false positives but risks missing actual churners.

Adjusting the decision threshold helps adapt model behavior to real-world needs where the *costs of errors are not equal*. In practice, lower thresholds increase sensitivity but reduce specificity. Higher thresholds increase specificity but reduce sensitivity.

This flexibility is essential in domains such as fraud detection or medical screening.

Tuning the Classification Threshold

Fine-tuning the classification threshold helps tailor a model to practical constraints. Suppose we want a sensitivity of at least 90% to ensure most churners are flagged. By adjusting the threshold and recalculating evaluation metrics, we can determine the *operating point*, the threshold that satisfies this requirement.

Of course, this involves trade-offs. Lowering the threshold generally boosts recall but may harm precision. Raising it does the opposite. For instance, a threshold of 0.9 might lead to excellent specificity but fail to detect most actual churners.

In short, threshold tuning transforms classification from a one-size-fits-all decision rule into a flexible tool that aligns model output with domain-specific priorities.

Manual tuning is helpful, but a more comprehensive view is needed to fully assess a model’s discriminatory power. To build on this idea, the next section introduces two powerful tools—the *Receiver Operating Characteristic (ROC) curve* and the *Area Under the Curve (AUC)*—which allow us to evaluate model performance across all possible thresholds.

8.5 ROC Curve

How can we fairly compare models when performance depends on the threshold we choose? Imagine building a medical diagnostic tool: adjusting the classification threshold might improve sensitivity but worsen specificity. To understand how models behave across all thresholds—and to compare them systematically—we turn to the *Receiver Operating Characteristic (ROC) curve* and its associated metric, the *Area Under the Curve (AUC)*.

The ROC curve visually represents the trade-off between sensitivity (true positive rate) and the false positive rate ($1 - \text{specificity}$) across various classification thresholds. It plots sensitivity (true positive rate) on the vertical axis against the false positive rate on the horizontal axis. Originally developed for radar signal detection during World War II, the ROC curve is now widely used in machine learning to evaluate classifier performance. In practice, ROC curves are especially useful for comparing different classification models—such as logistic regression, decision trees, and random forests. We will return to this idea in upcoming case studies, where ROC curves and AUC scores help us select the best-performing model.

Figure 8.1 illustrates three common performance scenarios:

- *Optimal Performance (Green Curve)*: A model with near-perfect performance reaches the top-left corner, indicating high sensitivity and specificity.
- *Good Performance (Blue Curve)*: A well-performing model stays closer to the top-left corner, though not perfectly.
- *Random Classifier (Red Diagonal Line)*: The dashed diagonal represents random guessing, with no predictive power.

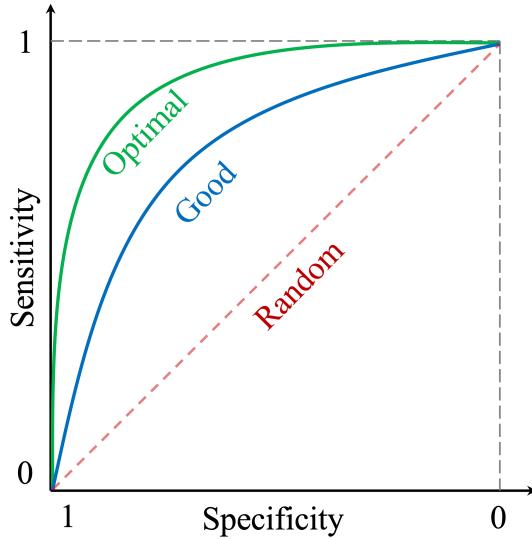


Figure 8.1: The ROC curve illustrates the trade-off between sensitivity and specificity at different thresholds. The diagonal line represents a classifier with no predictive value (red dashed line), while the curves represent varying levels of performance: green for optimal and blue for good.

Each point on the ROC curve corresponds to a specific threshold. As the threshold changes, the sensitivity and false positive rate vary, tracing out the curve. A curve that hugs the top-left corner reflects stronger performance, whereas curves closer to the diagonal suggest weak or random classification.

This visualization helps illustrate key trade-offs. For example, in *medical diagnostics*, maximizing sensitivity is important to avoid missing cases, even at the cost of some false positives. In *fraud detection*, high specificity reduces unnecessary investigations by avoiding false positives.

To construct an ROC curve, predicted probabilities for the positive class and the actual class labels are needed. Correctly predicted positives move the curve upward (increased sensitivity), while false positives move it to the right (increased false positive rate). Let us now see how this works in practice by applying it to the kNN model.

We return to the *kNN* model in Section 8.4, where we obtained probability scores for the positive class (`churn = yes`). We now use these probabilities to generate the ROC curve. The **pROC** package in R provides tools to compute and visualize ROC curves. If it is not already installed, you can do so using `install.packages("pROC")`.

To construct the ROC curve, we use the `roc()` function from the **pROC** package. It requires two main arguments: `response`, which contains the actual

class labels from the test set, and `predictor`, a numeric vector of predicted probabilities for the *positive* class. In our case, `test_labels` contains the true labels, and `kNN_prob[, 1]` extracts the predicted probabilities for `churn = yes` from the first column of the `kNN_prob` matrix. The first column contains probabilities for the positive class, which is essential for computing the ROC curve.

```
library(pROC)
roc_knn <- roc(response = test_labels, predictor = kNN_prob[, 1])
```

We then use the `ggroc()` function to visualize the ROC curve. It takes an ROC object and generates a `ggplot2` visualization for easier customization and clearer display:

```
ggroc(roc_knn, colour = "blue") +
  ggtitle("ROC curve for kNN with k = 9, based on churn data")
```

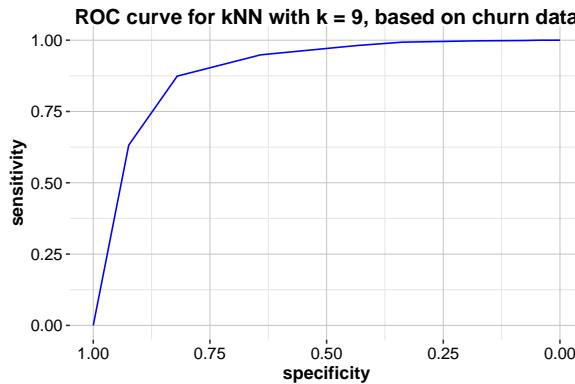


Figure 8.2: ROC curve for kNN with $k = 9$, based on churn data.

This ROC curve shows how the model's true positive rate and false positive rate vary as we adjust the classification threshold. A curve closer to the top-left corner indicates better classification performance. In this case, the ROC curve suggests that the kNN model is effective at distinguishing churners from non-churners.

While the ROC curve provides a visual summary of a model's classification performance across all thresholds, we often need a single numeric value to compare models more directly. This is where the *Area Under the Curve (AUC)* becomes valuable.

8.6 Area Under the Curve (AUC)

While the ROC curve provides a visual summary of a model's performance across all thresholds, it is often useful to quantify this performance with a single number. The *Area Under the Curve (AUC)* serves this purpose. It measures how well the model ranks positive cases higher than negative ones, independent of any particular threshold.

Mathematically, AUC is defined as:

$$\text{AUC} = \int_0^1 \text{TPR}(t) d\text{FPR}(t)$$

where t denotes the classification threshold. A larger AUC value indicates better discrimination between the positive and negative classes across all possible thresholds.

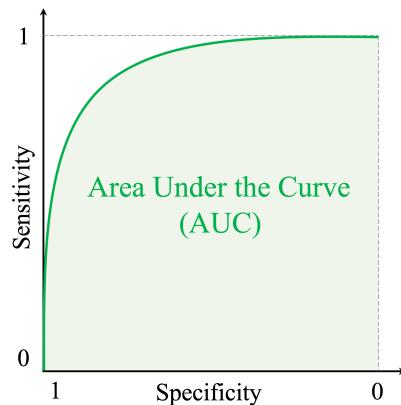


Figure 8.3: The AUC summarizes the ROC curve into a single number, representing the model's ability to rank positive cases higher than negative ones. AUC = 1: Perfect model. AUC = 0.5: No better than random guessing.

As shown in Figure 8.3, the AUC ranges from 0 to 1. A value of 1 indicates a perfect model, 0.5 corresponds to random guessing, and values between 0.5 and 1 reflect varying degrees of predictive power.

To compute the AUC in R, we use the `auc()` function from the **pROC** package. This function takes an ROC object, such as the one created earlier using `roc()`, and returns a numeric value:

```
auc(roc_knn)
Area under the curve: 0.897
```

Here, `roc_knn` is the ROC object based on predicted probabilities for churn = yes. The resulting value represents the model's ability to rank churners above non-churners. For example, the AUC for the kNN model is 0.897, meaning that it ranks churners higher than non-churners with a probability of 0.897.

AUC is especially useful when comparing multiple models or when the costs of false positives and false negatives differ. Unlike accuracy, AUC is *threshold-independent*, providing a more holistic measure of model quality.

In summary, the ROC curve and AUC offer a robust framework for evaluating classifiers, particularly on imbalanced datasets or in applications where the balance between sensitivity and specificity matters. In the next section, we explore how these ideas extend to *multi-class classification*, where evaluation requires new strategies to handle more than two outcome categories.

8.7 Metrics for Multi-Class Classification

So far, we have evaluated binary classifiers using metrics like precision, recall, and AUC. But what if your model needs to go beyond “spam or not” and predict whether an image contains a dog, a cat, or a bird? Many real-world problems involve *multi-class classification*, where the target variable has three or more categories. Examples include classifying types of tumors, predicting modes of transportation, or identifying product categories in retail.

In multi-class settings, the confusion matrix expands into a square grid where rows represent actual classes and columns represent predicted classes, as shown in Figure 8.4. The left matrix in the figure shows a binary (2×2) case, while the right matrix displays a three-class (3×3) case. Correct predictions lie along the diagonal; off-diagonal cells highlight common misclassifications between classes and reveal performance bottlenecks.

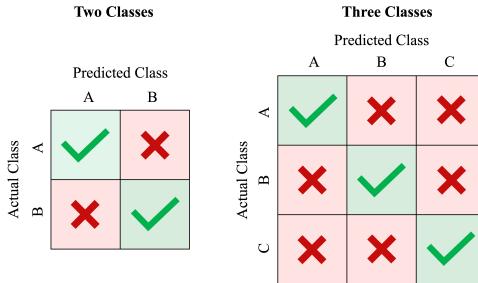


Figure 8.4: Confusion matrices for binary (left) and multi-class (right) classification. Diagonal cells show correct predictions; off-diagonal cells show misclassifications. Matrix size grows with the number of classes.

To compute metrics such as precision, recall, or F1-score in multi-class settings, we use a *one-vs-all* strategy. This approach treats each class in turn as the positive class and groups all other classes as negative. This strategy allows us to isolate performance for each class, which is useful for diagnosing strengths and weaknesses in specific categories using the familiar binary classification framework.

Since multi-class problems involve multiple binary evaluations, we need ways to summarize these into a single score to compare models effectively. To do this, we use different averaging strategies:

- *Macro-average*: Computes the unweighted mean of the per-class metrics, treating all classes equally. This is useful when misclassifying any class is equally costly—such as in disease subtype classification.
- *Micro-average*: Aggregates the individual true positives, false positives, and false negatives across all classes, then calculates metrics from these totals. This method favors majority classes and reflects the model’s overall predictive effectiveness across all examples, as might be desirable in quality control applications.
- *Weighted-average*: Computes a weighted mean of the per-class metrics, where the weight corresponds to the number of true instances (support) for each class. This balances attention between common and rare classes and is especially useful in class-imbalanced problems such as fraud detection.

These averaging techniques ensure that performance evaluation remains meaningful even when class distribution is skewed or certain categories are more important. When interpreting averaged metrics, it is essential to consider how class imbalance or business priorities might influence the final interpretation.

While ROC curves and AUC are inherently binary, they can be extended to multi-class problems using a one-vs-all approach. This generates separate ROC curves and AUC scores for each class. However, interpreting multiple ROC plots can be cumbersome. In many applications, macro- or weighted-averaged F1-scores provide a more concise and interpretable summary.

Many R packages such as **caret**, **yardstick**, and **MLmetrics** support multi-class metrics and facilitate the computation and visualization of evaluation results.

By leveraging one-vs-all metrics and appropriate averaging strategies, we can gain a nuanced view of model performance in multi-class tasks. These tools help identify weaknesses, compare models, and align evaluation with the goals of the application.

In the next section, we turn to *regression models*, where the outcome variable is continuous rather than categorical, and different evaluation tools are required.

8.8 Evaluation Metrics for Continuous Targets

Suppose you want to predict a house's selling price, a patient's recovery time, or the temperature for next week. These are examples of *regression problems*, where the outcome is a continuous value rather than a discrete class. In such cases, classification metrics do not apply. Instead, we use evaluation metrics specifically designed to assess the accuracy of continuous predictions.

One commonly used metric is the *Mean Squared Error (MSE)*:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Here, y_i is the actual value, \hat{y}_i is the predicted value, and n is the number of observations. MSE computes the average of the squared differences between predicted and observed values. Since squaring exaggerates larger errors, MSE is especially sensitive to outliers. Smaller values indicate better predictive accuracy, with zero representing a perfect model.

Another important metric is the *Mean Absolute Error (MAE)*:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

MAE calculates the average absolute difference between predictions and actual outcomes. Unlike MSE, all errors contribute equally, making MAE more robust to extreme values and easier to interpret. This is especially useful when the data contain outliers or when interpretability is a priority.

A third key metric is the *coefficient of determination*, or R^2 :

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

where \bar{y} is the mean of the actual values. The R^2 score measures the proportion of variance in the outcome that is explained by the model. A value of $R^2 = 1$ indicates a perfect fit, while $R^2 = 0$ means the model performs no better than predicting the mean for every observation.

Each of these metrics captures a different aspect of model performance:

- *MSE* penalizes large errors more heavily, making it sensitive to outliers.
- *MAE* offers a more intuitive measure and is less affected by extreme values.
- R^2 provides a scale-free measure of how well the model captures variation in the data.

The choice of metric depends on the goals of the analysis and the characteristics of the dataset. For example, in contexts where large prediction errors are costly, MSE may be more appropriate. When interpretability or robustness to outliers is important, MAE might be preferable. These tools form the foundation for assessing regression models, which we will explore in depth in Chapter 10.

8.9 Chapter Summary and Takeaways

No model is complete until it has been evaluated. A machine learning model is only as useful as its ability to perform reliably on unseen data. In this chapter, we examined the essential task of model evaluation—the process of determining whether a model meets practical requirements and can be trusted in real-world settings. Beginning with foundational concepts, we introduced a range of evaluation metrics for binary classification, multi-class classification, and regression problems.

Unlike other chapters in this book, this one does not include a standalone case study. This is intentional: model evaluation is not a one-time activity

but a recurring component of every modeling task. All subsequent case studies—spanning Naive Bayes, logistic regression, decision trees, and beyond—will include model evaluation as a core element. The tools introduced here will be applied repeatedly, reinforcing their role in sound data science practice.

This chapter also marks the completion of **Step 6: Model Evaluation** in the Data Science Workflow introduced in Chapter 2 and illustrated in Figure 2.3. By evaluating model performance with appropriate metrics, we close the loop between modeling and decision-making—ensuring that models are not only built but validated and aligned with practical goals. In future chapters, as we explore more advanced methods, we will continue to revisit this step in new modeling contexts.

Key Takeaways from this chapter include:

- *Binary classification metrics*: The confusion matrix provides a foundation for computing accuracy, sensitivity, specificity, precision, and the F1-score, each revealing different aspects of model performance.
- *Threshold tuning strategies*: Adjusting classification thresholds shifts the balance between sensitivity and specificity, helping models align with domain-specific priorities.
- *ROC curve and AUC*: These tools offer a threshold-independent view of classifier performance and are especially valuable for model comparison in imbalanced settings.
- *Evaluation for multi-class classification*: One-vs-all strategies and macro, micro, and weighted averages extend binary metrics to tasks with more than two outcome categories.
- *Regression metrics*: MSE, MAE, and the R^2 score provide complementary insights into prediction accuracy for continuous outcomes.

Table 8.2 provides a compact reference for the evaluation metrics discussed in this chapter. You may find this table helpful as a recurring guide throughout the rest of the book.

There is no single metric that universally defines model quality. Evaluation must reflect the goals of the application, balancing trade-offs such as interpretability, fairness, and the costs of different types of errors. By mastering these strategies, you are now prepared to assess models critically, choose thresholds thoughtfully, and compare competing approaches with confidence. In the chapters ahead, we will apply these techniques repeatedly as we build, refine, and evaluate more advanced models.

Now that you have a solid foundation in model evaluation, it is time to put it into practice. In the exercises that follow, you will use the bank dataset to explore how these metrics behave in real-world scenarios.

Table 8.2: Summary of evaluation metrics introduced in this chapter. Each captures a distinct aspect of model performance and should be chosen based on task-specific goals and constraints.

| Metric | Type | Description | When.to.Use |
|----------------------|----------------|--|--|
| Confusion Matrix | Classification | Counts of true positives, false positives, true negatives, and false negatives | Foundation for most classification metrics |
| Accuracy | Classification | Proportion of correct predictions | Balanced datasets, general overview |
| Sensitivity (Recall) | Classification | Proportion of actual positives correctly identified | When missing positives is costly (e.g., disease detection) |
| Specificity | Classification | Proportion of actual negatives correctly identified | When false positives are costly (e.g., spam filters) |
| Precision | Classification | Proportion of predicted positives that are actually positive | When false positives are costly (e.g., fraud alerts) |
| F1-score | Classification | Harmonic mean of precision and recall | Imbalanced data, or when balancing precision and recall |
| AUC (ROC) | Classification | Overall ability to distinguish positives from negatives | Model comparison, imbalanced data |
| MSE | Regression | Average squared error; penalizes large errors | When large prediction errors are critical |
| MAE | Regression | Average absolute error; more interpretable and robust to outliers | When interpretability and robustness matter |
| R^2 score | Regression | Proportion of variance explained by the model | To assess overall fit |

8.10 Exercises

The following exercises reinforce the core concepts of model evaluation introduced in this chapter. Start with conceptual questions to solidify your understanding, continue with hands-on tasks using the *bank* dataset to apply evaluation techniques in practice, and finish with critical thinking and reflection prompts to connect metrics to real-world decision-making.

Conceptual Questions

1. Why is model evaluation important in machine learning?
2. Explain the difference between training accuracy and test accuracy.

3. What is a confusion matrix, and why is it useful?
4. How does the choice of the positive class impact evaluation metrics?
5. What is the difference between sensitivity and specificity?
6. When would you prioritize sensitivity over specificity? Provide an example.
7. What is precision, and how does it differ from recall?
8. Why do we use the F1-score instead of relying solely on accuracy?
9. Explain the trade-off between precision and recall. How does changing the classification threshold impact them?
10. What is an ROC curve, and how does it help compare different models?
11. What does the Area Under the Curve (AUC) represent? How do you interpret different AUC values?
12. How can adjusting classification thresholds optimize model performance for a specific business need?
13. Why is accuracy often misleading for imbalanced datasets? What alternative metrics can be used?
14. What are macro-average and micro-average F1-scores, and when should each be used?
15. Explain how multi-class classification evaluation differs from binary classification.
16. What is Mean Squared Error (MSE), and why is it used in regression models?
17. How does Mean Absolute Error (MAE) compare to MSE? When would you prefer one over the other?
18. What is the R^2 score in regression, and what does it indicate?
19. Can an R^2 score be negative? What does it mean if this happens?
20. Why is it important to evaluate models using multiple metrics instead of relying on a single one?

Hands-On Practice: Evaluating Models with the bank Dataset

For these exercises, we will use the *bank* dataset from the **liver** package. This dataset contains information on customer demographics and financial

details, with the target variable *deposit* indicating whether a customer subscribed to a term deposit. It reflects a typical customer decision-making problem, making it ideal for practicing classification evaluation.

Load the necessary package and dataset:

```
library(liver)

# Load the dataset
data(bank)

# View the structure of the dataset
str(bank)
'data.frame': 4521 obs. of 17 variables:
 $ age      : int 30 33 35 30 59 35 36 39 41 43 ...
 $ job      : Factor w/ 12 levels "admin.", "blue-collar", ...: 11 8 5 5 2 5
   ↵ 7 10 3 8 ...
 $ marital  : Factor w/ 3 levels "divorced", "married", ...: 2 2 3 2 2 3 2 2
   ↵ 2 2 ...
 $ education: Factor w/ 4 levels "primary", "secondary", ...: 1 2 3 3 2 3 3 2
   ↵ 3 1 ...
 $ default  : Factor w/ 2 levels "no", "yes": 1 1 1 1 1 1 1 1 1 1 ...
 $ balance  : int 1787 4789 1350 1476 0 747 307 147 221 -88 ...
 $ housing  : Factor w/ 2 levels "no", "yes": 1 2 2 2 2 1 2 2 2 2 ...
 $ loan     : Factor w/ 2 levels "no", "yes": 1 2 1 2 1 1 1 1 1 2 ...
 $ contact  : Factor w/ 3 levels "cellular", "telephone", ...: 1 1 1 3 3 1 1
   ↵ 1 3 1 ...
 $ day      : int 19 11 16 3 5 23 14 6 14 17 ...
 $ month    : Factor w/ 12 levels "apr", "aug", "dec", ...: 11 9 1 7 9 4 9 9 9
   ↵ 1 ...
 $ duration : int 79 220 185 199 226 141 341 151 57 313 ...
 $ campaign : int 1 1 1 4 1 2 1 2 2 1 ...
 $ pdays    : int -1 339 330 -1 -1 176 330 -1 -1 147 ...
 $ previous : int 0 4 1 0 0 3 2 0 0 2 ...
 $ poutcome : Factor w/ 4 levels "failure", "other", ...: 4 1 1 4 4 1 2 4 4 1
   ↵ ...
 $ deposit  : Factor w/ 2 levels "no", "yes": 1 1 1 1 1 1 1 1 1 1 ...
```

Data Preparation

21. Load the *bank* dataset and identify the target variable and predictor variables.
22. Check for class imbalance in the target variable (*deposit*). How many customers subscribed to a term deposit versus those who did not?
23. Apply one-hot encoding to categorical variables using `one.hot()`.
24. Partition the dataset into 80% training and 20% test sets using `partition()`.

25. Validate the partitioning by comparing the class distribution of *deposit* in the training and test sets.
26. Apply min-max scaling to numerical variables to ensure fair distance calculations in kNN models.

Model Training and Evaluation

27. Train a kNN model using the training set and predict *deposit* for the test set.
28. Generate a confusion matrix for the test set predictions using `conf.mat()`. Interpret the results.
29. Compute the accuracy, sensitivity, and specificity of the kNN model.
30. Calculate precision, recall, and the F1-score for the model.
31. Use `conf.mat.plot()` to visualize the confusion matrix.
32. Experiment with different values of k (e.g., 3, 7, 15), compute evaluation metrics for each, and plot one or more metrics to visually compare performance.
33. Plot the ROC curve for the kNN model using the **pROC** package.
34. Compute the AUC for the model using the `auc()` function. What does the value indicate about performance?
35. Adjust the classification threshold (e.g., from 0.5 to 0.7) using the `cutoff` argument in `conf.mat()`. How does this impact sensitivity and specificity?

Critical Thinking and Real-World Applications

36. Suppose a bank wants to minimize false positives (incorrectly predicting a customer will subscribe). How should the classification threshold be adjusted?
37. If detecting potential subscribers is the priority, should the model prioritize precision or recall? Why?
38. If the dataset were highly imbalanced, what strategies could be used to improve model evaluation?

39. Consider a fraud detection system where false negatives (missed fraud cases) are extremely costly. How would you adjust the evaluation approach?
40. Imagine you are comparing two models: one has high accuracy but low recall, and the other has slightly lower accuracy but high recall. How would you decide which to use, and what contextual factors matter?
41. If a new marketing campaign resulted in a large increase in term deposit subscriptions, how might that affect the evaluation metrics?
42. Given the evaluation results from your model, what business recommendations would you make to a financial institution?

Self-Reflection

43. Which evaluation metric do you find most intuitive, and why?
44. Were there any metrics that initially seemed confusing or counterintuitive? How did your understanding change as you applied them?
45. In your own field or area of interest, what type of misclassification would be most costly? How would you design an evaluation strategy to minimize it?
46. How does adjusting the classification threshold shift your view of what makes a “good” model?
47. If you were to explain model evaluation to a non-technical stakeholder, what three key points would you highlight?

Chapter 9

Naive Bayes Classifier

How can we make fast, reasonably accurate predictions—using minimal data and computation? Imagine a bank deciding, in real time, whether to approve a loan based on a customer’s income, age, and mortgage status. Behind the scenes, such decisions must be made quickly, reliably, and at scale. The Naive Bayes classifier offers a remarkably simple yet surprisingly effective solution, relying on probability theory to make informed predictions in milliseconds.

In Chapter 7, we introduced *k-Nearest Neighbors* (kNN), a model that classifies based on similarity in feature space. In Chapter 8, we learned how to assess model performance using confusion matrices, sensitivity, specificity, ROC curves, and other evaluation metrics. Now, we turn to a fundamentally different approach: *Naive Bayes*, a *probabilistic* classifier grounded in Bayesian theory. Unlike kNN, which has no formal training phase, Naive Bayes builds a model from the data—estimating how likely each class is, given the features. It produces not just decisions but class probabilities, which integrate seamlessly with the evaluation tools we introduced earlier. This chapter gives us the chance to apply those tools while exploring a new perspective on classification.

At its core, Naive Bayes is built on *Bayes’ theorem* and makes a bold simplifying assumption: that all features are conditionally independent given the class label. This assumption is rarely true in practice—yet the model often works surprisingly well. Why? Because it enables fast training, efficient probability estimation, and interpretable outputs. The algorithm is especially well suited to high-dimensional data, such as text classification, where thousands of features are common. It is also ideal for real-time tasks like spam filtering or financial risk scoring, where speed and simplicity matter.

But every model has trade-offs. Naive Bayes assumes feature independence—an assumption often violated when features are strongly correlated. It also does not naturally handle continuous features unless a

specific distribution (often Gaussian) is assumed, which may misrepresent the data. And while it performs well in many scenarios, more flexible models—like decision trees or ensemble methods—can outperform it on datasets with complex feature interactions.

Despite these limitations, Naive Bayes remains a favorite in many real-world applications. In domains such as sentiment analysis, email filtering, and document classification, its assumptions hold well enough, and its simplicity becomes an asset. It is fast, easy to implement, and interpretable—qualities that make it a strong first-choice model and a valuable baseline in the early stages of model development.

The model’s power comes from its foundation in *Bayesian probability*, specifically *Bayes’ Theorem*, introduced by the 18th-century statistician Thomas Bayes (Bayes 1958). This theorem offers a principled way to update beliefs in light of new data—combining prior knowledge with observed evidence. It remains one of the most influential ideas in both statistics and machine learning.

What This Chapter Covers

This chapter explores how the Naive Bayes classifier leverages probability to make fast, interpretable predictions—even in high-dimensional and sparse settings. You will deepen your conceptual understanding of Bayesian reasoning while gaining hands-on experience implementing Naive Bayes models in R.

In particular, you will:

- Understand the mathematical foundation of Naive Bayes, including Bayes’ theorem and its role in classification.
- Work through step-by-step examples to see how the model estimates class probabilities from training data.
- Compare the main variants of Naive Bayes—Gaussian, Multinomial, and Bernoulli—and identify when each is appropriate.
- Analyze key assumptions, strengths, and limitations of the model in practical scenarios.
- Implement and evaluate a Naive Bayes model using the *risk* dataset from the **liver** package.

By the end of this chapter, you will be able to explain how Naive Bayes works, choose the right variant for a given task, and apply it effectively in

R. To begin, let us revisit the core principle that drives this classifier: Bayes' theorem.

9.1 Bayes' Theorem and Probabilistic Foundations

How should we update our beliefs when new evidence becomes available? Whether assessing financial risk, diagnosing medical conditions, or detecting spam, many real-world decisions require reasoning under uncertainty. Bayes' Theorem provides a formal, principled framework for refining probability estimates as new information emerges—making it a cornerstone of probabilistic learning and modern machine learning.

This framework underlies what is known as *Bayesian inference*: the process of starting with prior expectations based on historical data and updating them using new evidence to obtain a more accurate posterior belief. For example, when evaluating whether a loan applicant poses a financial risk, an institution might begin with general expectations derived from population statistics. As additional details—such as mortgage status or outstanding debts—are observed, Bayes' Theorem enables a systematic update of the initial risk assessment.

This idea traces back to Thomas Bayes, an 18th-century minister and self-taught mathematician. His pioneering work introduced a dynamic interpretation of probability—not merely as the frequency of outcomes, but as a belief that can evolve with new data. Readers interested in the broader implications of Bayesian reasoning may enjoy the book “[Everything Is Predictable: How Bayesian Statistics Explain Our World](#)”, which explores how this perspective informs real-life decisions.

Even earlier, the conceptual roots of probability theory developed from attempts to reason about chance in gambling, trade, and risk. In the 17th century, mathematicians such as Gerolamo Cardano, Blaise Pascal, and Pierre de Fermat laid the groundwork for formal probability theory. Cardano, for example, observed that some dice outcomes—such as sums of 9 versus 10—have unequal likelihoods due to differing numbers of permutations. These early insights into randomness and structure laid the intellectual foundation for modern approaches to modeling uncertainty, including the Naive Bayes classifier.

The Essence of Bayes' Theorem

Bayes' Theorem offers a systematic method for refining probabilistic beliefs as new evidence is observed, forming the theoretical foundation of Bayesian inference. It addresses a central question in probabilistic reasoning: *Given what is already known, how should our belief in a hypothesis change when new data are observed?*

The theorem is mathematically expressed as:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \quad (9.1)$$

Where:

- $P(A|B)$ is the *posterior probability*, the probability of event A (the hypothesis) given that event B (the evidence) has occurred.
- $P(A \cap B)$ is the *joint probability* that both events A and B occur.
- $P(B)$ is the *marginal probability* or *evidence*, quantifying the total probability of observing event B across all possible outcomes.

To better understand these components, Figure 9.1 offers a visual interpretation using a Venn diagram. The overlapping region of the two circles illustrates the joint probability $P(A \cap B)$, while the entire area of circle B represents $P(B)$.

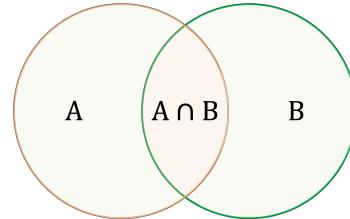


Figure 9.1: Venn diagram illustrating the joint and marginal probabilities in Bayes' Theorem.

The expression for Bayes' Theorem can also be derived by applying the definition of conditional probability. Specifically, $P(A \cap B)$ can be written as $P(A) \times P(B|A)$, leading to an alternative form:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = P(A) \times \frac{P(B|A)}{P(B)} \quad (9.2)$$

These equivalent expressions result from two ways of expressing the joint probability $P(A \cap B)$. This formulation highlights how a prior belief $P(A)$ is updated using the likelihood $P(B|A)$ and normalized by the marginal probability $P(B)$.

Bayes' Theorem thus provides a principled way to refine beliefs by incorporating new evidence. This principle underpins many probabilistic learning techniques, including the Naive Bayes classifier introduced in this chapter.

Let us now apply Bayes' Theorem to a practical example: estimating the probability that a customer has a good risk profile (A) given that they have a mortgage (B), using the `risk` dataset from the `liver` package.

We begin by loading the dataset and inspecting the relevant data:

```
library(liver)

data(risk)

xtabs(~ risk + mortgage, data = risk)
      mortgage
      risk   yes no
    good risk  81 42
    bad risk   94 29
```

To improve readability, we add row and column totals to the contingency table:

```
addmargins(xtabs(~ risk + mortgage, data = risk))
      mortgage
      risk   yes no Sum
    good risk  81 42 123
    bad risk   94 29 123
      Sum     175 71 246
```

Now, we define the relevant events: A is the event that a customer has a *good risk* profile, and B is the event that the customer has a mortgage (`mortgage = yes`). The prior probability of a customer having good risk is:

$$P(A) = \frac{\text{Total Good Risk Cases}}{\text{Total Cases}} = \frac{123}{246} = 0.5$$

Using Bayes' Theorem, we compute the probability of a customer being classified as good risk given that they have a mortgage:

$$\begin{aligned}
 P(\text{Good Risk} | \text{Mortgage} = \text{Yes}) &= \frac{P(\text{Good Risk} \cap \text{Mortgage} = \text{Yes})}{P(\text{Mortgage} = \text{Yes})} \\
 &= \frac{\text{Good Risk with Mortgage Cases}}{\text{Total Mortgage Cases}} \quad (9.3) \\
 &= \frac{81}{175} \\
 &= 0.463
 \end{aligned}$$

This result indicates that among customers with mortgages, the observed proportion of those with a good risk profile is lower than in the general population. Such insights help financial institutions refine credit risk models by incorporating new evidence systematically.

How Does Bayes' Theorem Work?

Imagine you are deciding whether to approve a loan application. You begin with a general expectation—perhaps most applicants with steady income and low debt are low risk. But what happens when you learn that the applicant has missed several past payments? Your belief shifts. This type of evidence-based reasoning is precisely what Bayes' Theorem formalizes.

Bayes' Theorem provides a structured method to refine our understanding of uncertainty as new information becomes available. In everyday decisions—whether assessing financial risk or evaluating the results of a medical test—we often begin with an initial belief and revise it in light of new evidence.

Bayesian reasoning plays a central role in many practical applications. In *financial risk assessment*, banks typically begin with prior expectations about borrower profiles, and then revise the risk estimate after considering additional information such as income, credit history, or mortgage status. In *medical diagnostics*, physicians assess the baseline probability of a condition and then update that estimate based on test results, incorporating both prevalence and diagnostic accuracy. In *spam detection*, email filters estimate the probability that a message is spam using features such as keywords, sender information, and formatting—and continually refine those estimates as new messages are processed.

Can you think of a situation where you made a decision based on initial expectations, but changed your mind after receiving new information? That shift in belief is the intuition behind Bayesian updating. Bayes' Theorem turns this intuition into a formal rule. It offers a principled mechanism for learning from data—one that underpins many modern tools for prediction and classification.

From Bayes’ Theorem to Naive Bayes

Bayes’ Theorem provides a mathematical foundation for updating probabilities as new evidence emerges. However, directly applying Bayes’ Theorem to problems involving many features becomes impractical, as it requires estimating a large number of joint probabilities from data—many of which may be sparse or unavailable.

The Naive Bayes classifier addresses this challenge by introducing a simplifying assumption: it treats all features as *conditionally independent* given the class label. While this assumption rarely holds exactly in real-world datasets, it dramatically simplifies the required probability calculations.

Despite its simplicity, Naive Bayes often delivers competitive results. For example, in financial risk prediction, a bank may evaluate a customer’s creditworthiness using multiple variables such as income, loan history, and mortgage status. Although these variables are often correlated, the independence assumption enables the classifier to estimate probabilities efficiently by breaking the joint distribution into simpler, individual terms.

This efficiency is particularly advantageous in domains like text classification, spam detection, and sentiment analysis, where the number of features can be very large and independence is a reasonable approximation.

Why does such a seemingly unrealistic assumption often work so well in practice? As we will see, this simplicity allows Naive Bayes to serve as a fast, interpretable, and surprisingly effective classifier—even in complex real-world settings.

9.2 Why Is It Called “Naive”?

When assessing a borrower’s financial risk using features such as income, mortgage status, and number of loans, it is reasonable to expect dependencies among them. For example, individuals with higher income may be more likely to have multiple loans or stable mortgage histories. However, Naive Bayes assumes that all features are conditionally independent given the class label (e.g., “good risk” or “bad risk”).

This simplifying assumption is what gives the algorithm its name. While features in real-world data are often correlated—such as income and age—assuming independence significantly simplifies probability calculations, making the method both efficient and scalable.

To illustrate this, consider the `risk` dataset from the `liver` package:

```

str(risk)
'data.frame': 246 obs. of 6 variables:
 $ age      : int 34 37 29 33 39 28 28 25 41 26 ...
 $ marital   : Factor w/ 3 levels "single","married",...: 3 3 3 3 3 3 3 3 3 3 ...
 $ ...       :
 $ income    : num 28061 28009 27615 27287 26954 ...
 $ mortgage  : Factor w/ 2 levels "yes","no": 1 2 2 1 1 2 2 2 2 2 ...
 $ nr.loans  : int 3 2 2 2 2 2 3 2 2 2 ...
 $ risk      : Factor w/ 2 levels "good risk","bad risk": 2 2 2 2 2 2 2 2 2 2 ...
 $ ...       :

```

This dataset includes financial indicators such as age, income, marital status, mortgage, and number of loans. Naive Bayes assumes that, given a person's risk classification, these features do not influence one another. Mathematically, the probability of a customer being in the good risk category given their attributes is expressed as:

$$P(Y = y_1 | X_1, X_2, \dots, X_5) = \frac{P(Y = y_1) \times P(X_1, X_2, \dots, X_5 | Y = y_1)}{P(X_1, X_2, \dots, X_5)}$$

Mathematically, computing the full joint likelihood of all features given a class label is challenging. Directly computing $P(X_1, X_2, \dots, X_5 | Y = y_1)$ is computationally expensive, especially as the number of features grows. In datasets with hundreds or thousands of features, storing and calculating joint probabilities for all possible feature combinations becomes impractical.

The naive assumption of conditional independence simplifies this problem by expressing the joint probability as the product of individual probabilities:

$$P(X_1, X_2, \dots, X_5 | Y = y_1) = P(X_1 | Y = y_1) \times P(X_2 | Y = y_1) \times P(X_5 | Y = y_1)$$

This transformation eliminates the need to compute complex joint probabilities, making the algorithm scalable even for high-dimensional data. Instead of handling an exponential number of feature combinations, Naive Bayes only requires computing simple conditional probabilities for each feature given the class label.

In practice, the independence assumption is rarely true, as features often exhibit some degree of correlation. Nevertheless, Naive Bayes remains widely used in domains where feature dependencies are sufficiently weak to preserve classification accuracy, where interpretability and computational efficiency are prioritized over capturing complex relationships, and where minor violations of the independence assumption do not substantially degrade predictive performance.

For example, in credit risk prediction, while income and mortgage status are likely correlated, treating them as independent still allows Naive Bayes to classify borrowers effectively. Similarly, in spam detection or text classification, where features (such as word occurrences) are often close to independent, the algorithm delivers fast and accurate predictions.

By reducing complex joint probability estimation to simpler conditional calculations, Naive Bayes offers a scalable solution. In the next section, we address a key practical issue: how to handle zero-probability problems when certain feature values are absent in the training data.

9.3 The Laplace Smoothing Technique

One challenge in Naive Bayes classification is handling feature values that appear in the test data but are missing from the training data for a given class. For example, suppose no borrowers labeled as “bad risk” are married in the training data. If a married borrower later appears in the test set, Naive Bayes would assign a probability of zero to $P(\text{bad risk}|\text{married})$. Because the algorithm multiplies probabilities when making predictions, this single zero would eliminate the bad risk class from consideration—leading to a biased or incorrect prediction.

This issue arises because Naive Bayes estimates conditional probabilities directly from frequency counts in the training set. If a category is absent for a class, its conditional probability becomes zero. To address this, *Laplace smoothing* (or *add-one smoothing*) is used. Named after [Pierre-Simon Laplace](#), the technique assigns a small non-zero probability to every possible feature-class combination, even if some combinations do not appear in the data.

To illustrate, consider the marital variable in the risk dataset. Suppose no customers labeled as bad risk are married. We can simulate this scenario:

| | risk | |
|---------|------|-----|
| marital | good | bad |
| single | 21 | 11 |
| married | 51 | 0 |
| other | 8 | 10 |

Without smoothing, the conditional probability becomes:

$$P(\text{bad risk}|\text{married}) = \frac{\text{count}(\text{bad risk} \cap \text{married})}{\text{count}(\text{married})} = \frac{0}{\text{count}(\text{married})} = 0$$

This would cause every married borrower to be classified as good `risk`, regardless of other features.

Laplace smoothing resolves this by adjusting the count of each category. A small constant k (typically $k = 1$) is added to each count, yielding:

$$P(\text{bad risk}|\text{married}) = \frac{\text{count}(\text{bad risk} \cap \text{married}) + k}{\text{count}(\text{bad risk}) + k \times \text{number of marital categories}}$$

This adjustment ensures that every possible feature-category pair has a non-zero probability, even if unobserved in the training set.

In **R**, you can apply Laplace smoothing using the `laplace` argument in the **naivebayes** package. By default, no smoothing is applied (`laplace = 0`). To apply smoothing, simply set `laplace = 1`:

```
library(naivebayes)

formula_nb = risk ~ age + income + marital + mortgage + nr.loans

model <- naive_bayes(formula = formula_nb, data = risk, laplace = 1)
```

This adjustment improves model robustness, especially when working with limited or imbalanced data. Curious to see how the **naivebayes** package performs in practice? In the case study later in this chapter, we will walk through how to train and evaluate a Naive Bayes model using the `risk` dataset—complete with **R** code, predicted probabilities, and performance metrics.

Laplace smoothing is a simple yet effective fix for the zero-probability problem in Naive Bayes. While $k = 1$ is a common default, the value can be tuned based on domain knowledge. By ensuring that all probabilities remain well-defined, Laplace smoothing makes Naive Bayes more reliable for real-world prediction tasks.

9.4 Types of Naive Bayes Classifiers

What if your dataset includes text, binary flags, and numeric values? Can a single Naive Bayes model accommodate them all? Not exactly. Different types of features require different probabilistic assumptions—this is where distinct variants of the Naive Bayes classifier come into play. The choice of variant depends on the structure and distribution of the predictors in your data.

Each of the three most common types of Naive Bayes classifiers is suited to a specific kind of feature:

- *Multinomial Naive Bayes* is designed for categorical or count-based features, such as word frequencies in text data. It models the probability of counts using a multinomial distribution. In the `risk` dataset, the `marital` variable—with levels such as `single`, `married`, and `other`—is an example where this variant is appropriate.
- *Bernoulli Naive Bayes* is intended for binary features that capture the presence or absence of a characteristic. This approach is common in spam filtering, where features often indicate whether a particular word is present. In the `risk` dataset, the binary `mortgage` variable (yes or no) fits this model.
- *Gaussian Naive Bayes* is used for continuous features that are assumed to follow a normal distribution. It models feature likelihoods using Gaussian densities and is well suited for variables like `age` and `income` in the `risk` dataset.

Selecting the appropriate variant based on your feature types ensures that the underlying probability assumptions remain valid and that the model produces reliable predictions.

The names *Bernoulli* and *Gaussian* refer to foundational distributions introduced by two prominent mathematicians: *Jacob Bernoulli*, known for early work in probability theory, and *Carl Friedrich Gauss*, associated with the normal distribution. Their contributions form the statistical backbone of different Naive Bayes variants.

In the next section, we apply Naive Bayes to the `risk` dataset and explore how these variants operate in practice.

9.5 Case Study: Predicting Financial Risk with Naive Bayes

How can a bank predict in advance whether an applicant is likely to repay a loan, or default, before making a lending decision? This is a daily challenge for financial institutions, where each loan approval carries both potential profit and risk. Making accurate predictions about creditworthiness helps banks protect their assets, comply with regulatory standards, and promote responsible lending practices.

In this case study, we apply the complete *Data Science Workflow* introduced in Chapter 2 (Figure 2.3), following each step, from understanding the problem and preparing the data to training, evaluating, and interpreting the model.

Using the *risk* dataset from the [liver](#) package in R, we build a Naive Bayes classifier to categorize customers as either *good risk* or *bad risk*. By walking through the workflow step-by-step, this example demonstrates how probabilistic classification can guide credit decisions and help institutions manage financial risk in a structured, data-driven manner.

Problem Understanding

How can financial institutions anticipate which applicants are likely to repay their loans and which may default before extending credit? This challenge lies at the heart of modern lending practices. Effective financial risk assessment requires balancing profitability with caution by using demographic and financial indicators to estimate the likelihood of default.

This case study builds on earlier chapters: Chapter [7](#) introduced classification with instance-based methods, and Chapter [8](#) covered how to assess model performance. We now extend these foundations by applying a probabilistic classification technique—Naive Bayes—to a real-world dataset.

Key business questions guiding this analysis include:

- Which financial and demographic features influence a customer's risk profile?
- How can we predict a customer's risk category before making a loan decision?
- In what ways can such predictions support more effective lending strategies?

By analyzing the *risk* dataset, we aim to develop a model that classifies customers as *good risk* or *bad risk* based on their likelihood of default. The results can inform data-driven credit scoring, guide responsible lending practices, and reduce non-performing loans.

Data Understanding

Before training a classification model, we begin by exploring the dataset to assess the structure of the variables, identify key distributions, and check for any anomalies that might affect modeling. As introduced earlier in Section

[9.2](#), the `risk` dataset from the `liver` package contains financial and demographic attributes used to assess whether a customer is a *good risk* or *bad risk*. It includes 246 observations across 6 variables.

The dataset consists of 5 predictors and a binary target variable, `risk`, which distinguishes between customers who are more or less likely to default. The key variables are:

- `age`: Customer's age in years.
- `marital`: Marital status (`single`, `married`, `other`).
- `income`: Annual income.
- `mortgage`: Indicates whether the customer has a mortgage (yes, no).
- `nr_loans`: Number of loans held by the customer.
- `risk`: The target variable (`good risk`, `bad risk`).

For additional details about the dataset, refer to its [documentation](#).

To obtain an overview of the variable distributions and check for missing values or outliers, we examine the dataset's summary statistics:

```
summary(risk)
   age          marital      income      mortgage      nr.loans
Min. :17.00    single :111    Min. :15301    yes:175    Min. :0.000
1st Qu.:32.00   married: 78   1st Qu.:26882   no : 71    1st Qu.:1.000
Median :41.00   other  : 57   Median :37662           Median :1.000
Mean   :40.64   Mean    :38790           Mean   :1.309
3rd Qu.:50.00   3rd Qu.:49398          3rd Qu.:2.000
Max.  :66.00    Max.   :78399           Max.  :3.000
risk
good risk:123
bad risk :123
```

As the summary indicates a clean and well-structured dataset with no apparent anomalies, we can proceed to data preparation before training the Naive Bayes classifier.

Data Setup for Modeling

Before training the Naive Bayes classifier, we begin by splitting the dataset into training and testing sets. This step allows us to evaluate how well the model generalizes to unseen data. We use an 80/20 split, allocating 80% of

the data for training and 20% for testing. To maintain consistency with previous chapters, we apply the `partition()` function from the `liver` package:

```
set.seed(5)

data_sets = partition(data = risk, ratio = c(0.8, 0.2))

train_set = data_sets$part1
test_set = data_sets$part2

test_labels = test_set$risk
```

Setting `set.seed(5)` ensures reproducibility so that the same partitioning is achieved each time the code is run. The `train_set` will be used to train the Naive Bayes classifier, while the `test_set` will serve as unseen data to evaluate the model's predictions. The `test_labels` vector contains the true class labels for the test set, which we will compare against the model's outputs.

As discussed in Section 6.5, it is important to check whether the training and test sets are representative of the original dataset. This can be done by comparing the distribution of the target variable or key predictors. Here, we illustrate the process by validating the `marital` variable across the two sets. As an exercise, you are encouraged to validate the partition based on the target variable `risk` to confirm that both classes—*good risk* and *bad risk*—are similarly distributed.

To check for representativeness, we use a chi-squared test to compare the distribution of marital statuses (`single`, `married`, `other`) in the training and test sets:

```
chisq.test(x = table(train_set$marital), y = table(test_set$marital))

Pearson's Chi-squared test

data: table(train_set$marital) and table(test_set$marital)
X-squared = 6, df = 4, p-value = 0.1991
```

This test evaluates whether the proportions of marital categories differ significantly between the two sets. The hypotheses are:

$$\begin{cases} H_0 : \text{The proportions of marital categories are the same in both sets.} \\ H_a : \text{At least one of the proportions is different.} \end{cases}$$

Since the p-value exceeds $\alpha = 0.05$, we fail to reject H_0 . This suggests that the marital status distribution is statistically similar between the training and test sets, indicating that the partition preserves the key structure of the dataset.

Unlike distance-based algorithms such as k-nearest neighbors, the Naive Bayes classifier does not rely on geometric distance calculations. Therefore, there is no need to scale numeric variables such as `age` or `income`, and no need to convert categorical variables like `marital` into dummy variables. The algorithm models probability distributions directly, making it robust to different variable types without requiring transformation. This illustrates how preprocessing steps must be tailored to the modeling technique in use.

In contrast, when applying kNN to this dataset (see Chapter 7), it would be necessary to scale numerical variables and encode categorical variables. These considerations are explored further in this chapter's exercises.

Applying the Naive Bayes Classifier

With the dataset partitioned and validated, we now proceed to train and evaluate the Naive Bayes classifier. This model is particularly well suited to problems like credit risk assessment because it is fast, interpretable, and effective even when variables are a mix of numerical and categorical types.

Several R packages provide implementations of Naive Bayes, with two commonly used options being `naivebayes` and `e1071`. In this case study, we use the `naivebayes` package, which offers a fast and flexible implementation that supports both categorical and continuous features.

The core function, `naive_bayes()`, estimates the required probability distributions during training and stores them in a model object. Based on the types of the predictors, the algorithm makes the following assumptions:

- *Categorical distributions* for nominal variables such as `marital` and `mortgage`;
- *Bernoulli distributions* for binary variables, which are a special case of categorical features;
- *Poisson distributions* for count variables (optionally enabled);
- *Gaussian distributions* for continuous features such as `age` and `income`;
- *Kernel density estimation* for continuous features when no parametric form is assumed.

Unlike the k-NN algorithm introduced in Chapter 7, which does not include an explicit training phase, Naive Bayes follows a two-step procedure:

1. *Training phase*: The model estimates class-conditional probability distributions from the training data.

2. *Prediction phase:* The trained model applies Bayes' theorem to compute posterior probabilities for new observations.

To train the model, we specify a formula where `risk` is the target variable and all other columns are treated as predictors:

```
formula = risk ~ age + income + mortgage + nr.loans + marital
```

We then fit the model using the `naive_bayes()` function:

```
library(naivebayes)

naive_bayes = naive_bayes(formula, data = train_set)

naive_bayes

===== Naive Bayes
↪ =====

Call:
naive_bayes.formula(formula = formula, data = train_set)

-----
↪ ----

Laplace smoothing: 0

-----
↪ ----

A priori probabilities:

good risk  bad risk
0.4923858 0.5076142

-----
↪ ----

Tables:

-----
↪ ----

:: age (Gaussian)
-----
↪ ----

age    good risk  bad risk
mean 46.453608 35.470000
sd    8.563513  9.542520
```

```

:: income (Gaussian)
-----
↪ ----

income good risk bad risk
mean 48888.987 27309.560
sd   9986.962  7534.639

-----
↪ ----
:: mortgage (Bernoulli)
-----
↪ ----

mortgage good risk bad risk
yes 0.6804124 0.7400000
no  0.3195876 0.2600000

-----
↪ ----
:: nr.loans (Gaussian)
-----
↪ ----

nr.loans good risk bad risk
mean 1.0309278 1.6600000
sd   0.7282057 0.7550503

-----
↪ ----
:: marital (Categorical)
-----
↪ ----

marital   good risk bad risk
single   0.38144330 0.49000000
married  0.52577320 0.11000000
other    0.09278351 0.40000000

```

This function automatically identifies the feature types and estimates appropriate probability distributions for each class. For instance:

- *Categorical features* (e.g., `marital`, `mortgage`) are modeled using class-conditional probabilities.
- *Numerical features* (e.g., `age`, `income`, `nr.loans`) are modeled using Gaussian distributions by default.

To inspect the learned parameters, we can use:

```
summary(naive_bayes)

=====
Naive Bayes
=====

- Call: naive_bayes(formula = formula, data = train_set)
- Laplace: 0
- Classes: 2
- Samples: 197
- Features: 5
- Conditional distributions:
  - Bernoulli: 1
  - Categorical: 1
  - Gaussian: 3
- Prior probabilities:
  - good risk: 0.4924
  - bad risk: 0.5076

=====
```

This summary shows the estimated means and standard deviations for numerical predictors and the conditional probabilities for categorical ones. These form the foundation of the model's predictions.

Note that the `nr.loans` variable is a count with values such as 0, 1, and 3. While the default setting uses a Gaussian distribution, it may be worth experimenting with the `usepoisson = TRUE` option to see whether a Poisson distribution offers a better fit. As an exercise, you are encouraged to compare model performance with and without this option.

Prediction and Model Evaluation

With the Naive Bayes classifier trained, we now evaluate its performance by applying it to the test set—data that was not used during training. The objective is to compare the model's predicted class probabilities against the actual outcomes stored in `test_labels`.

To generate predicted probabilities for each class, we use the `predict()` function from the **naivebayes** package, setting `type = "prob"` to return posterior probabilities instead of hard class labels:

```
prob_naive_bayes = predict(naive_bayes, test_set, type = "prob")
```

To explore the output, we display the first 6 rows and round the values to three decimal places:

```
round(head(prob_naive_bayes, n = 6), 3)
      good risk bad risk
[1,] 0.001 0.999
[2,] 0.013 0.987
[3,] 0.000 1.000
[4,] 0.184 0.816
[5,] 0.614 0.386
[6,] 0.193 0.807
```

The resulting matrix contains two columns: the first shows the predicted probability that a customer belongs to the “good risk” class, while the second shows the probability of being in the “bad risk” class. For example, if a customer receives a high probability for “bad risk,” it suggests that the model considers them more likely to default.

Rather than relying on a fixed decision threshold (such as 0.5), the model’s probabilities can be mapped to class labels using a threshold selected according to specific business needs. In the next subsection, we convert these probabilities into class predictions and evaluate performance using a confusion matrix and additional metrics.

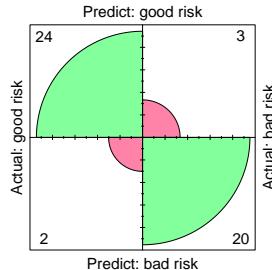
Confusion Matrix

To assess the classification performance of the Naive Bayes model, we compute a confusion matrix using the `conf.mat()` and `conf.mat.plot()` functions from the `liver` package:

```
# Extract probability of "good risk"
prob_naive_bayes = prob_naive_bayes[, 1]

conf.mat(prob_naive_bayes, test_labels, cutoff = 0.5, reference = "good
← risk")
      Actual
Predict      good risk bad risk
  good risk        24     3
  bad risk         2    20

conf.mat.plot(prob_naive_bayes, test_labels, cutoff = 0.5, reference = "good
← risk")
```



We apply a threshold of 0.5, classifying an observation as “good risk” if its predicted probability for that class exceeds 50%. The reference class is “good risk”, meaning that metrics such as sensitivity and precision are computed relative to this category.

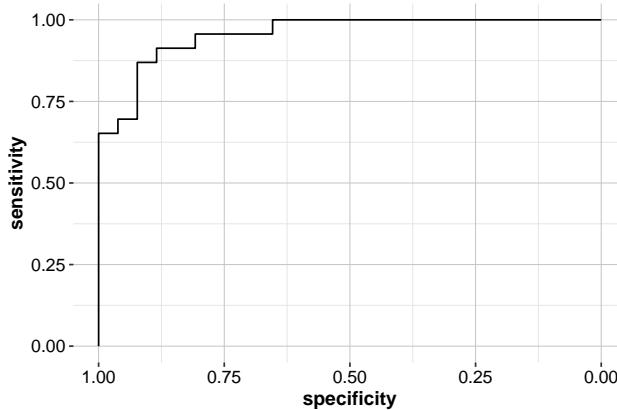
The resulting confusion matrix summarizes the model’s predictions compared to the actual outcomes, highlighting both correct classifications and misclassifications. For example, the matrix may indicate that 24 customers were correctly classified as “good risk” and 20 as “bad risk”, while 3 “bad risk” cases were misclassified as “good risk” and 2 “good risk” cases were misclassified as “bad risk”.

Want to explore the effect of changing the classification threshold? Try setting the cutoff to values such as 0.4 or 0.6 to examine how sensitivity, specificity, and overall accuracy shift under different decision criteria.

ROC Curve and AUC

To complement the confusion matrix, we use the *Receiver Operating Characteristic (ROC) curve* and the *Area Under the Curve (AUC)* to evaluate the classifier’s performance across all possible classification thresholds. While the confusion matrix reflects accuracy at a fixed cutoff (e.g., 0.5), ROC analysis provides a more flexible, threshold-agnostic view of model performance.

```
library(pROC)
roc_naive_bayes = roc(test_labels, prob_naive_bayes)
ggroc(roc_naive_bayes)
```



The ROC curve plots the *true positive rate* (sensitivity) against the *false positive rate* (1 - specificity) at various thresholds. A curve that bows toward the top-left corner indicates strong discriminative performance, reflecting a high sensitivity with a low false positive rate.

Next, we compute the AUC score:

```
round(auc(roc_naive_bayes), 3)
[1] 0.957
```

The AUC value, 0.957, quantifies the model's ability to distinguish between the two classes. Specifically, it represents the probability that a randomly selected "good risk" customer will receive a higher predicted probability than a randomly selected "bad risk" customer. An AUC of 1 indicates perfect separation, while an AUC of 0.5 reflects no discriminative power beyond random guessing.

Together, the ROC curve and AUC offer a comprehensive assessment of model performance, independent of any particular decision threshold. In the final section of this case study, we reflect on the model's practical strengths and limitations.

Takeaways from the Case Study

This case study illustrated how the Naive Bayes classifier can support financial risk assessment by classifying customers as *good risk* or *bad risk* based on demographic and financial attributes. Using tools such as the confusion matrix, ROC curve, and AUC, we evaluated the model's accuracy and ability to guide lending decisions.

Naive Bayes offers several practical advantages. Its simplicity and computational efficiency make it well suited for real-time decision-making. Despite its strong independence assumption, the algorithm often performs competitively, especially in high-dimensional settings or when feature correlations are weak. Moreover, the ability to output class probabilities allows institutions to adjust classification thresholds based on specific business goals—such as prioritizing sensitivity to minimize default risk or specificity to avoid rejecting reliable applicants.

Nonetheless, the conditional independence assumption can limit performance when predictors are strongly correlated. This limitation can be addressed by incorporating additional features (e.g., credit history), using more flexible probabilistic models, or transitioning to ensemble methods such as random forests or boosting.

By applying Naive Bayes to a real-world dataset, we demonstrated how probabilistic classification can support data-driven credit policy. Models like this help financial institutions strike a balance between risk management and fair lending practices.

Reflective prompt: How might this modeling approach transfer to other domains, such as healthcare or marketing? Could adjusting the classification threshold or selecting a different Naive Bayes variant improve outcomes in those settings? As you compare this method with others—such as k-nearest neighbors or logistic regression—consider when each model is most appropriate and why.

Chapter Summary and Takeaways

This chapter introduced the Naive Bayes classifier as a fast and interpretable approach to probabilistic classification. Grounded in Bayes' Theorem, the method estimates the likelihood that an observation belongs to a particular class, assuming conditional independence among features. While this assumption rarely holds exactly, Naive Bayes often performs surprisingly well in practice, especially in high-dimensional settings and text-based applications.

We examined three common variants—multinomial, Bernoulli, and Gaussian—each suited to different data types. Using the *risk* dataset, we applied Naive Bayes in R, evaluated its performance with confusion matrices, ROC curves, and AUC, and interpreted predicted probabilities to support threshold-based decisions.

Key takeaways:

- Naive Bayes is computationally efficient and scalable, making it well-suited for real-time applications.

- It offers transparent probabilistic outputs, enabling flexible decision-making and threshold adjustment.
- The model performs robustly even when the independence assumption is only approximately satisfied.

While this chapter focused on a generative probabilistic model, the next chapter introduces **logistic regression**, a discriminative linear model that estimates the log-odds of class membership. Logistic regression provides a useful complement to Naive Bayes, particularly when modeling predictor relationships and interpreting coefficients are central to the analysis.

9.6 Exercises

This section reinforces your understanding of Naive Bayes through conceptual questions, applied tasks, and real-world scenarios.

Conceptual Questions

1. Why is Naive Bayes considered a probabilistic classification model?
2. What is the difference between prior probability, likelihood, and posterior probability in Bayes' theorem?
3. What does it mean when we say Naive Bayes assumes feature independence?
4. In which situations does the feature independence assumption become problematic? Provide an example.
5. What are the key strengths of Naive Bayes? Why is it widely used in text classification and spam filtering?
6. What are the major limitations of Naive Bayes, and how do they impact its performance?
7. How does Laplace smoothing help in handling missing feature values in Naive Bayes? *Hint: See Section 9.3 for how smoothing helps prevent zero probabilities.*
8. When should you use multinomial Naive Bayes, Bernoulli Naive Bayes, or Gaussian Naive Bayes? *Hint: See Section 9.4 and consider feature types and their distributions.*

9. Compare the Naive Bayes classifier to the k-Nearest Neighbors algorithm (Chapter 7). How do their assumptions and outputs differ?
10. How does changing the probability threshold influence the predicted classes and performance metrics?
11. Why does Naive Bayes remain effective even when the independence assumption is violated?
12. What type of dataset characteristics make Naive Bayes perform poorly compared to other classifiers?
13. How does the Gaussian Naive Bayes classifier handle continuous data?
14. How can domain knowledge help improve Naive Bayes classification results?
15. How would Naive Bayes handle imbalanced datasets? What preprocessing techniques could help?
16. Explain how prior probabilities can be adjusted based on business objectives in a classification problem.

Hands-on Implementation with the Churn Dataset

For the following exercises, we will use the *churn* dataset from the **liver** package. This dataset contains information about customer subscriptions, and our goal is to predict whether a customer will churn (*churn* = yes/no) using the Naive Bayes classifier. In Section 4.3, we performed exploratory data analysis to understand the dataset's structure and key features.

Data Preparation

17. Load the **liver** package and the *churn* dataset:

```
library(liver)  
data(churn)
```

18. Display the structure and summary statistics of the dataset to examine its variables and their distributions.
19. Split the dataset into an 80% training set and a 20% test set using the `partition()` function from the **liver** package.

20. Confirm that the training and test sets have similar distributions of the `churn` variable by comparing proportions.

Training and Evaluating the Naive Bayes Classifier

21. Based on the exploratory data analysis in Section 4.3, select the following predictors for the Naive Bayes model: `account.length`, `voice.plan`, `voice.messages`, `intl.plan`, `intl.mins`, `day.mins`, `eve.mins`, `night.mins`, and `customer.calls`. Define the model formula:

```
formula = churn ~ account.length + voice.plan + voice.messages +
          intl.plan + intl.mins + day.mins + eve.mins +
          night.mins + customer.calls
```

22. Train a Naive Bayes classifier on the training set using the **naivebayes** package.
23. Summarize the trained model. What insights can you gain from the estimated class-conditional probabilities?
24. Use the trained model to predict class probabilities for the test set using the `predict()` function from the **naivebayes** package.
25. Extract and examine the first 10 probability predictions. Interpret what these values indicate about the likelihood of customer churn.
26. Compute the confusion matrix using the `conf.mat()` function with a classification threshold of 0.5. What does it reveal about prediction performance?
27. Visualize the confusion matrix using the `conf.mat.plot()` function from the **liver** package.
28. Compute key evaluation metrics, including accuracy, precision, recall, and F1-score, based on the confusion matrix.
29. Lower the classification threshold from 0.5 to 0.3 and recompute the confusion matrix. How does adjusting the threshold affect model performance?
30. Plot the ROC curve and compute the AUC value. What does the AUC tell you about the model's ability to distinguish between churn and non-churn customers?
31. Train a Naive Bayes model with Laplace smoothing (`laplace = 1`) and compare the results to the model without smoothing. How does smoothing affect predictions?

32. Compare the Naive Bayes classifier to the k-Nearest Neighbors algorithm (Chapter 7) trained on the same dataset. *Make sure both models use the same partitioning for fair comparison.* Evaluate their performance using accuracy, precision, recall, F1-score, and AUC. Which model performs better, and what factors might explain the differences in performance?
33. Experiment by removing one predictor variable at a time and retraining the model. How does this impact accuracy and other evaluation metrics?
34. Suppose the model performs unusually poorly on a subset of customers. How would you diagnose whether this is due to feature misrepresentation, class imbalance, or violations of the independence assumption?

Real-World Application and Critical Thinking

35. Suppose a telecommunications company wants to use this model to reduce customer churn. What business decisions could be made based on the model's predictions?
36. If incorrectly predicting a false negative (missed churner) is more costly than a false positive, how should the decision threshold be adjusted?
37. A marketing team wants to offer promotional discounts to customers predicted to churn. How would you use this model to target the right customers?
38. Suppose the dataset included a new feature: customer satisfaction score (on a scale from 1 to 10). How could this feature improve the model?
39. What steps would you take if the model performed poorly on new customer data?
40. Explain why feature independence may or may not hold in this dataset. How could feature correlation impact the model's reliability?
41. Would Naive Bayes be suitable for multi-class classification problems? If so, how would you extend this model to predict multiple churn reasons instead of just yes/no?
42. If given time-series data about customer interactions over months, would Naive Bayes still be appropriate? Why or why not?

Self-Reflection

43. In your own words, what are the key strengths and limitations of the Naive Bayes classifier?
44. How did the independence assumption shape the model's structure and influence your interpretation of the results?
45. Which stage of the case study—data preparation, training, or evaluation—most deepened your understanding of how Naive Bayes works in practice?
46. How confident are you in applying Naive Bayes to a new dataset containing both categorical and numerical variables?
47. If you were to extend this chapter, which topic would you explore further: smoothing techniques, alternative distributional assumptions, or methods for handling correlated features? Try one in a small experiment.
48. Compared to earlier models like kNN or logistic regression, when do you think Naive Bayes would be a preferable choice? What trade-offs are involved?

Chapter 10

Regression Analysis: Foundations and Applications

How can a company estimate the impact of digital ad spending on daily sales? How do age, income, and smoking habits relate to healthcare costs? Can we predict housing prices from a home's age, size, and location? These questions are central to regression analysis—one of the most powerful and widely used tools in data science. Regression models help us understand relationships between variables, uncover patterns, and make predictions grounded in evidence.

The roots of regression analysis can be traced back to the early 1700s, when Isaac Newton's method of fluxions laid the mathematical groundwork for continuous change—concepts that underpin modern optimization and calculus. The term *regression* was introduced by Sir Francis Galton in 1886 to describe how the heights of offspring tend to regress toward the mean height of their parents. Its mathematical foundations were later formalized by Legendre and Gauss through the method of least squares. What began as an observation in heredity has since evolved into a powerful tool for modeling relationships and making predictions from data. Thanks to advances in computing and tools like R, regression techniques are now scalable and accessible for solving complex, real-world problems.

Across domains such as economics, medicine, and engineering, regression models support data-driven decisions—whether estimating the impact of advertising on sales, predicting housing prices, or identifying risk factors for disease. As Charles Wheelan writes in *Naked Statistics* (Wheelan 2013), “Regression modeling is the hydrogen bomb of the statistics arsenal.” Used wisely, it can guide powerful decisions; misapplied, it can produce misleading conclusions. A thoughtful approach is essential to ensure that findings are valid, actionable, and aligned with the goals of a data science project.

In this chapter, we continue building upon the *Data Science Workflow* introduced in Chapter 2 and illustrated in Figure 2.3. So far, our journey has included data preparation, exploratory analysis, and the application of two classification algorithms—*k-Nearest Neighbors* (Chapter 7) and *Naive*

Bayes (Chapter 9)—followed by tools for evaluating predictive performance (Chapter 8). As introduced in Section 2.11, supervised learning includes both classification and regression tasks. Regression models expand our ability to predict numeric outcomes and understand relationships among variables.

This chapter also connects to the statistical foundation developed in Chapter 5, especially Section 5.11, which introduced correlation analysis and inference. Regression extends those ideas by quantifying relationships while accounting for multiple variables and allowing for formal hypothesis testing about the effects of specific predictors.

What This Chapter Covers

This chapter builds on your knowledge of the data science workflow and previous chapters on classification, model evaluation, and statistical inference. While earlier chapters focused on classification tasks—such as predicting churn or spam—regression models help us answer questions where the outcome is numeric and continuous.

You will begin by learning the fundamentals of simple linear regression, then extend to multiple regression and generalized linear models (GLMs), which include logistic and Poisson regression. You will also explore polynomial regression as a bridge to non-linear modeling. Along the way, we will use real-world datasets, including *marketing*, *house*, and *insurance*, to ground the techniques in practical applications.

You will also learn how to check model assumptions, evaluate regression performance, and select the most appropriate predictors using tools such as residual analysis and stepwise selection. These methods are introduced not just as statistical techniques, but as essential components of sound data-driven decision-making.

By the end of this chapter, you will be equipped to build, interpret, and evaluate regression models in R, and to understand when to use linear, generalized, or non-linear approaches depending on the nature of the data and modeling goals. We begin with the most fundamental regression technique: simple linear regression, which lays the groundwork for more advanced models introduced later in the chapter. These models will deepen your understanding of both prediction and explanation in data science.

10.1 Simple Linear Regression

Simple linear regression is the most fundamental form of regression modeling. It allows us to quantify the relationship between a *single predictor* and a *response variable*. By focusing on one predictor at a time, we develop an intuitive understanding of how regression models operate—how they estimate effects, assess fit, and make predictions—before progressing to more complex models with multiple predictors.

To illustrate simple linear regression in practice, we use the *marketing* dataset from the *liver* package. This dataset contains *daily digital marketing metrics* and their associated *revenue outcomes*, making it a realistic and relevant example. The data include key performance indicators (KPIs) such as advertising expenditure, user engagement, and transactional outcomes.

The dataset consists of 40 observations and 8 variables:

- `revenue`: Total daily revenue (response variable).
- `spend`: Daily expenditure on pay-per-click (PPC) advertising.
- `clicks`: Number of clicks on advertisements.
- `impressions`: Number of times ads were displayed to users.
- `transactions`: Number of completed transactions per day.
- `click.rate`: Click-through rate (CTR), calculated as the proportion of impressions resulting in clicks.
- `conversion.rate`: Conversion rate, representing the proportion of clicks leading to transactions.
- `display`: Whether a display campaign was active (yes or no).

We begin by loading the dataset and examining its structure:

```
library(liver)

data(marketing, package = "liver")

str(marketing)
'data.frame':   40 obs. of  8 variables:
 $ spend      : num  22.6 37.3 55.6 45.4 50.2 ...
 $ clicks     : int  165 228 291 247 290 172 68 112 306 300 ...
 $ impressions: int  8672 11875 14631 11709 14768 8698 2924 5919 14789
   .. 14818 ...
 $ display    : int  0 0 0 0 0 0 0 0 0 ...
 $ transactions: int  2 2 3 2 3 2 1 1 3 3 ...
 $ click.rate  : num  1.9 1.92 1.99 2.11 1.96 1.98 2.33 1.89 2.07 2.02
   .. ...
 $ conversion.rate: num  1.21 0.88 1.03 0.81 1.03 1.16 1.47 0.89 0.98 1 ...
 $ revenue     : num  58.9 44.9 141.6 209.8 197.7 ...
```

The dataset contains 8 variables and 40 observations. The response variable, revenue, is continuous, while the remaining 7 variables serve as potential predictors.

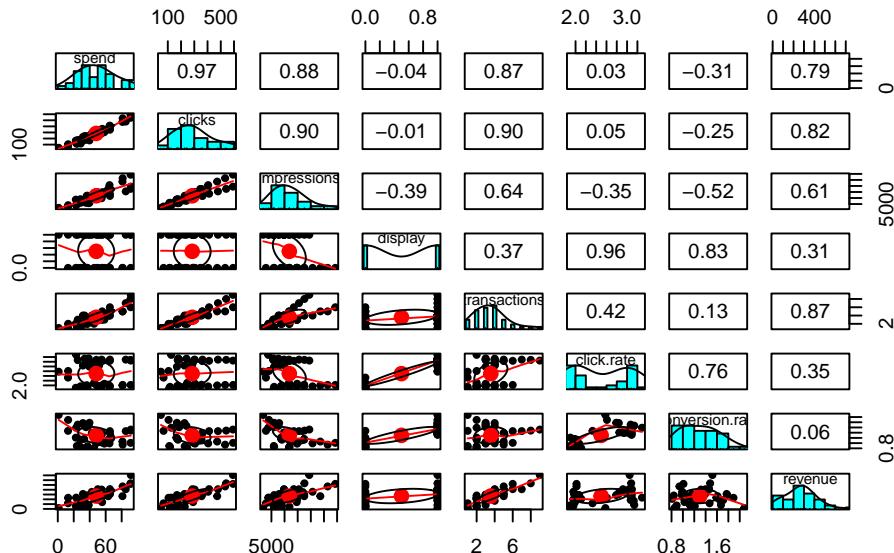
In the following section, we explore the relationship between advertising spend and revenue to determine whether a linear model is appropriate.

Exploring Relationships in the Data

Before constructing a regression model, we first explore the relationships between variables to ensure that our assumptions hold and to identify strong predictors. This step also helps assess whether the relationship between variables appears linear—a key assumption in simple linear regression.

A useful tool for this is the `pairs.panels()` function from the **psych** package, which provides a comprehensive overview of pairwise relationships:

```
library(psych)
pairs.panels(marketing)
```



This visualization includes:

- *Scatter plots* (lower triangle), showing how each predictor relates to the response variable.

- *Histograms* (diagonal), illustrating the distribution of each variable.
- *Correlation coefficients* (upper triangle), quantifying the strength and direction of linear associations.

From the correlation matrix, we observe that spend and revenue exhibit a *strong positive correlation* of 0.79. This indicates that *higher advertising expenditure is generally associated with higher revenue*, suggesting that spend is a promising predictor for modeling revenue. This correlation reflects the type of relationship we studied in Section 5.11, where we examined how to quantify and test linear associations between numeric variables.

In the next section, we formalize this relationship using a *simple linear regression model*.

Fitting a Simple Linear Regression Model

A logical starting point in regression analysis is to examine the relationship between a single predictor and the response variable. This helps build an intuitive understanding of how one variable influences another before moving on to more complex models. In this case, we explore how advertising expenditure (spend) affects daily revenue (revenue) using a simple linear regression model.

Before fitting the model, it is helpful to visualize the relationship between the two variables to assess whether a linear assumption is appropriate. A scatter plot with a fitted least-squares regression line provides insight into the strength and direction of the association:

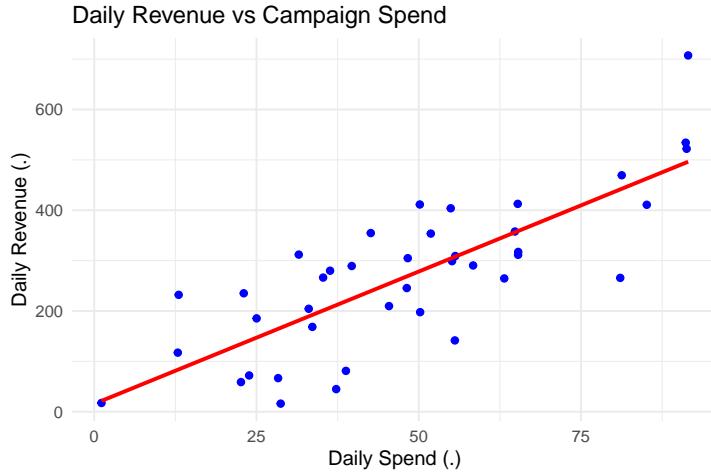


Figure 10.1: Scatter plot of daily revenue (€) versus daily spend (€) for 40 observations, with the fitted least-squares regression line (red) showing the linear relationship.

Figure 10.1 shows the empirical relationship between spend and revenue in the *marketing* dataset. The scatter plot suggests a positive association, indicating that increased advertising expenditure is generally linked to higher revenue—a pattern consistent with a linear relationship.

We model this association mathematically using a *simple linear regression model*, defined as:

$$\hat{y} = b_0 + b_1 x$$

where:

- \hat{y} is the predicted value of the response variable (revenue),
- x is the predictor variable (spend),
- b_0 is the intercept, representing the estimated revenue when no money is spent, and
- b_1 is the slope, indicating the expected change in revenue for a one-unit increase in spend.

To deepen your intuition, Figure 10.2 provides a *conceptual visualization* of this model. The red line shows the fitted regression line, the blue points represent observed data, and the vertical line illustrates a residual (error), calculated as the difference between the observed value y_i and its predicted value

$\hat{y}_i = b_0 + b_1 \times x_i$. Residuals quantify how much the model's predictions deviate from actual outcomes.

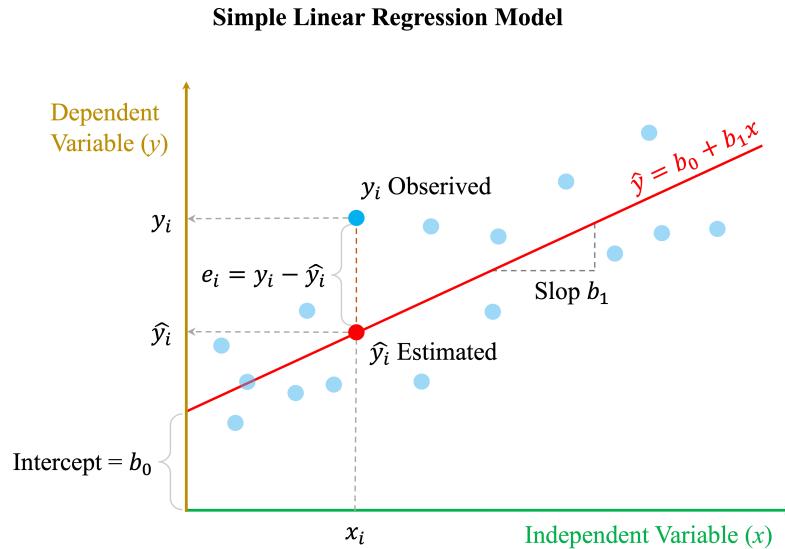


Figure 10.2: Conceptual view of a simple regression model: the red line shows the fitted regression line, blue points represent observed data, and the vertical line illustrates a residual (error), calculated as the difference between the observed value and its predicted value.

In the next subsection, we estimate the regression coefficients in **R** and interpret their meaning in the context of digital advertising and revenue.

Fitting the Simple Regression Model in R

Now that we understand the logic behind simple linear regression, let us put theory into practice. To estimate the regression coefficients, we use the `lm()` function in **R**, which fits a linear model using the least squares method. This function is part of base **R**, so there is no need to install any additional packages. Importantly, `lm()` works for both *simple* and *multiple* regression models, making it a flexible tool we will continue using in the upcoming sections.

The general syntax for fitting a regression model is:

```
lm(response_variable ~ predictor_variable, data = dataset)
```

In our case, we model revenue as a function of spend:

```
simple_reg = lm(revenue ~ spend, data = marketing)
```

Once the model is fitted, we can summarize the results using the `summary()` function:

```
summary(simple_reg)

Call:
lm(formula = revenue ~ spend, data = marketing)

Residuals:
    Min      1Q  Median      3Q     Max 
-175.640 -56.226   1.448  65.235 210.987 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 15.7058    35.1727   0.447   0.658    
spend        5.2517     0.6624   7.928 1.42e-09 ***  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 

Residual standard error: 93.82 on 38 degrees of freedom
Multiple R-squared:  0.6232, Adjusted R-squared:  0.6133 
F-statistic: 62.86 on 1 and 38 DF,  p-value: 1.415e-09
```

This output provides rich information about the model. At its core is the regression equation:

$$\widehat{\text{revenue}} = 15.71 + 5.25 \times \text{spend}$$

where:

- The *intercept* (b_0) is 15.71, representing the estimated revenue when no money is spent on advertising.
- The *slope* (b_1) is 5.25, indicating that for each additional €1 spent, revenue is expected to increase by about €5.25.

But there is more to unpack. The `summary()` output also reports several diagnostics that help us assess the model's reliability:

- *Estimate*: These are the regression coefficients—how much the response changes with a unit change in the predictor.

- *Standard error:* Reflects the precision of the coefficient estimates. Smaller values indicate more certainty.
- *t-value and p-value:* Help assess whether the coefficients are statistically different from zero. A small p-value (typically < 0.05) implies a meaningful relationship.
- *Multiple R-squared (R^2):* Indicates how well the model explains the variation in revenue. In our case, $R^2 = 0.623$, meaning that *62.3% of the variance in revenue is explained by advertising spend*.
- *Residual standard error (RSE):* Measures the average deviation of predictions from actual values. Here, $RSE = 93.82$, which provides a sense of the model's typical prediction error.

These results suggest a statistically significant and practically useful relationship between advertising expenditure and revenue. However, model fitting is only the first step. In the following sections, we explore how to apply this model for prediction, interpret residuals, and check whether key assumptions are met—an essential step for building trustworthy regression models.

Making Predictions with the Regression Line

One of the key advantages of a fitted regression model is its ability to generate predictions for new data. The regression line provides a mathematical approximation of the relationship between advertising spend and revenue, enabling us to estimate revenue based on different levels of expenditure.

Suppose a company wants to estimate the expected daily revenue when €25 is spent on pay-per-click (PPC) advertising. Using the fitted regression equation:

$$\begin{aligned}\widehat{\text{revenue}} &= b_0 + b_1 \times 25 \\ &= 15.71 + 5.25 \times 25 \\ &= 147\end{aligned}\tag{10.1}$$

Thus, if the company spends €25 on advertising, the model estimates a daily revenue of approximately **€147**.

This kind of predictive insight is particularly useful for marketing teams seeking to plan and evaluate advertising budgets. For example, if the objective is to maximize returns while staying within a cost constraint, the regres-

sion model offers a data-driven estimate of how revenue is likely to respond to changes in spending.

Note: Predictions from a regression model are most reliable when the input values are within the range of the observed data and when key model assumptions (e.g., linearity, homoscedasticity) hold.

As a short practice, try predicting the daily revenue if the company increases its advertising spend to €40 and €100. Use the regression equation with the estimated coefficients, and interpret your result. How does this compare to the €25 case? *Hint:* Keep in mind that linear models assume the relationship holds across the observed range—avoid extrapolating too far beyond the original data.

In practice, rather than manually plugging numbers into the regression formula, we can use the `predict()` function in R to estimate revenue more efficiently. You may recall using this same function in Chapter 9 to generate class predictions from a Naive Bayes model. The underlying idea is the same: once a model is fitted, `predict()` provides a simple interface to generate predictions for new data.

For example, to predict revenue for a day with €25 in advertising spend:

```
predict(simple_reg, newdata = data.frame(spend = 25))
```

To make predictions for multiple values (e.g., €25, €40, €100), supply a data frame with those values:

```
predict(simple_reg, newdata = data.frame(spend = c(25, 40, 100)))
```

This approach is especially helpful when working with larger datasets or integrating regression predictions into automated workflows.

Residuals and Model Fit

Residuals measure the difference between observed and predicted values, providing insight into how well the regression model fits the data. For a given observation i , the residual is calculated as:

$$e_i = y_i - \hat{y}_i$$

where y_i is the actual observed value and \hat{y}_i is the predicted value from the regression model. Figure 10.3 visually depicts these residuals as dashed lines connecting observed outcomes to the fitted regression line.

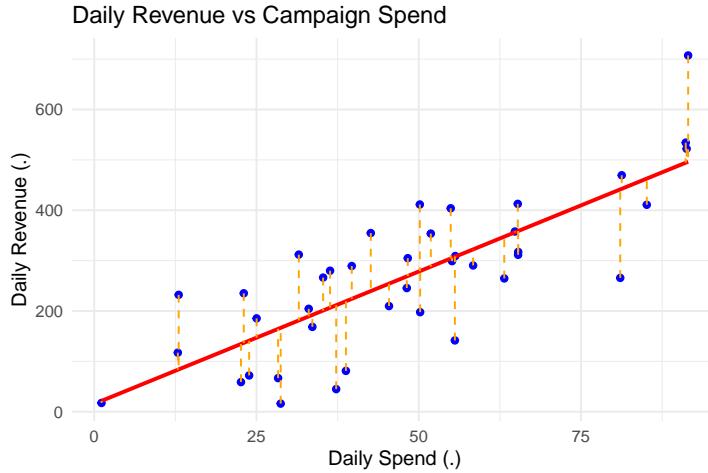


Figure 10.3: Scatter plot of daily revenue (€) versus daily spend (€) for 40 observations. The red line shows the fitted regression line, and the orange dashed lines represent residuals—the vertical distances between observed values and their predicted values on the line.

For example, suppose the 21st day in the dataset has a marketing spend of €25 and an actual revenue of 185.36. The residual for this observation is:

$$\begin{aligned}
 \text{Residual} &= y - \hat{y} \\
 &= 185.36 - 147 \\
 &= 38.36
 \end{aligned} \tag{10.2}$$

Residuals play a crucial role in assessing model adequacy. Ideally, they should be randomly distributed around zero, suggesting that the model appropriately captures the relationship between variables. However, if residuals show systematic patterns—such as curves, clusters, or increasing spread—this may indicate the need to include additional predictors, transform variables, or use a non-linear model.

The regression line is estimated using the *least squares* method, which finds the line that minimizes the *sum of squared residuals*, also known as the *sum of squared errors (SSE)*:

$$\text{SSE} = \sum_{i=1}^n (y_i - \hat{y}_i)^2 \tag{10.3}$$

where n is the number of observations. This quantity corresponds to the total squared length of the orange dashed lines in Figure 10.3. Minimizing SSE ensures that the estimated regression line best fits the observed data.

In summary, residuals provide critical feedback on model performance. By analyzing the *marketing* dataset, we have demonstrated how to calculate and interpret residuals and how they guide model refinement. This foundational understanding of simple linear regression prepares us to evaluate model quality and to extend the framework to models with multiple predictors in the following sections.

Now that we have fitted and interpreted a simple linear model, let us ask whether the observed relationships are statistically reliable.

10.2 Hypothesis Testing in Simple Linear Regression

Once we estimate a regression model, the next question is: *Is the relationship we found real, or could it have occurred by chance?* This is where *hypothesis testing* comes in—a core concept introduced in Chapter 5 and applied here to assess the statistical significance of regression coefficients.

In regression analysis, we are particularly interested in whether a predictor variable has a statistically significant relationship with the response variable. In simple linear regression, this involves testing whether the estimated slope b_1 from the sample provides evidence of a real linear association in the population, where the unknown population slope is denoted by β_1 .

The population regression model is

$$y = \beta_0 + \beta_1 x + \epsilon$$

where:

- β_0 is the *population intercept*: the expected value of y when $x = 0$,
- β_1 is the *population slope*: the expected change in y for a one-unit increase in x , and
- ϵ is the *error term*, accounting for variability not captured by the linear model.

The key question is: *Is β_1 significantly different from zero?* If $\beta_1 = 0$, then x has *no linear effect* on y , and the model reduces to:

$$y = \beta_0 + \epsilon.$$

We formalize this question using the following hypotheses:

$$\begin{cases} H_0 : \beta_1 = 0 & (\text{no linear relationship between } x \text{ and } y) \\ H_a : \beta_1 \neq 0 & (\text{a linear relationship exists between } x \text{ and } y) \end{cases}$$

To test these hypotheses, we compute the *t-statistic* for the slope:

$$t = \frac{b_1}{SE(b_1)},$$

$SE(b_1)$ is the standard error of the slope estimate (b_1). This statistic follows a *t-distribution* with $n - 2$ degrees of freedom (in simple regression, 2 parameters are estimated), where n is the number of observations. We then examine the *p-value*, which tells us how likely it would be to observe such a slope (or more extreme) if H_0 were true. A small p-value—typically below 0.05—leads us to reject the null hypothesis.

Let us return to our regression model predicting revenue from spend in the *marketing* dataset:

```
summary(simple_reg)

Call:
lm(formula = revenue ~ spend, data = marketing)

Residuals:
    Min      1Q  Median      3Q     Max 
-175.640 -56.226   1.448  65.235 210.987 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 15.7058    35.1727   0.447   0.658    
spend        5.2517     0.6624   7.928 1.42e-09 ***  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 

Residual standard error: 93.82 on 38 degrees of freedom
Multiple R-squared:  0.6232, Adjusted R-squared:  0.6133 
F-statistic: 62.86 on 1 and 38 DF,  p-value: 1.415e-09
```

From the output:

- The *estimated slope* $b_1 = 5.25\text{€}$.
- The *t-statistic* is 7.93.
- The *p-value* is 0 (rounded to three digits), which is lower than 0.05.

Since the p-value is well below our significance level ($\alpha = 0.05$), we reject the null hypothesis H_0 . This provides strong evidence of a *statistically significant association* between advertising spend and revenue. In practical terms:

For each additional €1 spent on advertising, the model predicts an average increase in daily revenue of approximately €5.25.

This confirms that spend is a meaningful predictor in our regression model.

Caution: Statistical significance does not imply *causation*. The observed relationship may be influenced by other factors not included in the model. Interpreting regression results responsibly requires considering possible confounders, omitted variables, and whether assumptions hold.

Statistical significance tells us the relationship is unlikely due to chance—but how well does the model actually perform? That is the focus of the next section. In the next sections, we explore how to *diagnose model quality* using residuals and *evaluate assumptions* that ensure the validity of regression results. We will then build on this foundation by introducing *multiple regression*, where more than one predictor is used to explain variation in the response variable.

10.3 Measuring the Quality of a Regression Model

Suppose your regression model shows that advertising spend has a statistically significant effect on daily revenue. That is useful—but is it enough? Can the model make accurate predictions, or is it just detecting a weak trend in noisy data?

Hypothesis tests tell us *if* a variable is related to the outcome, but they do not tell us *how well* the model performs as a whole. To evaluate a model's practical usefulness—whether for forecasting, decision-making, or understanding patterns—we need additional tools.

This section introduces two key metrics: the *Residual Standard Error (RSE)*, which measures average prediction error, and the R^2 (R-squared) statistic, which quantifies how much of the variation in the response variable is explained by the model. Together, they offer a more complete picture of model performance beyond statistical significance.

Residual Standard Error (RSE)

How far off are our predictions—on average—from the actual values? That is what the *Residual Standard Error (RSE)* tells us. It measures the typical size of the residuals: the differences between observed and predicted values; as presented in Figure 10.3 (orange dashed lines). In other words, RSE estimates the average prediction error of the regression model.

The formula for RSE is:

$$RSE = \sqrt{\frac{SSE}{n - m - 1}},$$

where SSE is defined in Equation 10.3, n is the number of observations, and m is the number of predictors. The denominator ($n - m - 1$) accounts for the degrees of freedom in the model, adjusting for the number of predictors being estimated.

A smaller RSE indicates more accurate predictions. For our simple linear regression model using the *marketing* dataset, the RSE is:

```
rse_value = sqrt(sum(simple_reg$residuals^2) / summary(simple_reg)$df[2])
round(rse_value, 2)
[1] 93.82
```

This value tells us the typical size of prediction errors, in euros. While lower values are preferred, RSE should always be interpreted in the context of the response variable's scale. For example, an RSE of 20 may be small or large depending on whether daily revenues typically range in the hundreds or thousands of euros.

R-squared (R^2)

If you could explain all the variation in revenue using just one line, how good would that line be? That is the idea behind *R-squared (R^2)*—a statistic that measures the proportion of variability in the response variable explained by the model.

The formula is:

$$R^2 = 1 - \frac{SSE}{SST},$$

where SSE is the sum of squared residuals (Equation 10.3) and SST is the total sum of squares, representing the total variation in the response. R^2 ranges from 0 to 1. A value of 1 means the model perfectly explains the variation in the outcome; a value of 0 means it explains none of it.

You can visualize this concept in Figure 10.1, where the red regression line summarizes how revenue changes with spend. The R^2 value quantifies how well this line captures the overall pattern in the *marketing* data:

```
round(summary(simple_reg)$r.squared, 3)
[1] 0.623
```

This means that approximately 62.3% of the variation in daily revenue is explained by advertising spend.

In simple linear regression, there is a direct connection between R^2 and the correlation coefficient introduced in Section 5.11 of Chapter 5. Specifically, R^2 is the square of the Pearson correlation coefficient r between the predictor and the response:

$$R^2 = r^2$$

Let us verify this in the *marketing* data:

```
round(cor(marketing$spend, marketing$revenue), 2)
[1] 0.79
```

Squaring this value:

```
round(cor(marketing$spend, marketing$revenue)^2, 2)
[1] 0.62
```

gives the same R^2 value, reinforcing that R^2 in simple regression reflects the strength of the linear association between two variables.

While a higher R^2 suggests a better fit, it does not guarantee that the model generalizes well or satisfies the assumptions of linear regression. Always examine residual plots, check for outliers, and interpret R^2 in context—not in isolation.

Adjusted R-squared

Adding more predictors to a regression model will always increase R^2 —even if those predictors are not truly useful. This is where *Adjusted R²* comes

in. It compensates for the number of predictors in the model, providing a more honest measure of model quality. Its formula is:

$$\text{Adjusted } R^2 = 1 - (1 - R^2) \times \frac{n - 1}{n - m - 1},$$

where n is the number of observations and m is the number of predictors.

In simple linear regression (where $m = 1$), Adjusted R^2 is nearly the same as R^2 . However, as we add more variables in multiple regression models, Adjusted R^2 becomes essential. It penalizes complexity and helps identify whether additional predictors genuinely improve model performance.

You will see Adjusted R^2 used more frequently in the next sections, especially when comparing alternative models with different sets of predictors.

Interpreting Model Quality

A strong regression model typically demonstrates the following qualities:

- *Low RSE*, indicating that predictions are consistently close to actual values;
- *High R^2* , suggesting that the model accounts for a substantial portion of the variability in the response;
- *High Adjusted R^2* , which reflects the model's explanatory power while penalizing unnecessary predictors.

However, these metrics do not tell the full story. For example, a high R^2 can result from overfitting or be distorted by outliers, while a low RSE may mask violations of regression assumptions. In applied settings, these statistics should be interpreted in conjunction with residual diagnostics, visual checks, and—when feasible—cross-validation.

Table 10.1 presents a summary of model quality metrics. Understanding these measures enables us to evaluate regression models more critically and prepares us to move beyond models with a single predictor.

Table 10.1: Overview of commonly used regression model quality metrics.

| Metric | What It Tells You | What to Look For |
|---------------------------|---|----------------------------------|
| RSE (Residual Std. Error) | Average prediction error | Lower is better |
| R^2 | Proportion of variance explained | Higher is better |
| Adjusted R^2 | R^2 adjusted for number of predictors | Higher (but realistic) is better |

In the next section, we extend the simple linear regression framework to include **multiple predictors**, allowing us to capture more complex relationships and improve predictive accuracy.

10.4 Multiple Linear Regression

We now move beyond simple linear regression and explore models with more than one predictor. This brings us into the realm of *multiple regression*, a framework that captures the simultaneous effects of multiple variables on an outcome. In most real-world scenarios, responses are rarely driven by a single factor—multiple regression helps us model this complexity.

To illustrate, we expand the previous model, which included only `spend` as a predictor, by adding `display`, an indicator of whether a *display (banner) advertising campaign* was active. This additional predictor allows us to assess its impact on revenue. The general equation for a multiple regression model with m predictors is:

$$\hat{y} = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_m x_m$$

where b_0 is the intercept, and b_1, b_2, \dots, b_m represent the estimated effects of each predictor on the response variable.

For our case, the equation with two predictors, `spend` and `display`, is:

$$\widehat{\text{revenue}} = b_0 + b_1 \times \text{spend} + b_2 \times \text{display}$$

where `spend` represents daily advertising expenditure and `display` is a categorical variable (yes or no), which **R** automatically converts into a binary indicator. In this case, `display = 1` corresponds to an active display campaign, while `display = 0` means no campaign was running. As with other

factor variables in **R**, the first level (no) serves as the reference category when converting to dummy (0/1) indicators (alphabetically, unless explicitly changed).

Fitting the Multiple Regression Model in R

To fit a multiple regression model in **R**, we continue using the `lm()` function—the same tool we used for simple regression. The only difference is that we now include more than one predictor on the right-hand side of the formula:

```
multiple_reg = lm(revenue ~ spend + display, data = marketing)

summary(multiple_reg)

Call:
lm(formula = revenue ~ spend + display, data = marketing)

Residuals:
    Min      1Q   Median      3Q      Max 
-189.420 -45.527    5.566   54.943  154.340 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -41.4377   32.2789  -1.284 0.207214    
spend        5.3556    0.5523   9.698 1.05e-11 ***  
display     104.2878   24.7353   4.216 0.000154 ***  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 

Residual standard error: 78.14 on 37 degrees of freedom
Multiple R-squared:  0.7455, Adjusted R-squared:  0.7317 
F-statistic: 54.19 on 2 and 37 DF,  p-value: 1.012e-11
```

This fits a model with both `spend` and `display` as predictors of `revenue`. The estimated regression equation is:

$$\widehat{\text{revenue}} = -41.44 + 5.36 \times \text{spend} + 104.29 \times \text{display}$$

Let us interpret each term:

- The *intercept* (b_0) is -41.44. This represents the estimated revenue when `spend = 0` and `display = "no"`—that is, when no advertising budget is spent and no display campaign is active.

- The coefficient for spend (b_1) is 5.36. It indicates that for every additional €1 spent on advertising, daily revenue increases by approximately 5.36, assuming the display campaign status remains unchanged.
- The coefficient for display (b_2) is 104.29. Since display is a binary variable (yes vs. no), this coefficient estimates the difference in average revenue between days with and without a display campaign—holding advertising spend constant.

These interpretations build on our earlier regression concepts, showing how multiple predictors can be incorporated and interpreted in a straightforward way.

Making Predictions

Consider a scenario where the company spends €25 on advertising while running a display campaign (display = 1). Using the regression equation, the predicted revenue is:

$$\widehat{\text{revenue}} = -41.44 + 5.36 \times 25 + 104.29 \times 1 = 196.74$$

Thus, the predicted revenue for that day is approximately €196.74.

The residual (prediction error) for a specific observation is calculated as the difference between the actual and predicted revenue:

$$\text{Residual} = y - \hat{y} = 185.36 - 196.74 = -11.49$$

The prediction error is smaller than that of the simple regression model, confirming that including display improves predictive accuracy.

In practice, rather than plugging numbers into the equation manually, we can use the `predict()` function in R to compute fitted values. This function works seamlessly with multiple regression models as it did with simple regression. For example, to predict revenue for a day with €25 in advertising spend and an active display campaign:

```
predict(multiple_reg, newdata = data.frame(spend = 25, display = "yes"))
```

This approach is especially useful when generating predictions for multiple new scenarios or automating analyses.

Practice: Try estimating the daily revenue under two new scenarios:

- Spending €40 with a display campaign (`display = "yes"`)
- Spending €100 with no display campaign (`display = "no"`)

Use the regression equation or the `predict()` function to compute these values.

What do your predictions suggest? Are they consistent with the €25 case?

Hint: Be cautious about extrapolation—stay within the range of the original data.

Evaluating Model Performance

How can we tell whether adding a new predictor—like `display`—actually improves a regression model? In the previous section, Table 10.1 outlined three key model evaluation metrics: *Residual Standard Error (RSE)*, R^2 , and *Adjusted R²*. Here, we apply those tools to compare the performance of our simple and multiple regression models. By doing so, we can assess whether the added complexity leads to genuine improvement.

- *Residual Standard Error (RSE)*: In the simple regression model, $RSE = 93.82$, whereas in the multiple regression model, $RSE = 78.14$. A lower RSE in the multiple model suggests that its predictions are, on average, closer to the actual values.
- R^2 (R-squared): The simple regression model had $R^2 = 62.3\%$, while the multiple regression model increased to $R^2 = 74.6\%$, indicating that more of the variance in revenue is explained when `display` is included.
- *Adjusted R²*: This metric penalizes unnecessary predictors. In the simple regression model, $\text{Adjusted } R^2 = 61.3\%$, while in the multiple regression model it rises to 73.2% . The increase confirms that adding `display` contributes meaningfully to model performance, beyond what might be expected by chance.

Taken together, these results show that model evaluation metrics do more than quantify fit—they also help guard against overfitting and inform sound modeling choices.

Practice: Try adding another variable—such as `clicks`—to the model. Does the Adjusted R^2 improve? What does that tell you about the added value of this new predictor?

You might now be wondering: *Should we include all available predictors in our regression model? Or is there an optimal subset that balances simplicity and performance?* These important questions will be addressed in Section 10.8, where we explore *stepwise regression* and other model selection strategies.

Same Data, Different Story: What Simpson's Paradox Can Teach Us

As we incorporate more variables into regression models, we must also be alert to how these variables interact. One cautionary tale is Simpson's Paradox. Suppose a university finds that within every department, female applicants are admitted at higher rates than males. Yet, when all departments are combined, it appears that male applicants are admitted more often. How can this be?

This is Simpson's Paradox—a phenomenon where trends within groups reverse when the groups are aggregated. It reminds us that context matters. The paradox often arises when a grouping variable influences both predictor and response but is omitted from the model.

In the plots below (Figure 10.4), the left panel displays a regression line fitted to all the data, yielding an overall correlation of -0.74, which ignores the underlying group structure. In contrast, the right panel reveals the true story: each group exhibits a positive correlation—Group 1: 0.79, Group 2: 0.71, Group 3: 0.62, Group 4: 0.66, Group 5: 0.75. This demonstrates how the apparent overall downward trend is misleading due to Simpson's Paradox.

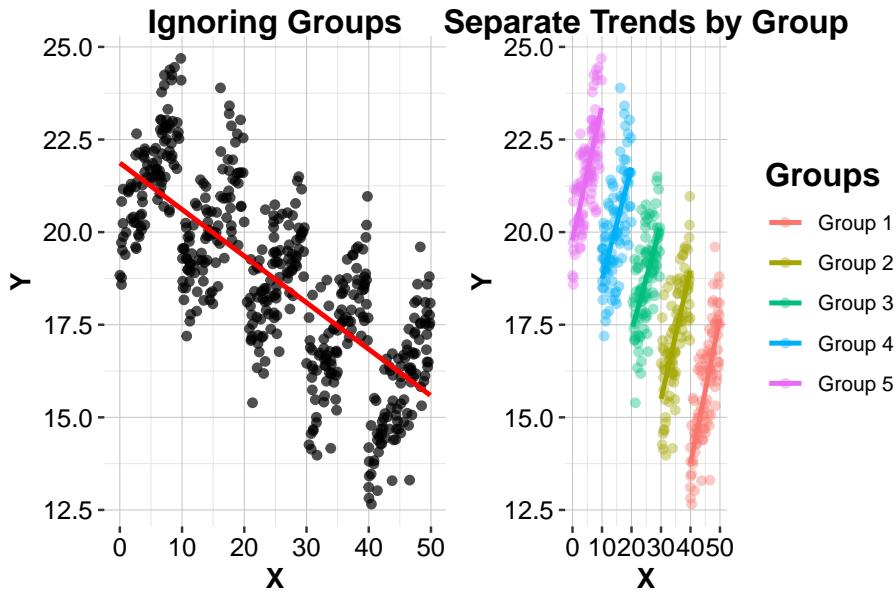


Figure 10.4: Simpson’s Paradox: The left plot shows a regression line fitted to the full dataset, ignoring group structure. The right plot fits separate regression lines for each group, revealing positive trends within groups that are hidden when data are aggregated.

This example underscores the importance of including relevant variables. Omitting key groupings can lead to flawed conclusions—even when regression coefficients appear statistically sound. This phenomenon directly connects to our earlier analysis of the *marketing* dataset. In the simple regression model, we considered only `spend` as a predictor of revenue. However, once we added `display` in the multiple regression model, the interpretation of `spend` changed. This shift reflects how omitted variables—like group membership or campaign status—can confound observed associations. Simpson’s Paradox reminds us that **a variable’s effect can reverse or diminish once other important predictors are included**. Careful modeling and exploratory analysis are essential to uncover these subtleties.

Reflection: Can you think of a situation in your domain—public health, marketing, or education—where combining groups might obscure important differences? How would you guard against this risk in your analysis?

Summary and Implications

In this section, we took our first step beyond simple linear regression and discovered the power of incorporating *multiple predictors*. By adding `display` to our original model with `spend`, we built a multiple regression model that offered clearer insights and better predictive performance.

The multiple regression model:

- *Improved model fit* by reducing prediction errors (lower RSE),
- *Explained more variance* in the outcome (higher R^2), and
- *Demonstrated true value* in adding a new variable (higher Adjusted R^2).

These gains are not automatic. As we expand our models, we also face new challenges:

- *Multicollinearity*: When predictors are strongly correlated with each other, it becomes difficult to isolate their individual effects. This can lead to unstable coefficient estimates and misleading interpretations.
- *Overfitting*: Adding too many predictors might improve performance on the training data but lead to poor generalization to new data.

The solution is not to include all available variables, but to build models thoughtfully. We need to ask: *Which predictors genuinely add value? Which combinations make sense given the context?*

Ready to build smarter, more reliable models? In the next section, we dive into diagnostic checks that help ensure your regression models stand up to real-world scrutiny.

10.5 Generalized Linear Models (GLMs)

What if your outcome is not continuous but binary—such as predicting whether a customer will churn—or count-based—like the number of daily transactions? Traditional linear regression is not suited for such cases. It assumes normally distributed errors, constant variance, and a linear relationship between predictors and response—all assumptions that break down with binary or count data.

Generalized Linear Models (GLMs) extend the familiar regression framework by introducing two powerful concepts:

- a *link function*, which transforms the mean of the response variable to be modeled as a linear function of the predictors,
- and a *variance function*, which allows the response to follow a distribution other than the normal.

These extensions make GLMs a flexible tool for modeling diverse types of response variables and are widely used in fields such as finance, healthcare, social sciences, and marketing.

GLMs preserve the core structure of linear regression but introduce three key components:

1. *Random component*: Specifies the probability distribution of the response variable, chosen from the exponential family (e.g., normal, binomial, Poisson).
2. *Systematic component*: Represents the linear combination of predictor variables.
3. *Link function*: Connects the expected value of the response variable to the linear predictor, enabling a broader range of outcome types to be modeled.

In the following sections, we introduce two commonly used GLMs:

- *Logistic regression*, for modeling binary outcomes (e.g., churn vs. no churn),
- *Poisson regression*, for modeling count data (e.g., number of customer service calls).

By extending regression beyond continuous responses, these models offer both interpretability and flexibility—key advantages for real-world data analysis. The next sections walk through their theoretical foundations and practical implementation in R.

10.6 Logistic Regression for Binary Classification

Can we predict whether a customer will leave a service based on their usage behavior? This is a classic binary classification problem—one we first encountered in Chapter 7 with k-Nearest Neighbors (kNN), and again in Chapter 9 with the Naive Bayes classifier. Those models provided flexible, data-driven solutions to classification, but now we shift to a *model-based* approach grounded in statistical theory: *logistic regression*.

Logistic regression is a generalized linear model specifically designed for binary outcomes. It estimates the *probability* that an observation belongs to a particular class (e.g., `churn = 1`) by applying the *logit function*, which transforms a linear combination of predictors into the log-odds of the outcome:

$$\text{logit}(p) = \ln\left(\frac{p}{1-p}\right) = b_0 + b_1x_1 + b_2x_2 + \cdots + b_mx_m,$$

where p is the probability that the outcome is 1. The logit transformation ensures that predictions remain between 0 and 1, making logistic regression well-suited for modeling binary events.

Unlike kNN and Naive Bayes, logistic regression provides interpretable model coefficients and naturally handles numeric and binary predictors. It also offers a foundation for many advanced models used in applied machine learning and data science.

In the next subsection, we bring logistic regression to life in R using the `churn` dataset, where you will learn how to fit the model, interpret its coefficients, and assess its usefulness for real-world decision-making.

Fitting a Logistic Regression Model in R

Let us now implement logistic regression in R and interpret its results in a real-world context. The `churn` dataset from the `liver` package—previously introduced in Section 7.7 as a case study for the k-Nearest Neighbors (kNN) algorithm—captures key aspects of customer behavior, including account length, plan types, usage metrics, and customer service interactions. The goal remains the same: to predict whether a customer has churned (yes) or not (no) based on these features.

For background on the dataset and exploratory analysis, see Section 4.3. We first inspect the structure of the data:

```
data(churn)
str(churn)
'data.frame': 5000 obs. of 20 variables:
 $ state      : Factor w/ 51 levels "AK","AL","AR",...: 17 36 32 36 37 2
   .. 20 25 19 50 ...
 $ area.code   : Factor w/ 3 levels "area_code_408",...: 2 2 2 1 2 3 3 2
   .. 1 2 ...
 $ account.length: int 128 107 137 84 75 118 121 147 117 141 ...
 $ voice.plan  : Factor w/ 2 levels "yes","no": 1 1 2 2 2 2 1 2 2 1 ...
 $ voice.messages: int 25 26 0 0 0 0 24 0 0 37 ...
```

```
$ intl.plan      : Factor w/ 2 levels "yes","no": 2 2 2 1 1 1 2 1 2 1 ...
$ intl.mins     : num 10 13.7 12.2 6.6 10.1 6.3 7.5 7.1 8.7 11.2 ...
$ intl.calls    : int 3 3 5 7 3 6 7 6 4 5 ...
$ intl.charge   : num 2.7 3.7 3.29 1.78 2.73 1.7 2.03 1.92 2.35 3.02 ...
$ day.mins      : num 265 162 243 299 167 ...
$ day.calls    : int 110 123 114 71 113 98 88 79 97 84 ...
$ day.charge   : num 45.1 27.5 41.4 50.9 28.3 ...
$ eve.mins      : num 197.4 195.5 121.2 61.9 148.3 ...
$ eve.calls    : int 99 103 110 88 122 101 108 94 80 111 ...
$ eve.charge   : num 16.78 16.62 10.3 5.26 12.61 ...
$ night.mins    : num 245 254 163 197 187 ...
$ night.calls   : int 91 103 104 89 121 118 118 96 90 97 ...
$ night.charge  : num 11.01 11.45 7.32 8.86 8.41 ...
$ customer.calls: int 1 1 0 2 3 0 3 0 1 0 ...
$ churn         : Factor w/ 2 levels "yes","no": 2 2 2 2 2 2 2 2 2 2 ...
```

The dataset is an **R** data frame with 5000 observations and 19 predictor variables. Based on earlier exploration, we select the following features for our logistic regression model:

`account.length`, `voice.plan`, `voice.messages`, `intl.plan`, `intl.mins`, `day.mins`, `eve.mins`, `night.mins`, and `customer.calls`.

We define a formula object to specify the relationship between the target variable (`churn`) and the predictors:

```
formula = churn ~ account.length + voice.messages + day.mins + eve.mins +
          night.mins + intl.mins + customer.calls + intl.plan +
          ↵ voice.plan
```

To fit the logistic regression model, we use the `glm()` function, which stands for *generalized linear model*. This function allows us to specify the family of distributions and link functions, making it suitable for logistic regression. This function is part of base **R**, so no need to install any additional packages. The general syntax for logistic regression is:

```
glm(response_variable ~ predictor_variables, data = dataset, family =
  ↵ binomial)
```

Here, `family = binomial` tells **R** to perform logistic regression.

```
data(churn)

glm_churn = glm(formula = formula, data = churn, family = binomial)
```

To examine the model output:

```
summary(glm_churn)

Call:
glm(formula = formula, family = binomial, data = churn)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 8.8917584 0.6582188 13.509 < 2e-16 ***
account.length -0.0013811 0.0011453 -1.206 0.2279
voice.messages -0.0355317 0.0150397 -2.363 0.0182 *
day.mins      -0.0136547 0.0009103 -15.000 < 2e-16 ***
eve.mins      -0.0071210 0.0009419 -7.561 4.02e-14 ***
night.mins     -0.0040518 0.0009048 -4.478 7.53e-06 ***
intl.mins      -0.0882514 0.0170578 -5.174 2.30e-07 ***
customer.calls -0.5183958 0.0328652 -15.773 < 2e-16 ***
intl.planno    2.0958198 0.1214476 17.257 < 2e-16 ***
voice.planno   -2.1637477 0.4836735 -4.474 7.69e-06 ***
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 4075.0 on 4999 degrees of freedom
Residual deviance: 3174.3 on 4990 degrees of freedom
AIC: 3194.3

Number of Fisher Scoring iterations: 6
```

This output includes:

- *Coefficients*, which indicate the direction and size of each predictor's effect on the log-odds of churn.
- *Standard errors*, which quantify the uncertainty around each coefficient.
- *z-values* and *p-values*, which test whether each predictor contributes significantly to the model.

A small p-value (typically < 0.05) suggests that the predictor has a statistically significant effect on churn. For example, if `account.length` has a large p-value, it may not be a strong predictor and could be removed to simplify the model.

Practice: Try removing one or more non-significant variables (e.g., `account.length`) and refit the model. Compare the new model's summary to the original. How do the coefficients or model fit statistics change?

Note: You might wonder why, in this example, we fit the logistic regression model to the entire `churn` dataset, while in Section 7.7, we first partitioned the data into training and test sets. That is because our current goal is to learn how to fit and interpret logistic regression models—not yet to evaluate out-of-sample predictive performance, which we will explore later.

Also, we did not manually create dummy variables for `intl.plan` and `voice.plan`. Unlike kNN, logistic regression in R automatically handles binary factors by converting them into 0/1 indicator variables, with the first level (e.g., "no") serving as the reference category.

Curious how logistic regression compares to kNN or Naive Bayes in terms of predictive accuracy? You will get to see that soon—in the case study later in this chapter. We will not only compare their accuracy, but also their interpretability and suitability for different types of decisions.

Finally, just like the `lm()` function, the `glm()` model supports the `predict()` function. In the case of logistic regression, `predict()` returns the *predicted probabilities* of the reference class (typically the first level of the outcome factor). We will explore how to interpret and apply these probabilities in the upcoming sections.

10.7 Poisson Regression for Modeling Count Data

Have you ever wondered how often an event will happen—like how many times a customer might call a support center in a given month? When the outcome is a count—how many, not how much—Poisson regression becomes a powerful modeling tool.

The Poisson distribution was first introduced by *Siméon Denis Poisson* (1781–1840) to describe the frequency of rare events, such as wrongful convictions in a legal system. Later, in one of its most famous applications, *Adalbert Bortkiewicz* used the distribution to model the number of soldiers in the Prussian army fatally kicked by horses. Despite the unusual subject, this analysis helped demonstrate how a well-chosen statistical model can make sense of seemingly random patterns.

Poisson regression builds on this foundation. It is a generalized linear model designed for *count data*, where the response variable represents how many times an event occurs in a fixed interval. Examples include the number of daily customer service calls, visits to a website per hour, or products purchased per customer.

Unlike linear regression, which assumes normally distributed residuals, Poisson regression assumes that the *conditional distribution* of the response variable (given the predictors) follows a *Poisson distribution*, and that the mean equals the variance. This makes it especially useful for modeling non-negative integers that represent event frequencies.

Like logistic regression, Poisson regression belongs to the family of generalized linear models (GLMs), extending the ideas introduced in the previous section.

The model is defined as:

$$\ln(\lambda) = b_0 + b_1 x_1 + b_2 x_2 + \dots + b_m x_m$$

where λ represents the expected count of events. The predictors x_1, x_2, \dots, x_m affect the log of λ . The `ln` symbol refers to the *natural logarithm*—a transformation that compresses large values and stretches small ones, helping to linearize relationships. Crucially, this transformation ensures that predicted counts are always positive, which aligns with the nature of count data.

In the next subsection, we will fit a Poisson regression model in **R** using the `churn` dataset to explore what drives customer service call frequency.

Fitting a Poisson Regression Model in R

How often do customers call the support line—and what factors drive that behavior? These are questions suited for modeling *count data*, where the outcome reflects how many times an event occurs, not how much of something is measured. Since the response is a non-negative integer, linear regression is no longer suitable. Instead, we turn to *Poisson regression*, a type of generalized linear model designed specifically for this kind of outcome.

To illustrate, we analyze customer service call frequency using the `churn` dataset. Our goal is to model the number of customer service calls (`customer.calls`) based on customer characteristics and service usage. Because `customer.calls` is a count variable, Poisson regression is more appropriate than linear regression.

In **R**, we fit a Poisson regression model using the `glm()` function, the same function we used for logistic regression. The syntax is:

```
glm(response_variable ~ predictor_variables, data = dataset, family =
  poisson)
```

Here, `family = poisson` tells **R** to fit a model under the assumption that the mean and variance of the response are equal, as expected under a Poisson distribution.

We fit the model as follows:

```
formula_calls = customer.calls ~ churn + voice.messages + day.mins +
  eve.mins +
  night.mins + intl.mins + intl.plan + voice.plan

reg_pois = glm(formula = formula_calls, data = churn, family = poisson)
```

To examine the model output:

```
summary(reg_pois)

Call:
glm(formula = formula_calls, family = poisson, data = churn)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 0.9957186 0.1323004 7.526 5.22e-14 ***
churnno     -0.5160641 0.0304013 -16.975 < 2e-16 ***
voice.messages 0.0034062 0.0028294 1.204 0.228646
day.mins     -0.0006875 0.0002078 -3.309 0.000938 ***
eve.mins      -0.0005649 0.0002237 -2.525 0.011554 *
night.mins    -0.0003602 0.0002245 -1.604 0.108704
intl.mins     -0.0075034 0.0040886 -1.835 0.066475 .
intl.planno   0.2085330 0.0407760 5.114 3.15e-07 ***
voice.planno  0.0735515 0.0878175  0.838 0.402284
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 5991.1 on 4999 degrees of freedom
Residual deviance: 5719.5 on 4991 degrees of freedom
AIC: 15592

Number of Fisher Scoring iterations: 5
```

The output provides:

- *Coefficients*, which quantify the effect of each predictor on the expected number of customer calls.
- *Standard errors*, which measure uncertainty around the estimates.
- *z-values* and *p-values*, which test whether each predictor significantly contributes to the model.

A small p-value (typically < 0.05) suggests that the predictor has a statistically significant effect on the call frequency. If a variable such as `voice.messages` has a large p-value, it may not add meaningful explanatory power and could be removed to simplify the model.

Interpreting coefficients in Poisson regression is different from linear regression. Coefficients are on the log scale: each unit increase in a predictor multiplies the expected count by e^b , where b is the coefficient. For instance, if the coefficient for `intl.plan` is 0.3:

$$e^{0.3} - 1 \approx 0.35$$

This means customers with an international plan are expected to make about 35% more service calls than those without one, holding other predictors constant.

Practice: Suppose a predictor has a coefficient of -0.2 . What is the expected percentage change in service calls? Compute $e^{-0.2} - 1$ and interpret the result.

Note: When the variance of the response variable is much greater than the mean, a condition called **overdispersion**, the standard Poisson model may not be suitable. In such cases, extensions like **quasi-Poisson** or **negative binomial regression** are better suited. Although we will not cover these models in detail here, they are valuable tools for analyzing real-world count data.

Tip: As with logistic regression, you can use the `predict()` function to generate predicted values from a Poisson model. These predictions return expected counts, which can be useful for estimating the number of calls for new customer profiles.

Poisson regression extends the linear modeling framework to a broader class of problems involving event frequency. It provides an interpretable, statistically grounded method for modeling count data, and—like logistic regression—it is part of the generalized linear model family.

10.8 Choosing the Right Predictors: Stepwise Regression in Action

“Which predictors should we include in our regression model—and which should we leave out?” This is one of the most important questions in applied data science. Including too few variables risks overlooking meaningful relationships, while including too many can lead to overfitting and diminished generalization performance.

Selecting appropriate predictors is essential for constructing a regression model that is both accurate and interpretable. This process, known as *model specification*, aims to preserve essential associations while excluding irrelevant variables. A well-specified model not only enhances predictive accuracy but also ensures that the resulting insights are meaningful and actionable.

In real-world applications—particularly in business and data science—datasets often contain a large number of potential predictors. Managing this complexity requires systematic approaches for identifying the most relevant variables. One such approach is *stepwise regression*, an iterative algorithm that evaluates predictors based on their contribution to the model. It adds or removes variables one at a time, guided by statistical significance and model evaluation criteria.

Stepwise regression builds on earlier stages in the data science workflow. In Chapter 4, we used visualizations and descriptive summaries to explore relationships among variables. In Chapter 5, we formally tested associations between predictors and the response. These initial steps offered valuable intuition. Stepwise regression builds upon that foundation, formalizing and automating feature selection using evaluation metrics.

Due to its structured procedure, stepwise regression is especially useful for small to medium-sized datasets, where it can improve model clarity without imposing excessive computational demands. In the next subsections, we will demonstrate how to perform stepwise regression in R, introduce model selection criteria such as AIC, and discuss both the strengths and limitations of this method.

How AIC Guides Model Selection

How do we know if a simpler model is better—or if we have left out something essential? This question lies at the heart of model selection. When faced with multiple competing models, we need a principled way to compare them, balancing model fit with interpretability.

One such tool is the *Akaike Information Criterion (AIC)*. AIC offers a structured trade-off between model complexity and goodness of fit: lower AIC values indicate a more favorable balance between explanatory power and simplicity. It is defined as

$$AIC = 2m + n \log\left(\frac{SSE}{n}\right),$$

where m denotes the number of estimated parameters in the model, n is the number of observations, and SSE is the sum of squared errors (as introduced in Equation 10.3), capturing the total unexplained variability in the response variable.

Unlike R^2 , which always increases as more predictors are added, AIC explicitly penalizes model complexity. This penalty helps prevent overfitting—where a model describes random noise rather than meaningful structure—by favoring simpler models that still provide a good fit. AIC serves as a model “scorecard,” rewarding goodness of fit while discouraging unnecessary complexity, much like preferring the simplest recipe that still delivers excellent flavor.

While AIC is widely used, it is not the only available criterion. An alternative is the *Bayesian Information Criterion (BIC)*, which applies a stronger penalty for model complexity. It is defined as

$$BIC = \log(n) \times m + n \log\left(\frac{SSE}{n}\right),$$

where the terms are as previously defined. The penalty in BIC grows with the sample size n , causing it to favor more parsimonious models as datasets become larger. BIC may be more appropriate when the goal is to identify the true underlying model, while AIC is often preferred for optimizing predictive accuracy. The choice depends on context, but both criteria reflect the same core idea: balancing fit with parsimony.

By default, the `step()` function in R uses AIC as its model selection criterion. We will demonstrate this process in the next subsection.

Stepwise Regression in Practice: Using `step()` in R

After introducing model selection criteria like AIC, we can implement them in practice using stepwise regression. In R, the `step()` function—part of base R—automates the selection of predictors to identify an optimal model. It iteratively evaluates predictors and includes or excludes them based on improvements in AIC.

The `step()` function takes a fitted model object (such as one created using `lm()` or `glm()`) and applies the stepwise selection algorithm. The general syntax is:

```
step(object, direction = c("both", "backward", "forward"))
```

where `object` is a model of class "`lm`" or "`glm`". The `direction` argument specifies the selection strategy:

- "forward": starts with no predictors and adds them one at a time;
- "backward": begins with all predictors and removes them sequentially;
- "both": combines forward selection and backward elimination.

To illustrate, we apply stepwise regression to the *marketing* dataset, which includes seven predictors. The goal is to construct a parsimonious model that predicts revenue while remaining interpretable.

We begin by fitting a full linear model using all available predictors:

```

data(marketing, package = "liver")

full_model = lm(revenue ~ ., data = marketing)

summary(full_model)

Call:
lm(formula = revenue ~ ., data = marketing)

Residuals:
    Min      1Q  Median      3Q     Max 
-138.00 -59.12  15.16  54.58 106.99 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -25.260020 246.988978 -0.102   0.919    
spend        -0.025807  2.605645  -0.010   0.992    
clicks         1.211912  1.630953   0.743   0.463    
impressions   -0.005308  0.021588  -0.246   0.807    
display        79.835729 117.558849   0.679   0.502    
transactions  -7.012069  66.383251  -0.106   0.917    
click.rate     -10.951493 106.833894  -0.103   0.919    
conversion.rate 19.926588 135.746632   0.147   0.884   

Residual standard error: 77.61 on 32 degrees of freedom
Multiple R-squared:  0.7829, Adjusted R-squared:  0.7354 
F-statistic: 16.48 on 7 and 32 DF,  p-value: 5.498e-09

```

Although the full model includes all available predictors, not all of them appear to meaningfully contribute to explaining variation in revenue. The summary output shows that all predictors have high p-values, which is unusual and suggests that none of them are statistically significant on their own, at least in the presence of the other predictors. For instance, the p-value for `spend` is 0.992, providing limited evidence that it is a meaningful predictor.

This pattern may be a sign of *multicollinearity*, a situation in which two or more predictors are highly correlated with one another. When multicollinearity is present, the regression algorithm has difficulty estimating the unique effect of each variable, because the predictors convey overlapping information. As a result, the standard errors of the coefficient estimates become inflated, and individual predictors may appear statistically insignificant—even though the model as a whole may still fit the data well (as indicated by a relatively high R^2 value).

Multicollinearity does not bias the regression coefficients, but it undermines the interpretability of the model and complicates variable selection. For a more detailed treatment of multicollinearity and its diagnostics, see Kutner et al. (2005).

This ambiguity reinforces the importance of model selection techniques, such as stepwise regression, which help identify a more stable and parsimonious subset of predictors that contribute meaningfully to the response.

We refine the model using the `step()` function with `direction = "both"`:

```
stepwise_model = step(full_model, direction = "both")
Start: AIC=355.21
revenue ~ spend + clicks + impressions + display + transactions +
    click.rate + conversion.rate

          Df Sum of Sq   RSS   AIC
- spend      1     0.6 192760 353.21
- click.rate  1    63.3 192822 353.23
- transactions 1    67.2 192826 353.23
- conversion.rate 1   129.8 192889 353.24
- impressions  1   364.2 193123 353.29
- display      1   2778.1 195537 353.79
- clicks       1   3326.0 196085 353.90
<none>                    192759 355.21

Step: AIC=353.21
revenue ~ clicks + impressions + display + transactions + click.rate +
    conversion.rate

          Df Sum of Sq   RSS   AIC
- click.rate  1     67.9 192828 351.23
- transactions 1     75.1 192835 351.23
- conversion.rate 1   151.5 192911 351.24
- impressions  1   380.8 193141 351.29
- display      1   2787.2 195547 351.79
- clicks       1   3325.6 196085 351.90
<none>                    192760 353.21
+ spend       1     0.6 192759 355.21

Step: AIC=351.23
revenue ~ clicks + impressions + display + transactions + conversion.rate

          Df Sum of Sq   RSS   AIC
- transactions 1     47.4 192875 349.24
- conversion.rate 1   129.0 192957 349.25
- impressions  1   312.9 193141 349.29
- clicks       1   3425.7 196253 349.93
- display      1   3747.1 196575 350.00
<none>                    192828 351.23
+ click.rate   1     67.9 192760 353.21
+ spend        1     5.2 192822 353.23

Step: AIC=349.24
revenue ~ clicks + impressions + display + conversion.rate

          Df Sum of Sq   RSS   AIC
- conversion.rate 1     89.6 192965 347.26
- impressions    1   480.9 193356 347.34
- display        1   5437.2 198312 348.35
```

```

<none>                               192875 349.24
+ transactions      1      47.4 192828 351.23
+ click.rate        1      40.2 192835 351.23
+ spend             1      13.6 192861 351.23
- clicks            1     30863.2 223738 353.17

Step: AIC=347.26
revenue ~ clicks + impressions + display

          Df Sum of Sq   RSS   AIC
- impressions    1      399 193364 345.34
<none>                               192965 347.26
- display        1     14392 207357 348.13
+ conversion.rate 1      90 192875 349.24
+ click.rate      1      52 192913 349.24
+ spend           1      33 192932 349.25
+ transactions    1      8 192957 349.25
- clicks          1     35038 228002 351.93

Step: AIC=345.34
revenue ~ clicks + display

          Df Sum of Sq   RSS   AIC
<none>                               193364 345.34
+ impressions    1      399 192965 347.26
+ transactions   1      215 193149 347.29
+ conversion.rate 1      8 193356 347.34
+ click.rate      1      6 193358 347.34
+ spend           1      2 193362 347.34
- display         1     91225 284589 358.80
- clicks          1     606800 800164 400.15

```

The algorithm evaluates each variable's contribution, removing those that do not improve the AIC score. The process continues until no further improvement is possible, terminating after 6 iterations.

AIC values track the progression of model refinement. The initial full model has an AIC of 355.21, while the final selected model achieves a lower AIC of 345.34, indicating a better balance between model fit and complexity.

To examine the final model, we use:

```

summary(stepwise_model)

Call:
lm(formula = revenue ~ clicks + display, data = marketing)

Residuals:
    Min      1Q  Median      3Q      Max 
-141.89 -55.92  16.44  52.70  115.46 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  193364.00   192965.00   1.0000  0.3162    
clicks       606800.00   800164.00   0.7500  0.4550    
display      91225.00   284589.00   0.3200  0.7447    

```

```
(Intercept) -33.63248   28.68893  -1.172 0.248564
clicks       0.89517    0.08308   10.775 5.76e-13 ***
display      95.51462   22.86126   4.178 0.000172 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 72.29 on 37 degrees of freedom
Multiple R-squared:  0.7822, Adjusted R-squared:  0.7704
F-statistic: 66.44 on 2 and 37 DF,  p-value: 5.682e-13
```

Stepwise regression yields a simpler model with just two predictors: `clicks` and `display`. The resulting regression equation is:

$$\widehat{\text{revenue}} = -33.63 + 0.9 \times \text{clicks} + 95.51 \times \text{display}$$

Model performance improves on several fronts. The **Residual Standard Error (RSE)**, which measures average prediction error, decreases from 77.61 to 72.29. The **Adjusted R-squared** increases from 74% to 77%, suggesting that the final model explains a greater proportion of variability in revenue with fewer predictors—achieving the goal of improved parsimony and interpretability.

Practice: Try running stepwise regression on a different dataset, or compare "forward" and "backward" directions to "both". Do all approaches lead to the same final model?

Strengths, Limitations, and Considerations for Stepwise Regression

Stepwise regression offers a systematic approach to model selection, striking a balance between interpretability and computational efficiency. By iteratively refining the set of predictors, it helps identify a streamlined model without manually testing every possible combination. This makes it especially useful for moderate-sized datasets where full subset selection would be computationally intensive.

However, stepwise regression also has important limitations. The algorithm proceeds sequentially, evaluating one variable at a time rather than considering all subsets of predictors exhaustively. As a result, it may miss interactions or combinations of variables that jointly improve model performance. It is also susceptible to *overfitting*, particularly when applied to small datasets with many predictors. In such cases, the model may capture random noise rather than meaningful relationships, reducing its ability to generalize to

new data. Additionally, *multicollinearity* among predictors can distort coefficient estimates and inflate p-values, leading to misleading conclusions.

For high-dimensional datasets or situations requiring more robust predictor selection, alternative methods such as *LASSO* (Least Absolute Shrinkage and Selection Operator) and *Ridge Regression* are often more effective. These regularization techniques introduce penalties for model complexity, which stabilizes coefficient estimates and improves predictive accuracy. For a detailed introduction to these methods, see [An Introduction to Statistical Learning with Applications in R](#) (Gareth et al. 2013).

Thoughtful model specification remains a crucial part of regression analysis. By selecting predictors using principled criteria and validating model performance on representative data, we can construct models that are both interpretable and predictive. While stepwise regression has limitations, it remains a valuable tool—particularly for moderate-sized problems—when used with care and awareness of its assumptions.

10.9 Extending Linear Models to Capture Non-Linear Relationships

Imagine trying to predict house prices using the age of a property. A brand-new home might be more expensive than one that is 20 years old—but what about a 100-year-old historic house? In practice, relationships like this are rarely straight lines. Yet standard linear regression assumes exactly that: a constant rate of change between predictors and the response.

Linear regression models are valued for their simplicity, interpretability, and ease of implementation. They work well when the relationship between variables is approximately linear. However, when data shows curvature or other non-linear patterns, a linear model may underperform—resulting in poor predictions and misleading interpretations.

Earlier in this chapter, we used stepwise regression (Section 10.8) to refine model specification and reduce complexity. But while stepwise regression helps us choose which variables to include, it does not address how variables relate to the outcome. It assumes that relationships are linear in form. To address this limitation while preserving interpretability, we turn to *polynomial regression*—an extension of linear regression that captures non-linear trends by transforming predictors.

The Need for Non-Linear Regression

Linear regression assumes a constant rate of change, represented as a straight line. However, many real-world datasets show more complex dynamics. Consider the scatter plot in Figure 10.5, which shows the relationship between `unit.price` (price per unit area) and `house.age` in the `house` dataset. The orange line represents a simple linear regression fit—but it clearly misses the curvature present in the data.

As seen in the plot, the linear model underestimates prices for newer homes and overestimates them for older ones. This mismatch highlights the limitations of a strictly linear model.

To better model the observed trend, we can introduce non-linear terms into the regression equation. If the relationship resembles a curve, a quadratic model may be appropriate:

$$\text{unit.price} = b_0 + b_1 \times \text{house.age} + b_2 \times \text{house.age}^2$$

This formulation includes both the original predictor and its squared term, allowing the model to bend with the data. Although it includes a non-linear transformation, the model remains a *linear regression model* because it is linear in the parameters (b_0, b_1, b_2). The coefficients are still estimated using ordinary least squares.

The blue curve in Figure 10.5 shows the improved fit from a quadratic regression. Unlike the straight-line model, it adapts to the curvature in the data, producing a more accurate and visually aligned fit.

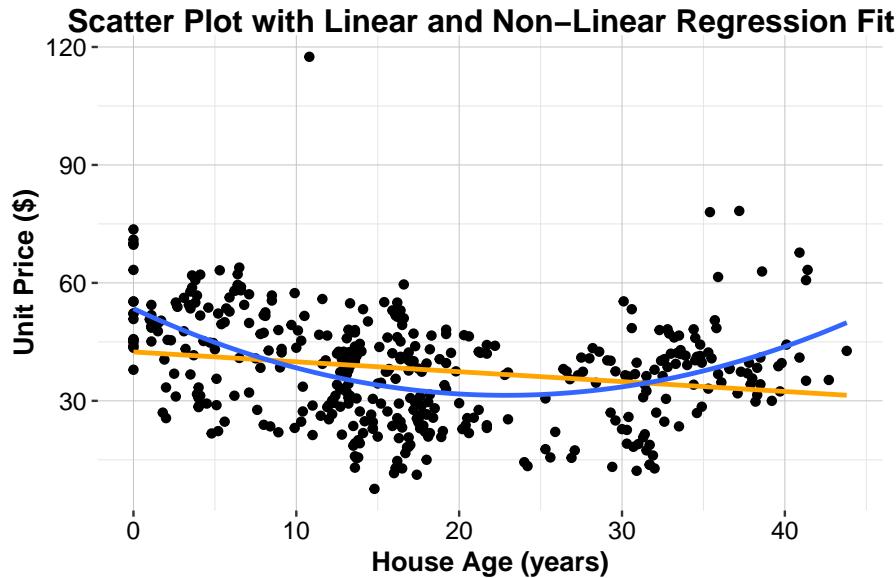


Figure 10.5: Scatter plot of house price (\$) versus house age (years) for the house dataset, with the fitted simple linear regression line in orange and the quadratic regression curve in blue.

This example illustrates the importance of adapting model structure when the linearity assumption does not hold. Polynomial regression extends our modeling vocabulary—allowing us to describe more realistic shapes in the data while keeping the model framework interpretable and statistically tractable.

Side Note: Is It Still Linear?

Although polynomial regression models curves, they are still called *linear models* because they are linear in their parameters. This is why tools like least squares and AIC remain valid—even when the relationship between the predictor and outcome is curved.

Now that we have seen how polynomial regression can capture non-linear relationships while preserving the linear modeling framework, we turn to its practical implementation in R. In the next section, we will fit polynomial regression models, interpret their output, and compare their performance to simpler linear models.

10.10 Polynomial Regression in Practice

Polynomial regression extends linear regression by incorporating higher-degree terms of the predictor variable, such as squared (x^2) or cubic (x^3) terms. This allows the model to capture non-linear relationships while remaining *linear in the coefficients*, meaning the model can still be estimated using ordinary least squares. The general form of a polynomial regression model is:

$$\hat{y} = b_0 + b_1 \times x + b_2 \times x^2 + \cdots + b_d \times x^d$$

where d represents the degree of the polynomial. While polynomial regression increases modeling flexibility, high-degree polynomials ($d > 3$) risk overfitting—capturing random noise, especially near the boundaries of the predictor range.

To illustrate polynomial regression, we use the `house` dataset from the `liver` package. This dataset includes housing prices and features such as age, proximity to public transport, and local amenities. Our goal is to model `unit.price` (price per unit area) as a function of `house.age` and compare the performance of simple linear and polynomial regression.

First, we load the dataset and examine its structure:

```
data(house)

str(house)
'data.frame':   414 obs. of  6 variables:
 $ house.age      : num  32 19.5 13.3 13.3 5 7.1 34.5 20.3 31.7 17.9 ...
 $ distance.to.MRT: num  84.9 306.6 562 562 390.6 ...
 $ stores.number   : int  10 9 5 5 5 3 7 6 1 3 ...
 $ latitude        : num  25 25 25 25 25 ...
 $ longitude       : num  122 122 122 122 122 ...
 $ unit.price      : num  37.9 42.2 47.3 54.8 43.1 32.1 40.3 46.7 18.8 22.1
  ..'
```

The dataset consists of 414 observations and 6 variables. The target variable is `unit.price`, while predictors include `house.age` (years), `distance.to.MRT`, `stores.number`, `latitude`, and `longitude`.

We begin by fitting a simple linear regression model:

```
simple_reg_house = lm(unit.price ~ house.age, data = house)

summary(simple_reg_house)

Call:
lm(formula = unit.price ~ house.age, data = house)
```

```

Residuals:
    Min      1Q  Median      3Q      Max
-31.113 -10.738   1.626   8.199  77.781

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 42.43470   1.21098 35.042 < 2e-16 ***
house.age   -0.25149   0.05752 -4.372 1.56e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 13.32 on 412 degrees of freedom
Multiple R-squared:  0.04434, Adjusted R-squared:  0.04202
F-statistic: 19.11 on 1 and 412 DF, p-value: 1.56e-05

```

The *R-squared* (R^2) value for this model is 0.04, indicating that only 4.43% of the variability in house prices is explained by `house.age`. This suggests the linear model may not fully capture the relationship.

Next, we fit a quadratic polynomial regression model to allow for curvature:

$$\text{unit.price} = b_0 + b_1 \times \text{house.age} + b_2 \times \text{house.age}^2$$

In **R**, this can be implemented using the `poly()` function, which fits orthogonal polynomials by default. These have numerical stability benefits but are less interpretable than raw powers:

```

reg_nonlinear_house = lm(unit.price ~ poly(house.age, 2), data = house)

summary(reg_nonlinear_house)

Call:
lm(formula = unit.price ~ poly(house.age, 2), data = house)

Residuals:
    Min      1Q  Median      3Q      Max
-26.542 -9.085 -0.445  8.260  79.961

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 37.980     0.599  63.406 < 2e-16 ***
poly(house.age, 2)1 -58.225    12.188 -4.777 2.48e-06 ***
poly(house.age, 2)2 109.635    12.188  8.995 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 12.19 on 411 degrees of freedom
Multiple R-squared:  0.2015, Adjusted R-squared:  0.1977
F-statistic: 51.87 on 2 and 411 DF, p-value: < 2.2e-16

```

The quadratic model yields a higher *Adjusted R-squared* (R^2) value of 0.2, compared to the simple model. Additionally, the *Residual Standard Error* (RSE) decreases from 13.32 to 12.19, indicating improved predictive accuracy. These improvements confirm that introducing a quadratic term helps capture the underlying curvature in the data.

Polynomial regression enhances linear models by allowing for flexible, curved fits. However, choosing the appropriate degree is critical—too low may underfit, while too high may overfit. More advanced methods, such as splines and generalized additive models, provide further flexibility with better control over complexity. These techniques are discussed in Chapter 7 of [An Introduction to Statistical Learning with Applications in R](#) (Gareth et al. 2013).

In the next sections, we will explore model validation and diagnostic techniques that help assess reliability and guide model improvement.

10.11 Diagnosing and Validating Regression Models

Before deploying a regression model, it is essential to validate its assumptions. Ignoring these assumptions is akin to constructing a house on an unstable foundation—predictions based on a flawed model can lead to misleading conclusions and costly mistakes. Model diagnostics ensure that the model is robust, reliable, and appropriate for inference and prediction.

Linear regression relies on several key assumptions:

1. *Linearity*: The relationship between the predictor(s) and the response should be approximately linear. Scatter plots or residuals vs. fitted plots help assess this.
2. *Independence*: Observations should be independent of one another. That is, the outcome for one case should not influence another.
3. *Normality*: The residuals (errors) should follow a normal distribution. This is typically checked using a Q-Q plot.
4. *Constant Variance (Homoscedasticity)*: The residuals should have roughly constant variance across all levels of the predictor(s). A residuals vs. fitted plot is used to examine this.

Violations of these assumptions undermine the reliability of coefficient estimates and associated inferential statistics. Even a model with a high R^2 may be inappropriate if its assumptions are violated.

To demonstrate model diagnostics, we evaluate the multiple regression model constructed in the example of Section 10.8 using the *marketing* dataset. The fitted model predicts daily revenue (revenue) based on clicks and display.

We generate diagnostic plots using:

```
stepwise_model = lm(revenue ~ clicks + display, data = marketing)
plot(stepwise_model)
```

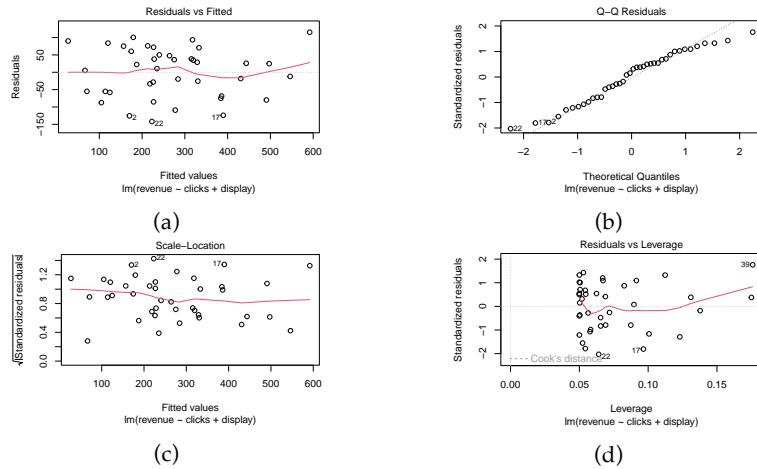


Figure 10.6: Diagnostic plots for assessing regression model assumptions.

These diagnostic plots provide visual checks of model assumptions:

- The *Normal Q-Q plot* (upper-right) evaluates whether residuals follow a normal distribution. Points that fall along the diagonal line support the normality assumption. In this case, the residuals align well with the theoretical quantiles.
- The *Residuals vs. Fitted plot* (upper-left) assesses both linearity and homoscedasticity. A random scatter with uniform spread supports both assumptions. Here, no strong patterns or funnel shapes are evident, suggesting that the assumptions are reasonable.
- The *Independence assumption* is not directly tested via plots but should be evaluated based on the study design. In the *marketing* dataset, each day's revenue is assumed to be independent from others, making this assumption plausible.

When examining these plots, ask yourself: - Do the residuals look randomly scattered, or do you notice any patterns? - Do the points in the Q-Q plot fall along the line, or do they curve away? - Is there a visible funnel shape in the residuals vs. fitted plot that might suggest heteroscedasticity?

Actively interpreting these patterns helps reinforce your understanding of model assumptions and deepens your statistical intuition.

Taken together, these diagnostics suggest that the fitted model satisfies the necessary assumptions for inference and prediction. When applying regression in practice, it is important to visually inspect these plots and ensure the assumptions hold.

When assumptions are violated, alternative modeling strategies may be necessary. *Robust regression* techniques can handle violations of normality or constant variance. *Non-linear models*, such as polynomial regression or splines, help address violations of linearity. *Transformations* (e.g., logarithmic or square root) can be applied to stabilize variance or normalize skewed residuals.

Validating regression models is fundamental to producing reliable, interpretable, and actionable results. By following best practices in model diagnostics and validation, we strengthen the statistical foundation of our analyses and build models that can be trusted for decision-making.

10.12 Case Study: Comparing Classifiers to Predict Customer Churn

Customer churn—when a customer discontinues service with a company—is a key challenge in subscription-based industries such as telecommunications, banking, and online platforms. Accurately predicting which customers are at risk of churning supports proactive retention strategies and can significantly reduce revenue loss.

In this case study, we apply three classification models introduced in earlier chapters of this book to predict churn using the *churn* dataset: Logistic Regression, Naive Bayes Classifier from Chapter 9, k-Nearest Neighbors (kNN) from Chapter 7.

Each model represents a distinct approach to classification. Logistic regression provides interpretable coefficients and probabilistic outputs. kNN is a non-parametric, instance-based learner that classifies observations based on similarity to their nearest neighbors. Naive Bayes offers a fast, probabilistic model that assumes conditional independence among predictors.

Guiding Questions: How do these models differ in how they handle decision boundaries and uncertainty? Which one do you think will perform best—and why?

Our goal is to compare these models using ROC curves and AUC, which offer threshold-independent measures of classification performance, as discussed in Chapter 8. To ensure a fair comparison, we use the same set of features and preprocessing steps for all three models.

The selected features, motivated by our earlier exploration in Section 4.3, include:

`account.length`, `voice.plan`, `voice.messages`, `intl.plan`, `intl.mins`, `intl.calls`, `day.mins`, `day.calls`, `eve.mins`, `eve.calls`, `night.mins`, `night.calls`, and `customer.calls`.

These features capture core aspects of a customer’s usage behavior and plan characteristics, making them informative for modeling churn.

We define the modeling formula used across all three classifiers:

```
formula = churn ~ account.length + voice.plan + voice.messages +  
    intl.plan + intl.mins + intl.calls +  
    day.mins + day.calls + eve.mins + eve.calls +  
    night.mins + night.calls + customer.calls
```

Try it yourself: Load the `churn` dataset using `data(churn, package = "liver")`, then use `str(churn)` to inspect its structure. What stands out about the variables? What is the distribution of `churn`?

This case study follows the Data Science Workflow introduced in Chapter 2. For research context, data understanding, and exploratory analysis of the `churn` dataset, see Section 4.3. In the next subsection, we begin the data preparation process by partitioning the dataset into training and testing sets.

Partitioning and Preprocessing

Partitioning the data into training and test sets is a standard step in predictive modeling that allows us to estimate how well a model will generalize to new observations. A carefully structured split helps ensure that model evaluation is both valid and unbiased.

To ensure consistency across chapters and reproducible results, we use the same partitioning strategy as in Chapter 7.7. The `partition()` function from the `liver` package splits the dataset into two non-overlapping subsets according to a specified ratio. Setting a random seed ensures that the partitioning results are reproducible:

```
set.seed(42)

data_sets = partition(data = churn, ratio = c(0.8, 0.2))

train_set = data_sets$part1
test_set = data_sets$part2

test_labels = test_set$churn
```

This setup assigns 80% of the data to the training set and reserves 20% for evaluation. The response labels from the test set are stored separately in `test_labels` for later comparison.

Reflection: Why do we partition the data before training, rather than evaluating the model on the full dataset?

In the next subsection, we train each classification model using the same formula and training data. We then generate predictions and evaluate their performance using ROC curves and AUC.

Training the Logistic Regression Model

We begin with logistic regression, a widely used baseline model for binary classification. It estimates the probability of customer churn using a linear combination of the selected predictors.

We fit the model using the `glm()` function, specifying the `binomial` family to model the binary outcome:

```
logistic_model = glm(formula = formula, data = train_set, family = binomial)
```

Next, we generate predicted probabilities on the test set:

```
logistic_probs = predict(logistic_model, newdata = test_set, type =
    "response")
```

The `predict()` function returns the estimated probability that each customer in the test set has churned—that is, the probability of `churn = "yes"`.

Reflection: How might we convert these predicted probabilities into binary class labels? What threshold would you use?

Training the Naive Bayes Model

We briefly introduced the Naive Bayes classifier and its probabilistic foundations in Chapter 9. Here, we apply the model to predict customer churn using the same features as the other classifiers.

Naive Bayes is a fast, probabilistic classifier that works well for high-dimensional and mixed-type data. It assumes that predictors are conditionally independent given the response, which simplifies the computation of class probabilities.

We fit a Naive Bayes model using the `naive_bayes()` function from the `naivebayes` package:

```
library(naivebayes)  
  
bayes_model = naive_bayes(formula, data = train_set)
```

Next, we use the `predict()` function to generate predicted class probabilities for the test set:

```
bayes_probs = predict(bayes_model, test_set, type = "prob")
```

The output `bayes_probs` is a matrix where each row corresponds to a test observation, and each column provides the predicted probability of belonging to either class (no or yes).

Reflection: How might Naive Bayes perform differently from logistic regression on this dataset, given its assumption of predictor independence?

Training the kNN Model

k-Nearest Neighbors (kNN) is a non-parametric method that classifies each test observation based on the majority class of its k closest neighbors in the training set. Because it relies on distance calculations, it is particularly sensitive to the scale of the input features.

We train a kNN model using the `kNN()` function from the `liver` package, setting $k = 5$. This choice is based on the results reported in Section 7.7, where $k = 5$ achieved the highest classification accuracy on the `churn` dataset.

We apply min-max scaling and binary encoding using the `scaler = "minmax"` option:

```
knn_probs = kNN(formula = formula, train = train_set,
                 test = test_set, k = 5, scaler = "minmax", type = "prob")
```

This ensures that all numeric predictors are scaled to the [0, 1] range, and binary categorical variables are appropriately encoded for use in distance computations.

For additional details on preprocessing and parameter selection, refer to Section 7.7.

Reflection: How might the model's performance change if we chose a much smaller or larger value of k ?

With predictions generated from all three models—logistic regression, Naive Bayes, and kNN—we are now ready to evaluate their classification performance using ROC curves and AUC.

Model Evaluation and Comparison

To evaluate and compare the performance of our classifiers across all possible classification thresholds, we use ROC curves and the Area Under the Curve (AUC) metric. As discussed in Chapter 8, the ROC curve plots the true positive rate against the false positive rate, and the AUC summarizes the curve into a single number—closer to 1 indicates better class separation.

ROC analysis is particularly useful when class distributions are imbalanced or when different classification thresholds need to be considered, as is often the case in churn prediction problems.

We compute the ROC curves using the pROC package:

```
library(pROC)

roc_logistic = roc(test_labels, logistic_probs)
roc_bayes   = roc(test_labels, bayes_probs[, "yes"])
roc_knn     = roc(test_labels, knn_probs[, "yes"])
```

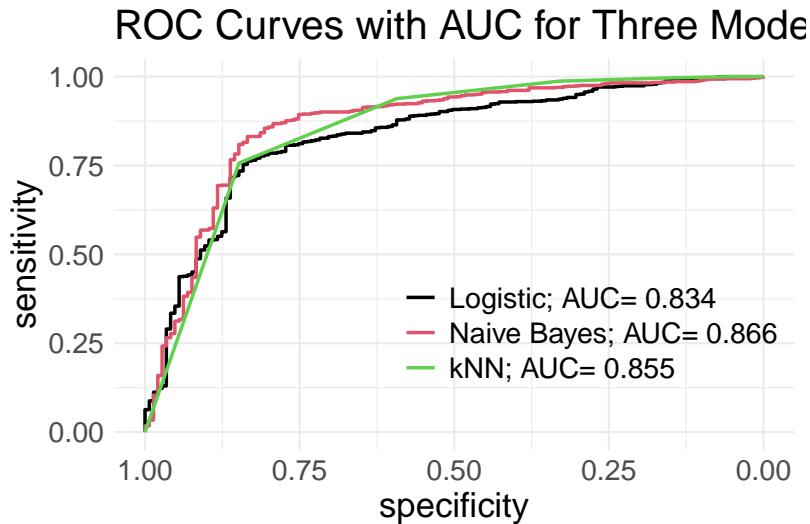
We visualize all three ROC curves in a single plot:

```
ggroc(list(roc_logistic, roc_bayes, roc_knn), size = 0.8) +
  theme_minimal() + ggtitle("ROC Curves with AUC for Three Models") +
  scale_color_manual(values = 1:3,
  labels = c(paste("Logistic; AUC=", round(auc(roc_logistic), 3)),
            paste("Naive Bayes; AUC=", round(auc(roc_bayes), 3))),
```

```

paste("kNN; AUC=", round(auc(roc_knn), 3))) +
theme(legend.title = element_blank()) +
theme(legend.position = c(.7, .3), text = element_text(size = 17))

```



In the ROC plot, each curve represents the performance of one classifier: logistic regression, Naive Bayes, and kNN. Higher curves and larger AUC values indicate stronger predictive performance.

The AUC values are:

- Logistic Regression: AUC = 0.834;
- Naive Bayes: AUC = 0.866;
- kNN: AUC = 0.855.

While kNN shows a slightly higher AUC, the differences are modest, and all three classifiers perform comparably on this task. This suggests that logistic regression and Naive Bayes remain viable choices—especially when interpretability, simplicity, or speed are more important than marginal performance gains.

Reflection: Would your choice of model change if interpretability or ease of deployment were more important than AUC?

Having compared the models based on ROC curves and AUC values, we now reflect on their overall strengths, limitations, and practical implications in the context of churn prediction.

Reflections and Takeaways

This case study demonstrates how comparing classification models involves more than just evaluating predictive accuracy—it also requires balancing interpretability, scalability, and domain-specific needs. While kNN achieved slightly higher AUC in this example, all three models—logistic regression, Naive Bayes, and kNN—performed similarly and offer different strengths.

- *Logistic regression* provides interpretable coefficients and is widely used in applied settings.
- *Naive Bayes* is computationally efficient, simple to implement, and performs well with categorical data.
- *kNN* is flexible and can capture non-linear relationships, but it is sensitive to feature scaling and computationally intensive for large datasets.

Model selection should reflect the broader goals of the analysis, including how results will be interpreted and used in decision-making. This reinforces the final step of the Data Science Workflow—*delivering results in context*.

Self-Reflection:

How do the modeling decisions made in this case study—such as the choice of predictors, model types (logistic regression, kNN, Naive Bayes), and evaluation methods (ROC curves, AUC)—apply to your own projects?

- When might you prioritize interpretability over accuracy?
- What would guide your selection of features or modeling approaches in your own work?
- Are there trade-offs in your field between transparency, speed, and predictive performance?

Taking a moment to map these ideas to your domain helps solidify what you've learned and prepares you to apply regression modeling effectively in practice.

10.13 Chapter Summary and Takeaways

This chapter introduced regression analysis as a central tool in data science for modeling relationships and making predictions. We began with simple linear regression, progressed to multiple regression, and then extended the framework through generalized linear models and polynomial regression.

Along the way, we explored how to:

- Interpret regression coefficients within the context of the problem,
- Assess model assumptions using diagnostic plots and residuals,
- Evaluate model performance with residual standard error (RSE), R-squared (R^2), and adjusted R^2 ,
- Select meaningful predictors using stepwise regression guided by model selection criteria such as AIC and BIC,
- Adapt regression models to binary and count outcomes using logistic and Poisson regression,
- Compare classifier performance using ROC curves and AUC in a practical case study on customer churn.

These techniques build upon earlier chapters and reinforce the importance of model transparency, reliability, and alignment with domain-specific goals. Regression models are not only statistical tools—they are instruments for reasoning about data and supporting informed decisions.

Reflection: Which type of regression model would be most appropriate for your next project? How does your choice depend on the type of outcome, the nature of the predictors, and your goal—interpretation or prediction?

In the next chapter, we explore decision trees and random forest methods. These models offer a different perspective—one that prioritizes interpretability through tree structures and improves performance through model aggregation.

10.14 Exercises

These exercises reinforce key ideas from the chapter, combining conceptual questions, interpretation of regression outputs, and practical implementation in R. The datasets used are included in the `liver` package.

Simple and Multiple Linear Regression (House, Insurance, and Cereal Datasets)

Conceptual Understanding

1. How does simple linear regression differ from multiple linear regression?
2. List the key assumptions of linear regression. Why do they matter?
3. What does the R-squared (R^2) value tell us about a regression model?
4. Compare Residual Standard Error (RSE) and R^2 . What does each measure?
5. What is multicollinearity, and how does it affect regression models?
6. Why is Adjusted R^2 preferred over R^2 in models with multiple predictors?
7. How are categorical variables handled in regression models in R?

Applications Using the House Dataset

```
data(house, package = "liver")
```

8. Fit a model predicting unit.price using house.age. Summarize the results.
9. Add distance.to.MRT and stores.number as predictors. Interpret the updated model.
10. Predict unit.price for homes aged 10, 20, and 30 years.
11. Evaluate whether including latitude and longitude improves model performance.
12. Report the RSE and R^2 . What do they suggest about the model's fit?
13. Create a residual plot. What does it reveal about model assumptions?
14. Use a Q-Q plot to assess the normality of residuals.

Applications Using the Insurance Dataset

```
data(insurance, package = "liver")
```

15. Model charges using age, bmi, children, and smoker.
16. Interpret the coefficient for smoker.
17. Include an interaction between age and bmi. Does it improve the model?
18. Add region as a predictor. Does Adjusted R^2 increase?
19. Use stepwise regression to find a simpler model with comparable performance.

Applications Using the Cereal Dataset

```
data(cereal, package = "liver")
```

20. Model rating using calories, protein, sugars, and fiber.
21. Which predictor appears to have the strongest impact on rating?
22. Should sodium be included in the model? Support your answer.
23. Compare the effects of fiber and sugars.
24. Use stepwise regression to identify a more parsimonious model.

Polynomial Regression (House Dataset)

Conceptual Understanding

25. What is polynomial regression, and how does it extend linear regression?
26. Why is polynomial regression still considered a linear model?
27. What risks are associated with using high-degree polynomials?
28. How can you determine the most appropriate polynomial degree?
29. What visual or statistical tools can help detect overfitting?

Applications Using the House Dataset

30. Fit a quadratic model for `unit.price` using `house.age`. Compare it to a linear model.
31. Fit a cubic model. Is there evidence of improved performance?
32. Plot the linear, quadratic, and cubic fits together.
33. Use cross-validation to select the optimal polynomial degree.
34. Interpret the coefficients of the quadratic model.

Logistic Regression (Bank Dataset)

Conceptual Understanding

35. What distinguishes logistic regression from linear regression?
36. Why does logistic regression use the logit function?
37. Explain how to interpret an odds ratio.
38. What is a confusion matrix, and how is it used?
39. Distinguish between precision and recall in classification evaluation.

Applications Using the Bank Dataset

```
data(bank, package = "liver")
```

40. Predict `y` using `age`, `balance`, and `duration`.
41. Interpret model coefficients as odds ratios.
42. Estimate the probability of subscription for a new customer.
43. Generate a confusion matrix to assess prediction performance.
44. Report accuracy, precision, recall, and F1-score.
45. Apply stepwise regression to simplify the model.
46. Plot the ROC curve and compute the AUC.

Stepwise Regression (House Dataset)

47. Use stepwise regression to model `unit.price`.
48. Compare the stepwise model to the full model.
49. Add interaction terms. Do they improve model performance?

Model Diagnostics and Validation

50. Check linear regression assumptions for the multiple regression model on `house`.
51. Generate diagnostic plots: residuals vs fitted, Q-Q plot, and scale-location plot.
52. Apply cross-validation to compare model performance.
53. Compute and compare mean squared error (MSE) across models.
54. Does applying a log-transformation improve model accuracy?

Self-Reflection

55. Think of a real-world prediction problem you care about—such as pricing, health outcomes, or consumer behavior. Which regression technique covered in this chapter would be most appropriate, and why?

Chapter 11

Decision Trees and Random Forests

Imagine a bank evaluating loan applications. Given details such as income, age, credit history, and debt-to-income ratio, how does the bank decide whether to approve or reject a loan? Similarly, how do online retailers recommend products based on customer preferences? These decisions, which mimic human reasoning, are often powered by *decision trees*—a simple yet powerful machine learning technique that classifies data by following a series of logical rules.

Decision trees are used across diverse domains—from medical diagnosis and fraud detection to customer segmentation and automation. Their intuitive nature makes them highly interpretable, enabling data-driven decisions without requiring deep mathematical expertise. However, while individual trees are easy to understand, they are prone to overfitting, capturing noise in the data rather than general patterns. *Random forests* address this limitation by combining multiple decision trees to produce a more accurate and stable model.

In the previous chapter, you learned how to build and evaluate regression models to predict continuous outcomes. In this chapter, we return to supervised learning from a new angle: *tree-based models*, which unify classification and regression under a single, flexible framework. Decision trees can automatically capture nonlinear relationships and interactions between variables—without requiring manual feature engineering or transformations.

To see decision trees in action, consider the example in Figure 11.1, which predicts whether a customer’s credit risk is classified as “good” or “bad” based on features such as age and income. This tree is trained on the *risk* dataset, introduced in Chapter 9, and consists of decision nodes representing yes/no questions, such as whether yearly income is below €36,000 ($\text{income} < 36e+3$) or whether age is greater than 29. The final classification is determined at the terminal nodes, also known as leaves.

Curious how this tree was built from real data? In the next sections, we will walk through each step of the process—from data to decision.

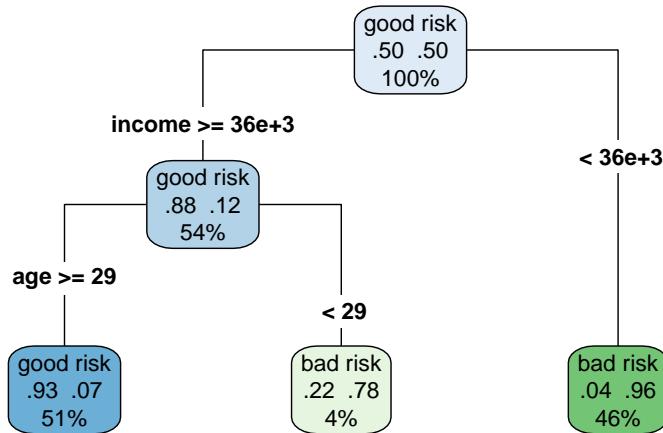


Figure 11.1: A classification tree built using the CART algorithm on the risk dataset to predict credit risk based on age and income. Terminal nodes display predicted class and class probabilities—highlighting CART’s transparent, rule-based structure.

Decision trees are highly interpretable, making them especially valuable in domains such as finance, healthcare, and marketing, where understanding model decisions is as important as accuracy. Their structured form allows for easy visualization of decision pathways, helping businesses with customer segmentation, risk assessment, and process optimization.

In this chapter, we continue building on the *Data Science Workflow* introduced in Chapter 2. So far, we have learned how to prepare and explore data, apply classification methods—*k*-Nearest Neighbors (Chapter 7) and *Naive Bayes* (Chapter 9)—as well as regression models (Chapter 10), and evaluate performance (Chapter 8). Decision trees and random forests now offer a powerful, non-parametric modeling strategy. They can handle both classification and regression tasks effectively.

What This Chapter Covers

This chapter continues the modeling journey by building on what you learned in the previous chapters—both classification methods such as

k-Nearest Neighbors and *Naive Bayes*, and regression models. Decision trees offer a flexible, non-parametric approach that can model complex relationships and interactions without requiring predefined equations. Their adaptability often leads to strong predictive performance in both classification and regression settings, especially when extended to ensemble methods like random forests.

You will begin by learning how decision trees make predictions by recursively splitting the data into increasingly homogeneous subsets. We introduce two widely used algorithms—CART and C5.0—and explore how they differ in structure, splitting criteria, and performance. From there, you will discover *random forests*, an ensemble approach that builds multiple trees and aggregates their predictions for improved accuracy and generalization.

This chapter includes hands-on modeling examples using datasets on credit risk, income prediction, and customer churn. You will learn to interpret decision rules, assess model complexity, tune hyperparameters, and evaluate models using tools such as confusion matrices, ROC curves, and variable importance plots.

By the end of this chapter, you will be able to build, interpret, and evaluate tree-based models for both categorical and numeric outcomes. You will also understand when decision trees and random forests are the right tools for your data science problems, especially when balancing interpretability with predictive power.

11.1 How Decision Trees Work

Are you interested in learning how to build decision trees like the one in Figure 11.1, trained on real-world data? In this section, we unpack the core ideas behind decision trees—how they decide where to split, how they grow, and how they ultimately classify or predict outcomes.

A decision tree makes predictions by recursively partitioning the data into increasingly homogeneous groups based on feature values. At each split, it chooses the question that best separates the data, gradually forming a tree-like structure of decision rules. This *divide-and-conquer* approach is intuitive, flexible, and capable of modeling both categorical and numerical outcomes—making decision trees a popular choice in many data science applications.

The quality of a split is assessed using a metric such as the *Gini Index* or *Entropy*, which are introduced in the following sections. The tree continues growing until it meets a stopping criterion—for example, a maximum depth,

a minimum number of observations per node, or a lack of further improvement in predictive power.

To see this process in action, consider a simple dataset with two features (x_1 and x_2) and two classes (Class A and Class B), as shown in Figure 11.2. The dataset consists of 50 data points, and the goal is to classify them into their respective categories.

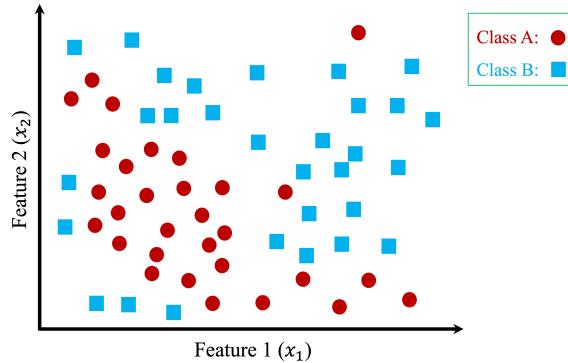


Figure 11.2: A toy dataset with two features and two classes (Class A and Class B) with 50 observations (points). This example is used to illustrate the construction of a decision tree.

The process begins by identifying the feature and threshold that best separate the two classes. The algorithm evaluates all possible splits and selects the one that most improves the homogeneity in the resulting subsets. For this dataset, the optimal split occurs at $x_1 = 10$, dividing the dataset into two regions:

- The left region contains data points where $x_1 < 10$, with 80% belonging to Class A and 20% to Class B.
- The right region contains data points where $x_1 \geq 10$, with 28% in Class A and 72% in Class B.

This first split is illustrated in Figure 11.3, where the decision boundary is drawn at $x_1 = 10$.

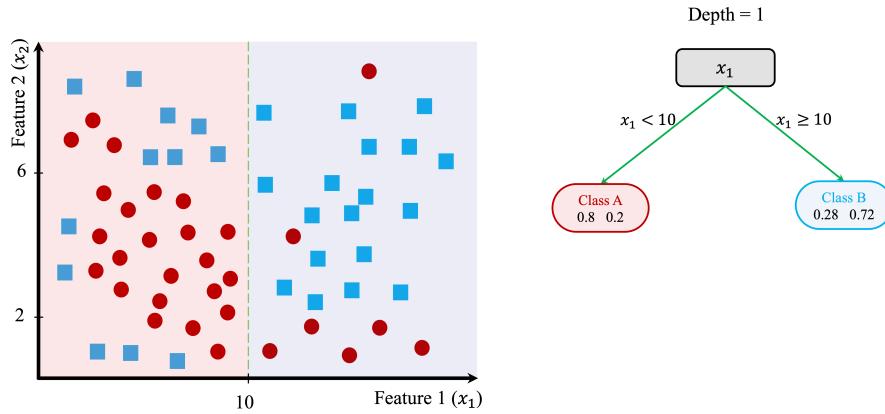


Figure 11.3: Left: Decision boundary for a tree with depth 1. Right: The corresponding Decision Tree.

Although this split improves class separation, some overlap remains, suggesting that further refinement is needed. The tree-building process continues by introducing additional splits based on x_2 , creating smaller, more homogeneous groups.

In Figure 11.4, the algorithm identifies new thresholds: $x_2 = 6$ for the left region and $x_2 = 8$ for the right region. These additional splits refine the classification process, improving the model's ability to distinguish between the two classes.

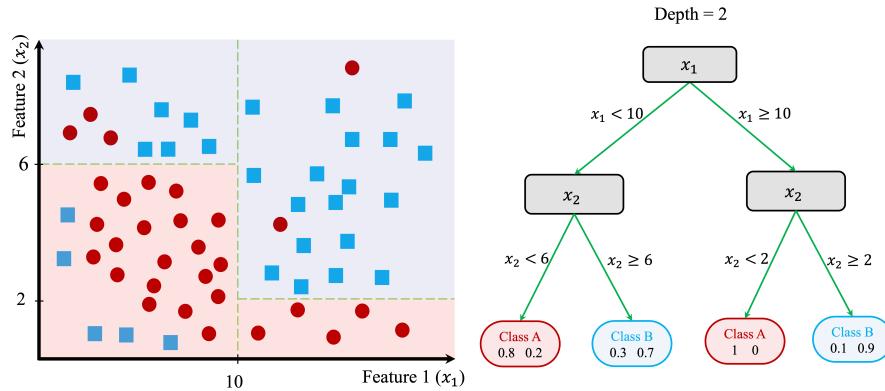


Figure 11.4: Left: Decision boundary for a tree with depth 2. Right: The corresponding Decision Tree.

This recursive process continues until the tree reaches a stopping criterion. Figure 11.5 shows a fully grown tree with a depth of 5, demonstrating how decision trees create increasingly refined decision boundaries.

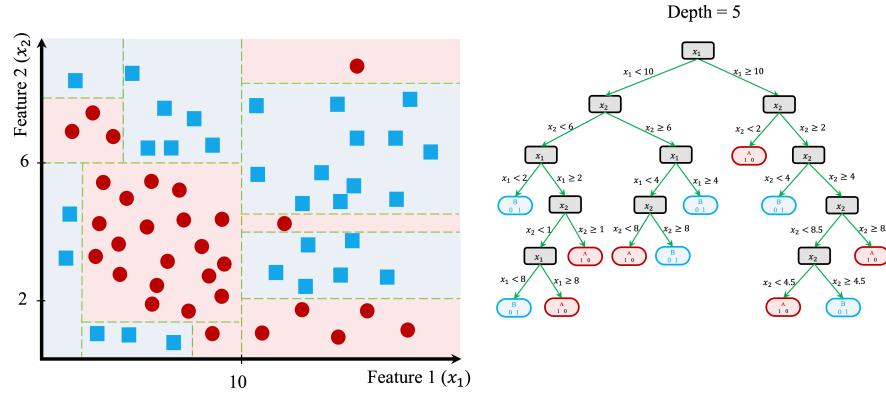


Figure 11.5: Left: Decision boundary for a tree with depth 5. Right: The corresponding Decision Tree.

At this depth, the tree has created highly specific decision boundaries that closely match the training data. While this deep tree perfectly classifies the training data, it may not generalize well to new observations. The model has likely captured not just meaningful patterns but also noise, a problem known as *overfitting*. Overfitted trees perform well on training data but struggle to make accurate predictions on unseen data.

In the next subsection, we explore how decision trees make predictions and how their structure influences interpretability.

Making Predictions with a Decision Tree

After a decision tree is built, making predictions involves following the decision rules from the root node down to a leaf. Each split narrows the possibilities, leading to a final classification or numeric prediction at the leaf.

For classification tasks, the tree assigns a new observation to the most common class in the leaf where it ends up. For regression tasks, the predicted outcome is the average target value of the data points in that leaf.

To illustrate, consider a new data point with $x_1 = 8$ and $x_2 = 4$ in Figure 11.4. The tree classifies it by following these steps:

1. Since $x_1 = 8$, the point moves to the left branch ($x_1 < 10$).
2. Since $x_2 = 4$, the point moves to the lower-left region ($x_2 < 6$).
3. The final leaf node assigns the point to Class A with 80% confidence.

This step-by-step path makes decision trees highly interpretable—especially in settings where knowing *why* a decision was made is just as important as the prediction itself.

Controlling Tree Complexity

Have you ever wondered why a decision tree that performs perfectly on training data sometimes fails miserably on new data? This is the classic pitfall of *overfitting*—where a model becomes so tailored to the training data that it mistakes noise for signal.

Like a gardener shaping a tree, we must decide how much growth to allow. If we let the branches grow unchecked, the tree captures every detail—but may become too complex and fragile. To strike the right balance, decision trees rely on techniques that control complexity and improve generalization.

One approach is *pre-pruning*, which restricts tree growth during training. The algorithm stops splitting when it hits limits such as a maximum depth, a minimum number of observations per node, or insufficient improvement in the splitting criterion. Pre-pruning acts like early shaping—preventing the model from becoming too specific too soon.

Another approach is *post-pruning*, where the tree is first grown to its full depth and then trimmed. After training, branches that add little to predictive accuracy are removed or merged. Post-pruning is like sculpting the tree after seeing its full form—often resulting in simpler, more interpretable models.

Which pruning strategy works best depends on the problem and dataset. In either case, the way we assess splits—using criteria like the *Gini Index* or *Entropy*—shapes the tree’s structure and performance. We will delve into these splitting metrics next.

11.2 How CART Builds Decision Trees

How do decision trees actually decide where to split? One of the most influential algorithms that answers this question is *CART*—short for *Classification and Regression Trees*. Introduced by Breiman et al. in 1984 (Breiman et

al. 1984), CART remains a foundational tool in both academic research and applied machine learning. Let us take a closer look at how it works—and why it continues to be so popular.

CART generates *binary trees*, meaning that each decision node always results in two branches. It recursively splits the dataset into subsets of records that are increasingly similar with respect to the target variable. This is achieved by choosing the split that results in the *purest* possible child nodes—where each node contains mostly one class.

For classification tasks, CART typically uses the *Gini index* to measure node impurity. The Gini index is defined as:

$$Gini = 1 - \sum_{i=1}^k p_i^2$$

where p_i represents the proportion of samples in the node that belong to class i , and k is the total number of classes. A node is considered pure when all data points in it belong to a single class, resulting in a Gini index of zero. During tree construction, CART selects the feature and threshold that result in the largest reduction in impurity, splitting the data to create two more homogeneous child nodes.

A simple example of a CART decision tree can be seen in Figure Figure 11.1.

The recursive nature of CART can lead to highly detailed trees that fit the training data perfectly. While this minimizes the error rate on the training set, it often results in overfitting, where the tree becomes overly complex and fails to generalize to unseen data. To mitigate this, CART employs pruning techniques to simplify the tree.

Pruning involves trimming branches that do not contribute meaningfully to predictive accuracy on a validation set. This is achieved by finding an adjusted error rate that penalizes overly complex trees with too many leaf nodes. The goal of pruning is to balance accuracy and simplicity, enhancing the tree's ability to generalize to new data. The pruning process is discussed in detail by Breiman et al. (Breiman et al. 1984).

Despite its simplicity, CART is widely used in practice due to its interpretability, versatility, and ability to handle both classification and regression tasks. The tree structure provides an intuitive way to visualize decision-making, making it highly explainable. Additionally, CART works well with both numerical and categorical data, making it applicable across a range of domains.

However, CART has limitations. The algorithm tends to produce deep trees that may overfit the training data, particularly when the dataset is small or noisy. Its reliance on greedy splitting can also result in suboptimal splits, as

it evaluates one feature at a time rather than considering all possible combinations.

To address these shortcomings, more advanced algorithms have been developed, such as *C5.0*, which incorporates improvements in splitting and pruning techniques, and *random forests*, which combine multiple decision trees to create more robust models. These approaches build on the foundations of *CART*, improving performance and reducing susceptibility to overfitting. The following sections explore these methods in detail.

11.3 C5.0: More Flexible Decision Trees

How can we improve on the classic decision tree? *C5.0*, developed by J. Ross Quinlan, offers an answer through smarter splitting, more flexible tree structures, and greater computational efficiency. As an evolution of earlier algorithms such as *ID3* and *C4.5*, it introduces enhancements that have made it widely used in both research and real-world applications. While a commercial version of *C5.0* is available through [RuleQuest](#), open-source implementations are integrated into **R** and other data science tools.

C5.0 differs from other decision tree algorithms, such as *CART*, in several key ways:

- Multi-way splits: Unlike *CART*, which constructs strictly binary trees, *C5.0* allows for multi-way splits, particularly for categorical attributes. This flexibility often results in more compact and interpretable trees.
- Entropy-based splitting: *C5.0* uses entropy and information gain, concepts from information theory, to evaluate node purity—whereas *CART* relies on the Gini index or variance reduction.

Entropy measures the degree of disorder in a dataset. Higher entropy indicates more class diversity; lower entropy suggests more homogeneous groups. *C5.0* aims to find splits that reduce this disorder, creating purer subsets at each node. For a variable x with k classes, entropy is defined as:

$$\text{Entropy}(x) = - \sum_{i=1}^k p_i \log_2(p_i),$$

where p_i is the proportion of observations in class i .

A dataset with all observations in one class has entropy 0 (maximum purity), while equal class distribution yields maximum entropy. When a dataset is

split, the entropy of each resulting subset is weighted by its size and combined:

$$H_S(T) = \sum_{i=1}^c \frac{|T_i|}{|T|} \times Entropy(T_i),$$

where T is the original dataset, and T_1, \dots, T_c are the resulting subsets from split S . The information gain from the split is then:

$$gain(S) = H(T) - H_S(T).$$

This value quantifies the improvement in class purity. C5.0 evaluates all possible splits and chooses the one that maximizes information gain.

A Simple C5.0 Example

To illustrate how C5.0 constructs decision trees, consider its application to the *risk* dataset, which classifies a customer's credit risk as *good* or *bad* based on features such as age and income. Figure Figure 11.6 shows the tree generated by the `C5.0()` function from the `C50` package in R.

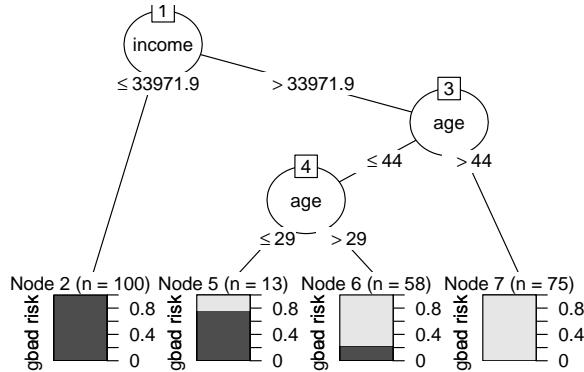


Figure 11.6: C5.0 Decision Tree trained on the risk dataset. Unlike CART, this tree allows multi-way splits and uses entropy-based splitting criteria to classify credit risk.

This tree illustrates several of C5.0's features. While the earlier CART model in Figure Figure 11.1 used only binary splits, C5.0 enables multi-way splits

when appropriate—especially useful when working with categorical features that have many levels. This often produces shallower trees that are easier to interpret without sacrificing accuracy.

Advantages and Limitations

C5.0 offers several advantages over earlier decision tree algorithms. It is computationally efficient, making it suitable for large datasets and high-dimensional feature spaces. Its ability to perform multi-way splits leads to more compact trees, particularly when working with categorical variables that have many levels. Additionally, C5.0 includes mechanisms for weighting features, enabling the model to prioritize the most informative predictors. The algorithm also incorporates automatic pruning during training, which helps prevent overfitting and improves generalizability.

Despite these strengths, C5.0 is not without limitations. The trees it produces can become overly complex, especially in the presence of irrelevant predictors or categorical attributes with many distinct values. Furthermore, the evaluation of multi-way splits can be computationally demanding, particularly when the number of candidate splits grows large. Nonetheless, the internal optimizations of the algorithm help mitigate these concerns in practice.

In summary, C5.0 builds on the strengths of earlier decision tree models by combining entropy-based splitting with flexible tree structures. Its capacity to adapt to diverse data types while maintaining interpretability makes it a valuable tool for a wide range of classification problems. In the next section, we shift focus to *random forests*—an ensemble technique that aggregates many decision trees to further improve predictive performance.

11.4 Random Forests: Boosting Accuracy with an Ensemble of Trees

What if you could take a room full of decision trees—each trained slightly differently—and let them vote on the best prediction? This is the idea behind *random forests*, one of the most popular and effective ensemble methods in modern machine learning.

While individual decision trees offer clarity and interpretability, they are prone to overfitting, especially when allowed to grow deep and unpruned. Random forests overcome this limitation by aggregating the predictions of

many diverse trees, each trained on a different subset of the data and using different subsets of features. This ensemble approach leads to models that are both more robust and more accurate.

Two key sources of randomness lie at the heart of random forests. The first is *bootstrap sampling*, where each tree is trained on a randomly sampled version of the training data (with replacement). The second is *random feature selection*, where only a subset of predictors is considered at each split. These two ingredients encourage diversity among the trees and prevent any single predictor or pattern from dominating the ensemble.

Once all trees are trained, their predictions are aggregated. In classification tasks, the forest chooses the class that receives the most votes across trees. For regression, it averages the predictions. This aggregation smooths over individual errors, reducing variance and improving generalization.

Strengths and Limitations of Random Forests

Random forests are known for their strong predictive performance, particularly on datasets with complex interactions, nonlinear relationships, or high dimensionality. They typically outperform individual decision trees and are less sensitive to noise and outliers. Importantly, they also provide *variable importance scores*, helping analysts identify the most influential features in a model.

However, these strengths come with trade-offs. Random forests are less interpretable than single trees. Although we can assess overall variable importance, it is difficult to trace how a specific prediction was made. In addition, training and evaluating hundreds of trees can be computationally demanding, especially for large datasets or time-sensitive applications.

Nonetheless, the balance random forests strike between accuracy and robustness has made them a cornerstone of predictive modeling. Whether predicting customer churn, disease outcomes, or financial risk, random forests offer a powerful and reliable tool.

In the next section, we move from theory to practice. Using a real-world income dataset, we compare decision trees and random forests to explore how ensemble learning enhances performance and why it often becomes the go-to choice in applied data science.

11.5 Case Study: Who Can Earn More Than \$50K Per Year?

Predicting income levels is a common task in fields such as finance, marketing, and public policy. Banks use income models to assess creditworthiness, employers rely on them to benchmark compensation, and governments use them to inform taxation and welfare programs. In this case study, we apply decision trees and random forests to classify individuals based on their likelihood of earning more than \$50,000 annually.

The analysis is based on the *adult* dataset, a widely used benchmark from the [US Census Bureau](#), available in the **liver** package. This dataset, introduced earlier in Section [Section 3.16](#), includes demographic and employment-related attributes such as education, work hours, marital status, and occupation—factors that influence earning potential.

Following the *Data Science Workflow* introduced in Chapter [2](#) and illustrated in Figure [2.3](#), we guide you through each stage of the process—from data preparation to modeling and evaluation. You will learn how to apply three tree-based algorithms—*CART*, *C5.0*, and *random forest*—to a real-world classification problem using **R**. Each step is grounded in the workflow to ensure reproducibility, clarity, and alignment with best practices in data science.

Overview of the Dataset

The *adult* dataset, included in the **liver** package, is a classic benchmark in predictive modeling. It originates from the US Census Bureau and contains demographic and employment information about individuals, making it ideal for studying income classification problems in a realistic setting.

To begin, we load the dataset into **R** and generate a summary:

```
library(liver)

data(adult)

summary(adult)
  age          workclass      demogweight      education
  Min.   :17.0    ?       : 2794    Min.   : 12285  HS-grad   :15750
  1st Qu.:28.0   Gov     : 6536    1st Qu.: 117550 Some-college:10860
  Median :37.0   Never-worked:   10    Median : 178215 Bachelors  : 7962
  Mean   :38.6   Private   :33780    Mean   : 189685 Masters   : 2627
  3rd Qu.:48.0   Self-emp  : 5457    3rd Qu.: 237713 Assoc-voc  : 2058
  Max.   :90.0   Without-pay:    21   Max.   :1490400  11th      : 1812
                                         (Other)   : 7529
  education.num      marital.status      occupation

```

```

Min. : 1.00   Divorced    : 6613   Craft-repair : 6096
1st Qu.: 9.00   Married     :22847   Prof-specialty : 6071
Median :10.00   Never-married:16096   Exec-managerial: 6019
Mean   :10.06   Separated    : 1526   Adm-clerical  : 5603
3rd Qu.:12.00   Widowed     : 1516   Sales         : 5470
Max.   :16.00                               Other-service : 4920
                                         (Other)       :14419
                                         relationship   race      gender
Husband        :19537   Amer-Indian-Eskimo: 470   Female:16156
Not-in-family  :12546   Asian-Pac-Islander: 1504   Male  :32442
Other-relative : 1506   Black          : 4675
Own-child      : 7577   Other          : 403
Unmarried      : 5118   White          :41546
Wife           : 2314

capital.gain    capital.loss    hours.per.week      native.country
Min. : 0.0   Min. : 0.00   Min. : 1.00   United-States:43613
1st Qu.: 0.0   1st Qu.: 0.00   1st Qu.:40.00   Mexico      : 949
Median : 0.0   Median : 0.00   Median :40.00   ?          : 847
Mean   : 582.4  Mean  : 87.94  Mean  :40.37   Philippines : 292
3rd Qu.: 0.0   3rd Qu.: 0.00   3rd Qu.:45.00   Germany    : 206
Max.   :41310.0  Max. :4356.00  Max. :99.00   Puerto-Rico : 184
                                         (Other)       : 2507
                                         income
<=50K:37155
>50K :11443

```

The dataset contains 48598 observations and 15 variables. The target variable is `income`, a binary factor with two levels: `<=50K` and `>50K`. The remaining 14 variables provide rich predictive features, spanning demographic characteristics, employment details, financial indicators, and household context.

These predictors fall into the following thematic groups:

- *Demographics*: `age`, `gender`, `race`, and `native.country`.
- *Education and employment*: `education`, `education.num`, `workclass`, `occupation`, and `hours.per.week`.
- *Financial status*: `capital.gain` and `capital.loss`.
- *Household and relationships*: `marital.status` and `relationship`.

For example, `education.num` captures the total years of formal education, while `capital.gain` and `capital.loss` reflect financial investment outcomes—factors that plausibly affect earning potential. Some predictors,

such as `native.country`, include many unique categories (42 levels), which we will address during preprocessing.

This diversity of attributes makes the `adult` dataset well suited for exploring classification models like decision trees and random forests. For full documentation, see the [package reference](#).

Data Preparation

Before building predictive models, it is essential to clean and preprocess the dataset to address missing values and simplify complex categorical variables. The `adult` dataset includes several features that require attention, particularly for improving model interpretability and robustness.

As discussed in Chapter 3, data preparation is a foundational step in the Data Science Workflow. In this case study, we summarize the most relevant transformations applied to prepare the `adult` dataset for modeling.

Handling Missing Values

Missing values in the dataset are encoded as "?". These need to be replaced with standard NA values before proceeding. Unused factor levels are removed, and categorical variables with missing entries are imputed using random sampling from the observed categories—a simple but effective strategy for preserving variable distributions.

```
library(Hmisc)

# Replace "?" with NA and remove unused levels
adult[adult == "?"] = NA
adult = droplevels(adult)

# Impute missing categorical values using random sampling
adult$workclass      = impute(factor(adult$workclass), 'random')
adult$native.country = impute(factor(adult$native.country), 'random')
adult$occupation     = impute(factor(adult$occupation), 'random')
```

Transforming Categorical Variables

Several categorical variables, such as `native.country` and `occupation`, contain a large number of unique levels. To improve model interpretability and

reduce complexity, we group these levels into broader, conceptually meaningful categories.

The `native.country` variable is consolidated into five geographic regions:

```
library(forcats)

Europe = c("England", "France", "Germany", "Greece", "Holland-Netherlands",
          ↵ "Hungary",
          "Ireland", "Italy", "Poland", "Portugal", "Scotland", "Yugoslavia")

Asia = c("China", "Hong", "India", "Iran", "Cambodia", "Japan", "Laos",
        "Philippines", "Vietnam", "Taiwan", "Thailand")

N.America = c("Canada", "United-States", "Puerto-Rico")

S.America = c("Columbia", "Cuba", "Dominican-Republic", "Ecuador",
             ↵ "El-Salvador",
             "Guatemala", "Haiti", "Honduras", "Mexico", "Nicaragua",
             "Outlying-US(Guam-USVI-etc)", "Peru", "Jamaica",
             ↵ "Trinidad&Tobago")

# Reclassify into broader regions
adult$native.country = fct_collapse(adult$native.country,
                                      "Europe"      = Europe,
                                      "Asia"         = Asia,
                                      "N.America"    = N.America,
                                      "S.America"    = S.America,
                                      "Other"        = c("South"))
```

Other categorical variables are adjusted to improve consistency:

- The `workclass` variable includes rare categories that reflect a lack of formal employment. These are grouped as "Unemployed":

```
adult$workclass = fct_collapse(adult$workclass, "Unemployed" =
                                ↵ c("Never-worked", "Without-pay"))
```

- For `race`, long or uncommon labels are simplified:

```
adult$race = fct_recode(adult$race, "Amer-Indian" = "Amer-Indian-Eskimo",
                        "Asian" = "Asian-Pac-Islander")
```

These transformations reduce the risk of overfitting and help the tree-based models generalize better. With these steps complete, the dataset is ready for training and evaluation, which we turn to in the next section.

Setup Data for Modeling

With the dataset cleaned and categorical variables simplified, we are ready to set up the data for training and evaluation. This corresponds to **Step 4: Setup Data for Modeling** in the Data Science Workflow introduced in Chapter 2 and discussed in detail in Chapter 6. It marks the transition from data preparation to model development.

To evaluate how well our models generalize to new data, we divide the dataset into two parts: a *training set* (80%) for model building and a *testing set* (20%) for performance assessment. This ensures an unbiased estimate of model accuracy on unseen data. Following the convention used in previous chapters, we use the `partition()` function from the `liver` package:

```
set.seed(6)

data_sets = partition(data = adult, ratio = c(0.8, 0.2))

train_set = data_sets$part1
test_set = data_sets$part2

test_labels = test_set$income
```

Here, `set.seed()` ensures reproducibility of the split. The `train_set` is used to train the classification models, and `test_set` serves as a holdout sample for evaluation. The vector `test_labels` contains the true income classes for the test observations, which will later be compared with predicted values from the CART, C5.0, and Random Forest models.

To ensure the training and test sets reflect the structure of the original dataset, we verified that the distribution of the `income` variable remains balanced after partitioning. While we do not show this diagnostic here, we refer interested readers to Section 6.5 for more on validation techniques.

The set of predictors used for modeling includes variables spanning demographic, economic, and employment dimensions:

```
age, workclass, education.num, marital.status, occupation,
race, gender, capital.gain, capital.loss, hours.per.week, native.country
```

The following variables are excluded for the reasons below:

- `demogweight`: Treated as an identifier and does not contain predictive information.
- `education`: Duplicates the information in `education.num`, which captures years of education numerically.

- **relationship:** Strongly correlated with `marital.status` and unlikely to provide additional value.

We now define the formula that will be used across all three models:

```
formula = income ~ age + workclass + education.num + marital.status +
  occupation +
    race + gender + capital.gain + capital.loss + hours.per.week +
  native.country
```

This formula will be used consistently across CART, C5.0, and Random Forest models to facilitate a fair comparison of predictive performance.

Note: Similar to regression models discussed in the previous chapter, tree-based models such as CART, C5.0, and Random Forests do *not* require dummy variable encoding for nominal variables or rescaling of numeric features. These models can directly handle categorical variables and are invariant to monotonic transformations of numeric variables. However, this is not the case for algorithms such as k -Nearest Neighbors (see Chapter 7), which require both dummy encoding and feature scaling for optimal performance.

Building a Decision Tree with CART

What happens inside a decision tree once it starts learning from data? Let us walk through the building process using the CART algorithm. To build a decision tree with CART in R, we use the `rpart` package (Recursive Partitioning and Regression Trees), which provides a widely used implementation. This package includes functions for building, visualizing, and evaluating decision trees.

First, ensure that the `rpart` package is installed. If needed, install it with `install.packages("rpart")`. Then, load the package into your R session:

```
library(rpart)
```

Once the package is loaded, we can build the decision tree model using the `rpart()` function:

```
cart_model = rpart(formula = formula, data = train_set, method = "class")
```

The main arguments of `rpart()` are:

- `formula`: Defines the relationship between the target variable (`income`) and the predictors.

- **data:** Specifies the training dataset.
- **method:** Indicates the type of modeling task. Here, we use `method = "class"` to build a classification tree.

Note: The `method` argument can also be set to "anova" for regression tasks (predicting continuous outcomes), "poisson" for count data, or "exp" for survival analysis (exponential models). This flexibility allows you to use CART for a wide range of predictive modeling tasks.

In the next subsection, we will visualize the decision tree to better understand the structure and decision-making process learned from the data.

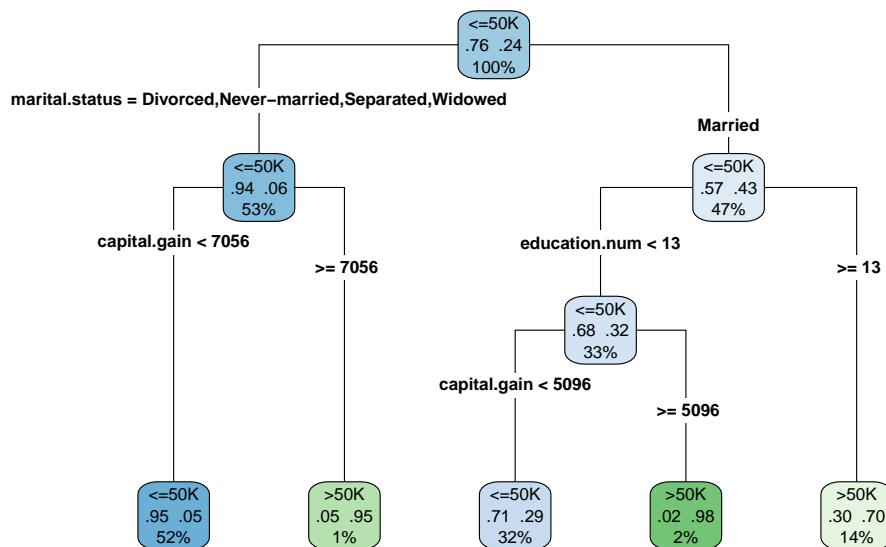
Visualizing the Decision Tree

After building the model, it is helpful to visualize the decision tree to better understand the learned decision rules. For this, we use the `rpart.plot` package, which provides intuitive graphical tools for displaying `rpart` models (install it with `install.packages("rpart.plot")` if needed):

```
library(rpart.plot)
```

The tree can be visualized with the following command:

```
rpart.plot(cart_model, type = 4, extra = 104)
```



The `type = 4` argument places decision rules inside the nodes for clarity, while the `extra = 104` argument displays both the predicted class and the probability of the most probable class at each terminal node.

If the tree is too large to fit within a single plot, an alternative is to examine a text-based structure using the `print()` function:

```
print(cart_model)
n= 38878

node), split, n, loss, yval, (yprob)
* denotes terminal node

1) root 38878 9217 <=50K (0.76292505 0.23707495)
   2) marital.status=Divorced,Never-married,Separated,Widowed 20580 1282
      ↵ <=50K (0.93770651 0.06229349)
        4) capital.gain< 7055.5 20261 978 <=50K (0.95172992 0.04827008) *
        5) capital.gain>=7055.5 319 15 >50K (0.04702194 0.95297806) *
       3) marital.status=Married 18298 7935 <=50K (0.56634605 0.43365395)
          6) education.num< 12.5 12944 4163 <=50K (0.67838381 0.32161619)
             12) capital.gain< 5095.5 12350 3582 <=50K (0.70995951 0.29004049) *
             13) capital.gain>=5095.5 594 13 >50K (0.02188552 0.97811448) *
            7) education.num>=12.5 5354 1582 >50K (0.29548001 0.70451999) *
```

This textual output lists the nodes, splits, and predicted outcomes, offering a compact summary when graphical space is limited.

Interpreting the Decision Tree

Now that we have visualized the tree, let us take a closer look at how it makes predictions. The model produces a binary tree with four decision nodes and five leaves. Among the twelve predictors, the algorithm selects three—`marital.status`, `capital.gain`, and `education.num`—as the most relevant for predicting income. The most influential predictor, `marital.status`, appears at the root node, meaning that marital status drives the first split in the tree.

The tree organizes individuals into five distinct groups, each represented by a terminal leaf. Blue leaves indicate those predicted to earn less than \$50,000 (`income <= 50K`), while green leaves represent those predicted to earn more than \$50,000 (`income > 50K`).

Consider the rightmost leaf of the tree: it classifies individuals who are married and have at least 13 years of education (`education.num >= 13`). This group represents 14% of the dataset, with 70% of them earning more than \$50,000 annually. The error rate for this leaf is 0.30, calculated as $1 - 0.70$.

In the next section, we apply the C5.0 algorithm to the same dataset and compare its structure and performance to the CART model.

Building a Decision Tree with C5.0

Now that we have seen how CART builds decision trees, let us turn to C5.0—an algorithm designed to build faster, deeper, and often more accurate trees. In this part of the case study, you will see how easily C5.0 can be applied to real-world data, using just a few lines of R code.

To fit a decision tree using the C5.0 algorithm in R, we use the **C50** package. If it is not already installed, it can be added with `install.packages("C50")`. After installation, load the package into your R session:

```
library(C50)
```

The model is constructed using the `C5.0()` function:

```
C50_model = C5.0(formula, data = train_set)
```

The key arguments are:

- `formula`: Specifies the relationship between the target variable (`income`) and the predictors.
- `data`: Defines the dataset used for training.

Compared to CART, C5.0 introduces several enhancements. It allows for multi-way splits, automatically assigns weights to predictors, and creates deeper but more compact trees when needed. This flexibility often results in stronger performance, especially when handling categorical variables with many levels.

Since the resulting tree can be quite large, we focus on summarizing the model rather than plotting its full structure. The `print()` function provides an overview:

```
print(C50_model)

Call:
C5.0.formula(formula = formula, data = train_set)

Classification Tree
Number of samples: 38878
Number of predictors: 11
```

```
Tree size: 120
Non-standard options: attempt to group attributes
```

The output displays important details, including the number of predictors used, the number of observations, and the total tree size. In this case, the tree consists of 74 decision nodes—substantially larger and potentially more powerful than the simpler CART tree.

By allowing richer splitting strategies and prioritizing informative features, C5.0 offers a step forward in sophistication over earlier decision tree algorithms.

In the next section, we explore *Random Forests*—an ensemble method that takes decision tree modeling to an entirely new level by combining the strengths of many trees.

Building a Random Forest Model

What if instead of relying on a single decision tree, we could build hundreds of trees and combine their predictions to make smarter decisions? *Random forests* offer exactly this approach, dramatically improving robustness and accuracy.

In R, random forests are implemented using the **randomForest** package, one of the most widely used and reliable implementations. If it is not already installed, add it with `install.packages("randomForest")`. Load the package into your R session:

```
library(randomForest)
```

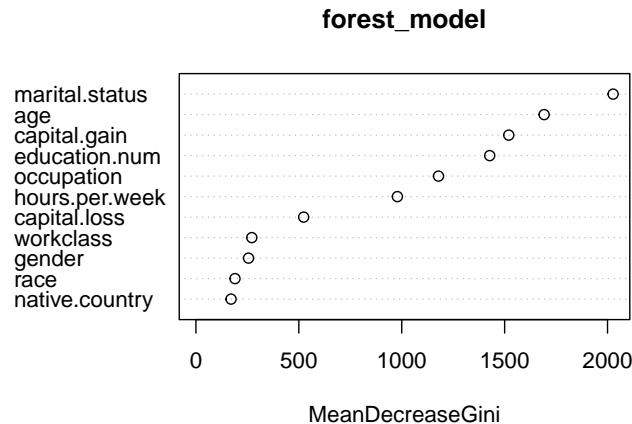
Using the same set of predictors as before, we construct a random forest model with 100 decision trees:

```
forest_model = randomForest(formula = formula, data = train_set, ntree = 100)
```

Here, `formula` specifies the relationship between the target variable (`income`) and the predictors, `data` defines the training dataset, and `ntree = 100` sets the number of trees to grow. Increasing `ntree` generally improves accuracy but requires more computational time.

To evaluate which predictors contribute most to model accuracy, we can visualize variable importance:

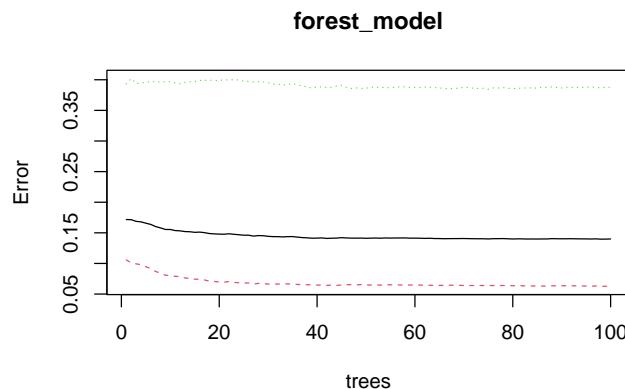
```
varImpPlot(forest_model)
```



The resulting plot ranks predictors by their influence. In this case, `marital.status` emerges as the most important, followed by `capital.gain` and `education.num`.

We can also assess how model error evolves as more trees are added:

```
plot(forest_model)
```



This plot shows classification error as a function of the number of trees. Notice that the error rate stabilizes after about 40 trees, suggesting that adding further trees yields diminishing returns.

By aggregating many trees trained on different subsets of the data and features, random forests reduce overfitting while preserving flexibility. They offer a powerful and reliable alternative to single-tree models.

In the next section, we compare the performance of the CART, C5.0, and Random Forest models side by side using evaluation metrics.

Model Evaluation and Comparison

Now that the models (CART, C5.0, and Random Forest) have been trained, it is time to see how well they perform when faced with new, unseen data. Model evaluation is the critical moment where we find out whether the patterns the models learned truly generalize—or whether they simply memorized the training set.

Following the evaluation techniques introduced in Chapter 8, we assess the models using three key tools:

- *Confusion matrices* to summarize classification errors,
- *ROC curves* to visualize model performance across different thresholds,
- *Area Under the Curve (AUC)* values to provide a concise, single-number summary of model quality.

To begin, we use the `predict()` function to generate predicted class probabilities for the test set. For all three models, we specify `type = "prob"` to obtain probabilities rather than discrete class labels:

```
cart_probs  = predict(cart_model,    test_set, type = "prob")[, "<=50K"]
C50_probs   = predict(C50_model,    test_set, type = "prob")[, "<=50K"]
forest_probs = predict(forest_model, test_set, type = "prob")[, "<=50K"]
```

The `predict()` function returns a matrix of probabilities for each class. The `[, "<=50K"]` notation extracts the probability of belonging to the $\leq 50K$ income class, which is important for evaluating the models' predictive accuracy.

In the sections that follow, we first examine confusion matrices to assess misclassification patterns, and then move on to ROC curves and AUC scores for a broader perspective on model performance.

Confusion Matrix and Classification Errors

How well do our models separate high earners from others? A confusion matrix gives us an immediate snapshot by showing how many predictions were correct—and what types of mistakes each model tends to make.

We generate confusion matrices for each model using the `conf.mat.plot()` function from the `liver` package, which creates easy-to-read graphical summaries:

```
conf.mat.plot(cart_probs, test_labels, cutoff = 0.5, reference = "<=50K",
  ↵ main = "CART Prediction")
conf.mat.plot(C50_probs, test_labels, cutoff = 0.5, reference = "<=50K",
  ↵ main = "C5.0 Prediction")
conf.mat.plot(forest_probs, test_labels, cutoff = 0.5, reference = "<=50K",
  ↵ main = "Random Forest Prediction")
```

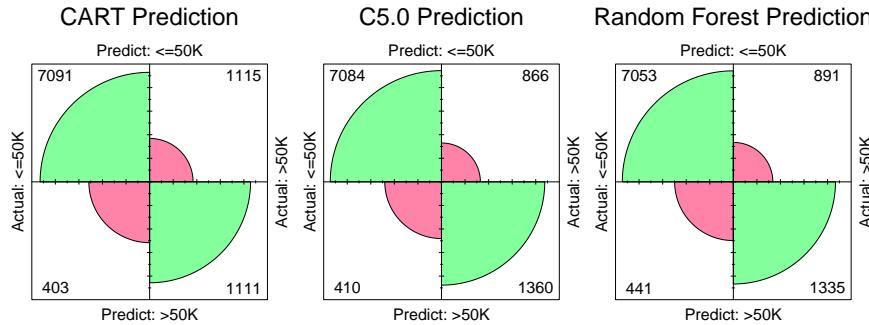


Figure 11.7: Confusion matrices for CART, C5.0, and Random Forest models using a cutoff value of 0.5. Each matrix summarizes the number of true positives, true negatives, false positives, and false negatives for the corresponding model.

In these plots, we set `cutoff = 0.5`, meaning that if the predicted probability for “ $\leq 50K$ ” is at least 0.5, the model predicts “ $\leq 50K$ ”; otherwise, it predicts “ $>50K$ ”. We also specify `reference = "<=50K"`, indicating that the $\leq 50K$ group is treated as the positive class.

Because confusion matrices depend on the cutoff value, changing the threshold would alter the number of true positives, false positives, and other entries. Thus, each confusion matrix reflects model performance at a particular decision point.

If you want to retrieve the numeric confusion matrices directly, you can use the `conf.mat()` function:

```
conf.mat(cart_probs,    test_labels, cutoff = 0.5, reference = "<=50K")
          Actual
          Predict <=50K >50K
          <=50K  7091 1115
          >50K   403 1111
conf.mat(C50_probs,    test_labels, cutoff = 0.5, reference = "<=50K")
```

```

      Actual
Predict <=50K >50K
<=50K  7084  866
>50K    410   1360
conf.mat(forest_probs, test_labels, cutoff = 0.5, reference = "<=50K")
      Actual
Predict <=50K >50K
<=50K  7053  891
>50K    441   1335

```

Here is the total number of correctly classified individuals for each model:

- *CART*: “ $7091 + 1111 = 8202$ ” correct predictions;
- *C5.0*: “ $7084 + 1360 = 8444$ ” correct predictions;
- *Random Forest*: “ $7053 + 1335 = 8388$ ” correct predictions.

Among the three models, C5.0 leads the pack, making the fewest misclassifications. Its flexible tree structure appears to provide a real advantage.

Try it yourself: What happens if you change the cutoff to 0.6 instead of 0.5? Re-run the `conf.mat.plot()` and `conf.mat()` functions with `cutoff = 0.6` and see how the confusion matrices shift. This small change can reveal important trade-offs between sensitivity and specificity—a topic we explore further with ROC curves and AUC values in the next part.

ROC Curve and AUC

What happens if we shift the decision threshold? Could our models behave differently? To answer this, we turn to the Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC)—two powerful tools that reveal how well a model separates the two income groups across all possible cutoff points.

We use the `pROC` package for these evaluations. If it is not installed yet, add it with `install.packages("pROC")`, and then load it:

```
library(pROC)
```

Next, we calculate the ROC curves for all three models:

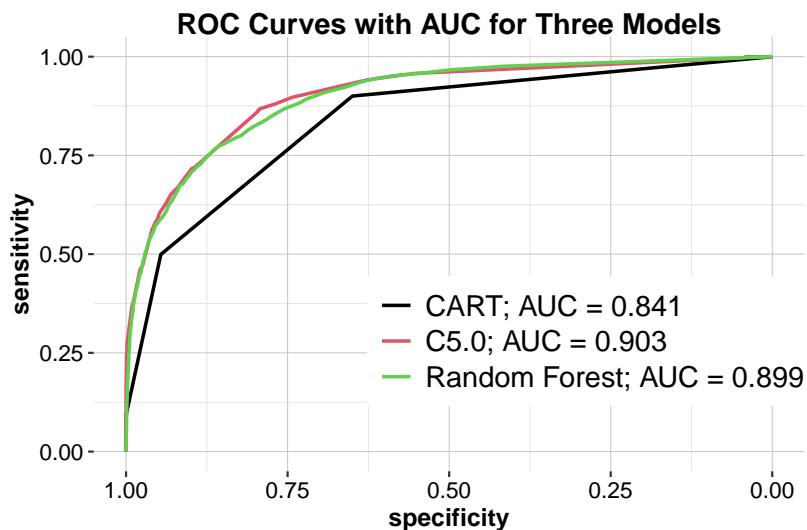
```

cart_roc  = roc(test_labels, cart_probs)
C50_roc  = roc(test_labels, C50_probs)
forest_roc = roc(test_labels, forest_probs)

```

Instead of viewing the models separately, let us put them all on the same plot using `ggroc()`:

```
ggroc(list(cart_roc, C50_roc, forest_roc), size = 0.8) +
  ggtitle("ROC Curves with AUC for Three Models") +
  scale_color_manual(values = 1:3,
    labels = c(paste("CART; AUC =", round(auc(cart_roc), 3)),
              paste("C5.0; AUC =", round(auc(C50_roc), 3)),
              paste("Random Forest; AUC =", round(auc(forest_roc), 3)))) +
  theme(legend.title = element_blank(),
    legend.position = c(.7, .3),
    text = element_text(size = 17))
```



In the ROC plot, the *black* curve represents CART, the *red* curve represents C5.0, and the *green* curve represents Random Forest. Take a moment to study the curves: Which model consistently stays closest to the top-left corner—the sweet spot for perfect classification?

The AUC values confirm what the eye suggests:

- CART: AUC = 0.841,
- C5.0: AUC = 0.903,
- Random Forest: AUC = 0.899.

C5.0 scores the highest, but notice that Random Forest is very close behind. While these AUC differences are real, they are relatively small—reminding us that model choice should also weigh factors like simplicity, speed, and interpretability.

In the next section, we wrap up the case study by reflecting on what these results mean for practical decision-making.

Reflections and Takeaways

This case study demonstrated how tree-based models—*CART*, *C5.0*, and *Random Forest*—can be applied to a real-world classification problem, following the Data Science Workflow from data preparation through model evaluation.

A major lesson from this analysis is the central importance of data preparation. Careful handling of missing values, consolidating categorical variables, and thoughtful selection of predictors were crucial for building models that are both interpretable and effective. Without these steps, even the most sophisticated algorithms would have struggled to find meaningful patterns.

The different tree-based algorithms each showed distinct strengths. *CART* offered a simple, easily interpretable structure but was more limited in flexibility. *C5.0* produced deeper and more nuanced trees, delivering the highest accuracy and AUC in this case. *Random Forest* demonstrated how combining multiple trees could achieve strong predictive performance with reduced overfitting, although at the cost of model interpretability.

Evaluating models through multiple lenses—confusion matrices, ROC curves, and AUC values—revealed important trade-offs that would have been invisible if we had focused only on overall accuracy. It also highlighted the effect of adjusting classification thresholds, showing how different cutoff points can shift the balance between sensitivity and specificity.

Finally, this case study emphasized that there is rarely a “one-size-fits-all” model. While *C5.0* slightly outperformed the others here, model choice always depends on the specific goals, resource constraints, and interpretability needs of a project.

11.6 Chapter Summary and Takeaways

This chapter introduced decision trees and random forests as flexible, non-parametric methods for supervised learning. Decision trees partition the feature space recursively based on splitting criteria such as the Gini index or entropy, offering interpretable models capable of capturing complex relationships. We explored two widely used algorithms—*CART* and *C5.0*—and demonstrated how *random forests* aggregate multiple trees to improve predictive performance and robustness.

The case study on income prediction illustrated the practical application of these methods, emphasizing the importance of careful data preparation,

thoughtful model evaluation, and the consideration of trade-offs between accuracy, interpretability, and computational efficiency. While C5.0 achieved the highest predictive performance in this example, model choice should always reflect the specific goals and constraints of the analysis.

Reflection: Tree-based models provide a natural balance between flexibility and interpretability. Single decision trees are transparent and easy to communicate but risk overfitting when grown too deeply. Ensemble methods such as random forests improve predictive accuracy at the cost of interpretability. As you move forward, consider how the complexity of a model aligns with the demands of the problem: when simplicity and explanation are paramount, shallow trees may suffice; when predictive power is critical, ensembles may be preferable.

You are encouraged to engage with the exercises provided at the end of the chapter, which reinforce the techniques discussed and build practical modeling skills. In the next chapter, the focus shifts to neural networks—extending the modeling toolkit to even more flexible, nonlinear approaches.

11.7 Exercises

Ready to put theory into practice? The exercises below invite you to test your understanding of Decision Trees and Random Forests through conceptual questions and hands-on modeling with real datasets.

Conceptual Understanding

1. Explain the basic structure of a Decision Tree and how it makes predictions.
2. What are the key differences between classification trees and regression trees?
3. What is the purpose of splitting criteria in Decision Trees? Describe the Gini Index, Entropy, and Variance Reduction.
4. Why are Decision Trees prone to overfitting? What techniques can be used to prevent it?
5. Define pre-pruning and post-pruning in Decision Trees. How do they differ?
6. Explain the bias-variance tradeoff in Decision Trees.

7. What are the advantages and disadvantages of Decision Trees compared to logistic regression for classification problems?
8. What is the role of the maximum depth parameter in a Decision Tree? How does it affect model performance?
9. Why might a Decision Tree favor continuous variables over categorical variables when constructing splits?
10. Explain the differences between CART (Classification and Regression Trees) and C5.0 Decision Trees.
11. What is the fundamental difference between Decision Trees and Random Forests?
12. How does bagging (Bootstrap Aggregation) improve Random Forest models?
13. Explain how majority voting works in a Random Forest classification task.
14. Why does a Random Forest tend to outperform a single Decision Tree?
15. How can we determine feature importance in a Random Forest model?
16. What are the limitations of Random Forests?
17. How does increasing the number of trees (`ntree`) affect model performance?

Hands-On: Classification with the Churn Dataset

In the case study of the previous chapter (Section 10.12), we fitted Logistic Regression, Naive Bayes, and k-Nearest Neighbors models to the churn dataset using the Data Science Workflow. Here, we extend the analysis by applying tree-based models: Decision Trees (CART and C5.0) and Random Forests. You can reuse the earlier data preparation code and directly compare the new models with those from the previous case study to deepen your understanding of classification techniques.

The *churn* dataset contains information about customer churn behavior in a telecommunications company. The goal is to predict whether a customer will churn based on various attributes.

Data Preparation

18. Load the *churn* dataset and generate a summary. Identify the target variable and the predictors to be used in the analysis.

19. Partition the dataset into a training set (80%) and a test set (20%) using the `partition()` function from the `liver` package. For reproducibility, use the same random seed as in Section 10.12.

Modeling with Decision Trees (CART)

20. Fit a Decision Tree using the CART algorithm, with `churn` as the response variable and the following predictors: `account.length`, `voice.plan`, `voice.messages`, `intl.plan`, `intl.mins`, `intl.calls`, `day.mins`, `day.calls`, `eve.mins`, `eve.calls`, `night.mins`, `night.calls`, and `customer.calls`. (See Section 10.12 for the rationale behind these predictor choices.)
21. Visualize the fitted Decision Tree using `rpart.plot()`. Interpret the main decision rules.
22. Identify the most important predictors in the tree.
23. Compute the confusion matrix and evaluate model performance.
24. Plot the ROC curve and compute the AUC for the CART model.
25. Evaluate the effect of pruning the tree by adjusting the complexity parameter (`cp`).

Modeling with Decision Trees (C5.0)

26. Fit a C5.0 Decision Tree using the same predictors as in the CART model.
27. Compare the structure and accuracy of the C5.0 tree with the CART tree.
28. Compare the confusion matrices and overall classification accuracies between the CART and C5.0 models.

Modeling with Random Forests

29. Fit a Random Forest model using the same predictors.
30. Identify the most important predictors using `varImpPlot()`.
31. Compare the accuracy of the Random Forest model to the CART and C5.0 models.
32. Compute the confusion matrix for the Random Forest model.

33. Plot the ROC curve and compute the AUC for the Random Forest model.
34. Set `ntree = 200` and assess whether increasing the number of trees improves accuracy.
35. Use `tuneRF()` to find the optimal value for `mtry`.
36. Predict churn probabilities for a new customer using the Random Forest model.
37. Train a Random Forest model using only the top three most important features.
38. Evaluate whether the simplified Random Forest model performs comparably to the full model.

Regression Trees and Random Forests: The redWines Dataset

We now turn to regression tasks, using the `redWines` dataset from `liver` package to predict wine quality.

Conceptual Questions: Regression Trees and Random Forests

39. How does a regression tree differ from a classification tree?
40. How is Mean Squared Error (MSE) used to evaluate regression trees?
41. Why is Random Forest regression often preferred over a single regression tree?

Hands-On: Regression with the redWines Dataset

Now, apply what you have learned to a real-world regression problem.

Data Preparation

Load the `redWines` dataset and partition it into a training set (70%) and a test set (30%).

```
data(redWines, package = "liver")
set.seed(42)
data_sets = partition(data = redWines, ratio = c(0.7, 0.3))
train_set = data_sets$part1
test_set = data_sets$part2
test_labels = test_set$quantity
```

Modeling and Evaluation

42. Fit a regression tree predicting quantity based on all predictors.
43. Visualize the regression tree and interpret the main decision rules.
44. Compute the Mean Squared Error (MSE) of the regression tree on the test set.
45. Fit a Random Forest regression model and compute its MSE.
46. Compare the predictive performance of the Random Forest and the regression tree models.
47. Identify the top three most important predictors in the Random Forest model.
48. Predict wine quality for a new observation with the following attributes:
fixed.acidity = 8.5, volatile.acidity = 0.4, citric.acid = 0.3,
residual.sugar = 2.0, chlorides = 0.08, free.sulfur.dioxide = 30,
total.sulfur.dioxide = 100, density = 0.995, pH = 3.2, sulphates = 0.6, alcohol = 10.5.
49. Perform cross-validation to compare the regression tree and Random Forest models.
50. Reflect: Based on your findings, does the Random Forest significantly improve prediction accuracy compared to the single regression tree?

Chapter 12

Neural Networks: The Building Blocks of Artificial Intelligence

Can machines think and learn like humans? This question has fascinated humanity for centuries, inspiring philosophers, inventors, and storytellers alike. From the mechanical automata of ancient Greece to the artificial beings of science fiction, visions of intelligent machines have long captured our imagination. Early inventors such as Hero of Alexandria designed self-operating devices, while myths like the golem and stories of automatons reflected a persistent desire to animate intelligence. What was once confined to myth and speculation has now materialized as *Artificial Intelligence (AI)*—a transformative force reshaping industries, societies, and daily life.

AI is no longer a futuristic fantasy. Today, it powers technologies that touch nearly every aspect of modern life, from recommendation systems and fraud detection to autonomous vehicles and generative AI (GenAI) models capable of producing text, images, and music. These advancements have been fueled by exponential growth in computational power, the explosion of data availability, and breakthroughs in machine learning algorithms. At the heart of this revolution lies a class of models known as *neural networks*—the foundational technology behind deep learning.

Neural networks are computational models inspired by the structure and function of the human brain. Just as biological neurons connect to form intricate networks that process information, artificial neural networks consist of layers of interconnected nodes that learn patterns from data. This design enables them to recognize complex structures, extract meaningful insights, and make predictions. Neural networks are particularly well-suited for problems involving *complex, high-dimensional, and unstructured data*—such as images, speech, and natural language. Unlike traditional machine learning models, which rely on manually engineered features, neural networks can automatically discover representations in data, often outperforming classical approaches.

While deep learning—powered by sophisticated neural architectures—has led to groundbreaking advances in fields such as computer vision and natural language processing, its foundation rests on simpler models. In this chapter, we focus on *feed-forward neural networks*, also known as *multilayer perceptrons (MLPs)*. These fundamental architectures serve as the essential building blocks for understanding and developing more advanced deep learning systems.

In this chapter, we continue advancing through the *Data Science Workflow* introduced in Chapter 2. So far, we have learned how to prepare and explore data, and apply classification methods—including *k-Nearest Neighbors* (Chapter 7) and *Naive Bayes* (Chapter 9)—as well as regression models (Chapter 10) and tree-based models (Chapter 11). We have also discussed how to evaluate model performance (Chapter 8).

Neural networks now offer another powerful modeling strategy within supervised learning, capable of handling both classification and regression tasks with remarkable flexibility. They often uncover complex patterns that traditional models may struggle to detect.

Why Neural Networks Are Powerful

Why are neural networks the engine behind modern breakthroughs like self-driving cars, real-time translation, and medical image diagnostics? The answer lies in their remarkable ability to learn from data in ways that traditional models simply cannot match. Their strengths stem from three core capabilities:

1. *Recognizing Patterns in Complex Data:* Neural networks excel at identifying intricate structures in unstructured data—whether it is detecting faces in photos, understanding spoken language, or generating realistic images. These are tasks that traditional rule-based algorithms struggle to perform reliably.
2. *Resilience to Noise:* Thanks to their densely connected architecture and adaptive learning, neural networks can still make accurate predictions even when data is incomplete or noisy—such as audio recordings with background interference or blurry video frames.
3. *Scalability and Flexibility:* As data complexity grows, neural networks can scale accordingly. By adding more layers and neurons, they can model highly nonlinear relationships, making them ideal for large-scale applications like recommendation engines or credit scoring.

Of course, this power comes with trade-offs. Unlike decision trees, neural networks often behave like “*black boxes*”, making it difficult to trace how individual predictions are made. In domains where transparency is critical—such as healthcare or finance—this lack of interpretability can be a serious concern.

Training these models also demands significant computational resources, often requiring GPUs or TPUs to efficiently process large datasets.

Yet despite these challenges, the impact of neural networks is undeniable. Just as neurons in the brain work together to form thought and perception, artificial neurons collaborate to extract patterns, recognize context, and make predictions. This ability to adapt and generalize has made neural networks central to the ongoing evolution of intelligent systems.

What This Chapter Covers

Ever wondered how machines can recognize faces, translate languages, or generate music? Neural networks are the driving force behind these breakthroughs—and in this chapter, we unpack their foundations and show how to apply them in practice using R.

By the end of this chapter, you will understand how neural networks mimic biological intelligence, how they learn from data, and how to train your own network to solve real-world problems.

We will move step-by-step through the essentials:

- *Biological inspiration:* How the structure and function of the human brain inspired artificial neural networks.
- *Core architecture and mechanics:* The layers, nodes, and weights that allow networks to represent and process data.
- *Activation functions:* Why non-linearity is essential, and how different functions shape a network’s learning capacity.
- *Training algorithms:* How neural networks learn from data through iterative optimization.
- *Applied case study:* Using a neural network to predict customer behavior in the *bank marketing dataset*, bringing theory into practice.

Neural networks have revolutionized computing, enabling machines to tackle problems once considered unsolvable. From autonomous vehicles to

medical diagnostics, these models are shaping the future of AI. To understand how they function, we begin by exploring their biological origins and the inspiration drawn from the human brain.

12.1 The Biological Inspiration Behind Neural Networks

How can a machine recognize a cat, understand speech, or recommend a movie—all without being explicitly told the rules? The answer lies in the architecture of neural networks, which are inspired by one of nature’s most powerful systems: the human brain.

Biological neurons, the fundamental units of the brain, enable learning, perception, and decision-making through massive networks of interconnected cells. While each neuron is simple, the collective network is extraordinarily powerful. The human brain contains approximately 10^{11} neurons, each forming around 10,000 synaptic connections. This yields a staggering 10^{15} pathways—an intricate system capable of adaptive learning, pattern recognition, and high-level reasoning.

Artificial neural networks (ANNs) are simplified computational models that abstract key principles from this biological system. Although they do not replicate the brain’s full complexity, ANNs use interconnected processing units—artificial neurons—to learn from data. By adjusting the strengths of these connections (weights), neural networks can model complex, nonlinear relationships in domains such as image recognition, speech processing, and decision-making—areas where traditional models often struggle.

As shown in Figure 12.1, a biological neuron receives input signals through dendrites. These are aggregated and processed in the cell body. If the combined signal exceeds a certain threshold, the neuron “fires” and transmits an electrical signal through its axon. This nonlinear decision mechanism is central to the brain’s efficiency.

In a similar spirit, an artificial neuron—depicted in Figure 12.2—receives input features (x_i), multiplies them by adjustable weights (w_i), and sums the results. This weighted sum is passed through an activation function $f(\cdot)$ to produce an output (\hat{y}). This output may then feed into other neurons or serve as the network’s final prediction. The activation function introduces the essential non-linearity that allows neural networks to approximate complex patterns.

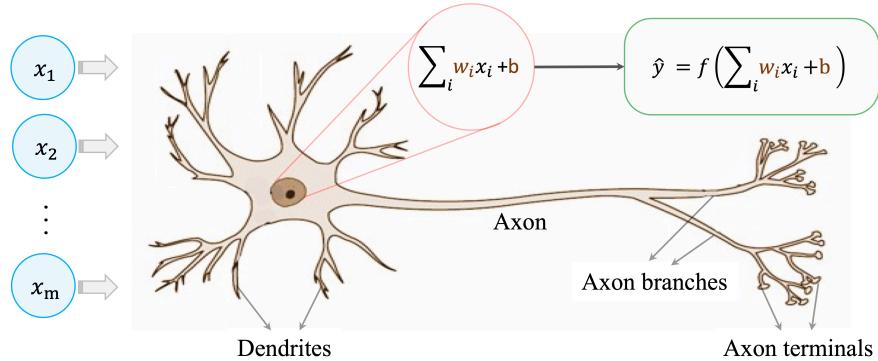


Figure 12.1: Visualization of a biological neuron, which processes input signals through dendrites and sends outputs through the axon.

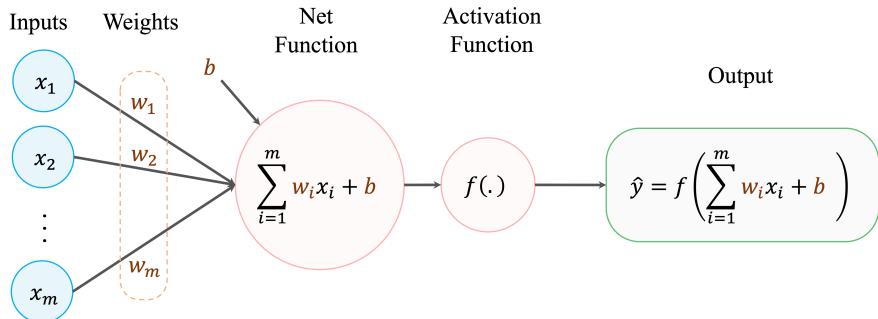


Figure 12.2: Illustration of an artificial neuron, designed to emulate the structure and function of a biological neuron in a simplified way.

One of the key strengths of neural networks is their ability to *generalize*—that is, to make accurate predictions on new, unseen data. Unlike traditional rule-based algorithms, which follow explicit instructions, neural networks learn flexibly from examples, discovering patterns even when data is noisy or incomplete.

This flexibility, however, comes with challenges. Neural networks are often regarded as *black boxes* by practitioners because their learned behavior is encoded in millions of parameters, making their decisions difficult to interpret. Additionally, training neural networks requires large datasets and substantial computational resources, often involving GPUs or TPUs for efficient learning.

In the following sections, we will explore how these models are constructed, trained, and applied in practice using R.

12.2 How Neural Networks Work

What if a model could not only fit a line—but build its own features to recognize faces, interpret speech, or detect anomalies in real-time data? Neural networks extend traditional linear models by incorporating multiple layers of processing to capture complex relationships in data. At their core, they build upon the fundamental concepts of linear regression, introduced in Chapter 10. As discussed in Section 10.4, a linear regression model makes predictions using the following equation:

$$\hat{y} = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_m x_m$$

where m represents the number of predictors, b_0 is the intercept, and b_1 to b_m are the learned coefficients. In this setup, \hat{y} is a weighted sum of the input features (x_1 to x_m), with the weights (b_1 to b_m) determining the influence of each feature on the prediction. This relationship is visualized in Figure 12.3, where predictors and outputs are shown as nodes, with the coefficients acting as connecting weights.

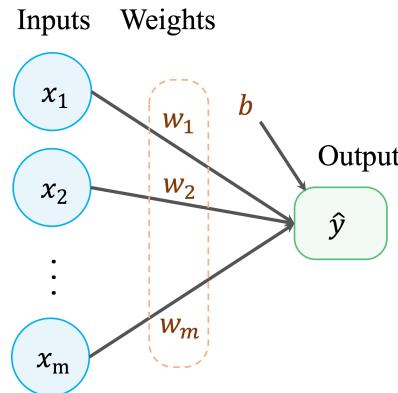


Figure 12.3: A graphical representation of a regression model: input features and predictions are shown as nodes, with the coefficients represented as connections between the nodes.

While this diagram illustrates the flow of information in a linear model, it also reveals a fundamental limitation: the model treats all features as contributing independently and linearly to the prediction. Linear models struggle to capture interactions between variables or hierarchical structures in data.

Neural networks address this limitation by introducing multiple layers of artificial neurons between the input and output layers, allowing them to model intricate, nonlinear relationships. This layered structure is illustrated in Figure 12.4.

The architecture of a neural network consists of the following key components:

- The *input layer* serves as the entry point for the data. Each node in this layer corresponds to an input feature, such as age, income, or pixel intensity.
- The *hidden layers* transform the data through multiple interconnected artificial neurons. Each hidden layer captures increasingly abstract features, allowing the network to learn patterns that are difficult to handcraft. Every neuron in a hidden layer is connected to neurons in both the preceding and succeeding layers, with each connection assigned a weight.
- The *output layer* produces the final prediction. In classification tasks, this is typically a probability; in regression tasks, it is a continuous value.

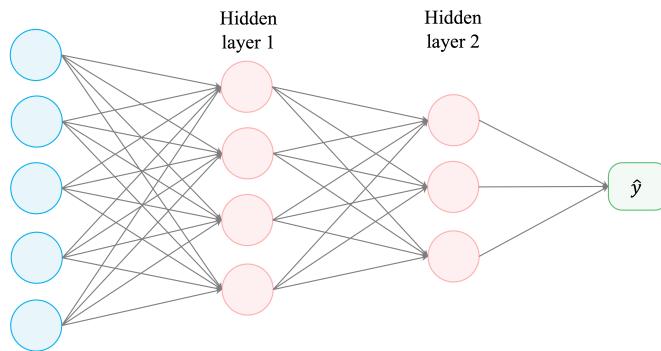


Figure 12.4: Visualization of a multilayer neural network model with two hidden layers.

In Figure 12.4, the input features flow into the network, are transformed by the hidden layers, and emerge as a final prediction from the output layer. Each connection is assigned a weight (w_i), which reflects the influence one

neuron has on another. During training, these weights are adjusted to minimize prediction error.

The computation performed by a single artificial neuron can be described mathematically as:

$$\hat{y} = f \left(\sum_{i=1}^p w_i x_i + b \right)$$

where x_i are the input features, w_i are their associated weights, b is a bias term that shifts the activation threshold, $f(\cdot)$ is the activation function, and \hat{y} is the neuron's output.

A critical feature of neural networks is the activation function, which introduces non-linearity. Without it, even deep networks would collapse into a simple linear model. This non-linear transformation is what gives neural networks their expressive power, enabling them to model intricate, real-world phenomena.

Key Characteristics of Neural Networks

Despite the wide variety of neural network designs, all networks share three essential characteristics that define how they learn and make predictions:

1. *Non-Linearity Through Activation Functions:* Activation functions transform a neuron's input into an output signal passed to the next layer. This non-linear transformation enables the network to capture complex relationships in the data. Common choices include the sigmoid, ReLU (Rectified Linear Unit), and tanh (hyperbolic tangent) functions.
2. *Capacity Defined by Network Architecture:* The number of layers and the number of neurons per layer determine the model's capacity to represent patterns. Deeper networks can learn more abstract, hierarchical representations—like recognizing edges in early layers and full objects in later ones.
3. *Learning via Optimization Algorithms:* Neural networks learn by iteratively updating their weights and biases to minimize a loss function. Optimization algorithms such as gradient descent compute how each parameter should change to improve predictions during training.

In the next sections, we take a closer look at each of these building blocks—starting with activation functions and their essential role in modeling non-linear patterns in data.

12.3 Activation Functions

What determines whether a neuron “fires”? And how can that simple decision give rise to models that recognize faces, understand speech, or generate images? The answer lies in the *activation function*, a fundamental component of neural networks. Much like biological neurons that integrate signals and fire when sufficiently stimulated, artificial neurons use activation functions to introduce non-linearity—enabling networks to model complex patterns beyond the reach of linear models.

Without activation functions, a neural network—even one with many layers—would reduce to a linear model, lacking the capacity to capture interactions, nonlinearities, or layered abstractions in data. Activation functions make it possible for networks to recognize abstract relationships in text, images, and time-series data.

In mathematical terms, an artificial neuron computes a weighted sum of its inputs and applies an activation function $f(x)$ to determine its output:

$$\hat{y} = f \left(\sum_{i=1}^p w_i x_i + b \right)$$

where x_i represents the input features, w_i are the corresponding weights, b is a bias term, and $f(x)$ is the activation function. The choice of activation function significantly impacts the network’s ability to learn and generalize.

The Threshold Activation Function

One of the earliest activation functions, the *threshold function*, was inspired by the all-or-nothing behavior of biological neurons. It outputs 1 when the input is zero or greater, and 0 otherwise:

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{if } x < 0. \end{cases}$$

This binary, step-like behavior is shown in Figure 12.5.

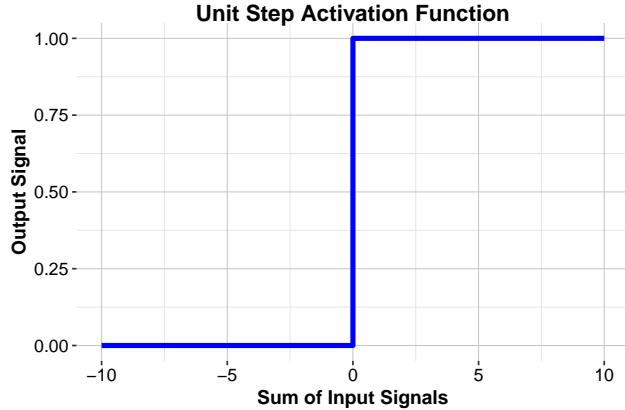


Figure 12.5: Visualization of the threshold activation function (unit step).

Although biologically intuitive, the threshold function is not differentiable, which prevents its use in gradient-based learning algorithms such as back-propagation. Its inability to capture nuanced input-output relationships also limits its effectiveness in modern neural networks.

The Sigmoid Activation Function

A widely used alternative to the threshold function is the sigmoid activation function, also known as the logistic function. It smoothly maps any real-valued input into the interval $(0, 1)$, making it suitable for binary classification problems where the output is interpreted as a probability. The function is defined as:

$$f(x) = \frac{1}{1 + e^{-x}},$$

where e is the base of the natural logarithm. The sigmoid function produces a characteristic S-shaped curve, as shown in Figure 12.6, and is both continuous and differentiable.

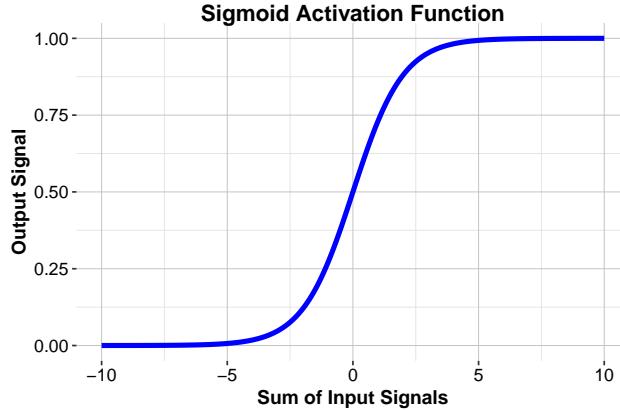


Figure 12.6: Visualization of the sigmoid activation function.

The sigmoid function is closely related to the logit function used in logistic regression (Section 10.6). In logistic regression, the log-odds of a binary outcome are modeled as a linear combination of input features:

$$\hat{y} = b_0 + b_1 x_1 + \cdots + b_m x_m.$$

The predicted probability is then computed using the sigmoid function:

$$p = \frac{1}{1 + e^{-\hat{y}}}.$$

This transformation from linear score to probability is mathematically identical to the operation of a neural network with a single output neuron and sigmoid activation. When paired with a cross-entropy loss function, such a network behaves similarly to logistic regression. However, the inclusion of hidden layers allows neural networks to model nonlinear decision boundaries that are beyond the capacity of logistic regression.

Despite its advantages, the sigmoid function has limitations. When input values are large in magnitude (positive or negative), the function saturates, and the gradient approaches zero. This vanishing gradient problem can slow or prevent learning in deeper networks. As a result, sigmoid is typically used only in output layers, while other functions are preferred in hidden layers.

Common Activation Functions in Deep Networks

While the sigmoid function was foundational in early neural networks, modern architectures benefit from activation functions that offer faster convergence and improved gradient flow. The most widely used alternatives are:

1. *Hyperbolic Tangent (tanh)*: Like sigmoid, but outputs values between -1 and 1 , making it zero-centered and often better suited for hidden layers.
2. *ReLU (Rectified Linear Unit)*: Defined as $f(x) = \max(0, x)$, ReLU is computationally efficient and helps mitigate the vanishing gradient problem.
3. *Leaky ReLU*: A variant of ReLU that allows a small negative output when $x < 0$, reducing the risk of inactive (“dead”) neurons.

Figure 12.7 visually compares the output shapes of sigmoid, tanh, and ReLU activation functions across a range of inputs.

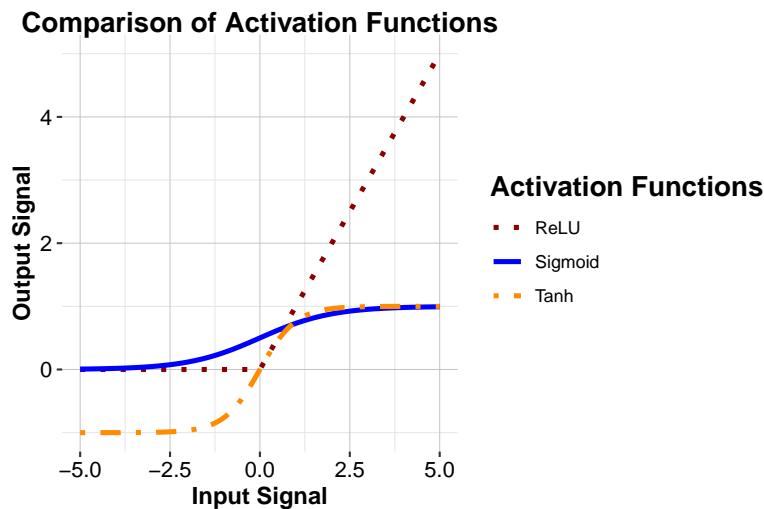


Figure 12.7: Comparison of common activation functions: sigmoid, tanh, and ReLU.

These activation functions each offer advantages depending on the architecture, input characteristics, and task complexity. The next subsection discusses how to select an appropriate activation function given the structure of the model and the goals of learning.

Choosing the Right Activation Function

Choosing an appropriate activation function is crucial for ensuring effective learning and model performance. The selection depends on both the learning task and the layer within the network:

- *Sigmoid*: Used in the output layer for binary classification tasks.
- *Tanh*: Preferred in hidden layers when zero-centered outputs aid convergence.
- *ReLU*: Commonly used in hidden layers due to computational efficiency and gradient propagation.
- *Leaky ReLU*: Applied when standard ReLU units risk becoming inactive.
- *Linear activation*: Used in the output layer for regression tasks involving continuous targets.

Both sigmoid and tanh functions can saturate when input values are very large or small, causing gradients to vanish and slowing learning. This issue can be mitigated through preprocessing steps such as normalization or standardization to keep inputs within effective ranges. In modern deep learning, ReLU and its variants are widely used due to their simplicity and effectiveness. However, the optimal choice of activation function often depends on the specific problem and should be evaluated empirically.

In the following section, we turn to the architecture of neural networks and examine how layers, neurons, and connections are structured to build models capable of learning from complex data.

12.4 Network Architecture

What makes some neural networks powerful enough to recognize faces, translate languages, or drive cars? The answer lies not just in the learning algorithm, but in the structure of the network itself—its architecture.

A neural network's architecture refers to the arrangement of neurons and their connections, which determine how data flows through the model. While network designs vary, three factors primarily characterize their structure:

- The number of layers in the network,
- The number of neurons in each layer, and

- The connectivity between neurons across layers.

This architecture defines the network's capacity to learn and generalize. Larger networks with more layers and neurons can model intricate relationships and decision boundaries. However, effectiveness depends not only on size but also on how these components are organized.

To understand this, consider a simple example shown in Figure 12.3. This basic network consists of:

- Input nodes, which receive raw feature values from the dataset. Each node corresponds to a feature.
- Output nodes, which provide the network's final prediction.

In this single-layer network, inputs are connected directly to outputs via a set of weights (w_1, w_2, \dots, w_m), which control how much influence each feature has. While such an architecture is suitable for basic regression or classification tasks, it is limited in its ability to capture complex, nonlinear patterns.

To overcome these limitations, additional layers can be introduced—known as *hidden layers*—as illustrated in Figure 12.4. These layers allow the network to learn intermediate features and nonlinear relationships, building toward more abstract representations of the data.

A multilayer network generally contains:

- An *input layer*, where raw data enters the network,
- One or more *hidden layers*, where features are transformed and abstracted,
- An *output layer*, which generates the model's prediction.

In fully connected networks, each neuron in one layer is connected to every neuron in the next. Each connection carries a weight that is updated during training to improve model performance.

Hidden layers allow for hierarchical processing of data. For example, early layers in an image recognition network may detect simple edges, while deeper layers recognize shapes or entire objects. Neural networks with multiple hidden layers are called *deep neural networks (DNNs)*, and training them forms the foundation of *deep learning*. This has enabled breakthroughs in fields such as computer vision, speech recognition, and language understanding.

The number of input and output nodes depends on the specific task:

- The number of *input nodes* equals the number of features in the dataset. A dataset with 20 features requires 20 input nodes.

- The number of *output nodes* depends on the type of task: one node for regression, or one per class in multi-class classification.

The number of *hidden nodes* is flexible and depends on problem complexity. While more hidden nodes increase capacity, they also raise the risk of *overfitting*—where the model performs well on training data but poorly on new data. Larger networks also demand more computation and training time.

Balancing complexity and generalization is essential. The principle of *Occam's Razor*—preferring the simplest model that performs well—often guides architecture choices. In practice, optimal architectures are found through experimentation and are often paired with techniques like cross-validation and regularization (e.g., dropout or weight decay) to avoid overfitting.

Although this section focuses on fully connected networks, alternative architectures provide further specialization:

- *Convolutional neural networks (CNNs)* are optimized for image data,
- *Recurrent neural networks (RNNs)* are designed for sequential tasks like speech or text.

These architectures build on the same principles but are tailored for specific data structures.

In summary, a network's architecture sets the stage for its learning capacity. From single-layer models to deep neural networks, choosing the right structure is a key part of building effective AI systems. As part of the modeling stage in the Data Science Workflow, selecting an appropriate architecture lays the foundation for training accurate and generalizable models.

12.5 How Neural Networks Learn

How does a neural network improve its predictions? Like a student learning from experience, a neural network begins with no prior knowledge and gradually refines its internal connections through exposure to data. These connections, represented as adjustable weights, are updated over time to help the network recognize patterns and make accurate predictions. Just as a child learns to identify objects through repeated encounters, a neural network improves by iteratively refining its internal parameters.

Training a neural network is a computationally intensive process that involves updating the weights between neurons. While the basic ideas date back to the mid-20th century, a major breakthrough occurred in the 1980s

with the introduction of the backpropagation algorithm, which made it feasible to train multilayer networks efficiently. Backpropagation enables the network to systematically learn from its mistakes, forming the foundation of modern neural network training used in areas such as image recognition and language modeling.

The learning process involves two main phases: a forward phase and a backward phase, and it proceeds iteratively over multiple passes through the training data, known as epochs.

In the forward phase, input data passes through the network, layer by layer. Each neuron applies its weights to the incoming signals, sums them, and applies an activation function. The final output is compared to the actual value, and an error is computed to quantify the discrepancy.

In the backward phase, this error is propagated backward through the network using the chain rule of calculus. The goal is to adjust the weights to reduce future prediction error. This adjustment is guided by gradient descent, which calculates how the error changes with respect to each weight. The network updates its weights in the direction that reduces the error most effectively—like descending a hill by following the steepest path downward.

The size of these weight updates is controlled by a parameter called the learning rate. A higher learning rate leads to larger, faster updates but may overshoot the optimal values. A lower rate yields more precise adjustments but may slow convergence. Modern training techniques often use adaptive learning rates to balance these trade-offs dynamically.

A key requirement for this process is that the activation functions must be differentiable. Common choices such as the sigmoid, tanh, and ReLU functions satisfy this condition and allow efficient gradient computation. Variants of gradient descent, including stochastic gradient descent (SGD) and Adam, further improve the speed and stability of training, especially for large datasets.

By repeating this cycle of forward and backward passes, the network progressively reduces its error and becomes better at generalizing to new data. Although the process may appear complex, modern tools such as TensorFlow and PyTorch automate these computations, allowing practitioners to focus on designing and evaluating models.

The development of backpropagation was a pivotal moment in neural network research. Coupled with advances in hardware like GPUs and TPUs, it has enabled the training of deep and complex networks that now drive many real-world AI applications. With this foundation in place, we turn next to practical examples of how neural networks can be applied to real-world data analysis.

12.6 Case Study: Predicting Term Deposit Subscriptions

Can we predict which customers will say “yes” before the call even happens? This case study explores how predictive modeling can help financial institutions run more effective marketing campaigns. We use data from a previous tele-marketing campaign to build a neural network model that predicts whether a customer will subscribe to a term deposit. The goal is to uncover patterns in customer demographics and campaign interactions that can guide future targeting efforts and reduce unnecessary outreach.

The dataset comes from the [UC Irvine Machine Learning Repository](#) and is available in the `liver` package. It was originally used by Moro, Cortez, and Rita (2014) in a study of data-driven strategies for bank marketing. The target variable indicates whether the customer subscribed to a term deposit, and the predictors include personal attributes and campaign details.

Following the Data Science Workflow introduced in Chapter 2 and illustrated in Figure 2.3, this case study guides you through each stage of the process—from understanding the problem to modeling and evaluation. You will learn how to apply a neural network to a real-world classification task using R, with each step grounded in the workflow to promote clarity, reproducibility, and adherence to sound data science practices.

Problem Understanding

Financial institutions face the challenge of identifying which customers are most likely to respond positively to marketing efforts. Two common strategies are used to promote financial products:

- *Mass campaigns*: Designed to reach a broad audience with minimal targeting, often resulting in low response rates (typically below 1%).
- *Directed marketing*: A targeted approach that focuses on individuals likely to show interest, improving conversion rates but raising potential privacy concerns.

This case study enhances directed marketing by analyzing data from past campaigns to identify behavioral patterns associated with term deposit subscriptions. By predicting which customers are more likely to subscribe, the bank can optimize outreach, reduce costs, and minimize intrusive communications—while improving overall effectiveness.

A *term deposit* is a fixed-term savings account that offers higher interest rates than regular savings accounts. It helps banks secure long-term capital while providing customers with a stable investment option.

Overview of the Dataset

The *bank* dataset includes information on direct phone-based marketing campaigns conducted by a financial institution. Customers were contacted multiple times within the same campaign. The objective of this dataset is to predict whether a customer will subscribe to a term deposit (`deposit = "yes"` or `"no"`).

We load the *bank* dataset directly into **R** and examine its structure using the following commands:

```
library(liver)    # Load the liver package
data(bank)        # Load the bank marketing dataset
str(bank)
'data.frame':   4521 obs. of  17 variables:
 $ age      : int  30 33 35 30 59 35 36 39 41 43 ...
 $ job      : Factor w/ 12 levels "admin.", "blue-collar", ...: 11 8 5 5 2 5
   ↪ 7 10 3 8 ...
 $ marital  : Factor w/ 3 levels "divorced", "married", ...: 2 2 3 2 2 3 2 2
   ↪ 2 2 ...
 $ education: Factor w/ 4 levels "primary", "secondary", ...: 1 2 3 3 2 3 3 2
   ↪ 3 1 ...
 $ default  : Factor w/ 2 levels "no", "yes": 1 1 1 1 1 1 1 1 1 1 ...
 $ balance  : int  1787 4789 1350 1476 0 747 307 147 221 -88 ...
 $ housing  : Factor w/ 2 levels "no", "yes": 1 2 2 2 2 1 2 2 2 2 ...
 $ loan     : Factor w/ 2 levels "no", "yes": 1 2 1 2 1 1 1 1 2 ...
 $ contact  : Factor w/ 3 levels "cellular", "telephone", ...: 1 1 1 3 3 1 1
   ↪ 1 3 1 ...
 $ day      : int  19 11 16 3 5 23 14 6 14 17 ...
 $ month    : Factor w/ 12 levels "apr", "aug", "dec", ...: 11 9 1 7 9 4 9 9 9
   ↪ 1 ...
 $ duration : int  79 220 185 199 226 141 341 151 57 313 ...
 $ campaign : int  1 1 1 4 1 2 1 2 2 1 ...
 $ pdays    : int  -1 339 330 -1 -1 176 330 -1 -1 147 ...
 $ previous : int  0 4 1 0 0 3 2 0 0 2 ...
 $ poutcome : Factor w/ 4 levels "failure", "other", ...: 4 1 1 4 4 1 2 4 4 1
   ↪ ...
 $ deposit  : Factor w/ 2 levels "no", "yes": 1 1 1 1 1 1 1 1 1 1 ...
```

The dataset contains 4521 observations and 17 variables. The target variable, `deposit`, is binary, with two categories: yes and no. Below is a summary of all features:

Demographic features:

- age: Age of the customer (numeric).
- job: Type of job (e.g., “admin.”, “blue-collar”, “management”).
- marital: Marital status (e.g., “married”, “single”).
- education: Level of education (e.g., “secondary”, “tertiary”).
- default: Whether the customer has credit in default (binary: “yes”, “no”).
- balance: Average yearly balance in euros (numeric).

Loan information:

- housing: Whether the customer has a housing loan (binary).
- loan: Whether the customer has a personal loan (binary).

Campaign details:

- contact: Type of communication used (e.g., “telephone”, “cellular”).
- day: Last contact day of the month (numeric).
- month: Last contact month of the year (categorical: “jan”, “feb”, …, “dec”).
- duration: Last contact duration in seconds (numeric).
- campaign: Total number of contacts made with the customer during the campaign (numeric).
- pdays: Days since the customer was last contacted (numeric).
- previous: Number of contacts before the current campaign (numeric).
- poutcome: Outcome of the previous campaign (e.g., “success”, “failure”).

Target variable:

- deposit: Indicates whether the customer subscribed to a term deposit (binary: “yes”, “no”).

This dataset contains a diverse set of features related to customer demographics and past interactions, making it well-suited for building predictive models to improve marketing strategies. Because the dataset is already clean and well-structured, we can skip the initial data cleaning steps. We now proceed to *Step 4: Set Up Data for Modeling* in the *Data Science Workflow* introduced in Chapter 2 and illustrated in Figure 2.3.

Setup Data for Modeling

How do we know if a model will perform well on customers it has never seen? The answer begins with how we split the data. This step corresponds to Stage

4 of the Data Science Workflow: *Setup Data for Modeling* (Chapter 2). Our goal is to divide the dataset into separate training and test sets, allowing us to build models on past data and evaluate how well they generalize to new customers. Although we use a neural network for this case study, the same data split can be used to train and compare other classification models introduced in previous chapters—such as logistic regression (Chapter 10), k-nearest neighbors (Chapter 7), or Naive Bayes (Chapter 9). This allows for a fair evaluation of model performance under the same conditions, as discussed in the chapter on Model Evaluation (Chapter 8).

We use an 80/20 split, allocating 80% of the data for training and 20% for testing. But why 80/20? Would a 70/30 or 90/10 split yield different results? There is no universally optimal ratio—it often depends on dataset size and the trade-off between training data and evaluation reliability. You are encouraged to try different splits and reflect on the results.

To maintain consistency with earlier chapters, we apply the `partition()` function from the `liver` package:

```
set.seed(500)

data_sets = partition(data = bank, ratio = c(0.8, 0.2))

train_set = data_sets$part1
test_set = data_sets$part2

test_labels = test_set$deposit
```

The `set.seed()` function ensures reproducibility. The `train_set` is used to train the neural network model introduced in this case study, while the `test_set` serves as unseen data for evaluation. The `test_labels` vector stores the true labels of the test set, which we will later compare to the model's predictions.

To validate this split, we compare the proportion of customers who subscribed (`deposit = "yes"`) in both subsets using a two-sample Z-test:

```
x1 = sum(train_set$deposit == "yes")
x2 = sum(test_set$deposit == "yes")

n1 = nrow(train_set)
n2 = nrow(test_set)

prop.test(x = c(x1, x2), n = c(n1, n2))

 2-sample test for equality of proportions with continuity correction

data: c(x1, x2) out of c(n1, n2)
X-squared = 0.0014152, df = 1, p-value = 0.97
alternative hypothesis: two.sided
```

```

95 percent confidence interval:
-0.02516048  0.02288448
sample estimates:
prop 1   prop 2
0.1150124 0.1161504

```

The test confirms that the proportions in both subsets are statistically similar (p -value > 0.05), validating our split. This gives us confidence that the split preserves the class distribution, making evaluation results more reliable (see Table 6.1 in Section 6.5 for why we use a two-sample Z-test).

Our objective is to classify customers as either likely (deposit = "yes") or unlikely (deposit = "no") to subscribe to a term deposit, based on the following predictors: age, marital, default, balance, housing, loan, duration, campaign, pdays, and previous.

You might wonder why we selected only a subset of the available predictors—why include variables like age, balance, and duration while leaving out others such as job, education, contact, day, month, and poutcome. That is a fair question. For clarity and simplicity, this case study focuses on predictors that require minimal preprocessing. Our primary goal in this case study is to demonstrate how a neural network can be applied to real-world data with minimal preprocessing. In the exercises at the end of this chapter, you are invited to expand the model by incorporating additional features from the *bank* dataset. This provides an opportunity to practice data preparation, potentially improve model accuracy, and explore how richer feature sets influence learning.

Now that the data are partitioned, we move on to preparing them for modeling. Two key steps—*encoding categorical predictors* and *scaling numerical features*—ensure that all predictors are represented in a format suitable for neural networks. This preparation is essential because neural networks require numerical inputs and are sensitive to differences in feature scale.

Encoding Binary and Nominal Predictors

Since neural networks require numerical inputs, categorical variables must be converted into numeric representations. For binary and nominal (unordered) predictors, one-hot encoding is a suitable method. It transforms each category into a separate binary column. We apply the `one.hot()` function from the **liver** package to convert selected binary and nominal features into a format compatible with a neural network.

```

categorical_vars = c("marital", "default", "housing", "loan")

train_onehot = one.hot(train_set, cols = categorical_vars)
test_onehot = one.hot(test_set, cols = categorical_vars)

str(test_onehot)
'data.frame':   904 obs. of  22 variables:
 $ age          : int  43 40 56 25 31 32 23 36 32 32 ...
 $ job          : Factor w/ 12 levels "admin.", "blue-collar", ...: 1 5 10
   ↘ 2 10 2 8 5 10 3 ...
 $ marital_divorced: int  0 0 0 0 0 0 0 0 0 0 ...
 $ marital_married : int  1 1 1 0 1 1 0 0 0 0 ...
 $ marital_single  : int  0 0 0 1 0 0 1 1 1 1 ...
 $ education     : Factor w/ 4 levels "primary", "secondary", ...: 2 3 2 1
   ↘ 2 2 3 3 3 1 ...
 $ default_no    : int  1 1 1 1 1 1 1 1 1 0 ...
 $ default_yes   : int  0 0 0 0 0 0 0 0 0 1 ...
 $ balance       : int  264 194 4073 -221 171 2089 363 553 2204 -849 ...
 $ housing_no    : int  0 1 1 0 1 0 0 1 0 0 ...
 $ housing_yes   : int  1 0 0 1 0 1 1 0 1 1 ...
 $ loan_no       : int  1 0 1 1 1 1 1 1 1 0 ...
 $ loan_yes      : int  0 1 0 0 0 0 0 0 0 1 ...
 $ contact       : Factor w/ 3 levels "cellular", "telephone", ...: 1 1 1 3
   ↘ 1 1 3 1 1 1 ...
 $ day           : int  17 29 27 23 27 14 30 11 21 4 ...
 $ month         : Factor w/ 12 levels "apr", "aug", "dec", ...: 1 2 2 9 2
   ↘ 10 9 2 10 4 ...
 $ duration      : int  113 189 239 250 81 132 16 106 11 204 ...
 $ campaign      : int  2 2 5 1 3 1 18 2 4 1 ...
 $ pdays         : int  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 ...
 $ previous      : int  0 0 0 0 0 0 0 0 0 0 ...
 $ poutcome      : Factor w/ 4 levels "failure", "other", ...: 4 4 4 4 4 4
   ↘ 4 4 4 4 ...
 $ deposit       : Factor w/ 2 levels "no", "yes": 1 1 1 1 1 2 1 1 1 1 ...

```

The `one.hot()` function expands each categorical variable into multiple binary columns. For instance, the `marital` variable, which has three categories (`married`, `single`, and `divorced`), is transformed into three binary indicator variables: `marital_married`, `marital_single`, and `marital_divorced`. To avoid multicollinearity (also known as the dummy variable trap), we include only two of these in the model formula—in this case, `marital_married` and `marital_single`. The third category (`marital_divorced`) becomes the reference group. The same principle applies to other nominal predictors, such as `default`, `housing`, and `loan`.

Note: Ordinal features (such as `education` level) are not always appropriate for one-hot encoding, as doing so may ignore their inherent order. Alternative encoding strategies may be more suitable; see Section 3.13 for more details.

Here is the formula used to specify the predictors for the neural network:

```
formula = deposit ~ marital_married + marital_single + default_yes +
  ↵ housing_yes + loan_yes + age + balance + duration + campaign + pdays +
  ↵ previous
```

This formula includes both the transformed categorical predictors and the numeric predictors we introduced earlier. With encoding complete, we now turn to scaling the numeric features to ensure they are on a comparable scale.

Feature Scaling for Numerical Predictors

Neural networks perform best when input features are on a similar scale. To achieve this, we normalize numerical variables using min-max scaling, transforming all inputs into a standardized range between 0 and 1. This prevents features with larger numerical ranges from dominating the learning process and helps improve model convergence.

To prevent data leakage, the scaling parameters (minimum and maximum values) are computed using only the training set and then applied consistently to the test set. This ensures that no information from the test data influences model training, thereby preserving the independence of the evaluation step. For a visual example of how improper scaling can distort evaluation, refer to Figure 7.6 in Section 7.5.

```
numeric_vars = c("age", "balance", "duration", "campaign", "pdays",
  ↵ "previous")

min_train = sapply(train_onehot[, numeric_vars], min)
max_train = sapply(train_onehot[, numeric_vars], max)

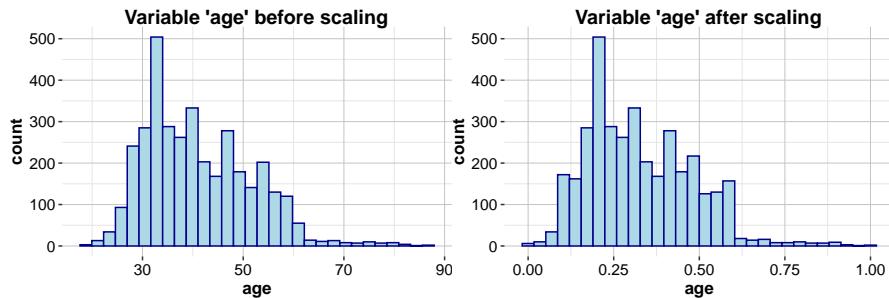
train_scaled = minmax(train_onehot, col = numeric_vars, min = min_train, max
  ↵ = max_train)
test_scaled = minmax(test_onehot, col = numeric_vars, min = min_train, max
  ↵ = max_train)
```

We use the `sapply()` function to compute the minimum and maximum values for each numeric variable across the training set. These values are then passed to the `minmax()` function from the `liver` package, which applies min-max scaling to both the training and test datasets using the same parameters.

To visualize the effect of scaling, we compare the distribution of age before and after transformation:

```
ggplot(data = train_set) +
  geom_histogram(mapping = aes(x = age), colour = "darkblue", fill =
  ↵ "lightblue") +
```

```
ggtitle("Variable 'age' before scaling")
ggplot(data = train_scaled) +
  geom_histogram(mapping = aes(x = age), colour = "darkblue", fill =
    "lightblue") +
  ggtitle("Variable 'age' after scaling")
```



The first histogram (left) shows the distribution of age in the training set before scaling, while the second histogram (right) shows the distribution after applying min-max scaling. The values are mapped to the interval $[0, 1]$ while preserving the original shape of the distribution.

Now that we have partitioned and prepared the data appropriately—through encoding and scaling—we are ready to fit our first neural network model. Are you ready to see how these ideas come together in practice?

Training a Neural Network Model in R

We use the **neuralnet** package in R to implement and visualize a feedforward neural network. This package provides a straightforward and flexible way to build neural networks, along with functionality for inspecting network topology. While **neuralnet** is a useful learning tool, it is also powerful enough for small- to medium-scale applications.

If the **neuralnet** package is not installed, you can install it using `install.packages("neuralnet")`. Then load it into the R session:

```
library(neuralnet)
```

We train the network using the `neuralnet()` function:

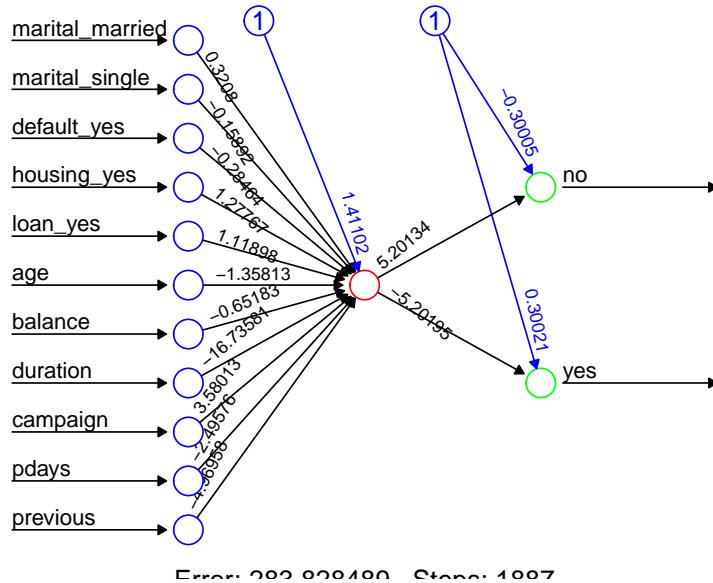
```
neuralnet_model = neuralnet(
  formula = formula,
  data = train_scaled,
  hidden = 1,           # Single hidden layer with 1 node
  err.fct = "sse",       # Loss function: Sum of Squared Errors
  linear.output = FALSE  # Logistic activation function for classification
)
```

Here's what each argument in the function call does:

- **formula**: Specifies the relationship between the target variable (`deposit`) and the predictors.
- **data**: Indicates the dataset used for training (`train_scaled`).
- **hidden = 1**: Defines the number of hidden layers and nodes (one hidden layer with a single node). For simplicity, we start with a minimal architecture.
- **err.fct = "sse"**: Specifies the sum of squared errors as the loss function. While SSE is commonly used, cross-entropy loss (`ce`) is often preferred for classification tasks.
- **linear.output = FALSE**: Ensures that the output layer uses a logistic activation function, which is appropriate when outputs represent probabilities.

After training, we can visualize the network architecture:

```
plot(neuralnet_model, rep = "best", fontsize = 10,
     col.entry = "blue", col.hidden = "red", col.out = "green")
```



This visualization shows that the network consists of:

- 11 input nodes, corresponding to the 11 predictors,
- 1 hidden layer containing a single node, and
- 2 output nodes representing the classification result (yes or no).

The training process converged after 1887 steps, indicating that the error function had stabilized. The final error rate is 283.83. An analysis of the network weights suggests that duration—the length of the last phone call—has the strongest influence on the outcome. This finding is consistent with earlier studies, where longer calls were associated with higher engagement.

To experiment with how network complexity affects learning, try adjusting the hidden parameter to include more nodes or additional layers:

```
# One hidden layer with 3 nodes
neuralnet_model_3 = neuralnet(
  formula = formula,
  data = train_scaled,
  hidden = 3,
  err.fct = "sse",
  linear.output = FALSE
)

# Two hidden layers: first with 3 nodes, second with 2 nodes
neuralnet_model_3_2 = neuralnet(
  formula = formula,
```

```
  data = train_scaled,  
  hidden = c(3, 2),  
  err.fct = "sse",  
  linear.output = FALSE  
)
```

You can visualize these more complex models using:

```
plot(neuralnet_model_3, rep = "best")  
plot(neuralnet_model_3_2, rep = "best")
```

Observe how architectural changes affect training convergence, model flexibility, and computation time. How do these changes influence performance? You are encouraged to compare results using the model evaluation techniques introduced in earlier chapters.

Note: The **neuralnet** package is ideal for learning and small-scale experiments, but it lacks support for GPU acceleration and may not scale well to large datasets or deep architectures. For more advanced applications, consider exploring packages such as **keras** or **torch**, which are well-supported in R and enable training deep neural networks efficiently.

This simple model demonstrates how a neural network processes input features through multiple layers to identify patterns and make predictions. In the next section, we evaluate the model's performance and interpret the results.

Prediction and Model Evaluation

How well does our neural network perform on customers it has never seen before? To answer this, we evaluate the model's predictive performance on the test set. This corresponds to the final step of the Data Science Workflow: assessing how well a model generalizes to new data (Chapter 8).

We begin by generating predictions for the test set using the `predict()` function:

```
neuralnet_probs = predict(neuralnet_model, test_scaled)
```

This produces the raw output activations for each observation. Since we are solving a binary classification problem, the network has two output nodes: one for `deposit = "yes"` and one for `deposit = "no"`. These are not normalized probabilities, but unscaled activations.

Let us examine the first few predicted outputs:

```
head(neuralnet_probs)
 [,1]      [,2]
 [1,] 0.9859681 0.01402658
 [2,] 0.9800866 0.01990667
 [3,] 0.8948246 0.10516268
 [4,] 0.9708354 0.02915618
 [5,] 0.9749485 0.02504382
 [6,] 0.9860860 0.01390872
```

To classify each customer, we compare the two activation values. If the activation for deposit = "yes" (column 2) is greater than the activation for no (column 1), we predict a subscription.

We extract the relevant values and create a confusion matrix using a threshold of 0.5:

```
# Extract predictions for 'deposit = "yes"'
neuralnet_probs_yes = neuralnet_probs[, 2]

conf.mat(neuralnet_probs_yes, test_labels, cutoff = 0.5, reference = "yes")
  Actual
Predict yes no
  yes  23 16
  no   82 783
```

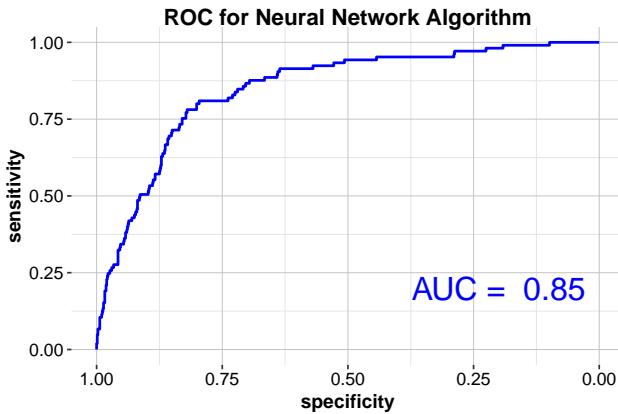
The confusion matrix summarizes the model's classification performance. It reveals how many predictions are true positives (correctly predicted subscribers), false positives (predicted yes but actually no), true negatives, and false negatives.

To further assess model performance across different thresholds, we plot the Receiver Operating Characteristic (ROC) curve and compute the Area Under the Curve (AUC):

```
library(pROC)

neuralnet_roc = roc(test_labels, neuralnet_probs_yes)

ggroc(neuralnet_roc, size = 0.8, colour = "blue") +
  ggtitle("ROC for Neural Network Algorithm") +
  annotate("text", x = .2, y = .2, size = 7, col = "blue",
           label = paste("AUC = ", round(auc(neuralnet_roc), 3))) +
  theme(legend.title = element_blank(),
        legend.position = c(.7, .3),
        text = element_text(size = 17))
```



The ROC curve visualizes the trade-off between sensitivity (true positive rate) and specificity (false positive rate) across thresholds. The AUC score—0.85—summarizes this performance into a single value. An AUC of 1 indicates perfect classification; a value close to 0.5 suggests random guessing.

This step completes our evaluation of the trained neural network. You are encouraged to:

- Try different cutoff values (e.g., 0.4 or 0.6) and compare confusion matrices.
- Identify false positives or false negatives and consider what features might lead to these misclassifications.
- Compare this model's ROC curve with those from previous chapters to see how neural networks perform relative to simpler classifiers.

In the next section, we reflect on key takeaways from this case study and discuss how neural networks can be extended to tackle more complex problems.

Reflections and Takeaways

This case study has walked you through the complete process of applying a neural network to a real-world classification problem—from understanding the data to evaluating model performance. Along the way, we followed the stages of the Data Science Workflow and explored practical considerations like feature encoding, data scaling, and model evaluation using confusion matrices and ROC curves.

Several key takeaways emerge:

- Neural networks can effectively learn patterns in marketing data, especially when variables like call duration are strong indicators of customer behavior.
- Proper data partitioning, transformation, and scaling are crucial to ensure model performance and generalizability.
- Model complexity matters. By adjusting the number of hidden layers and nodes, you can explore trade-offs between flexibility, training time, and overfitting.
- This model can be expanded by including additional features from the dataset (e.g., job, education, or contact). Doing so requires thoughtful preprocessing but may lead to performance gains.
- It is also instructive to compare the neural network's performance with simpler classifiers—such as logistic regression (Chapter 10), k-nearest neighbors (Chapter 7), Naive Bayes (Chapter 9), or tree-based models (Chapter 11). Doing so helps clarify when the added complexity of a neural network is justified.

While this case study focused on a binary classification task, neural networks are not limited to classification. They can also be applied to regression problems, time series forecasting, and more complex tasks such as image recognition and natural language processing—topics we will encounter in later chapters.

By experimenting with architectures, tuning hyperparameters, and comparing with other models, you will deepen your understanding of how neural networks behave in practice—and how to use them effectively to support data-driven decision making.

12.7 Chapter Summary and Takeaways

Can machines learn like humans? In this chapter, we explored how neural networks—mathematical models inspired by the human brain—form the backbone of many modern artificial intelligence systems. From their biological origins to their computational mechanics, we examined how these models process information, adapt through training, and uncover complex patterns in data.

Here are the key ideas you've learned:

- *Neural networks extend linear models* by introducing hidden layers and activation functions, enabling them to capture nonlinear and interactive effects.

- *Activation functions* like sigmoid, tanh, and ReLU inject non-linearity into the network, allowing it to model intricate relationships and make flexible decisions.
- *Learning happens iteratively* through gradient-based optimization, where weights are updated to reduce prediction error. This training process is guided by algorithms like gradient descent and relies on differentiable activation functions.
- *Architecture matters*: The number of layers and neurons shapes the model's capacity, interpretability, and computational demands.
- *Beyond classification*, neural networks can be applied to regression, time series forecasting, and deep learning tasks in vision, language, and beyond.

The term deposit case study demonstrated how to build and evaluate a neural network in **R**, reinforcing key ideas from theory with practical implementation. You saw how careful data preparation, architecture design, and model evaluation come together in real-world predictive modeling.

As you move into the exercises, consider experimenting with different architectures, adding more features, or comparing this model with others introduced in earlier chapters. Neural networks are powerful—but their effectiveness depends on your ability to apply them thoughtfully, tune them carefully, and evaluate them critically.

12.8 Exercises

These exercises help consolidate your understanding of neural networks by encouraging you to apply what you've learned, reflect on key concepts, and compare neural networks with alternative classification models. The exercises are grouped into three categories: conceptual questions, practical modeling tasks using the *bank* dataset, and comparative analysis using the *adult* dataset.

Conceptual questions

1. Describe how a neural network is structured, and explain the function of each layer (input, hidden, output).
2. What is the role of activation functions in a neural network, and why are non-linear functions essential for learning?

3. Compare and contrast ReLU, sigmoid, and tanh activation functions. In what types of problems would each be preferred?
4. Why are neural networks considered universal function approximators?
5. Outline how backpropagation works. What role does it play in adjusting the weights of a neural network?
6. Why do neural networks often require large datasets to achieve strong performance?
7. Define the bias-variance tradeoff in neural networks. How does model complexity influence bias and variance?
8. What is dropout regularization, and how does it help prevent overfitting?
9. What is the purpose of the loss function in neural network training?
10. What is weight initialization, and why does it matter for training stability?
11. Compare shallow and deep neural networks. What advantages do deeper architectures offer?
12. How does increasing the number of hidden layers affect a neural network's capacity to model complex patterns?
13. What are the signs that your neural network is underfitting or overfitting the data? What strategies can help address each issue?
14. Why is hyperparameter tuning important in neural networks? Which parameters are commonly tuned?
15. Compare the computational efficiency of decision trees and neural networks.
16. Reflect on a real-world application (e.g., fraud detection, voice recognition, or image classification). Why might a neural network be a suitable model? What trade-offs would you consider?

Practical exercises using the bank dataset

In the case study, we used a subset of predictors from the *bank* dataset to build a simple neural network model. This time, you will use **all** available features to further explore the dataset, build more sophisticated models, and compare performance across techniques.

Data preparation and model training

17. Load the *bank* dataset and examine its structure. Which variables are categorical? Which are numeric?
18. Split the dataset into training (70%) and testing (30%) sets. Validate the partition by comparing the proportion of customers who subscribed to a term deposit. Refer to Section 12.6 for guidance.
19. Apply one-hot encoding to all categorical features. How many new features are created?
20. Apply min-max scaling to numerical features. Why is feature scaling important for neural networks?
21. Train a feed-forward neural network with one hidden layer containing five neurons. Evaluate its classification accuracy on the test set.
22. Increase the number of neurons in the hidden layer to ten. How does this affect accuracy or training time?
23. Train a neural network with two hidden layers (five neurons in the first layer, three in the second). Compare its performance to the previous models.
24. Change the activation function from ReLU to sigmoid. How does this affect convergence and accuracy?
25. Train a model using cross-entropy loss instead of sum of squared errors. Which performs better?

Model evaluation and comparison with tree-based models

26. Compute the confusion matrix. Interpret the model's precision, recall, and F1-score.
27. Plot the ROC curve and calculate the AUC. How well does the model distinguish between classes?
28. Train a decision tree classifier (CART and C5.0) on the same data. Compare its performance with the neural network model.
29. Train a random forest model. How does its performance compare to the neural network and decision trees?
30. Train a logistic regression model. How does it perform relative to the other models?

31. Which model—decision tree, random forest, logistic regression, or neural network—performs best in terms of accuracy, precision, and recall? What trade-offs do you observe among these models?

Training Neural Networks on the *adult* Dataset

This set of exercises extends your neural network modeling skills to a second dataset—the *adult* dataset—commonly used to predict whether an individual earns more than \$50K per year based on demographic and employment attributes. For data preparation steps—including handling missing values, encoding categorical variables, and scaling numerical features—refer to the case study in the previous chapter (Section 11.5). Reusing the same preprocessing pipeline ensures a fair comparison between neural networks and tree-based models (CART, C5.0, and random forest).

32. Load the *adult* dataset and examine its structure. What are the key differences between this dataset and the *bank* dataset?
33. Preprocess the categorical features using one-hot encoding.
34. Normalize the numerical features using min-max scaling.
35. Split the dataset into training (80%) and testing (20%) sets.
36. Train a basic neural network with one hidden layer (five neurons) to predict income level ($\leq 50K$ or $> 50K$).
37. Increase the number of neurons in the hidden layer to ten. Does performance improve?
38. Train a deeper neural network with two hidden layers (ten and five neurons).
39. Compare ReLU, tanh, and sigmoid activation functions on model performance.
40. Train a decision tree on the *adult* dataset and compare its accuracy with the neural network model.
41. Train a random forest on the *adult* dataset and compare its performance to the neural network model.
42. Analyze the feature importance in the random forest model and compare it to the most influential features in the neural network model.
43. Compare the ROC curves of the neural network, decision tree, and random forest models. Which model has the highest AUC?
44. Which model performs better in predicting high-income individuals, and why?

Self-Reflection

45. Looking back at your work in this chapter, reflect on how model complexity, interpretability, and predictive performance differ across algorithms. What trade-offs arise in choosing between a neural network and a simpler model like logistic regression or a decision tree? How might these considerations influence model selection in real-world applications?
46. Which parts of the modeling pipeline (e.g., preprocessing, model selection, tuning, evaluation) did you find most challenging or insightful? How would you approach a new dataset differently based on what you have learned in this chapter?

Chapter 13

Clustering for Insight: Segmenting Data Without Labels

Imagine walking into a grocery store and seeing shelves lined with cereal boxes. Without reading a single label, you might instinctively group them by shape, size, or color. Clustering algorithms aim to replicate this kind of human intuition—grouping similar items based on shared characteristics, even when no categories are given.

How do apps know your habits when you never told them? From fitness trackers that sort users into behavioral types to streaming platforms that recommend shows tailored to your taste, machines often uncover structure in data without relying on labels. This process—known as *clustering*—enables systems to learn from raw, unlabeled information.

Clustering is a form of *unsupervised learning* that groups similar data points based on measurable traits, rather than predefined categories. It powers real-world applications such as customer segmentation, gene family discovery, and content recommendation. By organizing complex information into meaningful groups, clustering helps machines detect patterns and uncover structure hidden in the data.

Unlike classification, which predicts known labels (e.g., spam vs. not spam), clustering is *exploratory*. It reveals *hidden structures*—patterns that may not be immediately visible but are statistically meaningful. As such, clustering is a key part of the data scientist’s toolkit when the objective is *discovery* rather than prediction.

Clustering is widely used across domains, including customer segmentation for identifying distinct user groups for personalized marketing, market research for understanding consumer behavior to improve product recommendations, document organization for automatically grouping large text collections by topic or theme, and bioinformatics for uncovering functional relationships between genes through expression pattern similarity.

What This Chapter Covers

In this chapter, we introduce our first unsupervised learning technique—clustering—and continue progressing through the Data Science Workflow introduced in Chapter 2. So far, our focus has been on supervised learning, applying models for classification and regression tasks, including neural networks (Chapter 12), tree-based methods (Chapter 11), and regression analysis (Chapter 10).

Clustering now opens the door to data exploration when no labels are available—shifting our mindset from prediction to pattern discovery.

This chapter introduces the foundations of clustering, including:

- The core idea of clustering and how it differs from classification,
- How similarity is defined and measured in clustering algorithms,
- *K-means clustering*, one of the most intuitive and widely used methods,
- A hands-on case study: segmenting cereals based on their nutritional profiles.

By the end of the chapter, you will be able to apply clustering techniques to real-world datasets, evaluate cluster quality, and uncover meaningful patterns from unlabeled data.

13.1 What is Cluster Analysis?

Clustering is an unsupervised learning technique that organizes data into *clusters* of similar observations. Unlike supervised learning, which relies on labeled data, clustering is *exploratory* in nature, aiming to uncover *hidden patterns* or *latent structure* in raw data. A good clustering groups data points so that members of the same cluster are highly similar to one another, while those in different clusters are clearly distinct.

To better understand what makes clustering unique, it helps to compare it with *classification*, as introduced in Chapters 7 and 9. *Classification* assigns new observations to known categories based on past examples—like identifying an email as spam or not spam. *Clustering*, by contrast, discovers groupings from unlabeled data. It generates labels rather than predicting existing ones, which is why it is sometimes loosely referred to as *unsupervised classification*, even though no labels are provided during training. These cluster labels can also be used downstream—for example, as input features for neural networks or tree-based models.

The core objective of clustering is to ensure *high intra-cluster similarity* (points in the same cluster are alike) and *low inter-cluster similarity* (clusters are distinct). This idea is illustrated in Figure 13.1, where tight, well-separated groups represent an effective clustering.

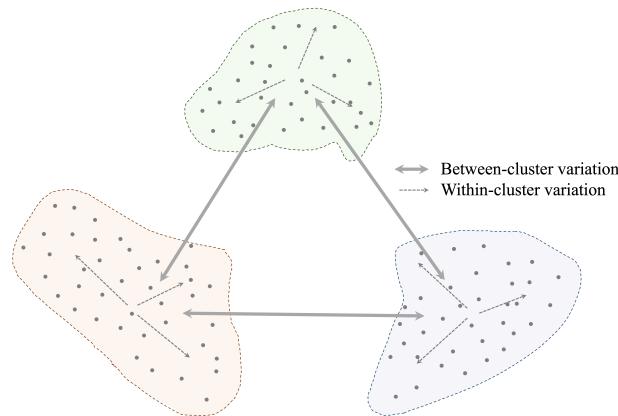


Figure 13.1: Clustering algorithms aim to minimize intra-cluster variation while maximizing inter-cluster separation.

Beyond helping us explore structure in data, clustering also plays a practical role in broader machine learning workflows. It is often used as a powerful *preprocessing tool*, summarizing a dataset into a smaller number of representative groups. This can:

- Reduce computation time for downstream models,
- Improve interpretability by simplifying complex data structures, and
- Enhance predictive performance by transforming raw inputs into structured features.

Before clustering can be applied effectively, the data often need to be pre-processed. Features measured on different scales can distort similarity measures, and categorical variables must be encoded numerically. These steps—such as scaling and one-hot encoding—not only improve algorithm performance but also ensure that the resulting clusters reflect meaningful structure.

What makes two observations feel similar—and how do machines measure that? Let us break it down in the next section.

How Do Clustering Algorithms Measure Similarity?

At the heart of clustering lies a fundamental question: *How similar are these data points?* Clustering algorithms answer this using *similarity measures*—quantitative tools for assessing how close or far apart two observations are. Choosing the right measure is essential for discovering meaningful clusters.

For numerical data, one of the most commonly used similarity measures is *Euclidean distance*—the straight-line distance between two points in space. You may recall this from the *k*-Nearest Neighbors algorithm (Section 7.4), where it helped identify the “nearest” neighbors. In clustering, it plays a similar role in grouping nearby observations together.

The Euclidean distance between two data points $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_n)$ with n features is calculated as:

$$\text{dist}(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

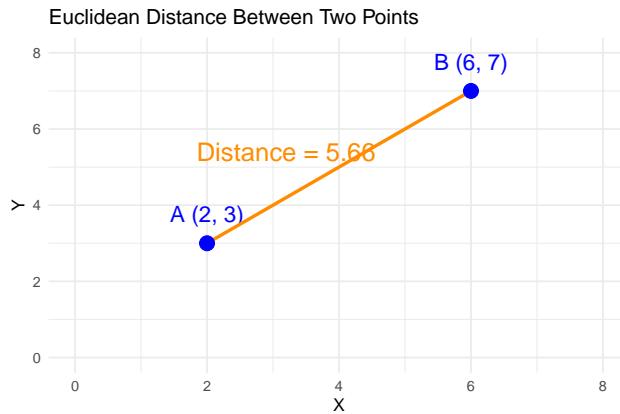


Figure 13.2: Visual representation of Euclidean distance between two points in 2D space.

In Figure 13.2, the line connecting Point A (2, 3) and Point B (6, 7) represents their Euclidean distance:

$$\text{dist}(A, B) = \sqrt{(6 - 2)^2 + (7 - 3)^2} = \sqrt{32} \approx 5.66.$$

While this is easy to visualize in two dimensions, clustering usually takes place in much higher dimensions—across dozens or even hundreds of features.

Before we can meaningfully apply distance-based clustering, we must prepare the data:

- *Feature scaling* (e.g., min-max scaling) ensures that no variable dominates the calculation simply because of its unit or range.
- *Categorical variables* must be numerically encoded (e.g., with one-hot encoding) to be included in distance computations.

Without these steps, even a good algorithm may find *spurious patterns* or miss *real ones*. Getting similarity right is the foundation of meaningful clustering.

Other similarity measures, such as *Manhattan distance* or *cosine similarity*, are also used in specific contexts—but Euclidean distance remains the default for many clustering tasks.

13.2 K-means Clustering

How does an algorithm decide which points belong together? K-means clustering answers this by iteratively grouping observations into k clusters, where each group contains data points that are similar to one another. The algorithm updates the cluster assignments and centers until the structure stabilizes, resulting in a set of well-separated clusters.

The K-means algorithm requires the user to specify the number of clusters, k , in advance. It proceeds through the following steps:

1. *Initialize*: Randomly select k data points as initial cluster centers.
2. *Assign*: Assign each data point to the nearest center based on distance.
3. *Update*: Recalculate each cluster's centroid (mean of its assigned points).
4. *Repeat*: Iterate steps 2 and 3 until no data points change clusters.

Although K-means is simple and efficient, it has limitations. The final clusters depend heavily on the initial choice of cluster centers, meaning different runs of the algorithm may produce different results. In addition, K-means assumes that clusters are spherical and of similar size, which may not always hold in real-world datasets. It is also sensitive to outliers, which can distort centroids and assignments.

To illustrate how K-means works, consider a dataset with 50 records and two features, x_1 and x_2 , as shown in Figure 13.3. The task is to partition the data into three clusters.

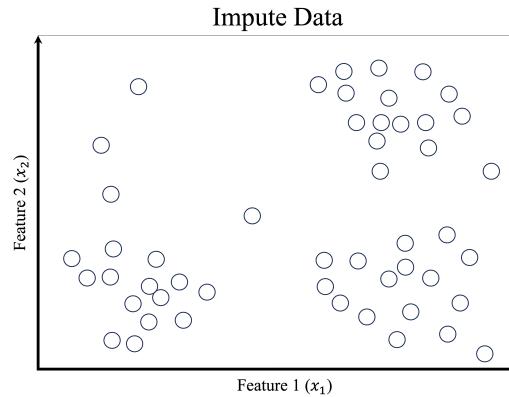


Figure 13.3: Scatter plot of 50 data points with two features, x_1 and x_2 , used as the starting point for K-means clustering.

In the first step, three data points are randomly selected as initial cluster centers (red stars), shown in the left panel of Figure 13.4. Each data point is then assigned to the nearest cluster, forming three groups labeled in blue (Cluster A), green (Cluster B), and orange (Cluster C). The right panel of the figure shows these initial assignments. The dashed lines depict the Voronoi diagram, which partitions the space into regions closest to each center.

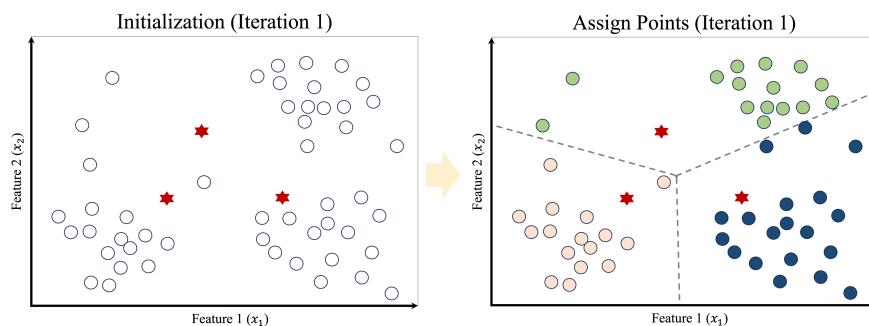


Figure 13.4: First iteration of K-means clustering. Left panel shows randomly initialized cluster centers (red stars); right panel shows the resulting initial assignments and Voronoi regions.

Because K-means is sensitive to initialization, poor placement of the initial cluster centers can lead to suboptimal results. To address this, the *K-means++* algorithm (Arthur and Vassilvitskii 2006) was introduced in 2007. It selects starting points in a more informed way, improving convergence and reducing variability across different initializations.

After the initial assignment, the algorithm enters the update phase. It recalculates the centroid of each cluster—that is, the mean position of all points in the group. The original cluster centers are updated by moving them to these new centroids, as shown in the left panel of Figure 13.5. The right panel shows how the Voronoi boundaries shift, causing some points to be reassigned.

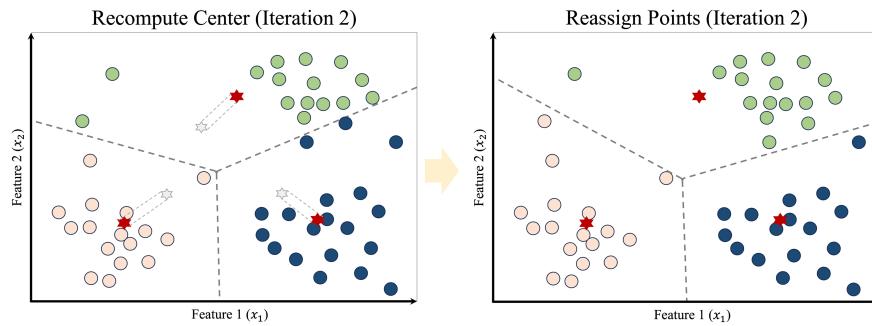


Figure 13.5: Second iteration of K-means clustering. Left panel shows updated cluster centroids; right panel displays new assignments and the corresponding Voronoi regions.

This process of reassigning points and updating centroids continues iteratively. After another update, some points switch clusters again, leading to a refined partition of the space, as seen in Figure 13.6.

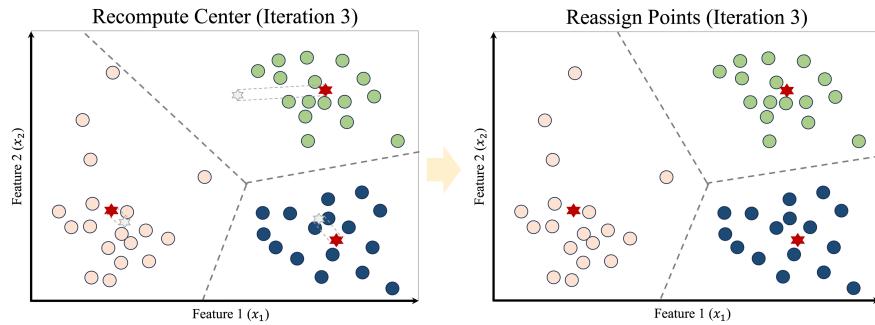


Figure 13.6: Third iteration of K-means clustering. Cluster centroids and point assignments are updated again as the algorithm continues refining the groupings.

The algorithm continues until no more data points switch clusters. At this point, it has converged, and the final clusters are established, as shown in Figure 13.7.

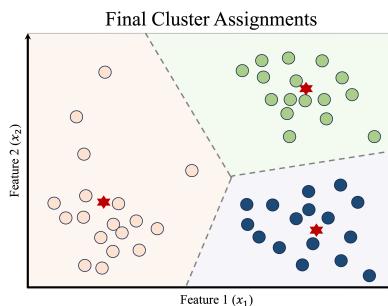


Figure 13.7: Final iteration of K-means clustering. Each data point is assigned to a stable cluster after convergence.

Once clustering is complete, the results can be summarized in two ways:

- *Cluster assignments*: Each data point is labeled as belonging to Cluster A, B, or C.
- *Centroid coordinates*: The final positions of the cluster centers can be used as representative points.

These centroids are particularly useful in applications such as customer segmentation, image compression, and document clustering, where the goal is to reduce complexity while preserving meaningful structure.

This simple example illustrates the core mechanics of K-means. But choosing how many clusters to use—our next topic—is just as critical to achieving meaningful results.

13.3 Selecting the Optimal Number of Clusters

A central challenge in applying K-means clustering is determining the appropriate number of clusters, k . This choice directly affects the outcome: too few clusters may obscure important structure, while too many may overfit the data and lead to fragmentation. Unlike supervised learning—where performance metrics such as accuracy or AUC guide model selection—clustering lacks an external ground truth, making the selection of k inherently more subjective.

In practice, domain knowledge can offer initial guidance. For instance, when clustering films, the number of well-established genres might suggest a reasonable starting point. In marketing, teams may set $k = 3$ if they aim to develop three targeted strategies. Similarly, logistical constraints—such as seating capacity at a conference—may dictate the desired number of groups. However, in many cases, no natural grouping is evident, and data-driven approaches are needed to inform the decision.

A widely used heuristic is the *elbow method*, which examines how within-cluster variation evolves as k increases. As additional clusters are introduced, the average similarity within clusters improves—up to a point. Beyond that, the marginal gain becomes negligible. The objective is to identify this point of diminishing returns, known as the *elbow*.

This concept is illustrated in Figure 13.8, where the total within-cluster sum of squares (WCSS) is plotted against the number of clusters. The “elbow point”—a bend in the curve—suggests a reasonable choice for k that balances simplicity with explanatory power.

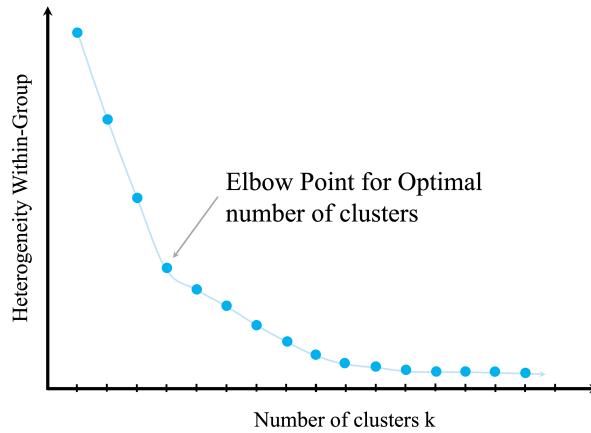


Figure 13.8: The elbow method visualizes the trade-off between the number of clusters and within-cluster variation, helping to identify an appropriate value for k .

While the elbow method is accessible and visually intuitive, it is not without limitations. Some datasets yield no clear inflection point, and evaluating many values of k can become computationally demanding in large-scale applications.

Alternative approaches can supplement or refine this decision:

- *Silhouette Score*: Quantifies how well each data point fits within its assigned cluster compared to others. Higher values indicate more coherent clusters.
- *Gap Statistic*: Compares the clustering result to that expected under a null reference distribution, helping assess whether structure exists at all.
- *Performance in downstream tasks*: When clustering is used as a preprocessing step—such as for customer segmentation—different values of k can be evaluated based on their impact on a subsequent predictive model.

Ultimately, the goal is not necessarily to find a mathematically optimal k , but to identify a clustering solution that is both interpretable and practically useful. Clustering is frequently employed in exploratory analysis, and observing how results change across values of k can itself be informative. Stable groupings that persist suggest meaningful structure; volatile groupings may reflect ambiguity in the data.

In the next section, we apply these ideas in practice using a real-world dataset. We explore how domain knowledge, visualization, and iterative experimentation can jointly inform the choice of k in applied settings.

13.4 Case Study: Segmenting Cereal Brands by Nutrition

Why do some cereals end up on the “healthy” shelf while others are marketed to kids? Behind such decisions lies data-driven product segmentation. In this case study, we use *K-means clustering* to uncover meaningful groupings based on nutritional content. Using the *cereal* dataset from the **liver** package, we cluster 77 cereal brands using variables such as calories, fat, protein, and sugar. While simplified, this example demonstrates how unsupervised learning techniques can support product design, marketing strategy, and consumer targeting.

This case study provides a focused, real-world example of how clustering works in practice—giving you hands-on experience applying K-means in R.

13.4.1 Overview of the Dataset

What do breakfast cereals reveal about nutritional marketing and consumer preferences? The *cereal* dataset offers a compact but information-rich glimpse into the world of packaged food products. It includes 77 breakfast cereals from major brands, described by 16 variables capturing nutritional content, product characteristics, and shelf placement. The dataset is included in the **liver** package and can be loaded with:

```
library(liver)
data(cereal)
```

To view the structure of the dataset:

```
str(cereal)
'data.frame': 77 obs. of 16 variables:
 $ name   : Factor w/ 77 levels "100% Bran","100% Natural Bran",...: 1 2 3
   ↪ 4 5 6 7 8 9 10 ...
 $ manuf  : Factor w/ 7 levels "A","G","K","N",...: 4 6 3 3 7 2 3 2 7 5 ...
 $ type   : Factor w/ 2 levels "cold","hot": 1 1 1 1 1 1 1 1 1 1 ...
 $ calories: int  70 120 70 50 110 110 110 130 90 90 ...
 $ protein: int  4 3 4 4 2 2 2 3 2 3 ...
 $ fat    : int  1 5 1 0 2 2 0 2 1 0 ...
 $ sodium : int  130 15 260 140 200 180 125 210 200 210 ...
 $ fiber   : num  10 2 9 14 1 1.5 1 2 4 5 ...
 $ carbo  : num  5 8 7 8 14 10.5 11 18 15 13 ...
 $ sugars  : int  6 8 5 0 8 10 14 8 6 5 ...
 $ potass : int  280 135 320 330 -1 70 30 100 125 190 ...
 $ vitamins: int  25 0 25 25 25 25 25 25 25 25 ...
 $ shelf   : int  3 3 3 3 3 1 2 3 1 3 ...
```

```
$ weight  : num 1 1 1 1 1 1 1.33 1 1 ...
$ cups    : num 0.33 1 0.33 0.5 0.75 0.75 1 0.75 0.67 0.67 ...
$ rating  : num 68.4 34 59.4 93.7 34.4 ...
```

Here is an overview of the key variables:

- name: Name of the cereal (categorical-nominal).
- manuf: Manufacturer (categorical-nominal).
- type: Cereal type—hot or cold (categorical-binary).
- calories: Calories per serving (numerical).
- protein: Grams of protein per serving (numerical).
- fat: Grams of fat per serving (numerical).
- sodium: Milligrams of sodium per serving (numerical).
- fiber: Grams of dietary fiber per serving (numerical).
- carbo: Grams of carbohydrates per serving (numerical).
- sugars: Grams of sugar per serving (numerical).
- potass: Milligrams of potassium per serving (numerical).
- vitamins: Percentage of recommended daily vitamins (categorical-ordinal: 0, 25, or 100).
- shelf: Display shelf position in stores (categorical-ordinal: 1, 2, or 3).
- weight: Weight of one serving in ounces (numerical).
- cups: Number of cups per serving (numerical).
- rating: Cereal rating score (numerical).

The dataset combines several feature types that reflect how real-world data is structured. It includes one binary variable (type), two nominal variables (name and manuf), and two ordinal variables (vitamins and shelf). The remaining variables are continuous numerical measures. Understanding these distinctions is essential for properly preparing the data for clustering.

Before clustering, we need to prepare the data by addressing missing values, selecting relevant features, and applying scaling—steps that ensure the algorithm focuses on meaningful nutritional differences rather than artifacts of data format.

13.4.2 Data Preprocessing

What makes some cereals more alike than others? Before we can explore that question with clustering, we must ensure that the data reflects meaningful similarities. This step corresponds to the second stage of the *Data Science Workflow* (Figure 2.3): Data Preparation (Section 3). Effective clustering

depends on distance calculations, which in turn rely on clean and consistently scaled inputs. Data preprocessing is therefore essential—especially when working with real-world datasets that often contain inconsistencies or hidden assumptions.

A summary of the cereal dataset reveals anomalous values in the sugars, carbo, and potass variables, where some entries are set to -1:

```
summary(cereal)
   name    manuf     type    calories
100% Bran      : 1 A: 1 cold:74 Min.  : 50.0
100% Natural Bran : 1 G:22 hot : 3 1st Qu.:100.0
All-Bran        : 1 K:23                   Median :110.0
All-Bran with Extra Fiber: 1 N: 6           Mean   :106.9
Almond Delight   : 1 P: 9           3rd Qu.:110.0
Apple Cinnamon Cheerios : 1 Q: 8           Max.   :160.0
(Other)         :71 R: 8

   protein      fat      sodium      fiber
Min.  :1.000  Min.  :0.000  Min.  : 0.0  Min.  : 0.000
1st Qu.:2.000 1st Qu.:0.000 1st Qu.:130.0 1st Qu.: 1.000
Median :3.000  Median :1.000  Median :180.0  Median : 2.000
Mean   :2.545  Mean   :1.013  Mean   :159.7  Mean   : 2.152
3rd Qu.:3.000 3rd Qu.:2.000 3rd Qu.:210.0 3rd Qu.: 3.000
Max.   :6.000  Max.   :5.000  Max.   :320.0  Max.   :14.000

   carbo      sugars      potass      vitamins
Min.  :-1.0  Min.  :-1.000  Min.  :-1.00  Min.  : 0.00
1st Qu.:12.0 1st Qu.: 3.000 1st Qu.: 40.00 1st Qu.: 25.00
Median :14.0  Median : 7.000  Median : 90.00  Median : 25.00
Mean   :14.6  Mean   : 6.922  Mean   : 96.08  Mean   : 28.25
3rd Qu.:17.0 3rd Qu.:11.000 3rd Qu.:120.00 3rd Qu.: 25.00
Max.   :23.0  Max.   :15.000  Max.   :330.00  Max.   :100.00

   shelf      weight      cups      rating
Min.  :1.000  Min.  :0.50  Min.  :0.250  Min.  :18.04
1st Qu.:1.000 1st Qu.:1.00 1st Qu.:0.670 1st Qu.:33.17
Median :2.000  Median :1.00  Median :0.750  Median :40.40
Mean   :2.208  Mean   :1.03  Mean   :0.821  Mean   :42.67
3rd Qu.:3.000 3rd Qu.:1.00 3rd Qu.:1.000 3rd Qu.:50.83
Max.   :3.000  Max.   :1.50  Max.   :1.500  Max.   :93.70
```

As discussed in Section 3.8, it is common for datasets to use codes like -1 or 999 to represent missing or unknown values—especially for attributes that should be non-negative. Since negative values are not valid for nutritional measurements, we treat these entries as missing:

```
cereal[cereal == -1] <- NA

find.na(cereal)
  row col
 [1,] 58  9
```

```
[2,] 58 10
[3,] 5 11
[4,] 21 11
```

The `find.na()` function from the `liver` package reports the locations of missing values. This dataset contains 4 such entries, with the first one appearing in row 58 and column 9.

To handle missing values in the `cereal` dataset, we apply *predictive imputation* using random forests, a method introduced in Section 3.8. This approach leverages the relationships among observed variables to estimate missing entries. We use the `mice()` function from the `mice` package that creates a predictive model for each variable with missing values, using the other variables as predictors. In this example, we use the "rf" method to perform random forest imputation and we generate a single imputed dataset using one iteration and a small number of trees for demonstration purposes:

```
library(mice)

imp <- mice(cereal, method = "rf", ntree = 3, m = 1, maxit = 1)

iter imp variable
  1   1  carbo sugars potass
cereal <- complete(imp)

find.na(cereal)
[1] "No missing values (NA) in the dataset."
```

The `complete()` function extracts the imputed dataset from the `mice` object. By default, it returns the first completed version when only one is created. The `mice()` function also supports a range of other imputation methods, including mean imputation ("mean"), predictive mean matching ("pmm"), and classification and regression trees ("cart"), allowing users to tailor the imputation strategy to their data and modeling needs.

The resulting `cereal` dataset contains no missing values, as confirmed by the `find.na()` function. This imputation step ensures that subsequent clustering analyses are not biased by incomplete records.

After imputation, no missing values remain, and the dataset is complete and ready for clustering. We now select the variables that will be used to group cereals. Three variables are excluded based on their role and structure:

- `name` is an identifier, functioning like an ID. It carries no analytical value for clustering.
- `manuf` is a nominal variable with seven categories. Encoding it would require six dummy variables, which may inflate dimensionality and distort distance metrics.

- rating reflects a subjective outcome (e.g., taste), rather than a feature of the cereal’s composition. It is more appropriate as a target variable in supervised learning than as an input for clustering.

```
selected_variables <- colnames(cereal)[-c(1, 2, 16)]
cereal_subset <- cereal[, selected_variables]
```

Because the numerical features span different scales (e.g., milligrams of sodium vs. grams of fiber), we apply min-max normalization using the `minmax()` function from the `liver` package:

```
cereal_mm <- minmax(cereal_subset, col = "all")
str(cereal_mm)
'data.frame': 77 obs. of 13 variables:
 $ type    : num  0 0 0 0 0 0 0 0 ...
 $ calories: num  0.182 0.636 0.182 0 0.545 ...
 $ protein : num  0.6 0.4 0.6 0.6 0.2 0.2 0.2 0.4 0.2 0.4 ...
 $ fat     : num  0.2 1 0.2 0 0.4 0.4 0 0.4 0.2 0 ...
 $ sodium  : num  0.4062 0.0469 0.8125 0.4375 0.625 ...
 $ fiber   : num  0.7143 0.1429 0.6429 1 0.0714 ...
 $ carbo   : num  0 0.167 0.111 0.167 0.5 ...
 $ sugars  : num  0.4 0.533 0.333 0 0.533 ...
 $ potass  : num  0.841 0.381 0.968 1 0.238 ...
 $ vitamins: num  0.25 0 0.25 0.25 0.25 0.25 0.25 0.25 0.25 ...
 $ shelf   : num  1 1 1 1 1 0 0.5 1 0 1 ...
 $ weight  : num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.83 0.5 0.5 ...
 $ cups    : num  0.064 0.6 0.064 0.2 0.4 0.4 0.4 0.6 0.4 0.336 0.336 ...
```

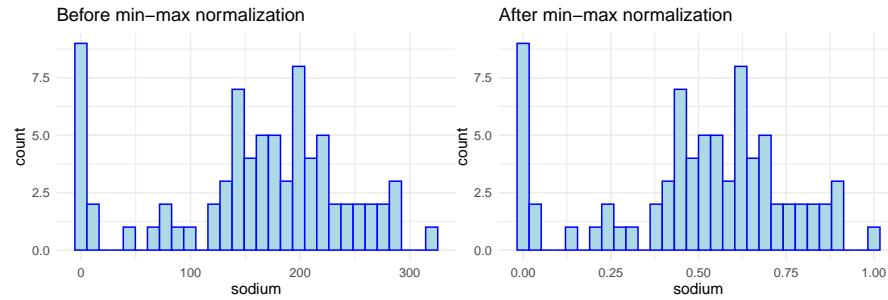
To visualize the effect of normalization, we compare the distribution of the sodium variable before and after scaling:

```
ggplot(data = cereal) +
  geom_histogram(aes(x = sodium), color = "blue", fill = "lightblue") +
  theme_minimal() + ggtitle("Before min-max normalization")

ggplot(data = cereal_mm) +
  geom_histogram(aes(x = sodium), color = "blue", fill = "lightblue") +
  theme_minimal() + ggtitle("After min-max normalization")
```

As shown in the histograms, normalization rescales all features to the [0, 1] range. This prevents variables like sodium or potassium—originally measured in large units—from overpowering the clustering process.

With the dataset cleaned, imputed, and normalized, we are now ready to explore how cereals naturally group together. But first, how many clusters should we use?



13.4.3 Selecting the Number of Clusters

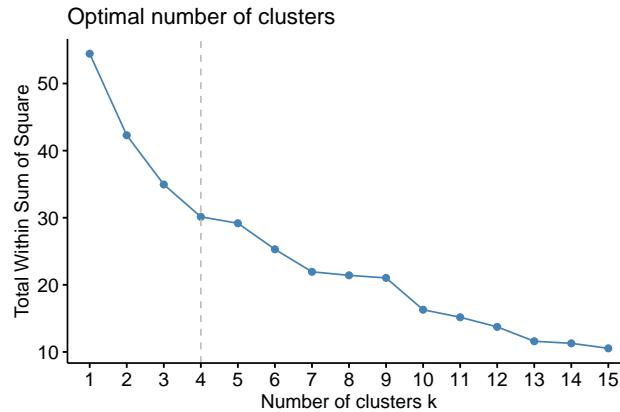
A key decision in clustering is selecting how many clusters (k) to use. Choosing too few clusters can obscure meaningful groupings, while too many may lead to overfitting or fragmented results. Because clustering is unsupervised, this decision must be guided by internal evaluation methods.

One widely used approach is the *elbow method*, which evaluates how the total within-cluster sum of squares (WCSS) decreases as k increases. Initially, adding more clusters significantly reduces WCSS, but beyond a certain point the improvement slows. The “elbow” in the plot—where the rate of decrease flattens—suggests a suitable value for k .

To create the elbow plot, we use the `fviz_nbclust()` function from the **factoextra** package. This package provides user-friendly tools for visualizing clustering results and evaluation metrics. The `fviz_nbclust()` function generates evaluation plots for different values of k based on methods such as WCSS or silhouette width.

```
library(factoextra)

fviz_nbclust(cereal_mm, kmeans, method = "wss", k.max = 15) +
  geom_vline(xintercept = 4, linetype = 2, color = "gray")
```



As shown in Figure 13.8, the WCSS drops sharply for small values of k , but levels off after $k = 4$. This suggests that four clusters may offer a reasonable balance between model complexity and within-cluster cohesion.

13.4.4 Performing K-means Clustering

With the number of clusters selected, we now apply the K-means algorithm to segment the cereals into four groups. We use the `kmeans()` function from base R, which does not require any additional packages. Its key arguments include the input data (`x`), the number of clusters (`centers`), and optional parameters such as the number of random starts (`nstart`), which helps avoid poor local optima.

We use `set.seed()` to ensure that the results are reproducible. Since the K-means algorithm involves random initialization of cluster centers, setting the seed guarantees that the same clusters are obtained each time the code is run.

```
set.seed(3) # Ensure reproducibility
cereal_kmeans <- kmeans(cereal_mm, centers = 4)
```

The `kmeans()` function returns several useful components, including:

- `cluster`: the cluster assignment for each observation,
- `centers`: the coordinates of the cluster centroids,
- `size`: the number of observations in each cluster,
- `tot.withinss`: the total within-cluster sum of squares (used earlier in the elbow method).

To check how the observations are distributed across the clusters:

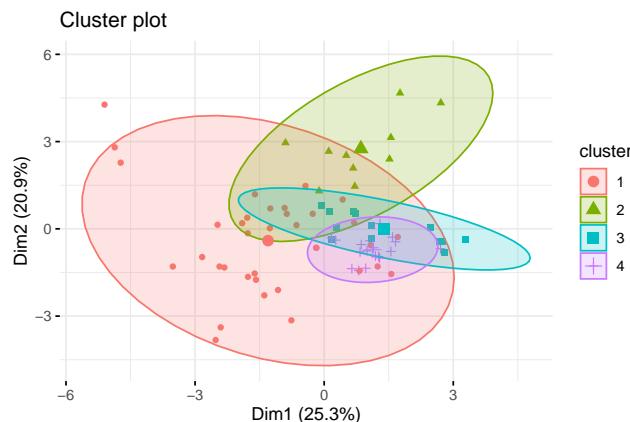
```
cereal_kmeans$size
[1] 36 10 13 18
```

The output shows the number of cereals assigned to each cluster, which can help us understand the distribution of products across the four groups.

Visualizing the Clusters

To gain insight into the clustering results, we use the `fviz_cluster()` function from the **factoextra** package to visualize the four groups:

```
fviz_cluster(cereal_kmeans, cereal_mm,
             geom = "point",
             ellipse.type = "norm",
             palette = "custom_palette",
             ggtheme = theme_minimal())
```



The resulting scatter plot displays the cluster structure, with each point representing a cereal. Colors indicate cluster membership, and ellipses represent the standard deviation around each cluster center. The plot is constructed using principal component analysis (PCA), which reduces the high-dimensional feature space to two principal components for visualization. Although some detail is inevitably lost, this projection helps reveal the overall shape and separation of the clusters.

Interpreting the Results

The clustering results reveal natural groupings among cereals based on their nutritional composition. For example:

- One cluster includes low-sugar, high-fiber cereals that are likely positioned for health-conscious consumers.
- Another contains high-calorie, high-sugar cereals typically marketed to children.
- A third represents balanced options with moderate levels of key nutrients.
- The fourth cluster combines cereals with higher protein or other distinctive profiles.

To examine which cereals belong to a particular cluster (e.g., Cluster 1), we can use:

```
cereal$name[cereal_kmeans$cluster == 1]
```

This command returns the names of cereals assigned to Cluster 1, allowing for further inspection and interpretation of that group's defining features.

13.4.5 Reflections and Takeaways

The cereal clustering analysis illustrates how K-means can be used to segment products based on measurable features—in this case, nutritional content. By combining careful preprocessing, feature scaling, and model evaluation, we identified coherent groupings that reflect distinct product profiles.

More generally, this example highlights the value of unsupervised learning in discovering hidden patterns when no outcome variable is available. Clustering is widely used in domains such as marketing, health analytics, and customer segmentation, where understanding natural structure in the data leads to better decisions and targeted strategies.

The process illustrated here—choosing relevant features, selecting the number of clusters, and interpreting results—forms the foundation for applying clustering techniques to other domains. Whether used for segmenting users, detecting anomalies, or grouping documents, clustering provides a flexible tool for uncovering structure and generating insights.

13.5 Chapter Summary and Takeaways

In this chapter, we introduced clustering as a fundamental technique for unsupervised learning—where the goal is to group observations based on similarity without using labeled outcomes.

We focused on the K-means algorithm, one of the most widely used clustering methods. You learned how K-means iteratively partitions data into k clusters by minimizing the within-cluster sum of squares. Selecting an appropriate number of clusters is crucial, and we explored common evaluation methods such as the elbow method.

We emphasized the importance of proper data preparation, including selecting relevant features, handling missing values, and applying scaling techniques to ensure fair distance calculations.

Through a case study using the cereal dataset, we demonstrated how to apply K-means in R, visualize the resulting clusters, and interpret their meaning in a real-world context. Unlike earlier chapters, we did not partition the dataset into training and testing sets. Since clustering is an unsupervised technique, there is no outcome variable to predict, and evaluation relies on internal measures such as within-cluster variance or silhouette scores.

This practical application highlighted the value of clustering in uncovering patterns and informing decisions. Clustering is a versatile tool with wide applications—from customer segmentation to product classification and beyond. A solid understanding of its strengths and limitations is essential for every data scientist.

13.6 Exercises

The exercises are grouped into two categories: conceptual questions and practical exercises using the *redWines* dataset, applying clustering techniques to real-world data.

Conceptual questions

1. What is clustering, and how does it differ from classification?
2. Explain the concept of similarity measures in clustering. What is the most commonly used distance metric for numerical data?

3. Why is clustering considered an unsupervised learning method?
4. What are some real-world applications of clustering? Name at least three.
5. Define the terms *intra-cluster similarity* and *inter-cluster separation*. Why are these important in clustering?
6. How does K-means clustering determine which data points belong to a cluster?
7. Explain the role of centroids in K-means clustering.
8. What happens if the number of clusters k in K-means is chosen too small? What if it is too large?
9. What is the elbow method, and how does it help determine the optimal number of clusters?
10. Why is K-means sensitive to the initial selection of cluster centers? How does K-means++ address this issue?
11. Describe a scenario where Euclidean distance might not be an appropriate similarity measure for clustering.
12. Why do we need to normalize or scale variables before applying K-means clustering?
13. How does clustering help in dimensionality reduction and preprocessing for supervised learning?
14. What are the key assumptions of K-means clustering?
15. How does the silhouette score help evaluate the quality of clustering?
16. Compare K-means with hierarchical clustering. What are the advantages and disadvantages of each?
17. Why is K-means not suitable for non-spherical clusters?
18. What is the difference between hard clustering (e.g., K-means) and soft clustering (e.g., Gaussian Mixture Models)?
19. What are outliers, and how do they affect K-means clustering?
20. What are alternative clustering methods that are more robust to outliers than K-means?

Practical Exercises

These exercises use the *redWines* dataset from the **liver** package, which contains chemical properties of red wines and their quality scores. Your goal is

to apply clustering techniques to uncover natural groupings in the wines—without using the quality label during clustering.

Data preparation and exploratory analysis

21. Load the *redWines* dataset from the **liver** package and inspect its structure.

```
library(liver)
data(redWines)
str(redWines)
```

22. Summarize the dataset using `summary()`. Identify any missing values.
23. Check the distribution of wine quality scores in the dataset. What is the most common wine quality score?
24. Since clustering requires numerical features, remove any non-numeric columns from the dataset.
25. Apply min-max scaling to normalize all numerical variables before clustering. Why is this step necessary?

Applying K-means clustering

26. Use the elbow method to determine the optimal number of clusters for the dataset.

```
library(factoextra)
fviz_nbclust(redWines, kmeans, method = "wss")
```

27. Based on the elbow plot, choose an appropriate value of k and perform K-means clustering.
28. Visualize the clusters using a scatter plot of two numerical features.
29. Compute the silhouette score to evaluate cluster cohesion and separation.
30. Identify the centroids of the final clusters and interpret their meaning.

Interpreting the clusters

31. Assign the cluster labels to the original dataset and examine the average chemical composition of each cluster.

32. Compare the wine quality scores across clusters. Do some clusters contain higher-quality wines than others?
33. Identify which features contribute most to defining the clusters.
34. Are certain wine types (e.g., high acidity, high alcohol content) concentrated in specific clusters?
35. Experiment with different values of k and compare the clustering results. Does increasing or decreasing k improve the clustering?
36. Visualize how wine acidity and alcohol content influence cluster formation.
37. (Optional) The **liver** package also includes a *whiteWines* dataset with the same structure as *redWines*. Repeat the clustering process on this dataset—from preprocessing and elbow method to K-means application and interpretation. How do the cluster profiles differ between red and white wines?

Self-reflection

38. Reflect on your experience applying K-means clustering to the *redWines* dataset. What challenges did you encounter in interpreting the clusters, and how might you validate or refine your results if this were a real-world project? What role do domain insights (e.g., wine chemistry, customer preferences) play in making clustering results actionable?

References

- Arthur, David, and Sergei Vassilvitskii. 2006. "K-Means++: The Advantages of Careful Seeding."
- Baumer, Benjamin S, Daniel T Kaplan, and Nicholas J Horton. 2017. *Modern Data Science with r*. Chapman; Hall/CRC.
- Bayes, Thomas. 1958. *Essay Toward Solving a Problem in the Doctrine of Chances*. Biometrika Office.
- Breiman, L, JH Friedman, R Olshen, and CJ Stone. 1984. "Classification and Regression Trees."
- Gareth, James, Witten Daniela, Hastie Trevor, and Tibshirani Robert. 2013. *An Introduction to Statistical Learning: With Applications in r*. Springer.
- Grolemund, Garrett. 2014. *Hands-on Programming with r: Write Your Own Functions and Simulations*. " O'Reilly Media, Inc.".
- James, Gareth, Daniela Witten, Trevor Hastie, Robert Tibshirani, et al. 2013. *An Introduction to Statistical Learning*. Vol. 112. 1. Springer.
- Kutner, Michael H, Christopher J Nachtsheim, John Neter, and William Li. 2005. *Applied Linear Statistical Models*. 5th ed. McGraw-Hill Education.
- Lantz, Brett. 2019. *Machine Learning with r: Expert Techniques for Predictive Modeling*. Packt publishing ltd.
- Messerli, Franz H. 2012. "Chocolate Consumption, Cognitive Function, and Nobel Laureates." *N Engl J Med* 367 (16): 1562–64.
- Moro, Sérgio, Paulo Cortez, and Paulo Rita. 2014. "A Data-Driven Approach to Predict the Success of Bank Telemarketing." *Decision Support Systems* 62: 22–31.
- Sutton, Richard S, Andrew G Barto, et al. 1998. *Reinforcement Learning: An Introduction*. Vol. 1. 1. MIT press Cambridge.
- Wheelan, Charles. 2013. *Naked Statistics: Stripping the Dread from the Data*. WW Norton & Company.
- Wickham, Hadley, Garrett Grolemund, et al. 2017. *R for Data Science*. Vol. 2. O'Reilly Sebastopol, CA.

- Wolfe, Douglas A, and Grant Schneider. 2017. *Intuitive Introductory Statistics*. Springer.