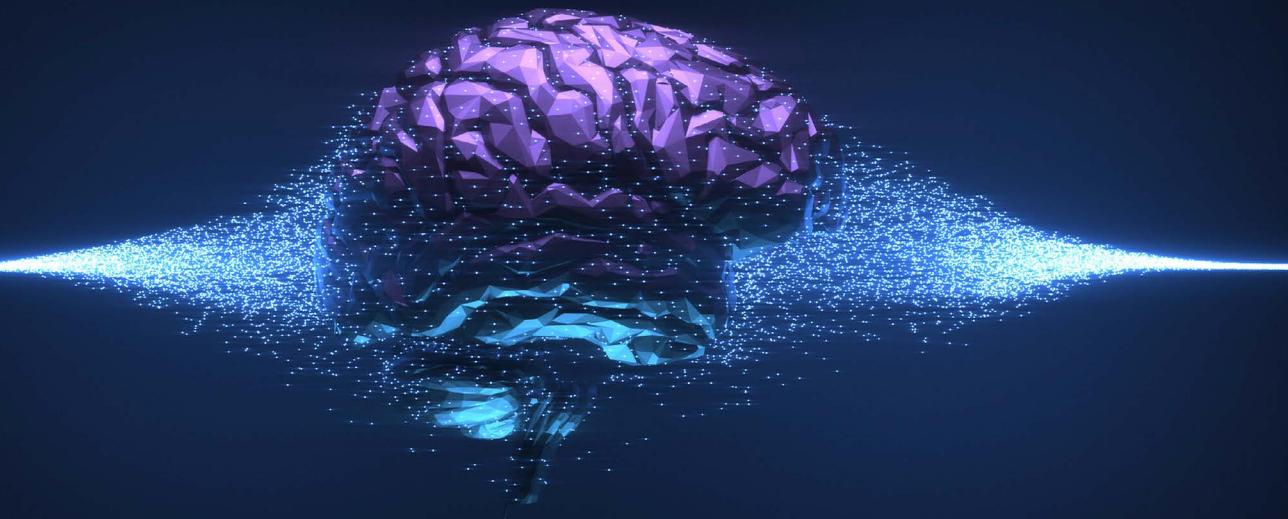


EXPERT INSIGHT

Machine Learning with R

Learn techniques for building and improving machine learning models, from data preparation to model tuning, evaluation, and working with big data

Fourth Edition



Brett Lantz

packt

saracliffel@gmail.com

Machine Learning with R

Fourth Edition

Learn techniques for building and improving machine learning models, from data preparation to model tuning, evaluation, and working with big data

Brett Lantz



BIRMINGHAM—MUMBAI

Machine Learning with R

Fourth Edition

Copyright © 2023 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Lead Senior Publishing Product Manager: Tushar Gupta

Acquisition Editor – Peer Reviews: Saby Dsilva

Project Editor: Janice Gonsalves

Content Development Editors: Bhavesh Amin and Elliot Dallow

Copy Editor: Safis Editor

Technical Editor: Karan Sonawane

Indexer: Hemangini Bari

Presentation Designer: Pranit Padwal

Developer Relations Marketing Executive: Monika Sangwan

First published: October 2013

Second edition: July 2015

Third edition: April 2019

Fourth edition: May 2023

Production reference: 1190523

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80107-132-1

www.packtpub.com

Contributors

About the author

Brett Lantz (@DataSpelunking) has spent more than 15 years using innovative data methods to understand human behavior. A sociologist by training, Brett was first captivated by machine learning while studying a large database of teenagers' social network profiles. Brett is a DataCamp instructor and has taught machine learning workshops around the world. He is known to geek out about data science applications for sports, video games, autonomous vehicles, and foreign language learning, among many other subjects, and hopes to eventually blog about such topics at dataspelunking.com.

It is hard to describe how much my world has changed since the first edition of this book was published nearly ten years ago! My sons Will and Cal were born amidst the first and second editions, respectively, and have grown alongside my career. This edition, which consumed two years of weekends, would have been impossible without the backing of my wife, Jessica. Many thanks are due also to the friends, mentors, and supporters who opened the doors that led me along this unexpected data science journey.

About the reviewer

Daniel D. Gutierrez is an independent consultant in data science through his firm AMULET Analytics. He's also a technology journalist, serving as Editor-in-Chief for [insideBIGDATA.com](#), where he enjoys keeping his finger on the pulse of this fast-paced industry. Daniel is also an educator, having taught data science, machine learning and R classes at university level for many years. He currently teaches data science for UCLA Extension. He has authored four computer industry books on database and data science technology, including his most recent title, *Machine Learning and Data Science: An Introduction to Statistical Learning Methods with R*. Daniel holds a BS in Mathematics and Computer Science from UCLA.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



Table of Contents

Preface	xvii
<hr/>	
Chapter 1: Introducing Machine Learning	1
The origins of machine learning	2
Uses and abuses of machine learning	5
Machine learning successes • 7	
The limits of machine learning • 8	
Machine learning ethics • 10	
How machines learn	14
Data storage • 16	
Abstraction • 16	
Generalization • 20	
Evaluation • 22	
Machine learning in practice	23
Types of input data • 24	
Types of machine learning algorithms • 26	
Matching input data to algorithms • 31	
Machine learning with R	32
Installing R packages • 33	
Loading and unloading R packages • 34	
Installing RStudio • 34	
Why R and why R now? • 36	
Summary	38

Chapter 2: Managing and Understanding Data	39
R data structures	40
Vectors • 40	
Factors • 43	
Lists • 44	
Data frames • 47	
Matrices and arrays • 51	
Managing data with R	52
Saving, loading, and removing R data structures • 52	
Importing and saving datasets from CSV files • 54	
Importing common dataset formats using RStudio • 56	
Exploring and understanding data	59
Exploring the structure of data • 60	
Exploring numeric features • 61	
<i>Measuring the central tendency – mean and median</i> • 62	
<i>Measuring spread – quartiles and the five-number summary</i> • 64	
<i>Visualizing numeric features – boxplots</i> • 66	
<i>Visualizing numeric features – histograms</i> • 68	
<i>Understanding numeric data – uniform and normal distributions</i> • 70	
<i>Measuring spread – variance and standard deviation</i> • 71	
Exploring categorical features • 73	
<i>Measuring the central tendency – the mode</i> • 74	
Exploring relationships between features • 76	
<i>Visualizing relationships – scatterplots</i> • 76	
<i>Examining relationships – two-way cross-tabulations</i> • 78	
Summary	82
Chapter 3: Lazy Learning – Classification Using Nearest Neighbors	83
Understanding nearest neighbor classification	84
The k-NN algorithm • 84	
<i>Measuring similarity with distance</i> • 88	

<i>Choosing an appropriate k</i> • 90	
<i>Preparing data for use with k-NN</i> • 91	
Why is the k-NN algorithm lazy? • 94	
Example – diagnosing breast cancer with the k-NN algorithm	95
Step 1 – collecting data • 96	
Step 2 – exploring and preparing the data • 96	
<i>Transformation – normalizing numeric data</i> • 98	
<i>Data preparation – creating training and test datasets</i> • 100	
Step 3 – training a model on the data • 101	
Step 4 – evaluating model performance • 103	
Step 5 – improving model performance • 104	
<i>Transformation – z-score standardization</i> • 104	
<i>Testing alternative values of k</i> • 106	
Summary	107
Chapter 4: Probabilistic Learning – Classification Using Naive Bayes	109
Understanding Naive Bayes	110
Basic concepts of Bayesian methods • 110	
<i>Understanding probability</i> • 111	
<i>Understanding joint probability</i> • 112	
<i>Computing conditional probability with Bayes' theorem</i> • 114	
The Naive Bayes algorithm • 117	
<i>Classification with Naive Bayes</i> • 118	
<i>The Laplace estimator</i> • 120	
<i>Using numeric features with Naive Bayes</i> • 122	
Example – filtering mobile phone spam with the Naive Bayes algorithm	123
Step 1 – collecting data • 124	
Step 2 – exploring and preparing the data • 125	
<i>Data preparation – cleaning and standardizing text data</i> • 126	
<i>Data preparation – splitting text documents into words</i> • 132	
<i>Data preparation – creating training and test datasets</i> • 135	

<i>Visualizing text data – word clouds</i> • 136	
<i>Data preparation – creating indicator features for frequent words</i> • 139	
Step 3 – training a model on the data • 141	
Step 4 – evaluating model performance • 143	
Step 5 – improving model performance • 144	
Summary	145
Chapter 5: Divide and Conquer – Classification Using Decision Trees and Rules	
147	
Understanding decision trees	148
Divide and conquer • 149	
The C5.0 decision tree algorithm • 153	
<i>Choosing the best split</i> • 154	
<i>Pruning the decision tree</i> • 157	
Example – identifying risky bank loans using C5.0 decision trees	158
Step 1 – collecting data • 159	
Step 2 – exploring and preparing the data • 159	
<i>Data preparation – creating random training and test datasets</i> • 161	
Step 3 – training a model on the data • 163	
Step 4 – evaluating model performance • 169	
Step 5 – improving model performance • 170	
<i>Boosting the accuracy of decision trees</i> • 170	
<i>Making some mistakes cost more than others</i> • 173	
Understanding classification rules	175
Separate and conquer • 176	
The 1R algorithm • 178	
The RIPPER algorithm • 180	
Rules from decision trees • 182	
What makes trees and rules greedy? • 183	
Example – identifying poisonous mushrooms with rule learners	185
Step 1 – collecting data • 186	

Step 2 – exploring and preparing the data • 186	
Step 3 – training a model on the data • 187	
Step 4 – evaluating model performance • 189	
Step 5 – improving model performance • 190	
Summary	194

Chapter 6: Forecasting Numeric Data – Regression Methods 197

Understanding regression	198
Simple linear regression • 200	
Ordinary least squares estimation • 203	
Correlations • 205	
Multiple linear regression • 207	
Generalized linear models and logistic regression • 212	
Example – predicting auto insurance claims costs using linear regression	218
Step 1 – collecting data • 219	
Step 2 – exploring and preparing the data • 220	
<i>Exploring relationships between features – the correlation matrix • 223</i>	
<i>Visualizing relationships between features – the scatterplot matrix • 224</i>	
Step 3 – training a model on the data • 227	
Step 4 – evaluating model performance • 230	
Step 5 – improving model performance • 232	
<i>Model specification – adding nonlinear relationships • 232</i>	
<i>Model specification – adding interaction effects • 233</i>	
<i>Putting it all together – an improved regression model • 233</i>	
<i>Making predictions with a regression model • 235</i>	
Going further – predicting insurance policyholder churn with logistic regression • 238	
Understanding regression trees and model trees	245
Adding regression to trees • 246	
Example – estimating the quality of wines with regression trees and model trees	248
Step 1 – collecting data • 249	
Step 2 – exploring and preparing the data • 250	

Step 3 – training a model on the data • 252	
<i>Visualizing decision trees</i> • 255	
Step 4 – evaluating model performance • 257	
<i>Measuring performance with the mean absolute error</i> • 257	
Step 5 – improving model performance • 259	
Summary	262
<hr/>	
Chapter 7: Black-Box Methods – Neural Networks and Support Vector Machines	
265	
<hr/>	
Understanding neural networks	266
From biological to artificial neurons • 267	
Activation functions • 269	
Network topology • 273	
<i>The number of layers</i> • 273	
<i>The direction of information travel</i> • 275	
<i>The number of nodes in each layer</i> • 277	
Training neural networks with backpropagation • 278	
Example – modeling the strength of concrete with ANNs	281
Step 1 – collecting data • 281	
Step 2 – exploring and preparing the data • 282	
Step 3 – training a model on the data • 284	
Step 4 – evaluating model performance • 287	
Step 5 – improving model performance • 288	
Understanding support vector machines	294
Classification with hyperplanes • 295	
<i>The case of linearly separable data</i> • 297	
<i>The case of nonlinearly separable data</i> • 299	
Using kernels for nonlinear spaces • 300	
Example – performing OCR with SVMs	302
Step 1 – collecting data • 303	
Step 2 – exploring and preparing the data • 304	

Step 3 – training a model on the data • 305	
Step 4 – evaluating model performance • 308	
Step 5 – improving model performance • 310	
<i>Changing the SVM kernel function</i> • 310	
<i>Identifying the best SVM cost parameter</i> • 311	
Summary	313

Chapter 8: Finding Patterns – Market Basket Analysis Using Association Rules	315
---	------------

Understanding association rules	316
The Apriori algorithm for association rule learning • 317	
Measuring rule interest – support and confidence • 319	
Building a set of rules with the Apriori principle • 320	
Example – identifying frequently purchased groceries with association rules	321
Step 1 – collecting data • 322	
Step 2 – exploring and preparing the data • 323	
<i>Data preparation – creating a sparse matrix for transaction data</i> • 324	
<i>Visualizing item support – item frequency plots</i> • 328	
<i>Visualizing the transaction data – plotting the sparse matrix</i> • 330	
Step 3 – training a model on the data • 331	
Step 4 – evaluating model performance • 335	
Step 5 – improving model performance • 339	
<i>Sorting the set of association rules</i> • 340	
<i>Taking subsets of association rules</i> • 341	
<i>Saving association rules to a file or data frame</i> • 342	
<i>Using the Eclat algorithm for greater efficiency</i> • 343	
Summary	345

Chapter 9: Finding Groups of Data – Clustering with k-means	347
--	------------

Understanding clustering	348
Clustering as a machine learning task • 348	

Clusters of clustering algorithms • 351	
The k-means clustering algorithm • 356	
<i>Using distance to assign and update clusters</i> • 357	
<i>Choosing the appropriate number of clusters</i> • 362	
Finding teen market segments using k-means clustering	364
Step 1 – collecting data • 364	
Step 2 – exploring and preparing the data • 365	
<i>Data preparation – dummy coding missing values</i> • 367	
<i>Data preparation – imputing the missing values</i> • 368	
Step 3 – training a model on the data • 370	
Step 4 – evaluating model performance • 373	
Step 5 – improving model performance • 377	
Summary	379
Chapter 10: Evaluating Model Performance	381
Measuring performance for classification	382
Understanding a classifier’s predictions • 383	
A closer look at confusion matrices • 387	
Using confusion matrices to measure performance • 389	
Beyond accuracy – other measures of performance • 391	
<i>The kappa statistic</i> • 393	
<i>The Matthews correlation coefficient</i> • 397	
<i>Sensitivity and specificity</i> • 400	
<i>Precision and recall</i> • 401	
<i>The F-measure</i> • 403	
Visualizing performance tradeoffs with ROC curves • 404	
<i>Comparing ROC curves</i> • 409	
<i>The area under the ROC curve</i> • 412	
<i>Creating ROC curves and computing AUC in R</i> • 413	
Estimating future performance	416
The holdout method • 417	

Cross-validation • 421	
Bootstrap sampling • 425	
Summary	427
<hr/>	
Chapter 11: Being Successful with Machine Learning	429
<hr/>	
What makes a successful machine learning practitioner?	430
What makes a successful machine learning model?	432
Avoiding obvious predictions • 436	
Conducting fair evaluations • 439	
Considering real-world impacts • 443	
Building trust in the model • 448	
Putting the “science” in data science	452
Using R Notebooks and R Markdown • 456	
Performing advanced data exploration • 460	
<i>Constructing a data exploration roadmap</i> • 461	
<i>Encountering outliers: a real-world pitfall</i> • 464	
<i>Example – using ggplot2 for visual data exploration</i> • 467	
Summary	480
<hr/>	
Chapter 12: Advanced Data Preparation	483
<hr/>	
Performing feature engineering	484
The role of human and machine • 485	
The impact of big data and deep learning • 489	
Feature engineering in practice	496
Hint 1: Brainstorm new features • 497	
Hint 2: Find insights hidden in text • 498	
Hint 3: Transform numeric ranges • 500	
Hint 4: Observe neighbors’ behavior • 501	
Hint 5: Utilize related rows • 503	
Hint 6: Decompose time series • 504	
Hint 7: Append external data • 509	

Exploring R’s tidyverse	511
Making tidy table structures with tibbles • 512	
Reading rectangular files faster with readr and readxl • 513	
Preparing and piping data with dplyr • 515	
Transforming text with stringr • 520	
Cleaning dates with lubridate • 526	
Summary	531
Chapter 13: Challenging Data – Too Much, Too Little, Too Complex	533
The challenge of high-dimension data	534
Applying feature selection • 536	
<i>Filter methods</i> • 538	
<i>Wrapper methods and embedded methods</i> • 539	
<i>Example – Using stepwise regression for feature selection</i> • 541	
<i>Example – Using Boruta for feature selection</i> • 545	
Performing feature extraction • 548	
<i>Understanding principal component analysis</i> • 548	
<i>Example – Using PCA to reduce highly dimensional social media data</i> • 553	
Making use of sparse data	562
Identifying sparse data • 562	
Example – Remapping sparse categorical data • 563	
Example – Binning sparse numeric data • 567	
Handling missing data	572
Understanding types of missing data • 573	
Performing missing value imputation • 575	
<i>Simple imputation with missing value indicators</i> • 576	
<i>Missing value patterns</i> • 577	
The problem of imbalanced data	579
Simple strategies for rebalancing data • 580	
Generating a synthetic balanced dataset with SMOTE • 583	
<i>Example – Applying the SMOTE algorithm in R</i> • 584	

Considering whether balanced is always better • 587	
Summary	589
<hr/>	
Chapter 14: Building Better Learners	591
<hr/>	
Tuning stock models for better performance	592
Determining the scope of hyperparameter tuning • 593	
Example – using caret for automated tuning • 598	
<i>Creating a simple tuned model</i> • 601	
<i>Customizing the tuning process</i> • 604	
Improving model performance with ensembles	609
Understanding ensemble learning • 610	
Popular ensemble-based algorithms • 613	
<i>Bagging</i> • 613	
<i>Boosting</i> • 615	
<i>Random forests</i> • 618	
<i>Gradient boosting</i> • 624	
<i>Extreme gradient boosting with XGBoost</i> • 629	
<i>Why are tree-based ensembles so popular?</i> • 636	
Stacking models for meta-learning	638
Understanding model stacking and blending • 640	
Practical methods for blending and stacking in R • 642	
Summary	645
<hr/>	
Chapter 15: Making Use of Big Data	647
<hr/>	
Practical applications of deep learning	648
Beginning with deep learning • 649	
<i>Choosing appropriate tasks for deep learning</i> • 650	
<i>The TensorFlow and Keras deep learning frameworks</i> • 653	
Understanding convolutional neural networks • 655	
<i>Transfer learning and fine tuning</i> • 658	
<i>Example – classifying images using a pre-trained CNN in R</i> • 659	

Unsupervised learning and big data	666
Representing highly dimensional concepts as embeddings • 667	
<i>Understanding word embeddings • 669</i>	
<i>Example – using word2vec for understanding text in R • 671</i>	
Visualizing highly dimensional data • 680	
<i>The limitations of using PCA for big data visualization • 681</i>	
<i>Understanding the t-SNE algorithm • 683</i>	
<i>Example – visualizing data's natural clusters with t-SNE • 686</i>	
Adapting R to handle large datasets	691
Querying data in SQL databases • 692	
<i>The tidy approach to managing database connections • 692</i>	
<i>Using a database backend for dplyr with dbplyr • 695</i>	
Doing work faster with parallel processing • 697	
<i>Measuring R's execution time • 699</i>	
<i>Enabling parallel processing in R • 699</i>	
<i>Taking advantage of parallel with foreach and doParallel • 702</i>	
<i>Training and evaluating models in parallel with caret • 704</i>	
Utilizing specialized hardware and algorithms • 705	
<i>Parallel computing with MapReduce concepts via Apache Spark • 706</i>	
<i>Learning via distributed and scalable algorithms with H2O • 708</i>	
<i>GPU computing • 710</i>	
Summary	712
Other Books You May Enjoy	715
<hr/> Index	721

Preface

Machine learning, at its core, describes algorithms that transform data into actionable intelligence. This fact makes machine learning well suited to the present-day era of big data. Without machine learning, it would be nearly impossible to make sense of the massive streams of information that are now all around us.

The cross-platform, zero-cost statistical programming environment called R provides an ideal pathway to start applying machine learning. R offers powerful but easy-to-learn tools that can assist you with finding insights in your own data.

By combining hands-on case studies with the essential theory needed to understand how these algorithms work, this book delivers all the knowledge you need to get started with machine learning and to apply its methods to your own projects.

Who this book is for

This book is aimed at people in applied fields—business analysts, social scientists, and others—who have access to data and hope to use it for action. Perhaps you already know a bit about machine learning, but have never used R; or, perhaps you know a little about R, but are new to machine learning. Maybe you are completely new to both! In any case, this book will get you up and running quickly. It would be helpful to have a bit of familiarity with basic math and programming concepts, but no prior experience is required. All you need is curiosity.

What this book covers

Chapter 1, Introducing Machine Learning, presents the terminology and concepts that define and distinguish machine learners, as well as a method for matching a learning task with the appropriate algorithm.

Chapter 2, Managing and Understanding Data, provides an opportunity to get your hands dirty working with data in R. Essential data structures and procedures used for loading, exploring, and understanding data are discussed.

Chapter 3, Lazy Learning – Classification Using Nearest Neighbors, teaches you how to understand and apply a simple yet powerful machine learning algorithm to your first real-world task: identifying malignant samples of cancer.

Chapter 4, Probabilistic Learning – Classification Using Naive Bayes, reveals the essential concepts of probability that are used in cutting-edge spam filtering systems. You'll learn the basics of text mining in the process of building your own spam filter.

Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules, explores a couple of learning algorithms whose predictions are not only accurate, but also easily explained. We'll apply these methods to tasks where transparency is important.

Chapter 6, Forecasting Numeric Data – Regression Methods, introduces machine learning algorithms used for making numeric predictions. As these techniques are heavily embedded in the field of statistics, you will also learn the essential metrics needed to make sense of numeric relationships.

Chapter 7, Black-Box Methods – Neural Networks and Support Vector Machines, covers two complex but powerful machine learning algorithms. Though the math may appear intimidating, we will work through examples that illustrate their inner workings in simple terms.

Chapter 8, Finding Patterns – Market Basket Analysis Using Association Rules, exposes the algorithm used in the recommendation systems employed by many retailers. If you've ever wondered how retailers seem to know your purchasing habits better than you know yourself, this chapter will reveal their secrets.

Chapter 9, Finding Groups of Data – Clustering with k-means, is devoted to a procedure that locates clusters of related items. We'll utilize this algorithm to identify profiles within an online community.

Chapter 10, Evaluating Model Performance, provides information on measuring the success of a machine learning project and obtaining a reliable estimate of the learner's performance on future data.

Chapter 11, Being Successful with Machine Learning, describes the common pitfalls faced when transitioning from textbook datasets to real world machine learning problems, as well as the tools, strategies, and soft skills needed to combat these issues.

Chapter 12, Advanced Data Preparation, introduces the set of “tidyverse” packages, which help wrangle large datasets to extract meaningful information to aid the machine learning process.

Chapter 13, Challenging Data – Too Much, Too Little, Too Complex, considers solutions to a common set of problems that can derail a machine learning project when the useful information is lost within a massive dataset, much like a needle in a haystack.

Chapter 14, Building Better Learners, reveals the methods employed by the teams at the top of machine learning competition leaderboards. If you have a competitive streak, or simply want to get the most out of your data, you’ll need to add these techniques to your repertoire.

Chapter 15, Making Use of Big Data, explores the frontiers of machine learning. From working with extremely large datasets to making R work faster, the topics covered will help you push the boundaries of what is possible with R, and even allow you to utilize the sophisticated tools developed by large organizations like Google for image recognition and understanding text data.

What you need for this book

The examples in this book were tested with R version 4.2.2 on Microsoft Windows, Mac OS X, and Linux, although they are likely to work with any recent version of R. R can be downloaded at no cost at <https://cran.r-project.org/>.

The RStudio interface, which is described in more detail in *Chapter 1, Introducing Machine Learning*, is a highly recommended add-on for R that greatly enhances the user experience. The RStudio Open Source Edition is available free of charge from Posit (<https://www.posit.co/>) alongside a paid RStudio Pro Edition that offers priority support and additional features for commercial organizations.

Download the example code files

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Machine-Learning-with-R-Fourth-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/TZ7os>.

Conventions used

Code in text: function names, filenames, file extensions, and R package names are shown as follows: “The `knn()` function in the `class` package provides a standard, classic implementation of the k-NN algorithm.”

R user input and output is written as follows:

```
> reg(y = launch$distress_ct, x = launch[2:4])
```

	estimate
Intercept	3.527093383
temperature	-0.051385940
field_check_pressure	0.001757009
flight_num	0.014292843

New terms and important words are shown in **bold**. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: “In RStudio, a new file can be created using the **File** menu, selecting **New File**, and choosing the **R Notebook** option.”



References to additional resources or background information appear like this.



Helpful tips and important caveats appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com, and mention the book’s title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

Share your thoughts

Once you've read *Machine Learning with R - Fourth Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/978-1-80107-132-1>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Introducing Machine Learning

If science-fiction stories are to be believed, the invention of artificial intelligence inevitably leads to apocalyptic wars between machines and their makers. The stories begin with today's reality: computers being taught to play simple games like tic-tac-toe and to automate routine tasks. As the stories go, machines are later given control of traffic lights and communications, followed by military drones and missiles. The machines' evolution takes an ominous turn once the computers become sentient and learn how to teach themselves. Having no more need for human programmers, humankind is then "deleted."

Thankfully, at the time of writing, machines still require user input.

Though your impressions of machine learning may be colored by these mass-media depictions, today's algorithms have little danger of becoming self-aware. The goal of today's machine learning is not to create an artificial brain, but rather to assist us with making sense of and acting on the world's rapidly accumulating data stores.

Putting popular misconceptions aside, by the end of this chapter, you will gain a more nuanced understanding of machine learning. You will also be introduced to the fundamental concepts that define and differentiate the most common machine learning approaches. You will learn:

- The origins, applications, ethics, and pitfalls of machine learning
- How computers transform data into knowledge and action
- The steps needed to match a machine learning algorithm with your data

The field of machine learning provides a set of algorithms that transform data into actionable knowledge. Keep reading to see how easy it is to use R to start applying machine learning to real-world problems.

The origins of machine learning

Beginning at birth, we are inundated with data. Our body's sensors—the eyes, ears, nose, tongue, and nerves—are continually assailed with raw data that our brain translates into sights, sounds, smells, tastes, and textures. Using language, we can share these experiences with others.

Since the advent of written language, humans have recorded their observations. Hunters monitored the movement of animal herds; early astronomers recorded the alignment of planets and stars; and cities recorded tax payments, births, and deaths. Today, such observations, and many more, are increasingly automated and recorded systematically in ever-growing computerized databases.

The invention of electronic sensors has additionally contributed to an explosion in the volume and richness of recorded data. Specialized sensors, such as cameras, microphones, chemical noses, electronic tongues, and pressure sensors mimic the human ability to see, hear, smell, taste, and feel. These sensors process the data far differently than a human being would. Unlike a human's limited and subjective attention, an electronic sensor never takes a break and has no emotions to skew its perception.



Although sensors are not clouded by subjectivity, they do not necessarily report a single, definitive depiction of reality. Some have an inherent measurement error due to hardware limitations. Others are limited by their scope. A black-and-white photograph provides a different depiction of its subject than one shot in color. Similarly, a microscope provides a far different depiction of reality than a telescope.

Between databases and sensors, many aspects of our lives are recorded. Governments, businesses, and individuals are recording and reporting information, from the monumental to the mundane. Weather sensors obtain temperature and pressure data; surveillance cameras watch sidewalks and subway tunnels; and all manner of electronic behaviors are monitored: transactions, communications, social media relationships, and many others.

This deluge of data has led some to state that we have entered an era of **big data**, but this may be a bit of a misnomer. Human beings have always been surrounded by large amounts of data—one would need only to look to the sky and attempt to count its stars to discover a virtually endless supply. What makes the current era unique is that we have vast amounts of *recorded* data, much of which can be directly accessed by computers. Larger and more interesting datasets are increasingly accessible at the tips of our fingers, only a web search away. This wealth of information has the potential to inform action, given a systematic way of making sense of it all.

The field of study dedicated to the development of computer algorithms for transforming data into intelligent action is known as **machine learning**. This field originated in an environment where the available data, statistical methods, and computing power rapidly and simultaneously evolved. Growth in the volume of data necessitated additional computing power, which in turn spurred the development of statistical methods for analyzing large datasets. This created a cycle of advancement, allowing even larger and more interesting data to be collected, and enabled today's environment in which endless streams of data are available on virtually any topic.

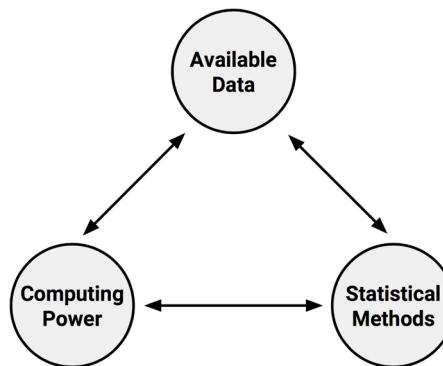


Figure 1.1: The cycle of advancement that enabled machine learning

A closely related sibling of machine learning, **data mining**, is concerned with the generation of novel insight from large databases. As the term implies, data mining involves a systematic hunt for nuggets of actionable intelligence. Although there is some disagreement over how widely machine learning and data mining overlap, one point of distinction is that machine learning focuses on teaching computers how to use data to solve a problem, while data mining focuses on teaching computers to identify patterns that humans then use to solve a problem.

Virtually all data mining involves the use of machine learning, but not all machine learning requires data mining. For example, you might apply machine learning to data mine automobile traffic data for patterns related to accident rates. On the other hand, if the computer is learning how to identify traffic signs, this is purely machine learning without data mining.



The phrase “data mining” is also sometimes used as a pejorative to describe the deceptive practice of cherry-picking data to support a theory.

Machine learning is also intertwined with the field of **artificial intelligence (AI)**, which is a nebulous discipline and, depending on whom you might ask, is simply machine learning with a strong marketing spin or a distinct field of study altogether. A cynic might suggest that the field of AI tends to exaggerate its importance such as by calling a simple predictive model an “AI bot,” while an AI proponent may point out that the field tends to tackle the most challenging learning tasks while aiming for human-level performance. The truth is somewhere in between.

Just as machine learning itself depends on statistical methods, artificial intelligence depends a great deal on machine learning, but the business contexts and applications tend to differ. The table that follows highlights some differentiators among traditional statistics, machine learning, and artificial intelligence; however, keep in mind that the lines between the three disciplines are often less rigid than they may appear.

	Traditional statistics	Machine learning	Artificial intelligence
Application	Hypothesis testing and insight	Prediction and knowledge generation	Automation
Success criterion	Greater understanding	Ability to intervene before things happen	Efficiency and cost savings
Success metric	Statistical significance	Trustworthiness of predictions	Return on investment (ROI)
Input data size	Smaller data	Medium data	Bigger data
Implementation	Reports and presentations for knowledge sharing	Scoring databases or interventions in business practices	Custom applications and automated processes

In this formulation, machine learning sits firmly at the intersection of human and computer partnership, whereas traditional statistics relies primarily on the human to drive insights and AI seeks to minimize human involvement as much as possible. Learning how to maximize the human-machine partnership and apply learning algorithms to real-world problems is the focus of this book. Understanding the use cases and limitations of machine learning is an important starting point in this journey.

Uses and abuses of machine learning

Most people have heard of Deep Blue, the chess-playing computer that in 1997 was the first to win a game against a world champion. Another famous computer, Watson, defeated two human opponents on the television trivia game show *Jeopardy!* in 2011. Based on these stunning accomplishments, some have speculated that computer intelligence will replace workers in information technology occupations, just as automobiles replaced horses and machines replaced workers in fields and assembly lines. Recently, these fears have become more pronounced as artificial intelligence-based algorithms, such as GPT-3 and DALL-E 2 from the OpenAI research group (<https://openai.com/>), have reached impressive milestones and are proving that computers are capable of writing text and creating artwork that is virtually indistinguishable from that produced by humans. Ultimately, this may lead to massive shifts in occupations like marketing, customer support, illustration, and so on, as creativity is outsourced to machines that can produce endless streams of material more cheaply than the former employees.

In this case, humans may still be necessary because the truth is that even as machines reach such impressive milestones, they are still relatively limited in their ability to thoroughly understand a problem, or understand how the work is going to be applied toward a real-world goal. Learning algorithms are pure intellectual horsepower without direction. A computer may be more capable than a human of finding subtle patterns in large databases, but it still needs a human to motivate the analysis and turn the result into meaningful action. In most cases, the human will determine whether the machine's output is valuable and will help the machine avoid creating a limitless supply of nonsense.



Without completely discounting the achievements of Deep Blue and Watson, it is important to note that neither is even as intelligent as a typical five-year-old. For more on why “comparing smarts is a slippery business,” see the *Popular Science* article FYI, *Which Computer Is Smarter, Watson Or Deep Blue?*, by Will Grunewald, 2012: <https://www.popsci.com/science/article/2012-12/fyi-which-computer-smarter-watson-or-deep-blue>.

Machines are not good at asking questions or even knowing what questions to ask. They are much better at answering them, provided the question is stated in a way that the computer can comprehend. Present-day machine learning algorithms partner with people much like a bloodhound works with its trainer: the dog's sense of smell may be many times stronger than its master's, but without being carefully directed, the hound may end up chasing its tail.

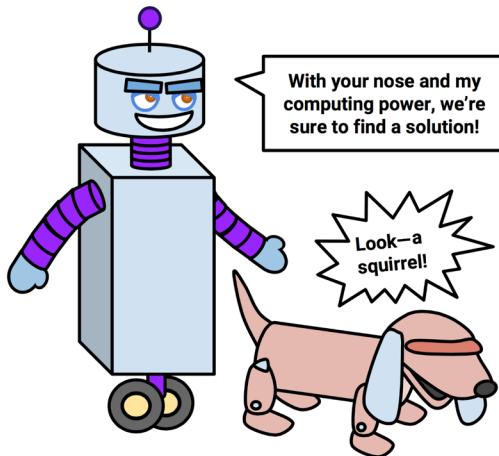


Figure 1.2: Machine learning algorithms are powerful tools that require careful direction

In the worst-case scenario, if machine learning were implemented carelessly, it might lead to what controversial tech billionaire Elon Musk provocatively called “summoning the demon.” This perspective suggests that we may be unleashing forces outside our control, despite the hubristic sense that we will be able to reign them in when needed. Given the power of artificial intelligence to automate processes and react to changing conditions much faster and more objectively than humans, there may come a point at which Pandora’s box has been opened and it is difficult or impossible to return to the old ways of life where humans are in control. As Musk describes:



“If AI has a goal and humanity just happens to be in the way, it will destroy humanity as a matter of course without even thinking about it. No hard feelings... It’s just like, if we’re building a road and an anthill just happens to be in the way, we don’t hate ants, we’re just building a road, and so, goodbye anthill.”

While this may seem to be a bleak portrayal, it is still the realm of far-future science fiction, as you will soon learn when reading about the present day’s state-of-the-art machine learning successes.

However, Musk's warning does help emphasize the importance of understanding the likelihood of machine learning and AI being a double-edged sword. For all of its benefits, there are some places where it still has room for improvement, and some situations where it may do more harm than good. If machine learning practitioners cannot be trusted to act ethically, it may be necessary for governments to intervene to prevent the greatest harm to society.



For more on Musk's fears of "summoning the demon" see the following 2018 article from CNBC: <https://www.cnbc.com/2018/04/06/elon-musk-warns-ai-could-create-immortal-dictator-in-documentary.html>.

Machine learning successes

Machine learning is most successful when it augments the specialized knowledge of a subject-matter expert rather than replacing the expert altogether. It works with medical doctors at the forefront of the fight to eradicate cancer; assists engineers with efforts to create smarter homes and automobiles; helps social scientists and economists build better societies; and provides business and marketing professionals with valuable insights. Toward these ends, it is employed in countless scientific laboratories, hospitals, companies, and governmental organizations. Any effort that generates or aggregates data likely employs at least one machine learning algorithm to help make sense of it.

Though it is impossible to list every successful application of machine learning, a selection of prominent examples is as follows:

- Identification of unwanted spam messages in email
- Segmentation of customer behavior for targeted advertising
- Forecasts of weather behavior and long-term climate changes
- Preemptive interventions for customers likely to churn (stop purchasing)
- Reduction of fraudulent credit card transactions
- Actuarial estimates of financial damage from storms and natural disasters
- Prediction of and influence over election outcomes
- Development of algorithms for auto-piloting drones and self-driving cars
- Optimization of energy use in homes and office buildings
- Projection of areas where criminal activity is most likely
- Discovery of genetic sequences useful for precision medicine

By the end of this book, you will understand the basic machine learning algorithms that are employed to teach computers to perform these tasks. For now, it suffices to say that no matter the context, the fundamental machine learning process is the same. In every task, an algorithm takes data and identifies patterns that form the basis for further action.

The limits of machine learning

Although machine learning is used widely and has tremendous potential, it is important to understand its limits. The algorithms used today—even those on the cutting edge of artificial intelligence—emulate a relatively limited subset of the capabilities of the human brain. They offer little flexibility to extrapolate outside of strict parameters and know no common sense. Considering this, one should be extremely careful to recognize exactly what an algorithm has learned before setting it loose in the real world.

Without a lifetime of past experiences to build upon, computers are limited in their ability to make simple inferences about logical next steps. Consider the banner advertisements on websites, which are served according to patterns learned by data mining the browsing history of millions of users. Based on this data, someone who views websites selling mattresses is interested in buying a mattress and should therefore see advertisements for mattresses. The problem is that this becomes a never-ending cycle in which, even after a mattress has been purchased, additional mattress advertisements are shown, rather than advertisements for pillows and bed sheets.

Many people are familiar with the deficiencies of machine learning’s ability to understand or translate language, or to recognize speech and handwriting. Perhaps the earliest example of this type of failure is in a 1994 episode of the television show *The Simpsons*, which showed a parody of the Apple Newton tablet. In its time, the Newton was known for its state-of-the-art handwriting recognition. Unfortunately for Apple, it would occasionally fail to great effect. The television episode illustrated this through a sequence in which a bully’s note to “Beat up Martin” was misinterpreted by the Newton as “Eat up Martha.”

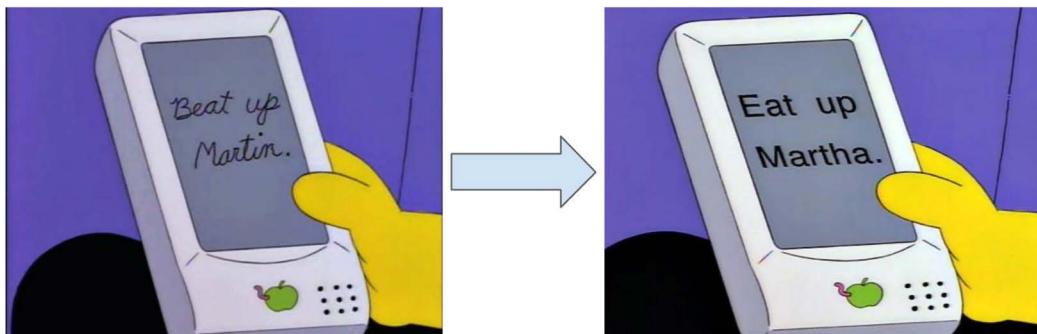


Figure 1.3: Screen captures from *Lisa on Ice*, *The Simpsons*, 20th Century Fox (1994)

Machine language processing has improved enough in the time since the Apple Newton that Google, Apple, and Microsoft are all confident in their ability to offer voice-activated virtual concierge services, such as Google Assistant, Siri, and Cortana. Still, these services routinely struggle to answer relatively simple questions. Furthermore, online translation services sometimes misinterpret sentences that a toddler would readily understand, and the predictive text feature on many devices has led to humorous “autocomplete fail” websites that illustrate computers’ ability to understand basic language but completely misunderstand context.

Some of these mistakes are to be expected. Language is complicated, with multiple layers of text and subtext, and even human beings sometimes misunderstand context. Although machine learning is rapidly improving at language processing, and current state-of-the-art algorithms like GPT-3 are quite good in comparison to prior generations, machines still make mistakes that are obvious to humans that know where to look. These predictable shortcomings illustrate the important fact that machine learning is only as good as the data it has learned from. If context is not explicit in the input data, then just like a human, the computer will have to make its best guess from its set of past experiences. However, the computer’s past experiences are usually much more limited than the human’s.

Machine learning ethics

At its core, machine learning is simply a tool that assists us with making sense of the world's complex data. Like any tool, it can be used for good or evil. Machine learning goes wrong mostly when it is applied so broadly, or so callously, that humans are treated as lab rats, automata, or mindless consumers. A process that may seem harmless can lead to unintended consequences when automated by an emotionless computer. For this reason, those using machine learning or data mining would be remiss not to at least briefly consider the ethical implications of the art.

Due to the relative youth of machine learning as a discipline and the speed at which it is progressing, the associated legal issues and social norms are often quite uncertain, and constantly in flux. Caution should be exercised when obtaining or analyzing data in order to avoid breaking laws, violating terms of service or data use agreements, or abusing the trust or violating the privacy of customers or the public. The informal corporate motto of Google, an organization that collects perhaps more data on individuals than any other, was at one time, "don't be evil." While this seems clear enough, it may not be sufficient. A better approach may be to follow the *Hippocratic Oath*, a medical principle that states, "above all, do no harm." Following the principle of "do no harm" may have helped avoid recent scandals at Facebook and other companies, such as the Cambridge Analytica controversy, which alleged that social media data was being used to manipulate elections.

Retailers routinely use machine learning for advertising, targeted promotions, inventory management, or the layout of items in a store. Many have equipped checkout lanes with devices that print coupons for promotions based on a customer's buying history. In exchange for a bit of personal data, the customer receives discounts on the specific products they want to buy. At first, this may appear relatively harmless, but consider what happens when this practice is taken a bit further.

One possibly apocryphal tale concerns a large retailer in the United States that employed machine learning to identify expectant mothers for coupon mailings. The retailer hoped that if these mothers-to-be received substantial discounts, they would become loyal customers who would later purchase profitable items such as diapers, baby formula, and toys. Equipped with machine learning methods, the retailer identified items in the customer purchase history that could be used to predict with a high degree of certainty not only whether a woman was pregnant, but also the approximate timing for when the baby was due.

After the retailer used this data for a promotional mailing, an angry man contacted the chain and demanded to know why his young daughter received coupons for maternity items. He was furious that the retailer seemed to be encouraging teenage pregnancy! As the story goes, when the retail chain called to offer an apology, it was the father who ultimately apologized after confronting his daughter and discovering that she was indeed pregnant!



For more detail on how retailers use machine learning to identify pregnancies, see the *New York Times Magazine* article titled *How Companies Learn Your Secrets*, by Charles Duhigg, 2012: <https://www.nytimes.com/2012/02/19/magazine/shopping-habits.html>.

Whether the story was completely true or not, the lesson learned from the preceding tale is that common sense should be applied before blindly applying the results of a machine learning analysis. This is particularly true in cases where sensitive information, such as health data, is concerned. With a bit more care, the retailer could have foreseen this scenario and used greater discretion when choosing how to reveal the pregnancy status its machine learning analysis had discovered. Unfortunately, as history tends to repeat itself, social media companies have been under fire recently for targeting expectant mothers with advertisements for baby products even after these mothers experience the tragedy of a miscarriage.

Because machine learning algorithms are developed with historical data, computers may learn some unfortunate behaviors of human societies. Sadly, this sometimes includes perpetuating race or gender discrimination and reinforcing negative stereotypes. For example, researchers found that Google's online advertising service was more likely to show ads for high-paying jobs to men than women and was more likely to display ads for criminal background checks to black people than white people. Although the machine may have correctly learned that men once held jobs that were not offered to most women, it is not desirable to have the algorithm perpetuate such injustices. Instead, it may be necessary to teach the machine to reflect society not as it currently is, but how it ought to be.



Sometimes, algorithms that are specifically designed with the intention of being "content-neutral" eventually come to reflect undesirable beliefs or ideologies. In one egregious case, a Twitter chatbot service developed by Microsoft was quickly taken offline after it began spreading Nazi and anti-feminist propaganda, which it may have learned from so-called "trolls" posting inflammatory content on internet forums and chat rooms. In another case, an algorithm created to reflect an objective conception of human beauty sparked controversy when it favored almost exclusively white people. Imagine the consequences if this had been applied to facial recognition software for criminal activity!

For more information about the real-world consequences of machine learning and discrimination see the *Harvard Business Review* article *Addressing the Biases Plaguing Algorithms*, by Michael Li, 2019: <https://hbr.org/2019/05/addressing-the-biases-plaguing-algorithms>.

To limit the ability of algorithms to discriminate illegally, certain jurisdictions have well-intentioned laws that prevent the use of racial, ethnic, religious, or other protected class data for business reasons. However, excluding this data from a project may not be enough because machine learning algorithms can still inadvertently learn to discriminate. If a certain segment of people tends to live in a certain region, buys a certain product, or otherwise behaves in a way that uniquely identifies them as a group, machine learning algorithms can infer the protected information from other factors. In such cases, you may need to *completely* de-identify these people by excluding any *potentially* identifying data in addition to the already-protected statuses.

In a recent example of this type of alleged algorithmic bias, the Apple credit card, which debuted in 2019, was almost immediately accused of providing substantially higher credit limits to men than to women—sometimes by 10 to 20 times the amount—even for spouses with joint assets and similar credit histories. Although Apple and the issuing bank, Goldman Sachs, denied that gender bias was at play and confirmed that no legally protected applicant characteristics were used in the algorithm, this did not slow speculation that perhaps some bias crept in unintentionally. It did not help matters that for competitive reasons, Apple and Goldman Sachs chose to keep the details of the algorithm secret, which led people to assume the worst. If the systematic bias allegations were untrue, being able to explain what was truly happening and exactly how the decisions were made might have alleviated much of the outrage. A potential worst-case scenario would have occurred if Apple and Goldman Sachs were investigated yet couldn't explain the result to regulators, due to the algorithm's complexity!



The Apple credit card fiasco is described in a 2019 BBC article, *Apple's 'sexist' credit card investigated by US regulator*: <https://www.bbc.com/news/business-50365609>.

Apart from the legal consequences, customers may feel uncomfortable or become upset if aspects of their lives they consider private are made public. The challenge is that privacy expectations differ across people and contexts. To illustrate this fact, imagine driving by someone's house and incidentally glancing through the window. This is unlikely to offend most people. In contrast, using a camera to take a picture from across the street is likely to make most feel uncomfortable; walking up to the house and pressing a face against the glass to peer inside is likely to anger virtually everybody. Although all three of these scenarios are arguably using “public” information, two of the three cross a line that will upset most people. In much the same way, it is possible to go too far with the use of data and cross a threshold that many will see as inconsiderate at best and creepy at worst.

Just as computing hardware and statistical methods kicked off the big data era, these methods also unlocked a **post-privacy era** in which many aspects of our lives that were once private are now public, or available to the public at a price. Even prior to the big data era, it would have been possible to learn a great deal about someone by observing public information. Watching their comings and goings may reveal information about their occupation or leisure activity, and a quick glance at their trash and recycling bins may reveal what they eat, drink, and read. A private investigator could learn even more with a bit of focused digging and observation. Companies applying machine learning methods to large datasets are essentially acting as large-scale private investigators, and while they claim to be working on anonymized datasets, many still argue that the companies have gone too far with their digital surveillance.

In recent years, some high-profile web applications have experienced a mass exodus of users who felt exploited when the applications' terms of service agreements changed, or their data was used for purposes beyond what the users had originally intended. The fact that privacy expectations differ by context, age cohort, and locale adds complexity to deciding the appropriate use of personal data. It would be wise to consider the cultural implications of your work before you begin on your project, in addition to being aware of ever-more-restrictive regulations such as the European Union's **General Data Protection Regulation (GDPR)** and the inevitable policies that will follow in its footsteps.



The fact that you *can* use data for a particular end does not always mean that you *should*.

Finally, it is important to note that as machine learning algorithms become progressively more important to our everyday lives, there are greater incentives for nefarious actors to work to exploit them. Sometimes, attackers simply want to disrupt algorithms for laughs or notoriety—such as “Google bombing,” the crowdsourced method of tricking Google’s algorithms to highly rank a desired page. Other times, the effects are more dramatic. A timely example of this is the recent wave of so-called fake news and election meddling, propagated via the manipulation of advertising and recommendation algorithms that target people according to their personality. To avoid giving such control to outsiders, when building machine learning systems, it is crucial to consider how they may be influenced by a determined individual or crowd.



Social media scholar danah boyd (styled lowercase) presented a keynote at the Strata Data Conference 2017 in New York City that discussed the importance of hardening machine learning algorithms against attackers. For a recap, refer to <https://points.datasociety.net/your-data-is-being-manipulated-a7e31a83577b>.

The consequences of malicious attacks on machine learning algorithms can also be deadly. Researchers have shown that by creating an “adversarial attack” that subtly distorts a road sign with carefully chosen graffiti, an attacker might cause an autonomous vehicle to misinterpret a stop sign, potentially resulting in a fatal crash. Even in the absence of ill intent, software bugs and human errors have already led to fatal accidents in autonomous vehicle technology from Uber and Tesla. With such examples in mind, it is of the utmost importance and ethical concern that machine learning practitioners should worry about how their algorithms will be used and abused in the real world.

How machines learn

A formal definition of machine learning, attributed to computer scientist Tom M. Mitchell, states that a machine learns whenever it utilizes its experience such that its performance improves on similar experiences in the future. Although this definition makes sense intuitively, it completely ignores the process of exactly how experience is translated into future action—and, of course, learning is always easier said than done!

Where human brains are naturally capable of learning from birth, the conditions necessary for computers to learn must be made explicit by the programmer hoping to utilize machine learning methods. For this reason, although it is not strictly necessary to understand the theoretical basis for learning, having a strong theoretical foundation helps the practitioner to understand, distinguish, and implement machine learning algorithms.



As you relate machine learning to human learning, you may find yourself examining your own mind in a different light.

Regardless of whether the learner is a human or a machine, the basic learning process is the same. It can be divided into four interrelated components:

- **Data storage** utilizes observation, memory, and recall to provide a factual basis for further reasoning.

- **Abstraction** involves the translation of stored data into broader representations and concepts.
- **Generalization** uses abstracted data to create knowledge and inferences that drive action in new contexts.
- **Evaluation** provides a feedback mechanism to measure the utility of learned knowledge and inform potential improvements.

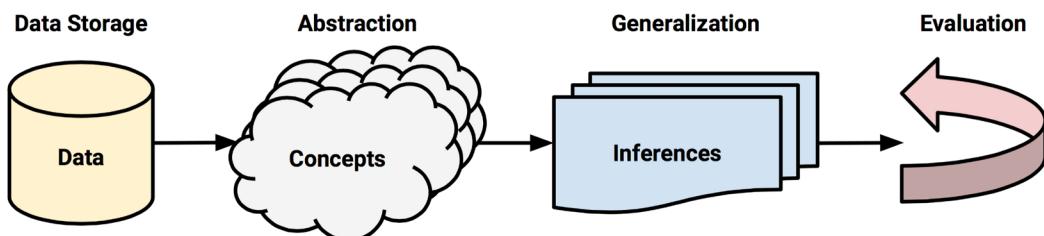


Figure 1.4: The four steps in the learning process

Although the learning process has been conceptualized here as four distinct components, they are merely organized this way for illustrative purposes. In reality, the entire learning process is inextricably linked. In human beings, the process occurs subconsciously. We recollect, deduce, induct, and intuit within the confines of our mind's eye, and because this process is hidden, any differences from person to person are attributed to a vague notion of subjectivity. In contrast, computers make these processes explicit, and because the entire process is transparent, the learned knowledge can be examined, transferred, utilized for future action, and treated as a data “science.”

The **data science** buzzword suggests a relationship between the data, the machine, and the people who guide the learning process. The term’s growing use in job descriptions and academic degree programs reflects its operationalization as a field of study concerned with both statistical and computational theory, as well as the technological infrastructure enabling machine learning and its applications. The field often asks its practitioners to be compelling storytellers, balancing an audacity in the use of data with the limitations of what one may infer and forecast from it. To be a strong data scientist, therefore, requires a strong understanding of how the learning algorithms work in the context of a business application, as we will discuss in greater detail in *Chapter 11, Being Successful with Machine Learning*.

Data storage

All learning begins with data. Humans and computers alike utilize **data storage** as a foundation for more advanced reasoning. In a human being, this consists of a brain that uses electrochemical signals in a network of biological cells to store and process observations for short- and long-term future recall. Computers have similar capabilities of short- and long-term recall using hard disk drives, flash memory, and **random-access memory (RAM)** in combination with a **central processing unit (CPU)**.

It may seem obvious, but the ability to store and retrieve data alone is insufficient for learning. Stored data is merely ones and zeros on a disk. It is a collection of memories, meaningless without a broader context. Without a higher level of understanding, knowledge is purely recall, limited to what has been seen before and nothing else.

To better understand the nuances of this idea, it may help to think about the last time you studied for a difficult test, perhaps for a university final exam or a career certification. Did you wish for an eidetic (photographic) memory? If so, you may be disappointed to know that perfect recall would unlikely be of much assistance. Even if you could memorize material perfectly, this rote learning would provide no benefit without knowing the exact questions and answers that would appear on the exam. Otherwise, you would need to memorize answers to every question that could *conceivably* be asked, on a subject in which there is likely to be an infinite number of questions. Obviously, this is an unsustainable strategy.

Instead, a better approach is to spend time selectively and memorize a relatively small set of representative ideas, while developing an understanding of how the ideas relate and apply to unforeseen circumstances. In this way, important broader patterns are identified, rather than you memorizing every detail, nuance, and potential application.

Abstraction

This work of assigning a broader meaning to stored data occurs during the **abstraction** process, in which raw data comes to represent a wider, more abstract concept or idea. This type of connection, say between an object and its representation, is exemplified by the famous René Magritte painting *The Treachery of Images*.



Figure 1.5: “This is not a pipe.” Source: <http://collections.lacma.org/node/239578>

The painting depicts a tobacco pipe with the caption *Ceci n'est pas une pipe* (“This is not a pipe”). The point Magritte was illustrating is that a representation of a pipe is not truly a pipe. Yet, despite the fact that the pipe is not real, anybody viewing the painting easily recognizes it as a pipe. This suggests that observers can connect the *picture* of a pipe to the *idea* of a pipe, to a memory of a *physical* pipe that can be held in the hand. Abstracted connections like this are the basis of **knowledge representation**, the formation of logical structures that assist with turning raw sensory information into meaningful insight.

Bringing this concept full circle, knowledge representation is what allows artificial intelligence-based tools like Midjourney (<https://www.midjourney.com>) to paint, virtually, in the style of René Magritte. The following image was generated entirely by artificial intelligence based on the algorithm’s understanding of concepts like “robot,” “pipe,” and “smoking.” If he were alive yet today, Magritte himself might find it surreal that his own surrealist work, which challenged human conceptions of reality and the connections between images and ideas, is now incorporated into the minds of computers and, in a roundabout way, is connecting machines’ ideas and images to reality. Machines learned what a pipe is, in part, by viewing images of pipes in artwork like Magritte’s.



Figure 1.6: “Am I a pipe?” image created by the Midjourney AI with the prompt of “robot smoking a pipe in the style of a René Magritte painting”

To reify the process of knowledge representation within an algorithm, the computer summarizes stored raw data using a **model**, an explicit description of the patterns within the data. Just like Magritte’s pipe, the model representation takes on a life beyond the raw data. It represents an idea greater than the sum of its parts.

There are many different types of models. You may already be familiar with some. Examples include:

- Mathematical equations
- Relational diagrams, such as trees and graphs
- Logical if/else rules
- Groupings of data known as clusters

The choice of model is typically not left up to the machine. Instead, the learning task and the type of data on hand inform model selection. Later in this chapter, we will discuss in more detail the methods for choosing the appropriate model type.

Fitting a model to a dataset is known as **training**. When the model has been trained, the data has been transformed into an abstract form that summarizes the original information. The fact that this step is called “training” rather than “learning” reveals a couple of interesting aspects of the process. First, note that the process of learning does not end with data abstraction—the learner must still generalize and evaluate its training. Second, the word “training” better connotes the fact that the human teacher trains the machine student to use the data toward a specific end.

The distinction between training and learning is subtle but important. The computer doesn’t “learn” a model, because this would imply that there is a single correct model to be learned. Of course, the computer must learn *something* about the data in order to complete its training, but it has some freedom in how or what exactly it learns. When training the learner using a given dataset, each learner finds its own way to model the data and identify the patterns that will be useful for the given task.

It is important to note that a learned model does not itself provide new data, yet it does result in new knowledge. How can this be? The answer is that imposing an assumed structure on the underlying data gives insight into the unseen. It supposes a new concept that describes a way in which existing data elements may be related.

Take, for instance, the discovery of gravity. By fitting equations to observational data, Sir Isaac Newton inferred the concept of gravity, but the force we now know as gravity was always present. It simply wasn’t recognized until Newton expressed it as an abstract concept that relates some data to other data—specifically, by becoming the g term in a model that explains observations of falling objects. The observations of the distance an object falls in various periods of time can be related to each other, via a simple model that applies a constant force of gravity to the object per unit of time.

Observations → Data → Model



Distance	Time
4.9m	1s
19.6m	2s
44.1m	3s
78.5m	4s

$$g = 9.8m/s^2$$

Figure 1.7: Models are abstractions that explain observed data

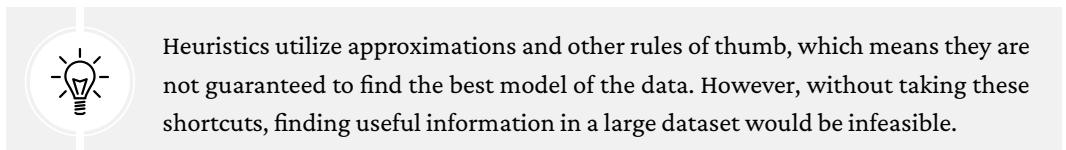
Most models will not result in the development of theories that shake up scientific thought for centuries. Still, your abstraction might result in the discovery of important, but previously unseen, patterns and relationships among data. A model trained on genomic data might find several genes that when combined are responsible for the onset of diabetes, a bank might discover a seemingly innocuous type of transaction that systematically appears prior to fraudulent activity, or a psychologist might identify a combination of personality characteristics indicating a new disorder. These underlying patterns were always present, but by presenting the information in a different format, a new idea is conceptualized.

Generalization

The third step in the learning process is to use the abstracted knowledge for future action. However, among the countless underlying patterns that may be identified during the abstraction process and the myriad ways to model those patterns, some patterns will be more useful than others. Unless the production of abstractions is limited to the useful set, the learner will be stuck where it started, with a large pool of information but no actionable insight.

Formally, the term **generalization** is defined as the process of turning abstracted knowledge into a form that can be utilized for future action on tasks that are similar, but not identical, to those the learner has seen before. It acts as a search through the entire set of models (that is, the theories or inferences) that *could* be established from the data during training. If you can imagine a hypothetical set containing every possible way the data might be abstracted, generalization involves the reduction of this set into a smaller and more manageable set of important findings.

In generalization, the learner is tasked with limiting the patterns it discovers to only those that will be most relevant to its future tasks. Normally, it is not feasible to reduce the number of patterns by examining them one by one and ranking them by future value. Instead, machine learning algorithms generally employ shortcuts that reduce the search space more quickly. To this end, the algorithm will employ **heuristics**, which are educated guesses about where to find the most useful inferences. Heuristics are routinely used by human beings to quickly generalize experience to new scenarios. If you have ever utilized your gut instinct to make a snap decision prior to fully evaluating your circumstances, you were intuitively using mental heuristics.



Heuristics utilize approximations and other rules of thumb, which means they are not guaranteed to find the best model of the data. However, without taking these shortcuts, finding useful information in a large dataset would be infeasible.

The incredible human ability to make quick decisions often relies not on computer-like logic, but rather on emotion-guided heuristics. Sometimes, this can result in illogical conclusions. For example, more people express fear of airline travel than automobile travel, despite automobiles being statistically more dangerous. This can be explained by the availability heuristic, which is the tendency for people to estimate the likelihood of an event by how easily examples can be recalled. Accidents involving air travel are highly publicized. Being traumatic events, they are likely to be recalled very easily, whereas car accidents barely warrant a mention in the newspaper.

The folly of misapplied heuristics is not limited to human beings. The heuristics employed by machine learning algorithms also sometimes result in erroneous conclusions. The algorithm is said to have a **bias** if the conclusions are *systematically* erroneous, which implies that they are wrong in a consistent or predictable manner. For example, suppose that a machine learning algorithm learned to identify faces by finding two circles representing eyes, positioned above a straight line indicating a mouth. The algorithm might then have trouble with, or be *biased against*, faces that do not conform to its model. Faces with glasses, turned at an angle, looking sideways, or with certain skin tones might not be detected by the algorithm. Similarly, it could be *biased toward* faces with other skin tones, face shapes, or characteristics that conform to its understanding of the world.

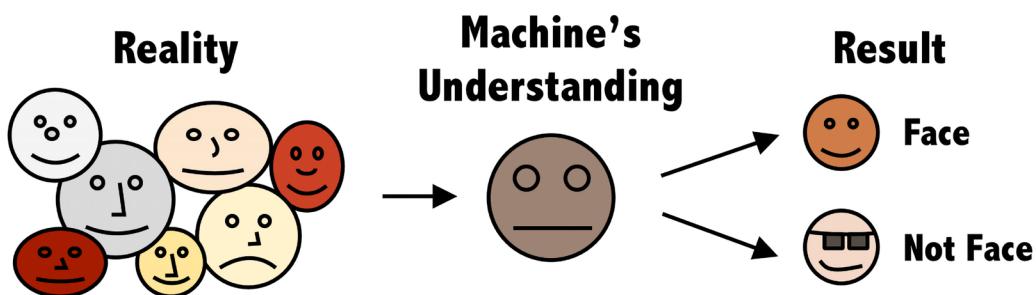


Figure 1.8: The process of generalizing a learner's experience results in a bias

In modern usage, the word “bias” has come to carry quite negative connotations. Various forms of media frequently claim to be free from bias and claim to report facts objectively, untainted by emotion. Still, consider for a moment the possibility that a little bias might be useful. Without a bit of arbitrariness, might it be a little difficult to decide among several competing choices, each with distinct strengths and weaknesses? Indeed, studies in the field of psychology have suggested that individuals born with damage to the portions of the brain responsible for emotion may be ineffectual at decision-making and might spend hours debating simple decisions, such as what color shirt to wear or where to eat lunch.

Paradoxically, bias is what blinds us from some information, while also allowing us to utilize other information for action. It is how machine learning algorithms choose among the countless ways to understand a set of data.

Evaluation

Bias is a necessary evil associated with the abstraction and generalization processes inherent in any learning task. In order to drive action in the face of limitless possibility, all learning must have a bias. Consequently, each learning strategy has weaknesses; there is no single learning algorithm to rule them all. Therefore, the final step in the learning process is to evaluate its success and to measure the learner's performance, despite its biases. The information gained in the evaluation phase can then be used to inform additional training if needed.



Once you've had success with one machine learning technique, you might be tempted to apply it to every task. It is important to resist this temptation because no machine learning approach is best for every circumstance. This fact is described by the **No Free Lunch** theorem, introduced by David Wolpert in 1996. Based on the popular adage that “there is no such thing as a free lunch,” the theorem suggests that there is a cost or trade-off in every decision that is made—a life lesson that is true more generally, even outside of machine learning! For more information, visit: <http://www.no-free-lunch.org>.

Generally, evaluation occurs after a model has been trained on an initial **training dataset**. Then, the model is evaluated on a separate **test dataset** to judge how well its characterization of the training data generalizes to new, unseen cases. It's worth noting that it is exceedingly rare for a model to perfectly generalize to every unforeseen case—mistakes are almost always inevitable.

In part, models fail to generalize perfectly due to the problem of **noise**, a term that describes unexplained or unexplainable variations in data. Noisy data is caused by seemingly random events, such as:

- Measurement error due to imprecise sensors that sometimes add or subtract a small amount from the readings
- Issues with human subjects, such as survey respondents reporting random answers to questions in order to finish more quickly
- Data quality problems, including missing, null, truncated, incorrectly coded, or corrupted values

- Phenomena that are so complex or so little understood that they impact the data in ways that appear to be random

Modeling noise is the basis of a problem called **overfitting**; because most noisy data is unexplainable by definition, attempts to explain the noise will result in models that do not generalize well to new cases. Efforts to explain the noise also typically result in more complex models that miss the true pattern the learner is trying to identify.

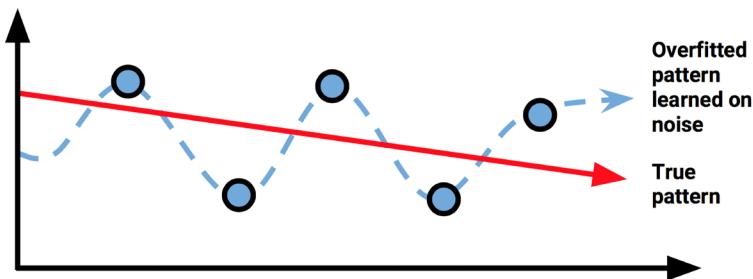


Figure 1.9: Modeling noise generally results in more complex models and misses underlying patterns

A model that performs relatively well during training but relatively poorly during evaluation is said to be **overfitted** to the training dataset because it does not generalize well to the test dataset. In practical terms, this means that it has identified a pattern in the data that is not useful for future action; the generalization process has failed. Solutions to the problem of overfitting are specific to each machine learning approach. For now, the important point is to be aware of the issue. How well the methods handle noisy data and avoid overfitting is an important point of distinction among them.

Machine learning in practice

So far, we've focused on how machine learning works in theory. To apply the learning process to real-world tasks, we'll use a five-step process. Regardless of the task, each machine learning algorithm uses the following series of steps:

1. **Data collection:** The data collection step involves gathering the learning material an algorithm will use to generate actionable knowledge. In most cases, the data will need to be combined into a single source, such as a text file, spreadsheet, or database.

2. **Data exploration and preparation:** The quality of any machine learning project is based largely on the quality of its input data. Thus, it is important to learn more about the data and its nuances. Data preparation involves fixing or cleaning so-called “messy” data, eliminating unnecessary data, and re-coding the data to conform to the learner’s expected inputs.
3. **Model training:** By the time the data has been prepared for analysis, you are likely to have a sense of what you are hoping and capable of learning from the data. The specific machine learning task chosen will inform the selection of an appropriate algorithm, and the algorithm will represent the data in the form of a model.
4. **Model evaluation:** Each machine learning model results in a biased solution to the learning problem, which means that it is important to evaluate how well the algorithm learned from its experience. Depending on the type of model used, you might be able to evaluate the accuracy of the model using a test dataset, or you may need to develop measures of performance specific to the intended application.
5. **Model improvement:** If better performance is needed, it becomes necessary to utilize more advanced strategies to augment the model’s performance. Sometimes it may be necessary to switch to a different type of model altogether. You may need to supplement your data with additional data or perform additional preparatory work, as in step 2 of this process.

After these steps have been completed, if the model appears to be performing well, it can be deployed for its intended task. You might utilize your model to provide score data for predictions (possibly in real time); for projections of financial data; to generate useful insight for marketing or research; or to automate tasks, such as mail delivery or flying aircraft. The successes and failures of the deployed model might even provide additional data to train your next-generation learner.

Types of input data

The practice of machine learning involves matching the characteristics of the input data to the biases of the available learning algorithms. Thus, before applying machine learning to real-world problems, it is important to understand the terminology that distinguishes input datasets.

The phrase **unit of observation** is used to describe the smallest entity with measured properties of interest for study. Commonly, the unit of observation is in the form of persons, objects or things, transactions, time points, geographic regions, or measurements. Sometimes, units of observation are combined to form units, such as person-years, which denote cases where the same person is tracked over multiple years, and each person-year comprises a person’s data for one year.



The unit of observation is related, but not identical, to the **unit of analysis**, which is the smallest unit from which inference is made. Although it is often the case, the observed and analyzed units are not always the same. For example, data observed from people (the unit of observation) might be used to analyze trends across countries (the unit of analysis).

Datasets that store the units of observation and their properties can be described as collections of:

- **Examples:** Instances of the unit of observation for which properties have been recorded
- **Features:** Recorded properties or attributes of examples that may be useful for learning

It is easiest to understand features and examples through real-world scenarios. For instance, to build a learning algorithm to identify spam emails, the unit of observation could be email messages, examples would be specific individual messages, and the features might consist of the words used in the messages. For a cancer detection algorithm, the unit of observation could be patients, the examples might include a random sample of cancer patients, and the features may be genomic markers from biopsied cells in addition to patient characteristics, such as weight, height, or blood pressure.

People and machines differ in the types of complexity they are suited to handle in the input data. Humans are comfortable consuming **unstructured data**, such as free-form text, pictures, or sound. They are also flexible in handling cases in which some observations have a wealth of features, while others have very little. On the other hand, computers generally require data to be **structured**, which means that each example of the phenomenon has exactly the same set of features, and these features are organized in a form that a computer may understand. Using the brute force of a machine on large, unstructured datasets usually requires a transformation of the input data to a structured form.

The following spreadsheet shows data that has been gathered in **matrix format**. In matrix data, each row is an example and each column is a feature. Here, the rows indicate examples of automobiles for sale, while the columns record the automobile's features, such as the price, mileage, color, and transmission type. Matrix format data is by far the most common form used in machine learning. As you will see in later chapters, when unstructured forms of data are encountered in specialized applications, they are ultimately transformed into a structured matrix format prior to machine learning.

The diagram illustrates a dataset matrix. At the top, a bracket labeled "features" spans across the first six columns. Below this, a large bracket on the right labeled "examples" spans vertically from the second row to the ninth row. The data is presented in a table with the following structure:

year	model	price	mileage	color	transmission
2011	SEL	21992	7413	Yellow	AUTO
2011	SEL	20995	10926	Gray	AUTO
2011	SEL	19995	7351	Silver	AUTO
2011	SEL	17809	11613	Gray	AUTO
2012	SE	17500	8367	White	MANUAL
2010	SEL	17495	25125	Silver	AUTO
2011	SEL	17000	27393	Blue	AUTO
2010	SEL	16995	21026	Silver	AUTO
2011	SES	16995	32655	Silver	AUTO

Figure 1.10: A simple dataset in matrix format describing automobiles for sale

A dataset's features may come in various forms. If a feature represents a characteristic measured in numbers, it is unsurprisingly called **numeric**. Alternatively, if a feature comprises a set of categories, the feature is called **categorical** or **nominal**. A special type of categorical feature is called **ordinal**, which designates a nominal feature with categories falling in an ordered list. One example of an ordinal feature is clothing sizes, such as small, medium, and large; another is a measurement of customer satisfaction on a scale from “not at all happy” to “somewhat happy” to “very happy.” For any given dataset, thinking about what the features represent, their types, and their units will assist with determining an appropriate machine learning algorithm for the learning task.

Types of machine learning algorithms

Machine learning algorithms are divided into categories according to their purpose. Understanding the categories of learning algorithms is an essential first step toward using data to drive the desired action.

A **predictive model** is used for tasks that involve, as the name implies, the prediction of one value using other values in a dataset. The learning algorithm attempts to discover and model the relationship between the **target** feature (the feature being predicted) and the other features. Despite the common use of the word “prediction” to imply forecasting, predictive models need not necessarily foresee events in the future. For instance, a predictive model could be used to predict past events, such as the date of a baby’s conception using the mother’s present-day hormone levels. Predictive models can also be used in real time to control traffic lights during rush hour.

Because predictive models are given clear instructions on what they need to learn and how they are intended to learn it, the process of training a predictive model is known as **supervised learning**. This supervision does not refer to human involvement, but rather to the fact that the target values provide a way for the learner to know how well it has learned the desired task. Stated more formally, given a set of data, a supervised learning algorithm attempts to optimize a function (the model) to find the best combination of feature values, resulting in a target output across all rows in the training data.

The often-used supervised machine learning task of predicting which category an example belongs to is known as **classification**. It is easy to think of potential uses for a classifier. For instance, you could predict whether:

- An email message is spam
- A person has cancer
- A football team will win or lose
- An applicant will default on a loan

In classification, the target feature to be predicted is a categorical feature known as the **class**, which is divided into categories called **levels**. A class can have two or more levels, and the levels may or may not be ordinal. Classification is so widely used in machine learning that there are many types of classification algorithms, with strengths and weaknesses suited for different types of input data. We will see examples of these later in this chapter and many times throughout this book. The first real-world application of classification appears in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, and additional examples appear in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*, and *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, among others.

Supervised learners can also be used to predict numeric data, such as income, laboratory values, test scores, or counts of items. To predict such numeric values, a common form of **numeric prediction** fits linear regression models to the input data. Although regression is not the only method for numeric prediction, it is by far the most widely used. Regression methods are widely used for forecasting, as they quantify in exact terms the association between the inputs and the target, including both the magnitude and uncertainty of the relationship. Many supervised learning algorithms can perform numeric prediction, but regression methods and numeric prediction are covered in detail in *Chapter 6, Forecasting Numeric Data – Regression Methods*.



Since it is easy to convert numbers to categories (for example, ages 13 to 19 are teenagers) and categories to numbers (for example, assign 1 to all males and 0 to all females), the boundary between classification models and numeric prediction models is not necessarily firm.

A **descriptive model** is used for tasks that would benefit from the insight gained from summarizing data in new and interesting ways. As opposed to predictive models that predict a target of interest, in a descriptive model, no single feature is of particular interest. Because there is no target to learn, the process of training a descriptive model is called **unsupervised learning**. Although it can be more difficult to think of applications for descriptive models—after all, what good is a learner that isn't learning anything in particular—they are used quite regularly for data mining.

For example, the descriptive modeling task called **pattern discovery** is used to identify useful associations within data. Pattern discovery is the goal of **market basket analysis**, which is applied to retailers' transactional purchase data. Here, retailers hope to identify items that are frequently purchased together, such that the learned information can be used to refine marketing tactics. For instance, if a retailer learns that swimsuits are commonly purchased at the same time as sunscreen, the retailer might reposition the items more closely in the store or run a promotion to "up-sell" associated items to customers. The methods needed to perform this type of analysis are included in *Chapter 8, Finding Patterns – Market Basket Analysis Using Association Rules*.



Originally used only in retail contexts, pattern discovery is now starting to be used in quite innovative ways. For instance, it can be used to detect patterns of fraudulent behavior, screen for genetic defects, or identify hotspots for criminal activity.

The descriptive modeling task of dividing a dataset into homogeneous groups is called **clustering**. This is sometimes used for **segmentation analysis**, which identifies groups of individuals with similar behavior or demographic information, in order to target them with advertising campaigns based on their shared characteristics. With this approach, a machine identifies the clusters, but human intervention is required to interpret them. For example, given a grocery store's five customer clusters, the marketing team will need to understand the differences among the groups in order to create a promotion that best suits each group. Despite this human effort, this is still less work than creating a unique appeal for each customer. This type of segmentation analysis is demonstrated on a real-world dataset in *Chapter 9, Finding Groups of Data – Clustering with k-means*.

Unsupervised learning can also be used to assist with supervised learning tasks where labeled data is unavailable or costly to obtain. A method called **semi-supervised learning** uses a small amount of labeled data in conjunction with an unsupervised learning analysis to help categorize the unlabeled records, which can then be used directly in a supervised learning model. For example, if it is expensive for a physician to label tumor samples as cancerous or non-cancerous, only a small portion of the patient records may have these labels. However, after performing unsupervised clustering on the patient data, it may be the case that the confirmed cancer and non-cancer patients fall into mostly separate groups, and thus the unlabeled records can inherit the labels of their cluster. Thus, a predictive model can be built on the complete set of data rather than the small portion that had been manually labeled. An application of semi-supervised learning is included in *Chapter 9, Finding Groups of Data – Clustering with k-means*.

An even more extreme version of this approach known as **self-supervised learning** requires no manually labeled data whatsoever; instead, it uses a two-step approach in which a sophisticated model first attempts to identify meaningful groupings among records, and the second model attempts to identify the key distinctions between the groups. This is a relatively recent innovation and is used primarily on large, unstructured data sources like audio, text, and image data. The building blocks of self-supervised learning are covered in *Chapter 7, Black-Box Methods – Neural Networks and Support Vector Machines*, and *Chapter 15, Making Use of Big Data*.

Lastly, a class of machine learning algorithms known as **meta-learners** is not tied to a specific learning task, but rather is focused on learning how to learn more effectively. Meta-learning can be beneficial for very challenging problems or when a predictive algorithm's performance needs to be as accurate as possible. All meta-learning algorithms use the result of past learning to inform additional learning. Most commonly, this encompasses algorithms that learn to work together in teams called **ensembles**. Just as complementary strengths and accumulated experiences are important factors for successful human teams, they are likewise valuable for machine learners, and ensembles are among the most powerful off-the-shelf algorithms available today. Several of the most popular ensemble learning algorithms are covered in *Chapter 14, Building Better Learners*.

A form of meta-learning involving algorithms that seem to evolve over time is called **reinforcement learning**. This technique involves a simulation in which the learner is rewarded for success or punished for failure and, over many iterations, seeks the highest cumulative reward. The algorithm improves at the desired learning task via the evolutionary pressure of rewards and punishment, applied to the simulated “offspring” of the learner that accumulates more beneficial random adaptations and jettisons the least helpful mutations over successive generations.

In contrast to supervised learning algorithms that are trained based on a labeled set of data, reinforcement learning can achieve performance even better than the human trainer on a learning task, as it is not limited to training on the data on hand. Stated differently, traditional supervised learners tend to mimic the existing data, but reinforcement learning can identify novel and unforeseen solutions to the task—sometimes to surprising or even comical results. For instance, reinforcement learners trained to play video games have discovered cheats or shortcuts to complete games like *Sonic the Hedgehog* much faster than humans, but in ways designers did not intend. Similarly, a learning algorithm trained to pilot a simulated moon lander discovered that a crash landing is faster than a gentle landing—apparently unphased by the consequences of what this would do to the fortunately hypothetical human passengers!

The reinforcement learning technique, although extremely powerful, is quite computationally expensive and is vastly different from the traditional learning methods described previously. It is generally applied to real-world cases in which the learner can perform the task quickly and repeatedly to determine its success. This means that it is less useful for predicting cancer, or business outcomes like churn and loan default, and more useful for stereotypical artificial intelligence applications like self-driving vehicles and other forms of automation, where success or failure is easily measurable, immediate, and can be simulated in a controlled environment. For such reasons, reinforcement learning is outside the scope of this book, although it is a fascinating topic to monitor closely in the future.



For more humorous stories about the unexpected solutions reinforcement learners have discovered, see Tom Simonite's article, *When Bots Teach Themselves to Cheat*, at <https://www.wired.com/story/when-bots-teach-themselves-to-cheat/>.

Clearly, much of the most exciting work being done in the field of machine learning today is in the domain of meta-learning. In addition to ensembles and reinforcement learning, the promising field of **adversarial learning** involves learning about a model's weaknesses to strengthen its future performance or harden it against malicious attacks. This may involve pitting algorithms against one another, such as building a reinforcement learning algorithm to produce fake photographs that can fool facial detection algorithms or identify transaction patterns that can bypass fraud detection. This is just one avenue of building better learning algorithms; there is also heavy investment in research and development efforts to make bigger and faster ensembles, which can model massive datasets using high-performance computers or cloud-computing environments. Starting with the base algorithms, we will touch on many of these fascinating innovations in the upcoming chapters.

Matching input data to algorithms

The following table lists the general types of machine learning algorithms covered in this book. Although this covers only a fraction of the entire set of learning algorithms, learning these methods will provide a sufficient foundation for making sense of any other methods you may encounter in the future.

Model	Learning task	Chapter
Supervised learning algorithms		
k-nearest neighbors	Classification	<i>Chapter 3</i>
Naive Bayes	Classification	<i>Chapter 4</i>
Decision trees	Classification	<i>Chapter 5</i>
Classification rule learners	Classification	<i>Chapter 5</i>
Linear regression	Numeric prediction	<i>Chapter 6</i>
Regression trees	Numeric prediction	<i>Chapter 6</i>
Model trees	Numeric prediction	<i>Chapter 6</i>
Logistic regression	Classification	<i>Chapter 6</i>
Neural networks	Dual use	<i>Chapter 7</i>
Support vector machines	Dual use	<i>Chapter 7</i>
Unsupervised learning algorithms		
Association rules	Pattern detection	<i>Chapter 8</i>
k-means clustering	Clustering	<i>Chapter 9</i>
Meta-learning algorithms		
Bagging	Dual use	<i>Chapter 14</i>
Boosting	Dual use	<i>Chapter 14</i>
Random forests	Dual use	<i>Chapter 14</i>
Gradient boosting	Dual use	<i>Chapter 14</i>

To begin applying machine learning to a real-world project, you will need to determine which of the four learning tasks your project represents: classification, numeric prediction, pattern detection, or clustering. The task will drive the choice of algorithm. For instance, if you are undertaking pattern detection, you are likely to employ association rules. Similarly, a clustering problem will likely utilize the k-means algorithm, and numeric prediction will utilize regression analysis or regression trees.

For classification, more thought is needed to match a learning problem to an appropriate classifier. In these cases, it is helpful to consider the various distinctions among the algorithms—distinctions that will only be apparent by studying each of the classifiers in depth. For instance, within classification problems, decision trees result in models that are readily understood, while the models of neural networks are notoriously difficult to interpret. If you were designing a credit scoring model, this could be an important distinction because the law often requires that the applicant must be notified about the reasons they were rejected for a loan. Even if the neural network is better at predicting loan defaults, if its predictions cannot be explained, then it is useless for this application.

To assist with algorithm selection, in every chapter the key strengths and weaknesses of each learning algorithm are listed. Although you will sometimes find that these characteristics exclude certain models from consideration, in many cases the choice of algorithm is arbitrary. When this is true, feel free to use whichever algorithm you are most comfortable with. Other times, when predictive accuracy is the primary goal, you may need to test several models and choose the one that fits best, or use a meta-learning approach that combines several different learners to utilize the strengths of each.

Machine learning with R

Many of the algorithms needed for machine learning are not included as part of the base R installation. Instead, the algorithms are available via a large community of experts who have shared their work freely. These must be installed on top of base R manually. Thanks to R's status as free open-source software, there is no additional charge for this functionality.

A collection of R functions that can be shared among users is called a **package**. Free packages exist for each of the machine learning algorithms covered in this book. In fact, this book only covers a small portion of all of R's machine learning packages.

If you are interested in the breadth of R packages, you can view a list at the **Comprehensive R Archive Network (CRAN)**, a collection of web and FTP sites located around the world to provide the most up-to-date versions of R software and packages. If you obtained the R software via download, it was most likely from CRAN. The CRAN website is available at <http://cran.r-project.org/index.html>.



If you do not already have R, the CRAN website also provides installation instructions and information on where to find help if you have trouble.

The **Packages** link on the left side of the CRAN page will take you to a page where you can browse the packages in alphabetical order or sorted by publication date. At the time of writing, a total of 18,910 packages were available—over 35 percent more than when the third edition of this book was written, nearly three times the number since the second edition, and over four times since the first edition roughly 10 years ago! Clearly, the R community has been thriving, and this trend shows no sign of slowing!

The **Task Views** link on the left side of the CRAN page provides a curated list of packages by subject area. The task view for machine learning, which lists the packages covered in this book (and many more), is available at <https://cran.r-project.org/view=MachineLearning>.

Installing R packages

Despite the vast set of available R add-ons, the package format makes installation and use a virtually effortless process. To demonstrate the use of packages, we will install and load the `gmodels` package maintained by Gregory R. Warnes, which contains a variety of functions to aid model fitting and data analysis. We'll use one of the package's functions throughout many of this book's chapters to compare model predictions to the true values. For more information on this package, see <https://cran.r-project.org/package=gmodels>.

The most direct way to install a package is via the `install.packages()` function. To install the `gmodels` package, at the R command prompt simply type:

```
> install.packages("gmodels")
```

R will then connect to CRAN and download the package in the correct format for your operating system. Many packages require additional packages to be installed before they can be used. These are called **dependencies**. By default, the installer will automatically download and install any dependencies.



The first time you install a package, R may ask you to choose a CRAN mirror. If this happens, choose the mirror residing at a location close to you. This will generally provide the fastest download speed.

The default installation options are appropriate for most systems. However, in some cases, you may want to install a package in another location. For example, if you do not have root or administrator privileges on your system, you may need to specify an alternative installation path.

This can be accomplished using the `lib` option as follows:

```
> install.packages("gmodels", lib = "/path/to/library")
```

The installation function also provides additional options for installing from a local file, installing from source, or using experimental versions. You can read about these options in the help file by using the following command:

```
> ?install.packages
```

More generally, the question mark operator can be used to obtain help on any R function. Simply type `?` before the name of the function.

Loading and unloading R packages

In order to conserve memory, R does not load every installed package by default. Instead, packages are loaded by users with the `library()` function as they are needed.



The name of this function leads some people to incorrectly use the terms “library” and “package” interchangeably. However, to be precise, a library refers to the location where packages are installed and never to a package itself.

To load the `gmodels` package installed previously, you can type the following:

```
> library(gmodels)
```

Aside from `gmodels`, there are many other R packages, which will be used in later chapters. A reminder to install these packages will be provided as these packages are needed.

To unload an R package, use the `detach()` function. For example, to unload the `gmodels` package shown previously, use the following command:

```
> detach("package:gmodels", unload = TRUE)
```

This will free up any resources used by the package.

Installing RStudio

After you have installed R from the CRAN website, it is highly recommended to also install the open-source **RStudio** desktop application. RStudio is an additional interface to R that includes functionalities that make it far easier, more convenient, and more interactive to work with R code.

The RStudio Open Source Edition is available free of charge from Posit (<https://www.posit.co/>) alongside a paid RStudio Pro Edition that offers priority support and additional features for commercial organizations.

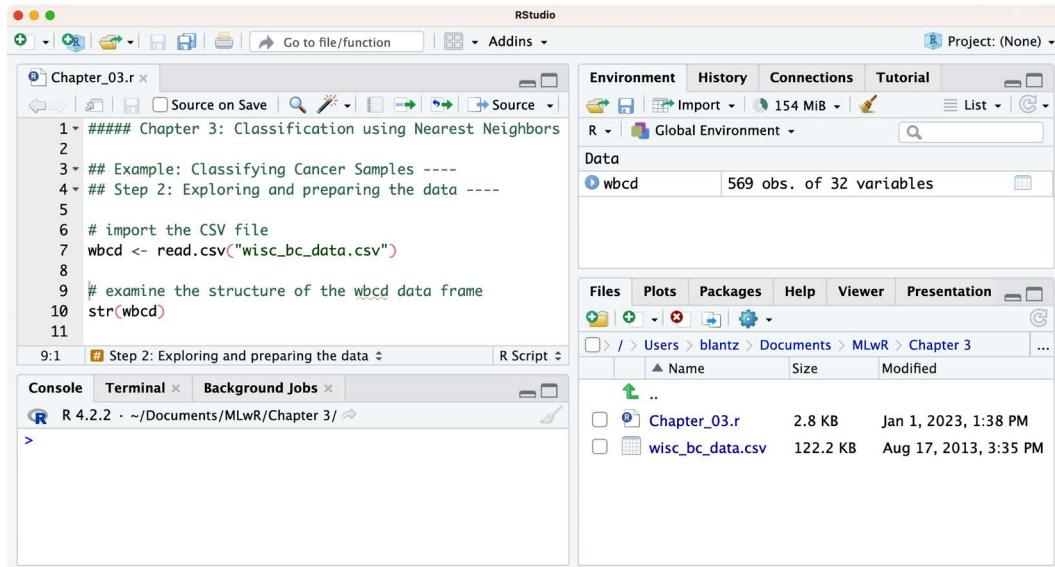


Figure 1.11: The RStudio desktop environment makes R easier and more convenient to use

The RStudio interface comprises an **integrated development environment (IDE)** including a code editor, an R command-line console, a file browser, and an R object browser. R code syntax is automatically colorized, and the code's output, plots, and graphics are displayed directly within the environment, which makes it much easier to follow long or complex statements and programs. More advanced features allow R project and package management; integration with source control or version control tools, such as Git and Subversion; database connection management; and the compilation of R output to HTML, PDF, or Microsoft Word formats. It is even possible to write Python code within RStudio!

RStudio is a key reason why R is a top choice for data scientists today. It wraps the power of R programming and its tremendous library of machine learning and statistical packages in an easy-to-use and easy-to-install development interface. It is not only ideal for learning R but can also grow with you as you learn R's more advanced functionality.



The RStudio Desktop software is developed by a company called Posit, which was formerly known itself as RStudio. The rebranding, which was somewhat surprising to fans of RStudio, occurred in late 2022 and is intended to reflect the company's broadening focus on Python as well as R. At the time of writing, the name of the desktop IDE software remains RStudio. For more information, see <https://posit.co/blog/rstudio-is-now-posit/>.

Why R and why R now?

Depending on whom you might ask, when the first edition of this book was published in 2013, R probably had a slight, if not substantial, lead over Python in user adoption for machine learning and what is now known as data science. In the time since, Python usage has grown substantially, and it would be hard to argue against the fact that Python is the new frontrunner, although the race may be closer than one might expect given the enthusiasm from Python fans supporting the new and shiny tool with greater hype.

Granted, in the past 10 years, Python has benefitted much from the rapid maturation of free add-ons like the scikit-learn machine learning framework, the pandas data structure library, the Matplotlib plotting library, and the Jupyter notebook interface, among numerous other open-source libraries that made it easier than ever to do data science in Python. Of course, these libraries merely brought Python to feature parity with what R and RStudio could do already! However, these add-ons, when combined with the comparatively fast and memory-efficient Python code—at least relative to R—may have contributed to the fact that Python is now undoubtedly the language most often taught in formal data science degree programs and has rapidly gained adoption in business domains.

Rather than indicating the impending death of R, the rise of Python may simply reflect the growth of the field. In fact, R usage is likewise growing quickly, and R and RStudio may be more popular than ever. Although students sometimes ask whether it is even worth starting with R rather than jumping straight into Python, there are still many good reasons one might choose to learn machine learning with R rather than the alternative. Note that these justifications are quite subjective and there is no single right answer for everyone, so I hesitate to even put this in writing! However, having supported this book for nearly a decade, and as someone who still uses R on a near-daily basis as part of my work for a large, international corporation, here are a few things I've noticed:

- R may be more intuitive and easier to learn for people with social science or business backgrounds (such as economics, marketing, and so on) whereas Python may make more sense to computer scientists and other types of engineers.

- R tends to be used more like a “calculator” in that you type a command, and something happens; in general, coding in Python tends to require more thought about loops and other program flow commands (this distinction is fading over time with the additional functionality in popular Python libraries).
- R uses relatively few types of data structures (those included are tailored for data analysis) and the often-used spreadsheet-like data format is a built-in data type; comparably, Python has many specialized data structures and uses libraries like NumPy or pandas for the matrix data format, each of which has their own syntax to learn.
- R and its packages may be easier to install and update than Python, in part because Python is managed by some operating systems by default, and keeping dependencies and environments separate is challenging (modern Python installation tools and package managers have addressed this while simultaneously, in some ways, making the problem worse!).
- R is typically slower and more memory-hungry than Python for data manipulation and iterating over large data structures, but if the data fits in memory, this difference is somewhat negligible; R has improved in this area (see *Chapter 12, Advanced Data Preparation*, for some ways in which R is making data preparation faster and easier), and for data that doesn’t fit in memory, there are workarounds (as described in *Chapter 15, Making Use of Big Data*), but this is admittedly one of Python’s major advantages.
- R has the support and vision of the team at Posit (formerly known as RStudio) driving innovation and making R easier and more pleasurable to use within a unified RStudio Desktop software environment; in contrast, Python’s innovations are occurring on multiple fronts, offering more “right” ways of accomplishing the same thing (for better or worse).

With any luck, the above reasons give you the confidence to begin your R journey! There’s no shame in starting here, and regardless of whether you remain with R for the long term, use it side by side with other languages like Python, or graduate to something else altogether, the foundational principles you learn in this book will transfer to whatever tools you choose. Although the book’s code is written in R, it is highly encouraged that you use the right tool for the job, whatever you feel that might be. You may find, as I have myself, that R and RStudio are your preferred tools for many real-world data science and machine learning projects—even if you still occasionally take advantage of Python’s unique strengths!

Summary

Machine learning originated at the intersection of statistics, database science, and computer science. It is a powerful tool, capable of finding actionable insight in large quantities of data. Still, as we have seen in this chapter, caution must be used in order to avoid common abuses of machine learning in the real world.

Conceptually, the learning process involves the abstraction of data into a structured representation, and the generalization of the structure into action that can be evaluated for utility. In practical terms, a machine learner uses data containing examples and features of the concept to be learned, then summarizes this data in the form of a model, which is used for predictive or descriptive purposes. These purposes can be grouped into tasks including classification, numeric prediction, pattern detection, and clustering. Among the many possible methods, machine learning algorithms are chosen based on the input data and the learning task.

R provides support for machine learning in the form of community-authored packages. These powerful tools are available to download at no cost but need to be installed before they can be used. Each chapter in this book will introduce such packages as they are needed.

In the next chapter, we will further introduce the basic R commands that are used to manage and prepare data for machine learning. Though you might be tempted to skip this step and jump directly into applications, a common rule of thumb suggests that 80 percent or more of the time spent on typical machine learning projects is devoted to the step of data preparation, also known as “data wrangling.” As a result, investing some effort into learning how to do this effectively will pay dividends for you later.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



2

Managing and Understanding Data

A key early component of any machine learning project involves managing and understanding data. Although this may not be as gratifying as building and deploying models—the stages in which you begin to see the fruits of your labor—it is unwise to ignore this important preparatory work.

Any learning algorithm is only as good as its training data, and in many cases, this data is complex, messy, and spread across multiple sources and formats. Due to this complexity, often the largest portion of effort invested in machine learning projects is spent on data preparation and exploration.

This chapter approaches data preparation in three ways. The first section discusses the basic data structures R uses to store data. You will become very familiar with these structures as you create and manipulate datasets. The second section is practical, as it covers several functions that are used for getting data in and out of R. In the third section, methods for understanding data are illustrated while exploring a real-world dataset.

By the end of this chapter, you will understand:

- How to use R’s basic data structures to store and manipulate values
- Simple functions to get data into R from common source formats
- Typical methods to understand and visualize complex data

The ways R handles data will dictate the ways you must work with data, so it is helpful to understand R’s data structures before jumping directly into data preparation. However, if you are already familiar with R programming, feel free to skip ahead to the section on data preprocessing.



All code files for this book can be found at <https://github.com/PacktPublishing/Machine-Learning-with-R-Fourth-Edition>

R data structures

There are numerous types of data structures found in programming languages, each with strengths and weaknesses suited to specific tasks. Since R is a programming language used widely for statistical data analysis, the data structures it utilizes were designed with this type of work in mind.

The R data structures used most frequently in machine learning are vectors, factors, lists, arrays, matrices, and data frames. Each is tailored to a specific data management task, which makes it important to understand how they will interact in your R project. In the sections that follow, we will review their similarities and differences.

Vectors

The fundamental R data structure is a **vector**, which stores an ordered set of values called **elements**. A vector can contain any number of elements. However, all of a vector's elements must be of the same type; for instance, a vector cannot contain both numbers and text. To determine the type of vector `v`, use the `typeof(v)` command. Note that R is a **case-sensitive** language, which means that lower-case `v` and upper-case `V` could represent two different vectors. This is also true for R's built-in functions and keywords, so be sure to always use the correct capitalization when typing R commands or expressions.

Several vector types are commonly used in machine learning: `integer` (numbers without decimals), `double` (numbers with decimals), `character` (text data, also commonly called “string” data), and `logical` (`TRUE` or `FALSE` values). Some R functions will report both `integer` and `double` vectors as `numeric`, while others distinguish between the two; generally, this distinction is unimportant. Vectors of logical values are used often in R, but notice that the `TRUE` and `FALSE` values must be written in all caps. This is slightly different from some other programming languages.

There are also two special values that are relevant to all vector types: `NA`, which indicates a *missing* value, and `NULL`, which is used to indicate the absence of *any* value. Although these two may seem to be synonymous, they are indeed slightly different. The `NA` value is a placeholder for something else and therefore has a length of one, while the `NULL` value is truly empty and has a length of zero.

It is tedious to enter large amounts of data by hand, but simple vectors can be created by using the `c()` combine function. The vector can also be given a name using the arrow `<-` operator. This is R's assignment operator, used much like the `=` assignment operator used in many other programming languages.



R also allows the use of the `=` operator for assignment, but it is considered a poor coding style according to commonly accepted style guidelines.

For example, let's construct a set of vectors containing data on three medical patients. We'll create a character vector named `subject_name` to store the three patient names, a numeric vector named `temperature` to store each patient's body temperature in degrees Fahrenheit, and a logical vector named `flu_status` to store each patient's diagnosis (TRUE if they have influenza, FALSE otherwise). As shown in the following code, the three vectors are:

```
> subject_name <- c("John Doe", "Jane Doe", "Steve Graves")
> temperature <- c(98.1, 98.6, 101.4)
> flu_status <- c(FALSE, FALSE, TRUE)
```

Values stored in R vectors retain their order. Therefore, data for each patient can be accessed using their position in the set, beginning at 1, then supplying this number inside square brackets (that is, `[` and `]`) following the name of the vector. For instance, to obtain the temperature value for patient Jane Doe, the second patient, simply type:

```
> temperature[2]
```

```
[1] 98.6
```

R offers a variety of methods to extract data from vectors. A range of values can be obtained using the colon operator. For instance, to obtain the body temperature of the second and third patients, type:

```
> temperature[2:3]
```

```
[1] 98.6 101.4
```

Items can be excluded by specifying a negative item number. To exclude the second patient's temperature data, type:

```
> temperature[-2]
```

```
[1] 98.1 101.4
```

It is also sometimes useful to specify a logical vector indicating whether each item should be included. For example, to include the first two temperature readings but exclude the third, type:

```
> temperature[c(TRUE, TRUE, FALSE)]
```

```
[1] 98.1 98.6
```

The importance of this type of operation is clearer with the realization that the result of a logical expression like `temperature > 100` is a logical vector. This expression returns TRUE or FALSE depending on whether the temperature is greater than 100 degrees Fahrenheit, which indicates a fever. Therefore, the following commands will identify the patients exhibiting a fever:

```
> fever <- temperature > 100
```

```
> subject_name[fever]
```

```
[1] "Steve Graves"
```

Alternatively, the logical expression can also be moved inside the brackets, which returns the same result in a single step:

```
> subject_name[temperature > 100]
```

```
[1] "Steve Graves"
```

As you will see shortly, the vector provides the foundation for many other R data structures and can be combined with programming expressions to complete more complex operations for selecting data and constructing new features. Therefore, knowing the various vector operations is crucial for working with data in R.

Factors

Recall from *Chapter 1, Introducing Machine Learning*, that nominal features represent a characteristic with categories of values. Although it is possible to use a character vector to store nominal data, R provides a data structure specifically for this task.

A **factor** is a special type of vector that is solely used for representing categorical or ordinal data. In the medical dataset we are building, we might use a factor to represent the patients' biological sex and record two categories: male and female.

Why use factors rather than character vectors? One advantage of factors is that the category labels are stored only once. Rather than storing MALE, MALE, FEMALE, the computer may store 1, 1, 2, which can reduce the memory needed to store the values. Additionally, many machine learning algorithms handle nominal and numeric features differently. Coding categorical features as factors allows R to treat the categorical features appropriately.



A factor should not be used for character vectors with values that don't truly fall into categories. If a vector stores mostly unique values such as names or identification codes like social security numbers, keep it as a character vector.

To create a factor from a character vector, simply apply the `factor()` function. For example:

```
> gender <- factor(c("MALE", "FEMALE", "MALE"))
> gender
```

```
[1] MALE   FEMALE MALE
Levels: FEMALE MALE
```

Notice that when the gender factor was displayed, R printed additional information about its levels. The levels comprise the set of possible categories the factor could take, in this case, MALE or FEMALE.

When we create factors, we can add additional levels that may not appear in the original data. Suppose we created another factor for blood type, as shown in the following example:

```
> blood <- factor(c("O", "AB", "A"),
  levels = c("A", "B", "AB", "O"))
> blood
```

```
[1] O  AB A
Levels: A B AB O
```

When we defined the blood factor, we specified an additional vector of four possible blood types using the `levels` parameter. As a result, even though our data includes only blood types O, AB, and A, all four types are retained with the blood factor, as the output shows. Storing the additional level allows for the possibility of adding patients with the other blood type in the future. It also ensures that if we were to create a table of blood types, we would know that type B exists, despite it not being found in our initial data.

The factor data structure also allows us to include information about the order of a nominal feature's categories, which provides a method for creating ordinal features. For example, suppose we have data on the severity of patient symptoms, coded in increasing order of severity from mild, to moderate, to severe. We indicate the presence of ordinal data by providing the factor's levels in the desired order, listed ascending from lowest to highest, and setting the `ordered` parameter to `TRUE` as shown:

```
> symptoms <- factor(c("SEVERE", "MILD", "MODERATE"),
  levels = c("MILD", "MODERATE", "SEVERE"),
  ordered = TRUE)
```

The resulting `symptoms` factor now includes information about the requested order. Unlike our prior factors, the levels of this factor are separated by `<` symbols to indicate the presence of a sequential order from `MILD` to `SEVERE`:

```
> symptoms
[1] SEVERE    MILD      MODERATE
Levels: MILD < MODERATE < SEVERE
```

A helpful feature of ordered factors is that logical tests work as you would expect. For instance, we can test whether each patient's symptoms are more severe than moderate:

```
> symptoms > "MODERATE"
[1] TRUE FALSE FALSE
```

Machine learning algorithms capable of modeling ordinal data will expect ordered factors, so be sure to code your data accordingly.

Lists

A **list** is a data structure, much like a vector, in that it is used for storing an ordered set of elements. However, where a vector requires all its elements to be the same type, a list allows different R data types to be collected. Due to this flexibility, lists are often used to store various types of input and output data and sets of configuration parameters for machine learning models.

To illustrate lists, consider the medical patient dataset we have been constructing, with data for three patients stored in six vectors. If we wanted to display all the data for the first patient, we would need to enter five R commands:

```
> subject_name[1]
[1] "John Doe"

> temperature[1]
[1] 98.1

> flu_status[1]
[1] FALSE

> gender[1]
[1] MALE
Levels: FEMALE MALE

> blood[1]
[1] 0
Levels: A B AB O

> symptoms[1]
[1] SEVERE
Levels: MILD < MODERATE < SEVERE
```

If we expect to examine the patient's data again in the future, rather than retyping these commands, a list allows us to group all the values into one object we can use repeatedly.

Similar to creating a vector with `c()`, a list is created using the `list()` function, as shown in the following example. One notable difference is that when a list is constructed, each component in the sequence should be given a name. The names are not strictly required, but allow the values to be accessed later by name rather than by numbered position and a mess of square brackets. To create a list with named components for the first patient's values, type the following:

```
> subject1 <- list(fullname = subject_name[1],
                     temperature = temperature[1],
                     flu_status = flu_status[1],
                     gender = gender[1],
```

```
blood = blood[1],  
symptoms = symptoms[1])
```

This patient's data is now collected in the `subject1` list:

```
> subject1  
  
$fullname  
[1] "John Doe"  
$temperature  
[1] 98.1  
$flu_status  
[1] FALSE  
$gender  
[1] MALE  
Levels: FEMALE MALE  
$blood  
[1] 0  
Levels: A B AB O  
$symptoms  
[1] SEVERE  
Levels: MILD < MODERATE < SEVERE
```

Note that the values are labeled with the names we specified in the preceding command. As a list retains order like a vector, its components can be accessed using numeric positions, as shown here for the `temperature` value:

```
> subject1[2]
```

```
$temperature  
[1] 98.1
```

The result of using vector-style operators on a list object is another list object, which is a subset of the original list. For example, the preceding code returned a list with a single `temperature` component. To instead return a single list item in its *native* data type, use double brackets (`[[` and `]]`) when selecting the list component. For example, the following command returns a numeric vector of length 1:

```
> subject1[[2]]
```

```
[1] 98.1
```

For clarity, it is often better to access list components by name, by appending a \$ and the component name to the list name as follows:

```
> subject1$temperature
```

```
[1] 98.1
```

Like the double-bracket notation, this returns the list component in its native data type (in this case, a numeric vector of length 1).



Accessing the value by name also ensures that the correct item is retrieved even if the order of the list elements is changed later.

It is possible to obtain several list items by specifying a vector of names. The following returns a subset of the `subject1` list, which contains only the `temperature` and `flu_status` components:

```
> subject1[c("temperature", "flu_status")]
```

```
$temperature
```

```
[1] 98.1
```

```
$flu_status
```

```
[1] FALSE
```

Entire datasets could be constructed using lists, and lists of lists. For example, you might consider creating a `subject2` and `subject3` list and grouping these into a list object named `pt_data`. However, constructing a dataset in this way is common enough that R provides a specialized data structure specifically for this task.

Data frames

By far the most important R data structure for machine learning is the **data frame**, a structure analogous to a spreadsheet or database in that it has both rows and columns of data. In R terms, a data frame can be understood as a list of vectors or factors, each having exactly the same number of values. Because the data frame is literally a list of vector-type objects, it combines aspects of both vectors and lists.

Let's create a data frame for our patient dataset. Using the patient data vectors we created previously, the `data.frame()` function combines them into a data frame:

```
> pt_data <- data.frame(subject_name, temperature,
                           flu_status, gender, blood, symptoms)
```

When displaying the `pt_data` data frame, we see that the structure is quite different from the data structures we've worked with previously:

```
> pt_data
```

	subject_name	temperature	flu_status	gender	blood	symptoms
1	John Doe	98.1	FALSE	MALE	O	SEVERE
2	Jane Doe	98.6	FALSE	FEMALE	AB	MILD
3	Steve Graves	101.4	TRUE	MALE	A	MODERATE

Compared to one-dimensional vectors, factors, and lists, a data frame has two dimensions and is displayed in a tabular format. Our data frame has one row for each patient and one column for each vector of patient measurements. In machine learning terms, the data frame's rows are the examples, and the columns are the features or attributes.

To extract entire columns (vectors) of data, we can take advantage of the fact that a data frame is simply a list of vectors. Like lists, the most direct way to extract a single element is by referring to it by name. For example, to obtain the `subject_name` vector, type:

```
> pt_data$subject_name
```

[1]	"John Doe"	"Jane Doe"	"Steve Graves"
-----	------------	------------	----------------

Like lists, a vector of names can be used to extract multiple columns from a data frame:

```
> pt_data[c("temperature", "flu_status")]
```

	temperature	flu_status
1	98.1	FALSE
2	98.6	FALSE
3	101.4	TRUE

When we request data frame columns by name, the result is a data frame containing all rows of data for the specified columns. The command `pt_data[2:3]` will also extract the `temperature` and `flu_status` columns. However, referring to the columns by name results in clear and easy-to-maintain R code, which will not break if the data frame is later reordered.

To extract specific values from the data frame, methods like those for accessing values in vectors are used. However, there is an important distinction—because the data frame is two-dimensional, both the desired rows and columns must be specified. Rows are specified first, followed by a comma, followed by the columns in a format like this: [rows, columns]. As with vectors, rows and columns are counted beginning at one.

For instance, to extract the value in the first row and second column of the patient data frame, use the following command:

```
> pt_data[1, 2]
```

```
[1] 98.1
```

If you would like more than a single row or column of data, specify vectors indicating the desired rows and columns. The following statement will pull data from the first and third rows and the second and fourth columns:

```
> pt_data[c(1, 3), c(2, 4)]
```

```
temperature gender
1      98.1    MALE
3     101.4    MALE
```

To refer to every row or every column, simply leave the row or column portion blank. For example, to extract all rows of the first column:

```
> pt_data[, 1]
```

```
[1] "John Doe"    "Jane Doe"    "Steve Graves"
```

To extract all columns for the first row:

```
> pt_data[1, ]
```

```
subject_name temperature flu_status gender blood symptoms
1      John Doe        98.1      FALSE    MALE      0    SEVERE
```

And to extract everything:

```
> pt_data[ , ]
```

```
subject_name temperature flu_status gender blood symptoms
1      John Doe        98.1      FALSE    MALE      0    SEVERE
2      Jane Doe        98.6      FALSE   FEMALE     AB     MILD
3 Steve Graves       101.4      TRUE    MALE      A MODERATE
```

Of course, columns are better accessed by name rather than position, and negative signs can be used to exclude rows or columns of data. Therefore, the output of the command:

```
> pt_data[c(1, 3), c("temperature", "gender")]
```

```
  temperature gender
1      98.1  MALE
3     101.4  MALE
```

is equivalent to:

```
> pt_data[-2, c(-1, -3, -5, -6)]
```

```
  temperature gender
1      98.1  MALE
3     101.4  MALE
```

We often need to create new columns in data frames—perhaps, for instance, as a function of existing columns. For example, we may need to convert the Fahrenheit temperature readings in the patient data frame into the Celsius scale. To do this, we simply use the assignment operator to assign the result of the conversion calculation to a new column name as follows:

```
> pt_data$temp_c <- (pt_data$temperature - 32) * (5 / 9)
```

To confirm the calculation worked, let's compare the new Celsius-based `temp_c` column to the previous Fahrenheit-scale `temperature` column:

```
> pt_data[c("temperature", "temp_c")]
```

```
  temperature   temp_c
1      98.1 36.72222
2      98.6 37.00000
3     101.4 38.55556
```

Seeing these side by side, we can confirm that the calculation has worked correctly.

As these types of operations are crucial for much of the work we will do in upcoming chapters, it is important to become very familiar with data frames. You might try practicing similar operations with the patient dataset, or even better, use data from one of your own projects—the functions to load your own data files into R will be described later in this chapter.

Matrices and arrays

In addition to data frames, R provides other structures that store values in tabular form. A **matrix** is a data structure that represents a two-dimensional table with rows and columns of data. Like vectors, R matrices can contain only one type of data, although they are most often used for mathematical operations and therefore typically store only numbers.

To create a matrix, simply supply a vector of data to the `matrix()` function, along with a parameter specifying the number of rows (`nrow`) or number of columns (`ncol`). For example, to create a 2×2 matrix storing the numbers one to four, we can use the `nrow` parameter to request the data be divided into two rows:

```
> m <- matrix(c(1, 2, 3, 4), nrow = 2)
> m
```

[,1]	[,2]
[1,]	1 3
[2,]	2 4

This is equivalent to the matrix produced using `ncol = 2`:

```
> m <- matrix(c(1, 2, 3, 4), ncol = 2)
> m
```

[,1]	[,2]
[1,]	1 3
[2,]	2 4

You will notice that R loaded the first column of the matrix first before loading the second column. This is called **column-major order**, which is R's default method for loading matrices.



To override this default setting and load a matrix by rows, set the parameter `byrow = TRUE` when creating the matrix.

To illustrate this further, let's see what happens if we add more values to the matrix. With six values, requesting two rows creates a matrix with three columns:

```
> m <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2)
> m
```

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

Requesting two columns creates a matrix with three rows:

```
> m <- matrix(c(1, 2, 3, 4, 5, 6), ncol = 2)
> m
```

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

As with data frames, values in matrices can be extracted using [row, column] notation. For instance, `m[1, 1]` will return the value 1 while `m[3, 2]` will extract 6 from the `m` matrix. Additionally, entire rows or columns can be requested:

```
> m[1, ]
```

```
[1] 1 4
```

```
> m[, 1]
```

```
[1] 1 2 3
```

Closely related to the matrix structure is the **array**, which is a multidimensional table of data. Where a matrix has rows and columns of values, an array has rows, columns, and one or more additional layers of values. Although we will occasionally use matrices in later chapters, the use of arrays is unnecessary within the scope of this book.

Managing data with R

One of the challenges faced while working with massive datasets involves gathering, preparing, and otherwise managing data from a variety of sources. Although we will cover data preparation, data cleaning, and data management in depth by working on real-world machine learning tasks in later chapters, this section highlights the basic functionality for getting data in and out of R.

Saving, loading, and removing R data structures

When you've spent a lot of time getting a data frame into the desired form, you shouldn't need to recreate your work each time you restart your R session.

To save data structures to a file that can be reloaded later or transferred to another system, the `save()` function can be used to write one or more R data structures to the location specified by the `file` parameter. R data files have an `.RData` or `.rda` extension.

Suppose you had three objects named `x`, `y`, and `z` that you would like to save to a permanent file. These might be vectors, factors, lists, data frames, or any other R object. To save them to a file named `mydata.RData`, use the following command:

```
> save(x, y, z, file = "mydata.RData")
```

The `load()` command can recreate any data structures that have been saved to an `.RData` file. To load the `mydata.RData` file created in the preceding code, simply type:

```
> load("mydata.RData")
```

This will recreate the `x`, `y`, and `z` data structures in your R environment.



Be careful what you are loading! All data structures stored in the file you are importing with the `load()` command will be added to your workspace, even if they overwrite something else you are working on.

Alternatively, the `saveRDS()` function can be used to save a single R object to a file. Although it is much like the `save()` function, a key distinction is that the corresponding `loadRDS()` function allows the object to be loaded with a different name to the original object. For this reason, `saveRDS()` may be safer to use when transferring R objects across projects, because it reduces the risk of accidentally overwriting existing objects in the R environment.

The `saveRDS()` function is especially helpful for saving machine learning model objects. Because some machine learning algorithms take a long time to train the model, saving the model to an `.rds` file can help avoid a long re-training process when a project is resumed. For example, to save a model object named `my_model` to a file named `my_model.rds`, use the following syntax:

```
> saveRDS(my_model, file = "my_model.rds")
```

To load the model, use the `readRDS()` function and assign the result an object name as follows:

```
> my_model <- readRDS("my_model.rds")
```

After you've been working in an R session for some time, you may have accumulated unused data structures. In RStudio, these objects are visible in the **Environment** tab of the interface, but it is also possible to access these objects programmatically using the listing function `ls()`, which returns a vector of all data structures currently in memory.

For example, if you've been following along with the code in this chapter, the `ls()` function returns the following:

```
> ls()
[1] "blood"        "fever"        "flu_status"    "gender"
[5] "m"            "pt_data"       "subject_name" "subject1"
[9] "symptoms"     "temperature"
```

R automatically clears all data structures from memory upon quitting the session, but for large objects, you may want to free up the memory sooner. The `remove` function `rm()` can be used for this purpose. For example, to eliminate the `m` and `subject1` objects, simply type:

```
> rm(m, subject1)
```

The `rm()` function can also be supplied with a character vector of object names to remove. This works with the `ls()` function to clear the entire R session:

```
> rm(list = ls())
```



Be very careful when executing the preceding code, as you will not be prompted before your objects are removed!

If you need to wrap up your R session in a hurry, the `save.image()` command will write your entire session to a file simply called `.RData`. By default, when quitting R or RStudio, you will be asked if you would like to create this file. R will look for this file the next time you start R, and if it exists, your session will be recreated just as you had left it.

Importing and saving datasets from CSV files

It is common for public datasets to be stored in text files. Text files can be read on virtually any computer or operating system, which makes the format nearly universal. They can also be exported and imported from and to programs such as Microsoft Excel, providing a quick and easy way to work with spreadsheet data.

A **tabular** (as in “table”) data file is structured in matrix form, such that each line of text reflects one example, and each example has the same number of features. The feature values on each line are separated by a predefined symbol known as a **delimiter**. Often, the first line of a tabular data file lists the names of the data columns. This is called a **header** line.

Perhaps the most common tabular text file format is the **comma-separated values (CSV)** file, which, as the name suggests, uses the comma as a delimiter. CSV files can be imported to and exported from many common applications. A CSV file representing the medical dataset constructed previously could be stored as:

```
subject_name,temperature,flu_status,gender,blood_type  
John Doe,98.1,TRUE,MALE,O  
Jane Doe,98.6,TRUE,FEMALE,AB  
Steve Graves,101.4,TRUE,MALE,A
```

Given a patient data file named `pt_data.csv` located in the R working directory, the `read.csv()` function can be used as follows to load the file into R:

```
> pt_data <- read.csv("pt_data.csv")
```

This will read the CSV file into a data frame titled `pt_data`. If your dataset resides outside the R working directory, the full path to the CSV file (for example, `"/path/to/mydata.csv"`) can be used when calling the `read.csv()` function.

By default, R assumes that the CSV file includes a header line listing the names of the features in the dataset. If a CSV file does not have a header, specify the option `header = FALSE` as shown in the following command, and R will assign generic feature names by numbering the columns sequentially as V1, V2, and so on:

```
> pt_data <- read.csv("pt_data.csv", header = FALSE)
```

As an important historical note, in versions of R prior to 4.0, the `read.csv()` function automatically converted all character type columns into factors due to a `stringsAsFactors` parameter that was set to `TRUE` by default. This feature was occasionally helpful, especially on the smaller and simpler datasets used in the earlier years of R. However, as datasets have become larger and more complex, this feature began to cause more problems than it solved. Now, starting with version 4.0, R sets `stringsAsFactors = FALSE` by default. If you are certain that every character column in a CSV file is truly a factor, it is possible to convert them using the following syntax:

```
> pt_data <- read.csv("pt_data.csv", stringsAsFactors = TRUE)
```

We will set `stringsAsFactors = TRUE` occasionally throughout the book, when working with datasets in which all character columns are truly factors.

Getting results data out of R can be almost as important as getting it in! To save a data frame to a CSV file, use the `write.csv()` function. For a data frame named `pt_data`, simply enter:

```
> write.csv(pt_data, file = "pt_data.csv", row.names = FALSE)
```

This will write a CSV file with the name `pt_data.csv` to the R working folder. The `row.names` parameter overrides R's default setting, which is to output row names in the CSV file. Generally, this output is unnecessary and will simply inflate the size of the resulting file.



For more sophisticated control over reading in files, note that `read.csv()` is a special case of the `read.table()` function, which can read tabular data in many different forms. This includes other delimited formats such as **tab-separated values (TSV)** and vertical bar (|) delimited files. For more detailed information on the `read.table()` family of functions, refer to the R help page using the `?read.table` command.

Importing common dataset formats using RStudio

For more complex importation scenarios, the RStudio Desktop software offers a simple interface, which will guide you through the process of writing R code that can be used to load the data into your project. Although it has always been relatively easy to load plaintext data formats like CSV, importing other common analytical data formats like Microsoft Excel (`.xls` and `.xlsx`), SAS (`.sas7bdat` and `.xpt`), SPSS (`.sav` and `.por`), and Stata (`.dta`) was once a tedious and time-consuming process, requiring knowledge of specific tricks and tools across multiple R packages. Now, the functionality is available via the **Import Dataset** command near the upper right of the RStudio interface, as shown in *Figure 2.1*:

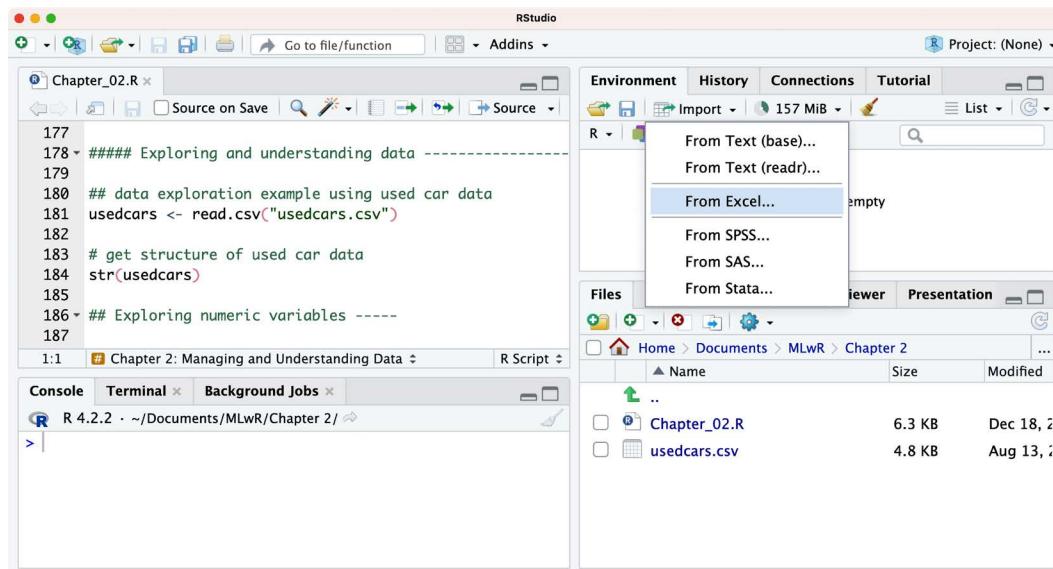


Figure 2.1: RStudio’s “Import Dataset” feature provides options to load data from a variety of common formats

Depending on the data format selected, you may be prompted to install R packages that are required for the functionality in question. Behind the scenes, these packages will translate the data format so that it can be used in R. You will then be presented with a dialog box allowing you to choose the options for the data import process and see a live preview of how the data will appear in R as these changes are made.

The following screenshot illustrates the process of importing a Microsoft Excel version of the used cars dataset using the `readxl` package (<https://readxl.tidyverse.org>), but the process is similar for any of the dataset formats:

The screenshot shows the 'Import Excel Data' dialog in RStudio. At the top, there's a 'File/URL:' input field containing the path `~/Documents/MLwR/Chapter 2/usedcars.xlsx`, with a 'Browse...' button to its right. Below this is a 'Data Preview:' section showing a table with 50 rows of car data. The columns are labeled: year (double), model (character), price (double), mileage (double), color (character), and transmission (character). The data includes years from 2011 to 2012, various models like SEL and SE, prices ranging from 17495 to 21992, mileages up to 21026, and colors like Yellow, Gray, Silver, and White. Below the preview is a note: 'Previewing first 50 entries.' Underneath the preview is an 'Import Options:' section with fields for Name (set to `usedcars`), Max Rows (empty), Skip (set to 0), and NA (empty). There are also checkboxes for 'First Row as Names' and 'Open Data Viewer'. To the right of this is a 'Code Preview:' section containing the R code:

```
library(readxl)  
usedcars <- read_excel("usedcars.xlsx")  
View(usedcars)
```

. At the bottom right of the dialog are 'Import' and 'Cancel' buttons.

Figure 2.2: The data import dialog provides a “Code Preview” that can be copy-and-pasted into your R code file

The **Code Preview** in the bottom-right of this dialog provides the R code to perform the importation with the specified options. Selecting the **Import** button will immediately execute the code; however, a better practice is to copy and paste the code into your R source code file, so that you can re-import the dataset in future sessions.

💡

The `read_excel()` function RStudio uses to load Excel data creates an R object called a “tibble” rather than a data frame. The differences are so subtle that you may not even notice! However, tibbles are an important R innovation enabling new ways to work with data frames. The tibble and its functionality are discussed in *Chapter 12, Advanced Data Preparation*.

The RStudio interface has made it easier than ever to work with data in a variety of formats, but more advanced functionality exists for working with large datasets. In particular, if you have data residing in database platforms like Microsoft SQL, MySQL, PostgreSQL, and others, it is possible to connect R to such databases to pull the data into R, or even utilize the database hardware itself to perform big data computations prior to bringing the results into R. *Chapter 15, Making Use of Big Data*, introduces these techniques and provides instructions for connecting to common databases using RStudio.

Exploring and understanding data

After collecting data and loading it into R data structures, the next step in the machine learning process involves examining the data in detail. It is during this step that you will begin to explore the data's features and examples and realize the peculiarities that make your data unique. The better you understand your data, the better you will be able to match a machine learning model to your learning problem.

The best way to learn the process of data exploration is by example. In this section, we will explore the `usedcars.csv` dataset, which contains actual data about used cars advertised for sale on a popular US website in the year 2012.



The `usedcars.csv` dataset is available for download on the Packt Publishing support page for this book. If you are following along with the examples, be sure that this file has been downloaded and saved to your R working directory.

Since the dataset is stored in CSV form, we can use the `read.csv()` function to load the data into an R data frame:

```
> usedcars <- read.csv("usedcars.csv")
```

Using the `usedcars` data frame, we will now assume the role of a data scientist who has the task of understanding the used car data. Although data exploration is a fluid process, the steps can be imagined as a sort of investigation in which questions about the data are answered. The exact questions may vary across projects, but the types of questions are always similar.

You should be able to adapt the basic steps of this investigation to any dataset you like, whether large or small.



Figure 2.3: “Do pricing algorithms get a test drive?” (image created by Midjourney AI with the prompt of “cute cartoon robot buying a used car”)

Exploring the structure of data

The first questions to ask in an investigation of a new dataset should be about how the dataset is organized. If you are fortunate, your source will provide a **data dictionary**, a document that describes the dataset’s features. In our case, the used car data does not come with this documentation, so we’ll need to create our own.

The `str()` function provides a method for displaying the structure of R objects, such as data frames, vectors, or lists. It can be used to create the basic outline for our data dictionary:

```
> str(usedcars)

'data.frame': 150 obs. of 6 variables:
 $ year      : int  2011 2011 2011 2011 ...
 $ model     : chr  "SEL" "SEL" "SEL" "SEL" ...
 $ price     : int  21992 20995 19995 17809 ...
 $ mileage   : int  7413 10926 7351 11613 ...
 $ color     : chr  "Yellow" "Gray" "Silver" "Gray" ...
 $ transmission: chr  "AUTO" "AUTO" "AUTO" "AUTO" ...
```

For such a simple command, we learn a wealth of information about the dataset. The statement **150 obs** informs us that the data includes 150 **observations**, which is just another way of saying that the dataset contains 150 rows or examples. The number of observations is often simply abbreviated as *n*.

Since we know that the data describes used cars, we can now presume that we have examples of $n = 150$ automobiles for sale.

The `6 variables` statement refers to the six features that were recorded in the data. The term **variable** is borrowed from the field of statistics, and simply means a mathematical object that can take various values—like the `x` and `y` variables you might solve for in an algebraic equation. These features, or variables, are listed by name on separate lines. Looking at the line for the feature called `color`, we note some additional details:

```
$ color      : chr  "Yellow" "Gray" "Silver" "Gray" ...
```

After the variable's name, the `chr` label tells us that the feature is the character type. In this dataset, three of the variables are character while three are noted as `int`, which refers to the integer type. Although the `usedcars` dataset includes only character and integer features, you are also likely to encounter `num`, or numeric type, when using non-integer data. Any factors would be listed as the `factor` type. Following each variable's type, R presents a sequence of the first few feature values. The values `"Yellow" "Gray" "Silver" "Gray"` are the first four values of the `color` feature.

Applying a bit of subject-area knowledge to the feature names and values allows us to make some assumptions about what the features represent. `year` could refer to the year the vehicle was manufactured, or it could specify the year the advertisement was posted. We'll have to investigate this feature later in more detail, since the four example values `(2011 2011 2011 2011)` could be used to argue for either possibility. The `model`, `price`, `mileage`, `color`, and `transmission` most likely refer to the characteristics of the car for sale.

Although our data appears to have been given meaningful names, this is not always the case. Sometimes datasets have features with nonsensical names or codes, like `V1`. In these cases, it may be necessary to do additional sleuthing to determine what a feature represents. Still, even with helpful feature names, it is always prudent to be skeptical about the provided labels. Let's investigate further.

Exploring numeric features

To investigate the numeric features in the used car data, we will employ a common set of measurements for describing values known as **summary statistics**. The `summary()` function displays several common summary statistics. Let's look at a single feature, `year`:

```
> summary(usedcars$year)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
2000	2008	2009	2009	2010	2012

Ignoring the meaning of the values for now, the fact that we see numbers such as 2000, 2008, and 2009 leads us to believe that `year` indicates the year of manufacture rather than the year the advertisement was posted, since we know the vehicle listings were obtained in 2012.

By supplying a vector of column names, we can also use the `summary()` function to obtain summary statistics for several numeric columns at the same time:

```
> summary(usedcars[c("price", "mileage")])
```

price	mileage
Min. : 3800	Min. : 4867
1st Qu.:10995	1st Qu.: 27200
Median :13592	Median : 36385
Mean :12962	Mean : 44261
3rd Qu.:14904	3rd Qu.: 55124
Max. :21992	Max. :151479

The six summary statistics provided by the `summary()` function are simple, yet powerful tools for investigating data. They can be divided into two types: measures of center and measures of spread.

Measuring the central tendency – mean and median

Measures of **central tendency** are a class of statistics used to identify a value that falls in the middle of a set of data. You are most likely already familiar with one common measure of center: the average. In common use, when something is deemed average, it falls somewhere between the extreme ends of the scale. An average student might have marks falling in the middle of their classmates'. An average weight is neither unusually light nor heavy. In general, an average item is typical and not too unlike the others in its group. You might think of it as an exemplar by which all the others are judged.

In statistics, the average is also known as the **mean**, which is a measurement defined as the sum of all values divided by the number of values. For example, to calculate the mean income in a group of three people with incomes of \$36,000, \$44,000, and \$56,000, we could type:

```
> (36000 + 44000 + 56000) / 3
```

```
[1] 45333.33
```

R also provides a `mean()` function, which calculates the mean for a vector of numbers:

```
> mean(c(36000, 44000, 56000))
```

```
[1] 45333.33
```

The mean income of this group of people is about \$45,333. Conceptually, this can be imagined as the income each person would have if the total income was divided equally across every person.

Recall that the preceding `summary()` output listed mean values for `price` and `mileage`. These values suggest that the typical used car in this dataset was listed at a price of \$12,962 and had an odometer reading of 44,261. What does this tell us about our data? We can note that because the average price is relatively low, we might expect that the dataset contains economy-class cars. Of course, the data could also include late-model luxury cars with high mileage, but the relatively low mean mileage statistic doesn't provide evidence to support this hypothesis. On the other hand, it doesn't provide evidence to ignore the possibility either. We'll need to keep this in mind as we examine the data further.

Although the mean is by far the most cited statistic for measuring the center of a dataset, it is not always the most appropriate one. Another commonly used measure of central tendency is the **median**, which is the value that occurs at the midpoint of an ordered list of values. As with the mean, R provides a `median()` function, which we can apply to our salary data as shown in the following example:

```
> median(c(36000, 44000, 56000))  
[1] 44000
```

So, because the middle value is 44000, the median income is \$44,000.



If a dataset has an even number of values, there is no middle value. In this case, the median is commonly calculated as the average of the two values at the center of the ordered list. For example, the median of the values 1, 2, 3, and 4 is 2.5.

At first glance, it seems like the median and mean are very similar measures. Certainly, the mean value of \$45,333 and the median value of \$44,000 are not very far apart. Why have two measures of central tendency? The reason relates to the fact that the mean and median are affected differently by values falling at the far ends of the range. In particular, the mean is highly sensitive to **outliers**, or values that are atypically high or low relative to the majority of data. A more nuanced take on outliers will be presented in *Chapter 11, Being Successful with Machine Learning*, but for now, we can consider them extreme values that tend to shift the mean higher or lower relative to the median, because the median is largely insensitive to the outlying values.

Recall again the reported median values in the `summary()` output for the used car dataset. Although the mean and median for price are similar (differing by approximately five percent), there is a much larger difference between the mean and median for mileage. For mileage, the mean of 44,261 is more than 20 percent larger than the median of 36,385. Since the mean is more sensitive to extreme values than the median, the fact that the mean is much higher than the median might lead us to suspect that there are some used cars with extremely high mileage values relative to the others in the dataset. To investigate this further, we'll need to add additional summary statistics to our analysis.

Measuring spread – quartiles and the five-number summary

The mean and median provide ways to quickly summarize values, but these measures of center tell us little about whether there is diversity in the measurements. To measure the diversity, we need to employ another type of summary statistics concerned with the **spread** of the data, or how tightly or loosely the values are spaced. Knowing about the spread provides a sense of the data's highs and lows, and whether most values are like or unlike the mean and median.

The **five-number summary** is a set of five statistics that roughly depict the spread of a feature's values. All five statistics are included in the `summary()` function output. Written in order, they are:

1. Minimum (Min.)
2. First quartile, or Q1 (1st Qu.)
3. Median, or Q2 (Median)
4. Third quartile, or Q3 (3rd Qu.)
5. Maximum (Max.)

As you would expect, the minimum and maximum are the most extreme feature values, indicating the smallest and largest values respectively. R provides the `min()` and `max()` functions to calculate these for a vector.

The span between the minimum and maximum value is known as the range. In R, the `range()` function returns both the minimum and maximum value:

```
> range(usedcars$price)
```

```
[1] 3800 21992
```

Combining `range()` with the difference function `diff()` allows you to compute the range statistic with a single line of code:

```
> diff(range(usedcars$price))
```

```
[1] 18192
```

One quarter of the dataset's values fall below the first quartile (Q1), and another quarter is above the third quartile (Q3). Along with the median, which is the midpoint of the data values, the quartiles divide a dataset into four portions, each containing 25 percent of the values.

Quartiles are a special case of a type of statistic called **quantiles**, which are numbers that divide data into equally sized quantities. In addition to quartiles, commonly used quantiles include **tertiles** (three parts), **quintiles** (five parts), **deciles** (10 parts), and **percentiles** (100 parts). Percentiles are often used to describe the ranking of a value; for instance, a student whose test score was ranked at the 99th percentile performed better than or equal to 99 percent of the other test takers.

The middle 50 percent of data, found between the first and third quartiles, is of particular interest because it is a simple measure of spread. The difference between Q1 and Q3 is known as the **interquartile range (IQR)**, and can be calculated with the `IQR()` function:

```
> IQR(usedcars$price)
```

```
[1] 3909.5
```

We could have also calculated this value by hand from the `summary()` output for the `usedcars$price` vector by computing $14904 - 10995 = 3909$. The small difference between our calculation and the `IQR()` output is due to the fact that R automatically rounds the `summary()` output.

The `quantile()` function provides a versatile tool for identifying quantiles for a set of values. By default, `quantile()` returns the five-number summary. Applying the function to the `usedcars$price` vector results in the same summary statistics as before:

```
> quantile(usedcars$price)
```

0%	25%	50%	75%	100%
3800.0	10995.0	13591.5	14904.5	21992.0



When computing quantiles, there are many methods for handling ties among sets of values with no single middle value. The `quantile()` function allows you to specify among nine different tie-breaking algorithms by specifying the `type` parameter. If your project requires a precisely defined quantile, it is important to read the function documentation using the `?quantile` command.

By supplying an additional `probs` parameter for a vector denoting cut points, we can obtain arbitrary quantiles, such as the 1st and 99th percentiles:

```
> quantile(usedcars$price, probs = c(0.01, 0.99))
1%      99%
5428.69 20505.00
```

The sequence function `seq()` generates vectors of evenly spaced values. This makes it easy to obtain other slices of data, such as the quintiles (five groups) shown in the following command:

```
> quantile(usedcars$price, seq(from = 0, to = 1, by = 0.20))
0%     20%     40%     60%     80%     100%
3800.0 10759.4 12993.8 13992.0 14999.0 21992.0
```

Equipped with an understanding of the five-number summary, we can re-examine the used car `summary()` output. For `price`, the minimum was \$3,800 and the maximum was \$21,992. Interestingly, the difference between the minimum and Q1 is about \$7,000, as is the difference between Q3 and the maximum; yet, the difference from Q1 to the median to Q3 is roughly \$2,000. This suggests that the lower and upper 25 percent of values are more widely dispersed than the middle 50 percent of values, which seem to be more tightly grouped around the center. We also see a similar trend with `mileage`. As you will learn later in this chapter, this pattern of spread is common enough that it has been called a “normal” distribution of data.

The spread of `mileage` also exhibits another interesting property—the difference between Q3 and the maximum is far greater than that between the minimum and Q1. In other words, the larger values are far more spread out than the smaller values. This finding helps explain why the mean value is much greater than the median. Because the mean is sensitive to extreme values, it is pulled higher, while the median stays in relatively the same place. This is an important property, which becomes more apparent when the data is presented visually.

Visualizing numeric features – boxplots

Visualizing numeric features can help diagnose data problems that might negatively affect machine learning model performance. A common visualization of the five-number summary is a **boxplot**, also known as a **box-and-whisker** plot. The boxplot displays the center and spread of a numeric variable in a format that allows you to quickly obtain a sense of its range and skew or compare it to other features.

Let's look at a boxplot for the used car price and mileage data. To obtain a boxplot for a numeric vector, we will use the `boxplot()` function. We will also specify a pair of extra parameters, `main` and `ylab`, to add a title to the figure and label the `y` axis (the vertical axis), respectively. The commands to create the `price` and `mileage` boxplots are:

```
> boxplot(usedcars$price, main = "Boxplot of Used Car Prices",
          ylab = "Price ($)")
> boxplot(usedcars$mileage, main = "Boxplot of Used Car Mileage",
          ylab = "Odometer (mi.)")
```

R will produce figures as follows:

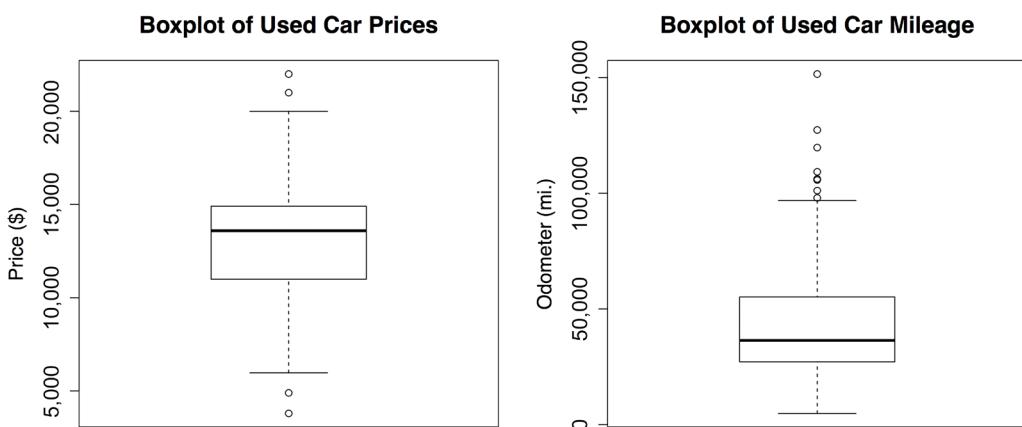


Figure 2.4: Boxplots of used car price and mileage data

A boxplot depicts the five-number summary using horizontal lines and dots. The horizontal lines forming the box in the middle of each figure represent Q1, Q2 (the median), and Q3 when reading the plot from bottom to top. The median is denoted by the dark line, which lines up with \$13,592 on the vertical axis for `price` and 36,385 mi. on the vertical axis for `mileage`.



In simple boxplots, such as those in the preceding diagram, the box width is arbitrary and does not illustrate any characteristic of the data. For more sophisticated analyses, it is possible to use the shape and size of the boxes to facilitate comparisons of the data across several groups. To learn more about such features, begin by examining the `notch` and `varwidth` options in the R `boxplot()` documentation by typing the `?boxplot` command.

The minimum and maximum values can be illustrated using whiskers that extend below and above the box; however, a widely used convention only allows the whiskers to extend to a minimum or maximum of 1.5 times the IQR below Q1 or above Q3. Any values that fall beyond this threshold are considered outliers and are denoted as circles or dots. For example, recall that the IQR for price was 3,909 with Q1 of 10,995 and Q3 of 14,904. An outlier is therefore any value that is less than $10995 - 1.5 * 3909 = 5131.5$ or greater than $14904 + 1.5 * 3909 = 20767.5$.

The price boxplot shows two outliers on both the high and low ends. On the mileage boxplot, there are no outliers on the low end and thus the bottom whisker extends to the minimum value of 4,867. On the high end, we see several outliers beyond the 100,000-mile mark. These outliers are responsible for our earlier finding, which noted that the mean value was much greater than the median.

Visualizing numeric features – histograms

A **histogram** is another way to visualize the spread of a numeric feature. It is like a boxplot in that it divides the feature values into a predefined number of portions or **bins**, which act as containers for values. Their similarities end there, however. Where a boxplot creates four portions containing the same number of values but varying in range, a histogram uses a larger number of portions of identical range and allows the bins to contain different numbers of values.

We can create a histogram for the used car price and mileage data using the `hist()` function. As we did with the boxplot, we will specify a title for the figure using the `main` parameter and label the `x` axis with the `xlab` parameter. The commands to create the histograms are:

```
> hist(usedcars$price, main = "Histogram of Used Car Prices",
       xlab = "Price ($)")
> hist(usedcars$mileage, main = "Histogram of Used Car Mileage",
       xlab = "Odometer (mi.)")
```

This produces the following diagrams:

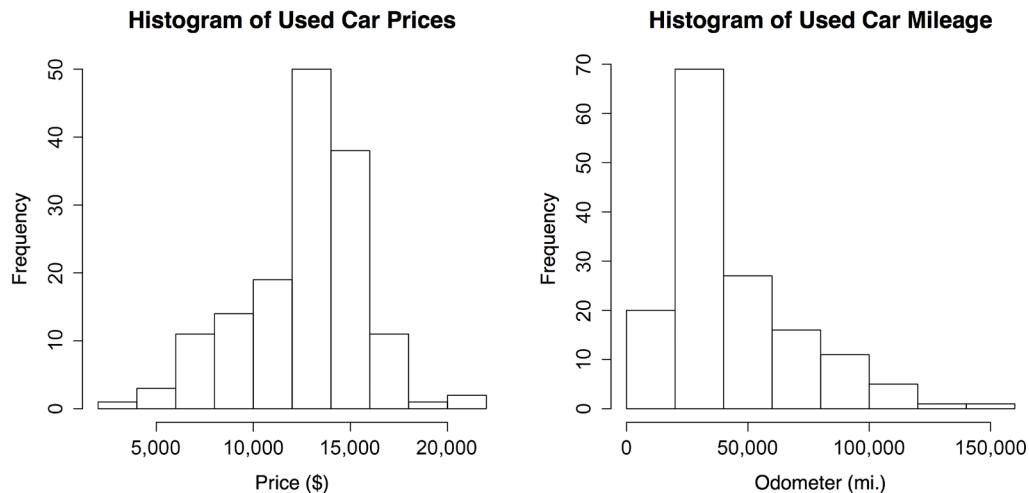


Figure 2.5: Histograms of used car price and mileage data

The histogram is composed of a series of bars with heights indicating the count, or **frequency**, of values falling within each of the equal-width bins partitioning the values. The vertical lines that separate the bars, as labeled on the horizontal axis, indicate the start and end points of the range of values falling within the bin.



You may have noticed that the preceding histograms have differing numbers of bins. This is because the `hist()` function attempts to identify the optimal number of bins for the feature's range. If you'd like to override this default, use the `breaks` parameter. Supplying an integer such as `breaks = 10` creates exactly 10 bins of equal width, while supplying a vector such as `c(5000, 10000, 15000, 20000)` creates bins that break at the specified values.

On the price histogram, each of the 10 bars spans an interval of \$2,000, beginning at \$2,000 and ending at \$22,000. The tallest bar in the center of the figure covers the range from \$12,000 to \$14,000 and has a frequency of 50. Since we know our data includes 150 cars, we know that one-third of all the cars are priced from \$12,000 to \$14,000. Nearly 90 cars—more than half—are priced from \$12,000 to \$16,000.

The mileage histogram includes eight bars representing bins of 20,000 miles each, beginning at 0 and ending at 160,000 miles. Unlike the price histogram, the tallest bar is not in the center of the data, but on the left-hand side of the diagram. The 70 cars contained in this bin have odometer readings from 20,000 to 40,000 miles.

You might also notice that the shape of the two histograms is somewhat different. It seems that the used car prices tend to be evenly divided on both sides of the middle, while the car mileages stretch further to the right.

This characteristic is known as **skew**, or more specifically right skew, because the values on the high end (right side) are far more spread out than the values on the low end (left side). As shown in the following diagram, histograms of skewed data look stretched on one of the sides:

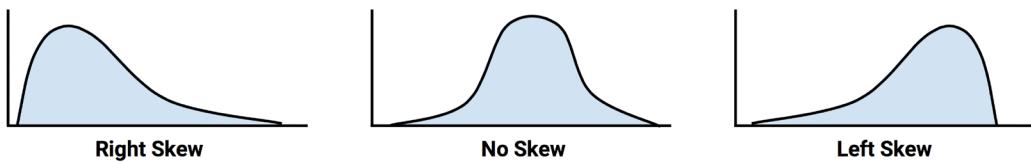


Figure 2.6: Three skew patterns visualized with idealized histograms

The ability to quickly diagnose such patterns in our data is one of the strengths of the histogram as a data exploration tool. This will become even more important as we start examining other patterns of spread in numeric data.

Understanding numeric data – uniform and normal distributions

Histograms, boxplots, and statistics describing the center and spread provide ways to examine the distribution of a feature's values. A variable's **distribution** describes how likely a value is to fall within various ranges.

If all values are equally likely to occur—say, for instance, in a dataset recording the values rolled on a fair six-sided die—the distribution is said to be uniform. A uniform distribution is easy to detect with a histogram because the bars are approximately the same height. The histogram may look something like the following diagram:

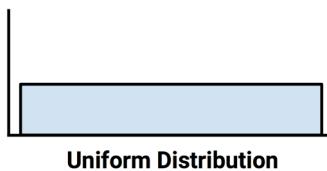


Figure 2.7: A uniform distribution visualized with an idealized histogram

It's important to note that not all random events are uniform. For instance, rolling a weighted six-sided trick die would result in some numbers coming up more often than others. While each roll of the die results in a randomly selected number, they are not equally likely.

The used car price and mileage data are also clearly not uniform, since some values are seemingly far more likely to occur than others. In fact, on the price histogram, it seems that values become less likely to occur as they are further away from both sides of the center bar, which results in a bell-shaped distribution of data. This characteristic is so common in real-world data that it is the hallmark of the so-called **normal distribution**. The stereotypical bell-shaped curve of the normal distribution is shown in the following diagram:

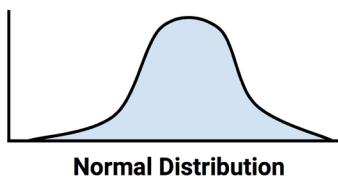


Figure 2.8: A normal distribution visualized with an idealized histogram

Although there are numerous types of non-normal distributions, many real-world phenomena generate data that can be described by the normal distribution. Therefore, the normal distribution's properties have been studied in detail.

Measuring spread – variance and standard deviation

Distributions allow us to characterize a large number of values using a smaller number of parameters. The normal distribution, which describes many types of real-world data, can be defined with just two: center and spread. The center of the normal distribution is defined by its mean value, which we have used before. The spread is measured by a statistic called the **standard deviation**.

To calculate the standard deviation, we must first obtain the **variance**, which is defined as the average of the squared differences between each value and the mean value. In mathematical notation, the variance of a set of n values in a set named x is defined by the following formula:

$$\text{Var}(X) = \sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2$$

In this formula, the Greek letter *mu* (written as μ) denotes the mean of the values, and the variance itself is denoted by the Greek letter *sigma* squared (written as σ^2).

The standard deviation is the square root of the variance, and is denoted by sigma (written as σ) as shown in the following formula:

$$\text{StdDev}(X) = \sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2}$$

In R, the `var()` and `sd()` functions save us the trouble of calculating the variance and standard deviation by hand. For example, computing the variance and standard deviation of the `price` and `mileage` vectors, we find:

```
> var(usedcars$price)
[1] 9749892
> sd(usedcars$price)
[1] 3122.482
> var(usedcars$mileage)
[1] 728033954
> sd(usedcars$mileage)
[1] 26982.1
```

When interpreting the variance, larger numbers indicate that the data is spread more widely around the mean. The standard deviation indicates, on average, how much each value differs from the mean.



If you compute these statistics by hand using the formulas in the preceding diagrams, you will obtain a slightly different result than the built-in R functions. This is because the preceding formulas use the population variance (which divides by n), while R uses the sample variance (which divides by $n - 1$). Except for very small datasets, the distinction is minor.

The standard deviation can be used to quickly estimate how extreme a given value is under the assumption that it came from a normal distribution. The **68–95–99.7 rule** states that 68 percent of values in a normal distribution fall within one standard deviation of the mean, while 95 percent and 99.7 percent of values fall within two and three standard deviations, respectively. This is illustrated in the following diagram:

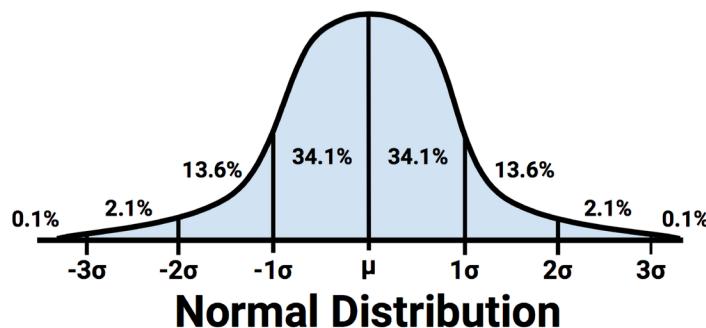


Figure 2.9: The percent of values within one, two, and three standard deviations of a normal distribution's mean

Applying this information to the used car data, we know that the mean and standard deviation of price were \$12,962 and \$3,122 respectively. Therefore, by assuming that the prices are normally distributed, approximately 68 percent of cars in our data were advertised at prices between $\$12,962 - \$3,122 = \$9,840$ and $\$12,962 + \$3,122 = \$16,804$.



Although, strictly speaking, the 68–95–99.7 rule only applies to normal distributions, the basic principle applies to almost any data; values more than three standard deviations away from the mean tend to be exceedingly rare events.

Exploring categorical features

If you recall, the used car dataset contains three categorical features: `model`, `color`, and `transmission`. Additionally, although `year` is stored as a numeric vector, each year can be imagined as a category applying to multiple cars. We might therefore consider treating it as categorical as well.

In contrast to numeric data, categorical data is typically examined using tables rather than summary statistics. A table that presents a single categorical feature is known as a **one-way table**. The `table()` function can be used to generate one-way tables for the used car data:

```
> table(usedcars$year)
```

2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012
3	1	1	1	3	2	6	11	14	42	49	16	1

```
> table(usedcars$model)
```

```
SE SEL SES
```

```
78 23 49
```

```
> table(usedcars$color)
```

Black	Blue	Gold	Gray	Green	Red	Silver	White	Yellow
35	17	1	16	5	25	32	16	3

The `table()` output lists the categories of the nominal variable and a count of the number of values falling into each category. Since we know there are 150 used cars in the dataset, we can determine that roughly one-third of all the cars were manufactured in 2010, given that $49/150 = 0.327$.

R can also perform the calculation of table proportions directly, by using the `prop.table()` command on a table produced by the `table()` function:

```
> model_table <- table(usedcars$model)
> prop.table(model_table)
```

SE	SEL	SES
0.5200000	0.1533333	0.3266667

The results of `prop.table()` can be combined with other R functions to transform the output. Suppose we would like to display the results in percentages with a single decimal place. We can do this by multiplying the proportions by 100, then using the `round()` function while specifying `digits = 1`, as shown in the following example:

```
> color_table <- table(usedcars$color)
> color_pct <- prop.table(color_table) * 100
> round(color_pct, digits = 1)
```

Black	Blue	Gold	Gray	Green	Red	Silver	White	Yellow
23.3	11.3	0.7	10.7	3.3	16.7	21.3	10.7	2.0

Although this includes the same information as the default `prop.table()` output, the changes make it easier to read. The results show that black is the most common color, with nearly a quarter (23.3 percent) of all advertised cars. Silver is a close second with 21.3 percent, and red is third with 16.7 percent.

Measuring the central tendency – the mode

In statistics terminology, the **mode** of a feature is the value occurring most often. Like the mean and median, the mode is another measure of central tendency. It is typically used for categorical data, since the mean and median are not defined for nominal variables.

For example, in the used car data, the mode of year is 2010, while the modes for the model and color variables are SE and Black, respectively. A variable may have more than one mode; a variable with a single mode is **unimodal**, while a variable with two modes is **bimodal**. Data with multiple modes is more generally called **multimodal**.

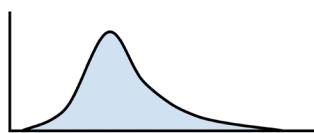


Although you might suspect that you could use the `mode()` function, R uses this to obtain the type of variable (as in numeric, list, and so on) rather than the statistical mode. Instead, to find the statistical mode, simply look at the `table()` output for the category with the greatest number of values.

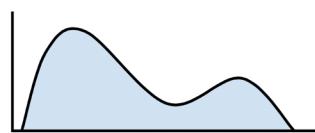
The mode or modes are used in a qualitative sense to gain an understanding of important values. Even so, it would be dangerous to place too much emphasis on the mode since the most common value is not necessarily a majority. For instance, although black was the single most common car color, it was only about a quarter of all advertised cars.

It is best to think about the modes in relation to the other categories. Is there one category that dominates all others, or are there several? Thinking about modes this way may help to generate testable hypotheses by raising questions about what makes certain values more common than others. If black and silver are common used car colors, we might believe that the data represents luxury cars, which tend to be sold in more conservative colors. Alternatively, these colors could indicate economy cars, which are sold with fewer color options. We will keep these questions in mind as we continue to examine this data.

Thinking about the modes as common values allows us to apply the concept of the statistical mode to numeric data. Strictly speaking, it would be unlikely to have a mode for a continuous variable, since no two values are likely to repeat. However, if we think about modes as the highest bars on a histogram, we can discuss the modes of variables such as `price` and `mileage`. It can be helpful to consider the mode when exploring numeric data, particularly to examine whether the data is multimodal.



Unimodal Distribution



Bimodal Distribution

Figure 2.10: Hypothetical distributions of numeric data with one and two modes

Exploring relationships between features

So far, we have examined variables one at a time, calculating only **univariate** statistics. During our investigation, we raised questions that we were unable to answer before:

- Does the `price` and `mileage` data imply that we are examining only economy-class cars, or are there luxury cars with high mileage?
- Do relationships between `model` and `color` provide insight into the types of cars we are examining?

These types of questions can be addressed by looking at **bivariate** relationships, which consider the relationship between two variables. Relationships of more than two variables are called **multivariate** relationships. Let's begin with the bivariate case.

Visualizing relationships – scatterplots

A **scatterplot** is a diagram that visualizes a bivariate relationship between numeric features. It is a two-dimensional figure in which dots are drawn on a coordinate plane using the values of one feature to provide the horizontal x coordinates, and the values of another feature to provide the vertical y coordinates. Patterns in the placement of dots reveal underlying associations between the two features.

To answer our question about the relationship between `price` and `mileage`, we will examine a scatterplot. We'll use the `plot()` function, along with the `main`, `xlab`, and `ylab` parameters used previously to label the diagram.

To use `plot()`, we need to specify x and y vectors containing the values used to position the dots on the figure. Although the conclusions would be the same regardless of the variable used to supply the x and y coordinates, convention dictates that the y variable is the one that is presumed to depend on the other (and is therefore known as the **dependent variable**). Since a seller cannot modify a car's odometer reading, mileage is unlikely to be dependent on the car's price. Instead, our hypothesis is that a car's price depends on the odometer mileage. Therefore, we will select `price` as the dependent y variable.

The full command to create our scatterplot is:

```
> plot(x = usedcars$mileage, y = usedcars$price,  
       main = "Scatterplot of Price vs. Mileage",  
       xlab = "Used Car Odometer (mi.)",  
       ylab = "Used Car Price ($)")
```

This produces the following scatterplot:

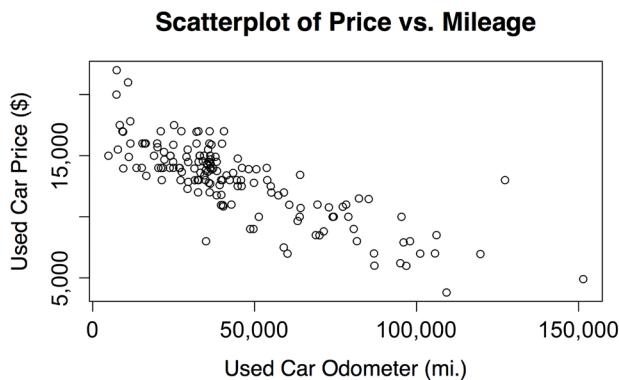


Figure 2.11: The relationship between used car price and mileage

Using the scatterplot, we notice a clear relationship between the price of a used car and the odometer reading. To read the plot, examine how the values of the y axis variable change as the values on the x axis increase. In this case, car prices tend to be lower as the mileage increases. If you have ever sold or shopped for a used car, this is not a profound insight.

Perhaps a more interesting finding is the fact that there are very few cars that have both high price and high mileage, aside from a lone outlier at about 125,000 miles and \$14,000. The absence of more points like this provides evidence to support the conclusion that our dataset is unlikely to include any high-mileage luxury cars. All the most expensive cars in the data, particularly those above \$17,500, seem to have extraordinarily low mileage, which implies that we could be looking at a single type of car that retails for a price of around \$20,000 when new.

The relationship we've observed between car prices and mileage is known as a negative association because it forms a pattern of dots in a line sloping downward. A positive association would appear to form a line sloping upward. A flat line, or a seemingly random scattering of dots, is evidence that the two variables are not associated at all. The strength of a linear association between two variables is measured by a statistic known as **correlation**. Correlations are discussed in detail in *Chapter 6, Forecasting Numeric Data – Regression Methods*, which covers methods for modeling linear relationships.



Keep in mind that not all associations form straight lines. Sometimes the dots form a U shape or V shape, while sometimes the pattern seems to be weaker or stronger for increasing values of the x or y variable. Such patterns imply that the relationship between the two variables is not linear, and thus correlation would be a poor measure of their association.

Examining relationships – two-way cross-tabulations

To examine a relationship between two nominal variables, a **two-way cross-tabulation** is used (also known as a **crosstab** or **contingency table**). A cross-tabulation is like a scatterplot in that it allows you to examine how the values of one variable vary by the values of another. The format is a table in which the rows are the levels of one variable, while the columns are the levels of another. Counts in each of the table's cells indicate the number of values falling into the row and column combination.

To answer our earlier question about whether there is a relationship between `model` and `color`, we will examine a crosstab. There are several functions to produce two-way tables in R, including `table()`, which we used before for one-way tables. The `CrossTable()` function in the `gmodels` package by Gregory R. Warnes is perhaps the most user-friendly, as it presents the row, column, and margin percentages in a single table, saving us the trouble of computing them ourselves. To install the `gmodels` package if you haven't already done so using the instructions in the prior chapter, use the following command:

```
> install.packages("gmodels")
```

After the package installs, type `library(gmodels)` to load the package. Although you only need to install the package once, you will need to load the package with the `library()` command during each R session in which you plan to use the `CrossTable()` function.

Before proceeding with our analysis, let's simplify our project by reducing the number of levels in the `color` variable. This variable has nine levels, but we don't really need this much detail. What we are truly interested in is whether the car's color is conservative. Toward this end, we'll divide the nine colors into two groups—the first group will include the conservative colors black, gray, silver, and white; the second group will include blue, gold, green, red, and yellow. We'll create a logical vector indicating whether the car's color is conservative by our definition. The following returns `TRUE` if the car is one of the four conservative colors, and `FALSE` otherwise:

```
> usedcars$conservative <-  
  usedcars$color %in% c("Black", "Gray", "Silver", "White")
```

You may have noticed a new command here. The `%in%` operator returns TRUE or FALSE for each value in the vector on the left-hand side of the operator, indicating whether the value is found in the vector on the right-hand side. In simple terms, you can translate this line as “is the used car color in the set of black, gray, silver, and white?”

Examining the `table()` output for our newly created variable, we see that about two-thirds of the cars have conservative colors while one-third do not:

```
> table(usedcars$conservative)
```

FALSE	TRUE
51	99

Now, let’s look at a cross-tabulation to see how the proportion of conservatively colored cars varies by model. Since we’re assuming that the model of car dictates the choice of color, we’ll treat the conservative color indicator as the dependent (*y*) variable. The `CrossTable()` command is therefore:

```
> CrossTable(x = usedcars$model, y = usedcars$conservative)
```

This results in the following table:

Cell Contents				
		N		
	Chi-square-contribution			
	N / Row Total			
	N / Col Total			
	N / Table Total			

Total Observations in Table: 150				
usedcars\$model	usedcars\$conservative			
	FALSE	TRUE	Row Total	
SE	27	51	78	
	0.009	0.004		
	0.346	0.654	0.520	
	0.529	0.515		
	0.180	0.340		

SEL	7	16	23	
	0.086	0.044		
	0.304	0.696	0.153	
	0.137	0.612		
	0.047	0.107		
SES	17	32	49	
	0.007	0.004		
	0.347	0.653	0.327	
	0.333	0.323		
	0.113	0.213		
Column Total	51	99	150	
	0.340	0.660		

The `CrossTable()` output is dense with numbers, but the legend at the top (labeled `Cell Contents`) indicates how to interpret each value. The table rows indicate the three models of used cars: SE, SEL, and SES (plus an additional row for the total across all models). The columns indicate whether the car's color is conservative (plus a column totaling across both types of color).

The first value in each cell indicates the number of cars with that combination of model and color. The proportions indicate each cell's contribution to the chi-square statistic, the row total, the column total, and the table's overall total.

What we are most interested in is the proportion of conservative cars for each model. The row proportions tell us that 0.654 (65 percent) of SE cars are colored conservatively, in comparison to 0.696 (70 percent) of SEL cars, and 0.653 (65 percent) of SES. These differences are relatively small, which suggests that there are no substantial differences in the types of colors chosen for each model of car.

The chi-square values refer to the cell's contribution in the **Pearson's chi-squared test** for independence between two variables. Although a complete discussion of the statistics behind this test is highly technical, the test measures how likely it is that the difference in cell counts in the table is due to chance alone, which can help us confirm our hypothesis that the differences by group are not substantial. Beginning by adding the cell contributions for the six cells in the table, we find $0.009 + 0.004 + 0.086 + 0.044 + 0.007 + 0.004 = 0.154$. This is the chi-squared test statistic.

To calculate the probability that this statistic is observed under the hypothesis that there is no association between the variables, we pass the test statistic to the `pchisq()` function as follows:

```
> pchisq(0.154, df = 2, lower.tail = FALSE)
```

```
[1] 0.9258899
```

The `df` parameter refers to the degrees of freedom, which is a component of the statistical test related to the number of rows and columns in the table; again, ignoring what it means, it can be computed as $(\text{rows} - 1) * (\text{columns} - 1)$, or 1 for a 2x2 table and 2 for the 3x2 table used here. Setting `lower.tail = FALSE` requests the right-tailed probability of about 0.926, which can be understood intuitively as the probability of obtaining a test statistic of at least 0.154 or greater due to chance alone.

If the probability of the chi-squared test is very low—perhaps below ten, five, or even one percent—it provides strong evidence that the two variables are associated, because the observed association in the table is unlikely to have happened by chance. In our case, the probability is much closer to 100 percent than to 10 percent, so it is unlikely we are observing an association between `model` and `color` for cars in this dataset.

Rather than computing this by hand, you can also obtain the chi-squared test results by adding an additional parameter specifying `chisq = TRUE` when calling the `CrossTable()` function. For example:

```
> CrossTable(x = usedcars$model, y = usedcars$conservative,
  chisq = TRUE)
```

```
Pearson's Chi-squared test
```

```
Chi^2 = 0.1539564    d.f. = 2    p = 0.92591
```

Note that, aside from slight differences due to rounding, this produces the same chi-squared test statistic and probability as was computed by hand.



The chi-squared test performed here is one of many types of formal hypothesis testing that can be performed using traditional statistics. If you've ever heard the phrase “statistically significant,” it means a statistical test like chi-squared (or one of many others) was performed, and it reached a “significant” level—typically, a probability less than five percent. Although hypothesis testing is somewhat beyond the scope of this book, it will be encountered again briefly in *Chapter 6, Forecasting Numeric Data – Regression Methods*.

Summary

In this chapter, we learned about the basics of managing data in R. We started by taking an in-depth look at the structures used for storing various types of data. The foundational R data structure is the vector, which is extended and combined into more complex data types, such as lists and data frames. The data frame is an R data structure that corresponds to the notion of a dataset having both features and examples. R provides functions for reading and writing data frames to spreadsheet-like tabular data files.

We then explored a real-world dataset containing the prices of used cars. We examined numeric variables using common summary statistics of center and spread, and visualized relationships between prices and odometer readings with a scatterplot. Next, we examined nominal variables using tables. In examining the used car data, we followed an exploratory process that can be used to understand any dataset. These skills will be required for the other projects throughout this book.

Now that we have spent some time understanding the basics of data management with R, you are ready to begin using machine learning to solve real-world problems. In the next chapter, we will tackle our first classification task using nearest neighbor methods. You may be surprised to discover that with just a few lines of R code, a machine can achieve human-like performance on a challenging medical diagnosis task.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



3

Lazy Learning – Classification Using Nearest Neighbors

A curious type of dining experience has appeared in cities around the world. Patrons are served in a completely darkened restaurant by waiters who move via memorized routes, using only their senses of touch and sound. The allure of these establishments is the belief that depriving oneself of sight will enhance the senses of taste and smell, and foods will be experienced in new ways. Each bite provides a sense of wonder while discovering the flavors the chef has prepared.

Can you imagine how a diner experiences the unseen food? Upon first bite, the senses are overwhelmed. What are the dominant flavors? Does the food taste savory or sweet? Does it taste like something they've eaten previously? Personally, I imagine this process of discovery in terms of a slightly modified adage—if it smells like a duck and tastes like a duck, then you are probably eating duck.

This illustrates an idea that can be used for machine learning—as does another maxim involving poultry—birds of a feather flock together. Stated differently, things that are alike tend to have properties that are alike. Machine learning uses this principle to classify data by placing it in the same category as similar or “nearest” neighbors. This chapter is devoted to classifiers that use this approach. You will learn:

- The key concepts that define nearest neighbor classifiers and why they are considered “lazy” learners
- Methods to measure the similarity of two examples using distance
- How to apply a popular nearest neighbor classifier called k-NN

If all of this talk about food is making you hungry, our first task will be to understand the k-NN approach by putting it to use while we settle a long-running culinary debate.

Understanding nearest neighbor classification

In a single sentence, **nearest neighbor** classifiers are defined by their characteristic of classifying unlabeled examples by assigning them the class of similar labeled examples. This is analogous to the dining experience described in the chapter introduction, in which a person identifies new foods through comparison to those previously encountered. With nearest neighbor classification, computers apply a human-like ability to recall past experiences to make conclusions about current circumstances. Despite the simplicity of this idea, nearest neighbor methods are extremely powerful. They have been used successfully for:

- Computer vision applications, including optical character recognition and facial recognition in still images and video
- Recommendation systems that predict whether a person will enjoy a movie or song
- Identifying patterns in genetic data to detect specific proteins or diseases

In general, nearest neighbor classifiers are well suited for classification tasks where relationships among the features and the target classes are numerous, complicated, or otherwise extremely difficult to understand, yet the items of similar class types tend to be fairly homogeneous. Another way of putting it would be to say that if a concept is difficult to define, but you know it when you see it, then nearest neighbors might be appropriate. On the other hand, if the data is noisy and thus no clear distinction exists among the groups, nearest neighbor algorithms may struggle to identify the class boundaries.

The k-NN algorithm

The nearest neighbors approach to classification is exemplified by the **k-nearest neighbors** algorithm (**k-NN**). Although this is perhaps one of the simplest machine learning algorithms, it is still used widely. The strengths and weaknesses of this algorithm are as follows:

Strengths	Weaknesses
<ul style="list-style-type: none"> • Simple and effective • Makes no assumptions about the underlying data distribution • Fast training phase 	<ul style="list-style-type: none"> • Does not produce a model, limiting the ability to understand how the features are related to the class • Requires selection of an appropriate k • Slow classification phase • Nominal features and missing data require additional processing

The k-NN algorithm gets its name from the fact that it uses information about an example's k nearest neighbors to classify unlabeled examples. The letter k is a variable implying that any number of nearest neighbors could be used. After choosing k , the algorithm requires a training dataset made up of examples that have been classified into several categories, as labeled by a nominal variable. Then, for each unlabeled record in the test dataset, k-NN identifies the k records in the training data that are the "nearest" in similarity. The unlabeled test instance is assigned the class representing the majority of the k nearest neighbors.

To illustrate this process, let's revisit the blind tasting experience described in the introduction. Suppose that prior to eating the mystery meal, we had created a dataset in which we recorded our impressions of a set of previously tasted ingredients. To keep things simple, we rated only two features of each ingredient. The first is a measure from 1 to 10 of how crunchy the ingredient is, and the second is a score from 1 to 10 measuring how sweet the ingredient tastes. We then labeled each ingredient as one of three types of food: fruits, vegetables, or proteins, ignoring other foods such as grains and fats.

The first few rows of such a dataset might be structured as follows:

Ingredient	Sweetness	Crunchiness	Food type
Apple	10	9	Fruit
Bacon	1	4	Protein
Banana	10	1	Fruit
Carrot	7	10	Vegetable
Celery	3	10	Vegetable

The k-NN algorithm treats the features as coordinates in a multidimensional **feature space**, which is a space comprising all possible combinations of feature values. Because the ingredient dataset includes only two features, its feature space is two-dimensional. We can plot two-dimensional data on a scatterplot, with the *x* dimension indicating the ingredient's sweetness and the *y* dimension indicating the crunchiness. After adding a few more ingredients to the taste dataset, the scatterplot might look like this:

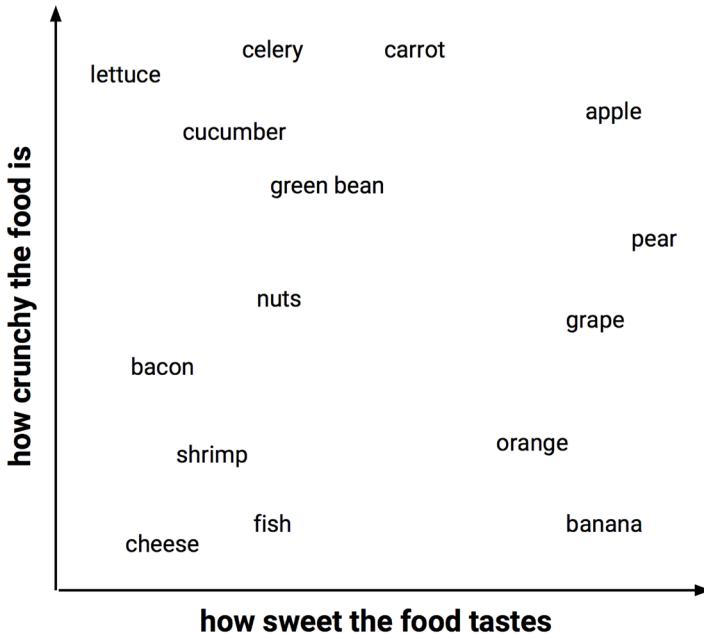


Figure 3.1: A scatterplot of selected foods' crunchiness versus sweetness

Do you notice a pattern? Similar types of food tend to be grouped closely together. As illustrated in *Figure 3.2*, vegetables tend to be crunchy but not sweet; fruits tend to be sweet and either crunchy or not crunchy; and proteins tend to be neither crunchy nor sweet:

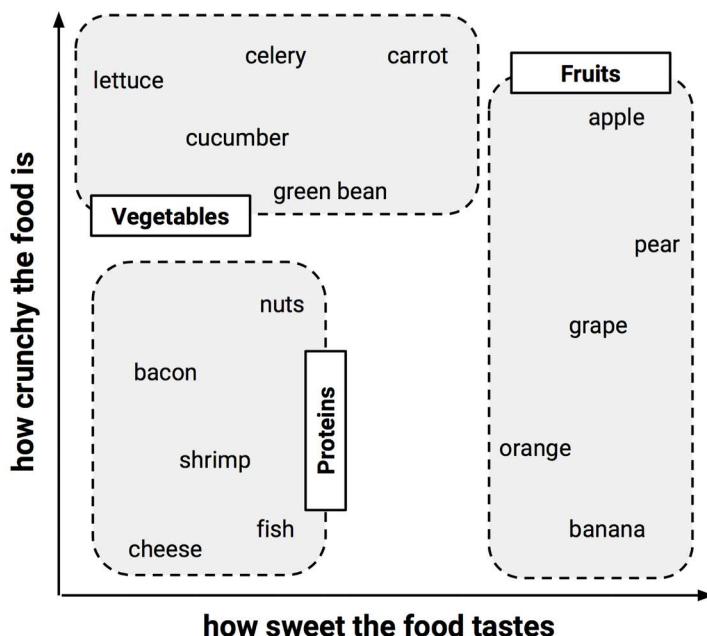


Figure 3.2: Foods that are similarly classified tend to have similar attributes

Suppose that after constructing this dataset, we decide to use it to settle the age-old question: is a tomato a fruit or a vegetable? We can use the nearest neighbor approach to determine which class is a better fit, as shown in *Figure 3.3*:

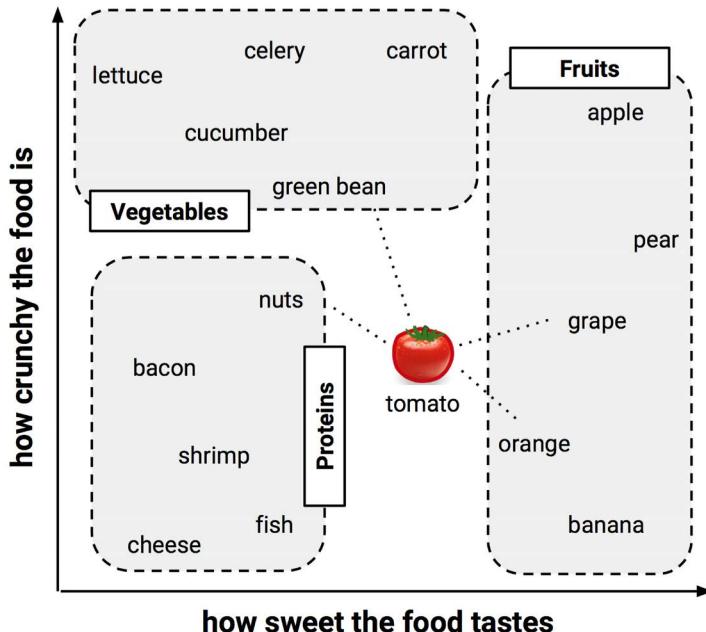


Figure 3.3: The tomato's nearest neighbors provide insight into whether it is a fruit or vegetable

Measuring similarity with distance

Locating the tomato's nearest neighbors requires a **distance function**, which is a formula that measures the similarity between two instances.

There are many ways to calculate distance. The choice of distance function may impact the model's performance substantially, although it is difficult to know which to use except by comparing them directly on the desired learning task. Traditionally, the k-NN algorithm uses **Euclidean distance**, which is the distance one would measure if it were possible to use a ruler to connect two points. Euclidean distance is measured “as the crow flies,” which implies the shortest direct route. This is illustrated in the previous figure by the dotted lines connecting the tomato to its neighbors.



Another common distance measure is **Manhattan distance**, which is based on the paths a pedestrian would take by walking city blocks. If you are interested in learning more about other distance measures, you can read the documentation for R's `distance` function using the `?dist` command.

Euclidean distance is specified by the following formula, where p and q are the examples to be compared, each having n features. The term p^1 refers to the value of the first feature of example p , while q^1 refers to the value of the first feature of example q :

$$\text{dist}(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2}$$

The distance formula involves comparing the values of each example's features. For example, to calculate the distance between the tomato (sweetness = 6, crunchiness = 4), and the green bean (sweetness = 3, crunchiness = 7), we can use the formula as follows:

$$\text{dist}(\text{tomato}, \text{green bean}) = \sqrt{(6 - 3)^2 + (4 - 7)^2} = 4.2$$

In a similar vein, we can calculate the distance between the tomato and several of its closest neighbors as follows:

Ingredient	Sweetness	Crunchiness	Food type	Distance to the tomato
Grape	8	5	Fruit	$\sqrt{(6 - 8)^2 + (4 - 5)^2} = 2.2$
Green bean	3	7	Vegetable	$\sqrt{(6 - 3)^2 + (4 - 7)^2} = 4.2$
Nuts	3	6	Protein	$\sqrt{(6 - 3)^2 + (4 - 6)^2} = 3.6$
Orange	7	3	Fruit	$\sqrt{(6 - 7)^2 + (4 - 3)^2} = 1.4$

To classify the tomato as a vegetable, protein, or fruit, we'll begin by assigning the tomato the food type of its single nearest neighbor. This is called 1-NN classification because $k = 1$. The orange is the single nearest neighbor to the tomato, with a distance of 1.4. Because an orange is a fruit, the 1-NN algorithm would classify a tomato as a fruit.

If we use the k-NN algorithm with $k = 3$ instead, it performs a vote among the three nearest neighbors: orange, grape, and nuts. Now, because the majority class among these neighbors is fruit (with two of the three votes), the tomato again is classified as a fruit.

Choosing an appropriate k

The decision of how many neighbors to use for k-NN determines how well the model will generalize to future data. The balance between overfitting and underfitting the training data is a problem known as the **bias-variance tradeoff**. Choosing a large k reduces the impact of variance caused by noisy data but can bias the learner such that it runs the risk of ignoring small but important patterns.

Suppose we took the extreme stance of setting a very large k , as large as the total number of observations in the training data. With every training instance represented in the final vote, the most common class always has a majority of the voters. The model would consequently always predict the majority class, regardless of the nearest neighbors.

On the opposite extreme, using a single nearest neighbor allows noisy data and outliers to unduly influence the classification of examples. For example, suppose some of the training examples were accidentally mislabeled. Any unlabeled example that happens to be nearest to the incorrectly labeled neighbor will be predicted to have the incorrect class, even if nine other nearby neighbors would have voted differently.

Obviously, the best k value is somewhere between these two extremes.

Figure 3.4 illustrates, more generally, how the decision boundary (depicted by a dashed line) is affected by larger or smaller k values. Smaller values allow more complex decision boundaries that more carefully fit the training data. The problem is that we do not know whether the straight boundary or the curved boundary better represents the true underlying concept to be learned.

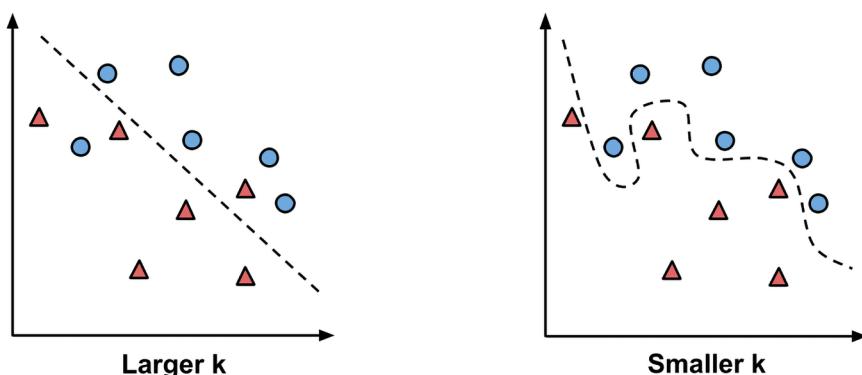
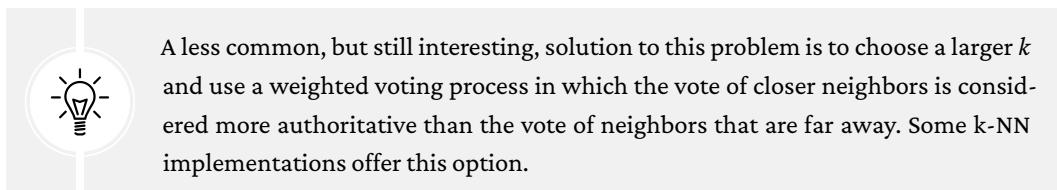


Figure 3.4: A larger k has higher bias and lower variance than a smaller k

In practice, the choice of k depends on the difficulty of the concept to be learned and the number of records in the training data. One common approach is to begin with k equal to the square root of the number of training examples. In the food classifier developed previously, we might set $k = 4$ because there were 15 example ingredients in the training data and the square root of 15 is 3.87.

However, such rules may not always result in the single best k . An alternative approach is to test several k values on a variety of test datasets and choose the one that delivers the best classification performance. That said, unless the data is very noisy, a large training dataset can make the choice of k less important. This is because even subtle concepts will have a sufficiently large pool of examples to vote as nearest neighbors.



A less common, but still interesting, solution to this problem is to choose a larger k and use a weighted voting process in which the vote of closer neighbors is considered more authoritative than the vote of neighbors that are far away. Some k-NN implementations offer this option.

Preparing data for use with k-NN

Features are typically transformed to a standard range prior to applying the k-NN algorithm. The rationale for this step is that the distance formula is highly dependent on how features are measured. In particular, if certain features have a much larger range of values than others, the distance measurements will be strongly dominated by the features with larger ranges. This wasn't a problem for the food tasting example, as both sweetness and crunchiness were measured on a scale from 1 to 10.

However, suppose that we added an additional feature to the dataset to represent a food's spiciness, which was measured using the Scoville scale. If you are unfamiliar with this metric, it is a standardized measure of spice heat, ranging from zero (not at all spicy) to over a million (for the hottest chili peppers). Since the difference between spicy and non-spicy foods can be over a million while the difference between sweet and non-sweet or crunchy and non-crunchy foods is at most 10, the difference in scale allows the spice level to impact the distance function much more than the other two factors. Without adjusting our data, we might find that our distance measures only differentiate foods by their spiciness; the impact of crunchiness and sweetness would be dwarfed by the contribution of spiciness.

The solution is to rescale the features by shrinking or expanding their range such that each one contributes relatively equally to the distance formula. For example, if sweetness and crunchiness are both measured on a scale from 1 to 10, we would also like spiciness to be measured on a scale from 1 to 10. There are several common ways to accomplish such scaling.

The traditional method of rescaling features for k-NN is **min-max normalization**. This process transforms a feature such that all values fall in a range between 0 and 1. The formula for normalizing a feature is as follows:

$$X_{new} = \frac{X - \min(X)}{\max(X) - \min(X)}$$

To transform each value of feature X , the formula subtracts the minimum X value and divides it by the range of X . The resulting normalized feature values can be interpreted as indicating how far, from 0 percent to 100 percent, the original value fell along the range between the original minimum and maximum.

Another common transformation is called **z-score standardization**. The following formula subtracts the mean value of feature X , and divides the result by the standard deviation of X :

$$X_{new} = \frac{X - \mu}{\sigma} = \frac{X - \text{Mean}(X)}{\text{StdDev}(X)}$$

This formula, which is based on properties of the normal distribution covered in *Chapter 2, Managing and Understanding Data*, rescales each of a feature's values in terms of how many standard deviations they fall above or below the mean. The resulting value is called a **z-score**. The z-scores fall in an unbounded range of negative and positive numbers. Unlike the normalized values, they have no predefined minimum and maximum.



The same rescaling method used on the k-NN training dataset must also be applied to the test examples that the algorithm will later classify. This can lead to a tricky situation for min-max normalization, as the minimum or maximum of future cases might be outside the range of values observed in the training data. If you know the theoretical minimum or maximum value ahead of time, you can use these constants rather than the observed minimum and maximum values. Alternatively, you can use z-score standardization under the assumption that the future examples are taken from a distribution with the same mean and standard deviation as the training examples.

The Euclidean distance formula is undefined for nominal data. Therefore, to calculate the distance between nominal features, we need to convert them into a numeric format. A typical solution utilizes **dummy coding**, where a value of 1 indicates one category, and 0 indicates the other. For instance, dummy coding for a male or non-male sex variable could be constructed as:

$$\text{male} = \begin{cases} 1 & \text{if } x = \text{male} \\ 0 & \text{otherwise} \end{cases}$$

Notice how dummy coding of the two-category (binary) sex variable results in a single new feature named male. There is no need to construct a separate feature for non-male. Since both are mutually exclusive, knowing one or the other is enough.

This is true more generally as well. An n -category nominal feature can be dummy coded by creating binary indicator variables for $n - 1$ levels of the feature. For example, dummy coding for a three-category temperature variable (for example, hot, medium, or cold) could be set up as $(3 - 1) = 2$ features, as shown here:

$$\text{hot} = \begin{cases} 1 & \text{if } x = \text{hot} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{medium} = \begin{cases} 1 & \text{if } x = \text{medium} \\ 0 & \text{otherwise} \end{cases}$$

Knowing that hot and medium are both 0 provides enough information to know that the temperature is cold, and thus, a third binary feature for the cold category is unnecessary. However, a widely used close sibling of dummy coding known as **one-hot encoding** creates binary features for all n levels of the feature, rather than $n - 1$ as with dummy coding. It is known as “one-hot” because only one attribute is coded as 1 and the others are set to 0.

In practice, there is virtually no difference between these two methods, and the results of machine learning will be unaffected by the choice of coding. This being said, one-hot encoding can cause problems with linear models, such as those described in *Chapter 6, Forecasting Numeric Data – Regression Methods*, and thus one-hot encoding is often avoided among statisticians or in fields like economics that rely heavily on such models. On the other hand, one-hot encoding has become prevalent in the field of machine learning and is often treated synonymously with dummy coding for the simple reason that the choice makes virtually no difference in the model fit; yet, in one-hot encoding, the model itself may be easier to understand since all levels of the categorical features are specified explicitly. This book uses only dummy coding since it can be used universally, but you may encounter one-hot encoding elsewhere.

A convenient aspect of both dummy and one-hot coding is that the distance between dummy-coded features is always one or zero, and thus, the values fall on the same scale as min-max normalized numeric data. No additional transformation is necessary.



If a nominal feature is ordinal (one could make such an argument for temperature), an alternative to dummy coding is to number the categories and apply normalization. For instance, cold, warm, and hot could be numbered as 1, 2, and 3, which normalizes to 0, 0.5, and 1. A caveat to this approach is that it should only be used if the steps between categories are equivalent. For instance, although income categories for poor, middle class, and wealthy are ordered, the difference between poor and middle class may be different than the difference between middle class and wealthy. Since the steps between groups are not equal, dummy coding is a safer approach.

Why is the k-NN algorithm lazy?

Classification algorithms based on nearest neighbor methods are considered **lazy learning** algorithms because, technically speaking, no abstraction occurs. The abstraction and generalization processes are skipped altogether, which undermines the definition of learning proposed in *Chapter 1, Introducing Machine Learning*.

Under the strict definition of learning, a lazy learner is not really learning anything. Instead, it merely stores the training data verbatim. This allows the training phase, which is not actually training anything, to occur very rapidly. Of course, the downside is that the process of making predictions tends to be relatively slow by comparison. Due to the heavy reliance on the training instances rather than an abstracted model, lazy learning is also known as **instance-based learning** or **rote learning**.

As instance-based learners do not build a model, the method is said to be in a class of **non-parametric** learning methods—no parameters are learned about the data. Without generating theories about the underlying data, non-parametric methods limit our ability to understand how the classifier is using the data, yet it can still make useful predictions. Non-parametric learning allows the learner to find natural patterns rather than trying to fit the data into a preconceived and potentially biased functional form.

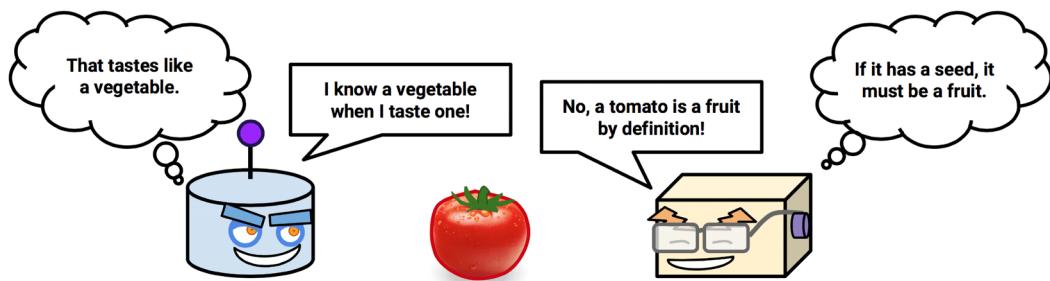


Figure 3.5: Machine learning algorithms have different biases and may come to different conclusions!

Although k-NN classifiers may be considered lazy, they are still quite powerful. As you will soon see, the simple principles of nearest neighbor learning can be used to automate the process of screening for cancer.

Example – diagnosing breast cancer with the k-NN algorithm

Routine breast cancer screening allows the disease to be diagnosed and treated prior to it causing noticeable symptoms. The process of early detection involves examining the breast tissue for abnormal lumps or masses. If a lump is found, a fine-needle aspiration biopsy is performed, which uses a hollow needle to extract a small sample of cells from the mass. A clinician then examines the cells under a microscope to determine whether the mass is likely to be malignant or benign.

If machine learning could automate the identification of cancerous cells, it would provide considerable benefit to the health system. Automated processes are likely to improve the efficiency of the detection process, allowing physicians to spend less time diagnosing and more time treating the disease. An automated screening system might also provide greater detection accuracy by removing the inherently subjective human component from the process.

Let's investigate the utility of machine learning for detecting cancer by applying the k-NN algorithm to measurements of biopsied cells from women with abnormal breast masses.

Step 1 – collecting data

We will utilize the Breast Cancer Wisconsin (Diagnostic) dataset from the UCI Machine Learning Repository at <http://archive.ics.uci.edu/ml>. This data was donated by researchers at the University of Wisconsin and includes measurements from digitized images of fine-needle aspirations of a breast mass. The values represent characteristics of the cell nuclei present in the digital image.



To read more about this dataset, refer to *Breast Cancer Diagnosis and Prognosis via Linear Programming*, Mangasarian OL, Street WN, Wolberg WH, *Operations Research*, 1995, Vol. 43, pp. 570-577.

The breast cancer data includes 569 examples of cancer biopsies, each with 32 features. One feature is an identification number, another is the cancer diagnosis, and 30 are numeric-valued laboratory measurements. The diagnosis is coded as “M” to indicate malignant or “B” to indicate benign.

The 30 numeric measurements comprise the mean, standard error, and worst (that is, largest) value for 10 different characteristics of the digitized cell nuclei, such as radius, texture, area, smoothness, and compactness. Based on the feature names, the dataset seems to measure the shape and size of the cell nuclei, but unless you are an oncologist, you are unlikely to know how each of these relates to benign or malignant masses. No such expertise is necessary, as the computer will discover the important patterns during the machine learning process.

Step 2 – exploring and preparing the data

By exploring the data, we may be able to shine some light on the relationships between the features and the cancer status. In doing so, we will prepare the data for use with the k-NN learning method.



If you plan on following along, download the code and `wisc_bc_data.csv` files from the GitHub repository and save them to your R working directory. For this book, the dataset was modified very slightly from its original form. In particular, a header line was added, and the rows of data were randomly ordered.

We'll begin by importing the CSV data file as we have done in previous chapters, saving the Wisconsin breast cancer data to the `wbcd` data frame:

```
> wbcn <- read.csv("wisc_bc_data.csv")
```

Using the command `str(wbcd)`, we can confirm that the data is structured with 569 examples and 32 features, as we expected. The first several lines of output are as follows:

```
> str(wbcd)
```

```
'data.frame': 569 obs. of 32 variables:  
 $ id           : int 87139402 8910251 905520 ...  
 $ diagnosis    : chr "B" "B" "B" "B" ...  
 $ radius_mean   : num 12.3 10.6 11 11.3 15.2 ...  
 $ texture_mean   : num 12.4 18.9 16.8 13.4 13.2 ...  
 $ perimeter_mean : num 78.8 69.3 70.9 73 97.7 ...  
 $ area_mean     : num 464 346 373 385 712 ...
```

The first feature is an integer variable named `id`. As this is simply a unique identifier (ID) for each patient in the data, it does not provide useful information, and we will need to exclude it from the model.



Regardless of the machine learning method, ID variables should always be excluded. Neglecting to do so can lead to erroneous findings because the ID can be used to correctly predict each example. Therefore, a model that includes an ID column will almost definitely suffer from overfitting and generalize poorly to future data.

Let's drop the `id` feature from our data frame. As it is in the first column, we can exclude it by making a copy of the `wbcid` data frame without column 1:

```
> wbcd <- wbcd[-1]
```

The next feature, `diagnosis`, is of particular interest as it is the target outcome we hope to predict. This feature indicates whether the example is from a benign or malignant mass. The `table()` output indicates that 357 masses are benign, while 212 are malignant:

```
> table(wbcd$diagnosis)
```

B M
357 212

Many R machine learning classifiers require the target feature to be coded as a factor, so we will need to recode the diagnosis column. We will also take this opportunity to give the "B" and "M" values more informative labels using the `labels` parameter:

When we look at the `prop.table()` output, we now see that the values have been labeled Benign and Malignant, with 62.7 percent and 37.3 percent of the masses, respectively:

```
> round(prop.table(table(wbcd$diagnosis)) * 100, digits = 1)
```

Benign	Malignant
62.7	37.3

The remaining 30 features are all numeric and, as expected, consist of three different measurements of 10 characteristics. For illustrative purposes, we will only take a closer look at three of these features:

```
> summary(wbcd[c("radius_mean", "area_mean", "smoothness_mean")])
```

radius_mean	area_mean	smoothness_mean
Min. : 6.981	Min. : 143.5	Min. : 0.05263
1st Qu.:11.700	1st Qu.: 420.3	1st Qu.: 0.08637
Median :13.370	Median : 551.1	Median : 0.09587
Mean :14.127	Mean : 654.9	Mean : 0.09636
3rd Qu.:15.780	3rd Qu.: 782.7	3rd Qu.: 0.10530
Max. :28.110	Max. :2501.0	Max. : 0.16340

Looking at the three side by side, do you notice anything problematic about the values? Recall that the distance calculation for k-NN is heavily dependent upon the measurement scale of the input features. Since smoothness ranges from 0.05 to 0.16, while area ranges from 143.5 to 2501.0, the impact of area is going to be much greater than smoothness in the distance calculation. This could potentially cause problems for our classifier, so let's apply normalization to rescale the features to a standard range of values.

Transformation – normalizing numeric data

To normalize these features, we need to create a `normalize()` function in R. This function takes a vector `x` of numeric values, and for each value in `x`, subtracts the minimum `x` value and divides it by the range of `x` values. Lastly, the resulting vector is returned. The code for the function is as follows:

```
> normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
```

After executing the previous code, the `normalize()` function is available for use in R. Let's test the function on a couple of vectors:

```
> normalize(c(1, 2, 3, 4, 5))  
[1] 0.00 0.25 0.50 0.75 1.00  
  
> normalize(c(10, 20, 30, 40, 50))  
[1] 0.00 0.25 0.50 0.75 1.00
```

The function appears to be working correctly. Even though the values in the second vector are 10 times larger than the first vector, after normalization, they are identical.

We can now apply the `normalize()` function to the numeric features in our data frame. Rather than normalizing each of the 30 numeric variables individually, we will use one of R's functions to automate the process.

The `lapply()` function takes a list and applies a specified function to each list element. As a data frame is a list of equal-length vectors, we can use `lapply()` to apply `normalize()` to each feature in the data frame. The final step is to convert the list returned by `lapply()` to a data frame using the `as.data.frame()` function. The full process looks like this:

```
> wbcd_n <- as.data.frame(lapply(wbcd[2:31], normalize))
```

In plain English, this command applies the `normalize()` function to columns 2 to 31 in the `wbcd` data frame, converts the resulting list to a data frame, and assigns it the name `wbcd_n`. The `_n` suffix is used here as a reminder that the values in `wbcd` have been normalized.

To confirm that the transformation was applied correctly, let's look at one variable's summary statistics:

```
> summary(wbcd_n$area_mean)  
Min. 1st Qu. Median Mean 3rd Qu. Max.  
0.0000 0.1174 0.1729 0.2169 0.2711 1.0000
```

As expected, the `area_mean` variable, which originally ranged from 143.5 to 2501.0, now ranges from 0 to 1.



To simplify data preparation for this example, min-max normalization was applied to the entire dataset—including the rows that will later become the test set. In a way, this violates our simulation of unseen future data since, in practice, one will generally not know the true minimum and maximum values at the time of model training and future values might fall outside the previously observed range. A better approach might be to normalize the test set using only the minimum and maximum values observed in the training data, and potentially even capping any future values at the prior minimum or maximum levels. This being said, whether normalization is applied to training and test sets together or separately is unlikely to notably impact the model's performance and does not do so here.

Data preparation – creating training and test datasets

Although all 569 biopsies are labeled with a benign or malignant status, it is not very interesting to predict what we already know. Additionally, any performance measures we obtain during training may be misleading, as we do not know the extent to which the data has been overfitted or how well the learner will generalize to new cases. For these reasons, a more interesting question is how well our learner performs on a dataset of unseen data. If we had access to a laboratory, we could apply our learner to measurements taken from the next 100 masses of unknown cancer status and see how well the machine learner's predictions compare to diagnoses obtained using conventional methods.

In the absence of such data, we can simulate this scenario by dividing our data into two portions: a training dataset that will be used to build the k-NN model and a test dataset that will be used to estimate the predictive accuracy of the model. We will use the first 469 records for the training dataset and the remaining 100 to simulate new patients.

Using the data extraction methods presented in *Chapter 2, Managing and Understanding Data*, we will split the `wbcd_n` data frame into `wbcd_train` and `wbcd_test`:

```
> wbcn_train <- wbcn_n[1:469, ]  
> wbcn_test <- wbcn_n[470:569, ]
```

If the previous commands are confusing, remember that data is extracted from data frames using the `[row, column]` syntax. A blank value for the row or column value indicates that all rows or columns should be included. Hence, the first line of code requests rows 1 to 469 and all columns, and the second line requests 100 rows from 470 to 569 and all columns.



When constructing training and test datasets, it is important that each dataset is a representative subset of the full set of data. The `wbcd` records were already randomly ordered, so we could simply extract 100 consecutive records to create a representative test dataset. This would not be appropriate if the data was ordered chronologically or in groups of similar values. In these cases, random sampling methods would be needed. Random sampling will be discussed in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*.

When we constructed our normalized training and test datasets, we excluded the target variable, `diagnosis`. For training the k-NN model, we will need to store these class labels in factor vectors, split between the training and test datasets:

```
> wbc_train_labels <- wbc[1:469, 1]  
> wbc_test_labels <- wbc[470:569, 1]
```

This code takes the `diagnosis` factor in the first column of the `wbc` data frame and creates the vectors `wbc_train_labels` and `wbc_test_labels`. We will use these in the next steps of training and evaluating our classifier.

Step 3 – training a model on the data

Equipped with our training data and vector of labels, we are now ready to classify our test records. For the k-NN algorithm, the training phase involves no model building; the process of training a so-called “lazy” learner like k-NN simply involves storing the input data in a structured format.

To classify our test instances, we will use a k-NN implementation from the `class` package, which provides a set of basic R functions for classification. If this package is not already installed on your system, you can install it by typing:

```
> install.packages("class")
```

To load the package during any session in which you wish to use the functions, simply enter the `library(class)` command.

The `knn()` function in the `class` package provides a standard, traditional implementation of the k-NN algorithm. For each instance in the test data, the function will identify the k nearest neighbors, using Euclidean distance, where k is a user-specified number. The test instance is classified by taking a “vote” among the k nearest neighbors—specifically, this involves assigning the class of the majority of neighbors. A tie vote is broken at random.



There are several other k-NN functions in other R packages that provide more sophisticated or more efficient implementations. If you run into limitations with `knn()`, search for k-NN on the CRAN website: <https://cran.r-project.org>.

Training and classification using the `knn()` function is performed in a single command that requires four parameters, as shown in the following table:

kNN classification syntax

using the `knn()` function in the `class` package

Building the classifier and making predictions:

```
p <- knn(train, test, class, k)
```

- **train** is a data frame containing numeric training data
 - **test** is a data frame containing numeric test data
 - **class** is a factor vector with the class for each row in the training data
 - **k** is an integer indicating the number of nearest neighbors

The function returns a factor vector of predicted classes for each row in the test data frame.

Example:

```
wbcd_pred <- knn(train = wbcd_train, test = wbcd_test,  
                   cl = wbcd_train_labels, k = 3)
```

Figure 3.6: kNN classification syntax

We now have nearly everything we need to apply the k-NN algorithm to this data. We've split our data into training and test datasets, each with the same numeric features. The labels for the training data are stored in a separate factor vector. The only remaining parameter is k , which specifies the number of neighbors to include in the vote.

As our training data includes 469 instances, we might try $k = 21$, an odd number roughly equal to the square root of 469. With a two-category outcome, using an odd number eliminates the possibility of ending with a tie vote.

Now we can use the `knn()` function to classify the test data:

The knn() function returns a factor vector of predicted labels for each of the examples in the wbcד_test dataset. We have assigned these predictions to wbcד_test_pred.

Step 4 – evaluating model performance

The next step of the process is to evaluate how well the predicted classes in the wbcד_test_pred vector match the actual values in the wbcד_test_labels vector. To do this, we can use the CrossTable() function in the gmodels package, which was introduced in *Chapter 2, Managing and Understanding Data*. If you haven't done so already, please install this package using the `install.packages("gmodels")` command.

After loading the package with the `library(gmodels)` command, we can create a cross tabulation indicating the agreement between the predicted and actual label vectors. Specifying `prop.chisq = FALSE` excludes the unnecessary chi-square values from the output:

```
> CrossTable(x = wbcד_test_labels, y = wbcד_test_pred,
             prop.chisq = FALSE)
```

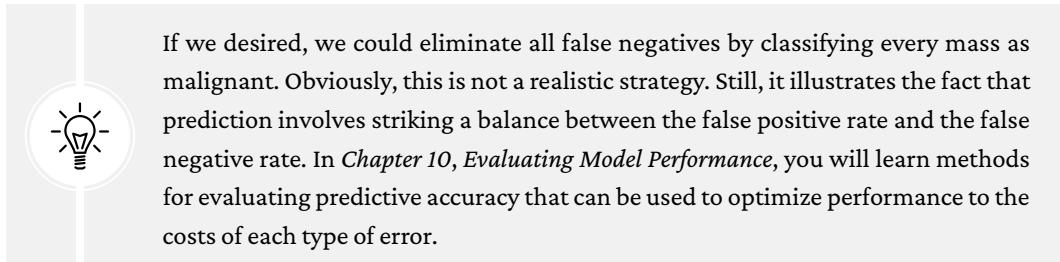
The resulting table looks like this:

		wbcד_test_pred		
Wbcד_test_labels		Benign	Malignant	Row Total
		-----	-----	-----
Benign		61	0	61
		1.000	0.000	0.610
		0.968	0.000	
		0.610	0.000	
		-----	-----	-----
Malignant		2	37	39
		0.051	0.949	0.390
		0.032	1.000	
		0.020	0.370	
		-----	-----	-----
Column Total		63	37	100
		0.630	0.370	

The cell percentages in the table indicate the proportion of values that fall into four categories. The top-left cell indicates the **true negative** results. These 61 of 100 values are cases where the mass was benign and the k-NN algorithm correctly identified it as such. The bottom-right cell indicates the **true positive** results, where the classifier and the clinically determined label agree that the mass is malignant. A total of 37 of 100 predictions were true positives.

The cells falling on the other diagonal contain counts of examples where the k-NN prediction disagreed with the true label. The two examples in the lower-left cell are **false negative** results; in this case, the predicted value was benign, but the tumor was actually malignant. Errors in this direction could be extremely costly, as they might lead a patient to believe that they are cancer-free, but in reality, the disease may continue to spread.

The top-right cell would contain the **false positive** results, if there were any. These values occur when the model has classified a mass as malignant when it actually was benign. Although such errors are less dangerous than a false negative result, they should also be avoided, as they could lead to additional financial burden on the health care system or stress for the patient, as unnecessary tests or treatment may be provided.



A total of 2 out of 100, or 2 percent of masses were incorrectly classified by the k-NN approach. While 98 percent accuracy seems impressive for a few lines of R code, we might try another iteration of the model to see if we can improve the performance and reduce the number of values that have been incorrectly classified, especially because the errors were dangerous false negatives.

Step 5 – improving model performance

We will attempt two simple variations on our previous classifier. First, we will employ an alternative method for rescaling our numeric features. Second, we will try several different k values.

Transformation – z-score standardization

Although normalization is commonly used for k-NN classification, z-score standardization may be a more appropriate way to rescale the features in a cancer dataset.

Since z-score standardized values have no predefined minimum and maximum, extreme values are not compressed towards the center. Even without medical training, one might suspect that a malignant tumor might lead to extreme outliers as tumors grow uncontrollably. With this in mind, it might be reasonable to allow the outliers to be weighted more heavily in the distance calculation. Let's see whether z-score standardization improves our predictive accuracy.

To standardize a vector, we can use R's built-in `scale()` function, which by default rescales values using the z-score standardization. The `scale()` function can be applied directly to a data frame, so there is no need to use the `lapply()` function. To create a z-score standardized version of the `wbcd` data, we can use the following command:

```
> wbcz_z <- as.data.frame(scale(wbcd[-1]))
```

This rescales all features with the exception of `diagnosis` in the first column and stores the result as the `wbcz_z` data frame. The `_z` suffix is a reminder that the values were z-score transformed.

To confirm that the transformation was applied correctly, we can look at the summary statistics:

```
> summary(wbcz_z$area_mean)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-1.4530	-0.6666	-0.2949	0.0000	0.3632	5.2460

The mean of a z-score standardized variable should always be zero, and the range should be fairly compact. A z-score less than -3 or greater than 3 indicates an extremely rare value. Examining the summary statistics with these criteria in mind, the transformation seems to have worked.

As we have done before, we need to divide the z-score-transformed data into training and test sets, and classify the test instances using the `knn()` function. We'll then compare the predicted labels to the actual labels using `CrossTable()`:

```
> wbcz_train <- wbcz_z[1:469, ]
> wbcz_test <- wbcz_z[470:569, ]
> wbcz_train_labels <- wbcd[1:469, 1]
> wbcz_test_labels <- wbcd[470:569, 1]
> wbcz_test_pred <- knn(train = wbcz_train, test = wbcz_test,
   cl = wbcz_train_labels, k = 21)
> CrossTable(x = wbcz_test_labels, y = wbcz_test_pred,
   prop.chisq = FALSE)
```

Unfortunately, in the following table, the results of our new transformation show a slight decline in accuracy. Using the same instances in which we had previously classified 98 percent of examples correctly, we now classified only 95 percent correctly. Making matters worse, we did no better at classifying the dangerous false negatives.

	wbcdf_test_pred		
Wbcd_test_labels	Benign	Malignant	Row Total
Benign	61	0	61
	1.000	0.000	0.610
	0.924	0.000	
	0.610	0.000	
Malignant	5	34	39
	0.128	0.872	0.390
	0.076	1.000	
	0.050	0.340	
Column Total	66	34	100
	0.660	0.340	

Testing alternative values of k

We may be able to optimize the performance of the k-NN model by examining its performance across various k values. Using the normalized training and test datasets, the same 100 records need to be classified using several different choices of k . Given that we are testing only six k values, these iterations can be performed most simply by using copy-and-paste of our previous `knn()` and `CrossTable()` functions. However, it is also possible to write a `for` loop that runs these two functions for each of the values in a vector named `k_values`, as demonstrated in the following code:

```
> k_values <- c(1, 5, 11, 15, 21, 27)
> for (k_val in k_values) {
  wbcdf_test_pred <- knn(train = wbcdf_train,
                           test = wbcdf_test,
                           cl = wbcdf_train_labels,
                           k = k_val)
  CrossTable(x = wbcdf_test_labels,
             y = wbcdf_test_pred,
             prop.chisq = FALSE)
}
```

The for loop can almost be read as a simple sentence: for each value named `k_val` in the `k_values` vector, run the `knn()` function while setting the parameter `k` to the current `k_val`, and then produce the `CrossTable()` for the resulting predictions.



A more sophisticated approach to looping using one of R's `apply()` functions is described in *Chapter 7, Black-Box Methods – Neural Networks and Support Vector Machines*, to test various values of a cost parameter and plot the result.

The false negatives, false positives, and overall error rate are shown for each iteration:

k value	False negatives	False positives	Error rate
1	1	3	4 percent
5	2	0	2 percent
11	3	0	3 percent
15	3	0	3 percent
21	2	0	2 percent
27	4	0	4 percent

Although the classifier was never perfect, the 1-NN approach was able to avoid some of the false negatives at the expense of adding false positives. It is important to keep in mind, however, that it would be unwise to tailor our approach too closely to our test data; after all, a different set of 100 patient records is likely to be somewhat different from those used to measure our performance.



If you need to be certain that a learner will generalize to future data, you might create several sets of 100 patients at random and repeatedly retest the result. Such methods to carefully evaluate the performance of machine learning models will be discussed further in *Chapter 10, Evaluating Model Performance*.

Summary

In this chapter, we learned about classification using k-NN. Unlike many classification algorithms, k-nearest neighbors does not do any learning—at least not according to the formal definition of machine learning. Instead, it simply stores the training data verbatim. Unlabeled test examples are then matched to the most similar records in the training set using a distance function, and the unlabeled example is assigned the label of its nearest neighbors.

Although k-NN is a very simple algorithm, it can tackle extremely complex tasks, such as the identification of cancerous masses. In a few simple lines of R code, we were able to correctly identify whether a mass was malignant or benign 98 percent of the time in an example using real-world data. Although this teaching dataset was designed to streamline the process of building a model, the exercise demonstrated the ability of learning algorithms to make accurate predictions much like a human can.

In the next chapter, we will examine a classification method that uses probability to estimate the likelihood that an observation falls into certain categories. It will be interesting to compare how this approach differs from k-NN. Later, in *Chapter 9, Finding Groups of Data – Clustering with k-means*, we will learn about a close relative to k-NN, which uses distance measures for a completely different learning task.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



4

Probabilistic Learning – Classification Using Naive Bayes

When a meteorologist provides a weather forecast, precipitation is typically described with phrases like “70 percent chance of rain.” Such forecasts are known as probability of precipitation reports. Have you ever considered how they are calculated? It is a puzzling question because, in reality, it will either rain or not with absolute certainty.

Weather estimates are based on probabilistic methods, which are those concerned with describing uncertainty. They use data on past events to extrapolate future events. In the case of the weather, the chance of rain describes the proportion of prior days with similar atmospheric conditions on which precipitation occurred. A 70 percent chance of rain implies that in 7 out of 10 past cases with similar conditions, precipitation occurred somewhere in the area.

This chapter covers the Naive Bayes algorithm, which uses probabilities in much the same way as a weather forecast. While studying this method, you will learn about:

- Basic principles of probability
- The specialized methods and data structures needed to analyze text data with R
- How to employ Naive Bayes to build a **Short Message Service (SMS)** junk message filter

If you’ve taken a statistics class before, some of the material in this chapter may be a review. Even so, it may be helpful to refresh your knowledge of probability. You will find out that these principles are the basis of how Naive Bayes got such a strange name.

Understanding Naive Bayes

The basic statistical ideas necessary to understand the Naive Bayes algorithm have existed for centuries. The technique descended from the work of the 18th-century mathematician Thomas Bayes, who developed foundational principles for describing the probability of events and how these probabilities should be revised in light of additional information. These principles formed the foundation for what are now known as **Bayesian methods**.

We will cover these methods in greater detail later. For now, it suffices to say that a probability is a number between zero and one (or from 0 to 100 percent) that captures the chance that an event will occur in light of the available evidence. The lower the probability, the less likely the event is to occur. A probability of zero indicates that the event will definitely not occur, while a probability of one indicates that the event will occur with absolute certainty. Life's most interesting events tend to be those with uncertain probability; estimating the chance that they will occur helps us make better decisions by revealing the most likely outcomes.

Classifiers based on Bayesian methods utilize training data to calculate the probability of each outcome based on the evidence provided by feature values. When the classifier is later applied to unlabeled data, it uses these calculated probabilities to predict the most likely class for the new example. It's a simple idea, but it results in a method that can have results on par with more sophisticated algorithms. In fact, Bayesian classifiers have been used for:

- Text classification, such as junk email (spam) filtering
- Intrusion or anomaly detection in computer networks
- Diagnosing medical conditions given a set of observed symptoms

Typically, Bayesian classifiers are best applied to problems for which the information from numerous attributes should be considered simultaneously to estimate the overall probability of an outcome. While many machine learning algorithms ignore features that have weak effects, Bayesian methods utilize all available evidence to subtly change the predictions. This implies that even if a large portion of features have relatively minor effects, their combined impact in a Bayesian model could be quite large.

Basic concepts of Bayesian methods

Before jumping into the Naive Bayes algorithm, it's worth spending some time defining the concepts that are used across Bayesian methods. Summarized in a single sentence, Bayesian probability theory is rooted in the idea that the estimated likelihood of an **event**, or potential outcome, should be based on the evidence at hand across multiple **trials**, or opportunities for the event to occur.

The following table illustrates events and trials for several real-world outcomes:

Event	Trial
Heads result	A coin flip
Rainy weather	A single day (or another time period)
Message is spam	An incoming email message
Candidate becomes president	A presidential election
Mortality	A hospital patient
Winning the lottery	A lottery ticket

Bayesian methods provide insights into how the probability of these events can be estimated from observed data. To see how, we'll need to formalize our understanding of probability.

Understanding probability

The probability of an event is estimated from observed data by dividing the number of trials in which the event occurred by the total number of trials. For instance, if it rained 3 out of 10 days with similar conditions as today, the probability of rain today can be estimated as $3 / 10 = 0.30$ or 30 percent. Similarly, if 10 out of 50 prior email messages were spam, then the probability of any incoming message being spam can be estimated as $10 / 50 = 0.20$ or 20 percent.

To denote these probabilities, we use notation in the form $P(A)$, which signifies the probability of event A . For example, $P(\text{rain}) = 0.30$ to indicate a 30 percent chance of rain or $P(\text{spam}) = 0.20$ to describe a 20 percent probability of an incoming message being spam.

Because a trial always results in some outcome happening, the probability of all possible outcomes of a trial must always sum to one. Thus, if the trial has exactly two outcomes and the outcomes cannot occur simultaneously, then knowing the probability of either outcome reveals the probability of the other. This is the case for many outcomes, such as heads or tails coin flips, or spam versus legitimate email messages, also known as “ham.” Using this principle, knowing that $P(\text{spam}) = 0.20$ allows us to calculate $P(\text{ham}) = 1 - 0.20 = 0.80$. This only works because spam and ham are **mutually exclusive and exhaustive events**, which implies that they cannot occur at the same time and are the only possible outcomes.

A single event cannot happen and not happen simultaneously. This means an event is always mutually exclusive and exhaustive with its **complement**, or the event comprising all other outcomes in which the event of interest does not happen. The complement of event A is typically denoted A^c or A' .

Additionally, the shorthand notation $P(A^c)$ or $P(\neg A)$ can be used to denote the probability of event A not occurring. For example, the notation $P(\neg \text{spam}) = 0.80$ suggests that the probability of a message not being spam is 80%.

To illustrate events and their complements, it is often helpful to imagine a two-dimensional space that is partitioned into probabilities for each event. In the following diagram, the rectangle represents the possible outcomes for an email message. The circle represents the 20 percent probability that the message is spam. The remaining 80 percent represents the complement $P(\neg \text{spam})$, or the messages that are not spam:

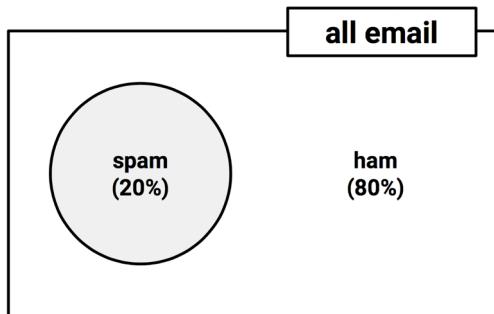


Figure 4.1: The probability space for all emails can be visualized as partitions of spam and ham

Understanding joint probability

Often, we are interested in monitoring several non-mutually exclusive events in the same trial. If certain events occur concurrently with the event of interest, we may be able to use them to make predictions. Consider, for instance, a second event based on the outcome that an email message contains the word *Viagra*. The preceding diagram, updated for this second event, might appear as shown in the following diagram:

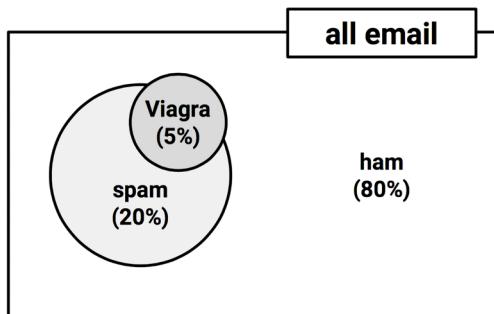


Figure 4.2: Non-mutually exclusive events are depicted as overlapping partitions

Notice in the diagram that the Viagra circle overlaps with both the spam and ham areas of the diagram and the spam circle includes an area not covered by the Viagra circle. This implies that not all spam messages contain the term Viagra and some messages with the term Viagra are ham. However, because this word appears very rarely outside spam, its presence in a new incoming message would be strong evidence that the message is spam.

To zoom in for a closer look at the overlap between these circles, we'll employ a visualization known as a **Venn diagram**. First used in the late 19th century by mathematician John Venn, the diagram uses circles to illustrate the overlap between sets of items. As in most Venn diagrams, the size and degree of overlap of the circles in the depiction is not meaningful. Instead, it is used as a reminder to allocate probability to all combinations of events. A Venn diagram for spam and Viagra might be depicted as follows:

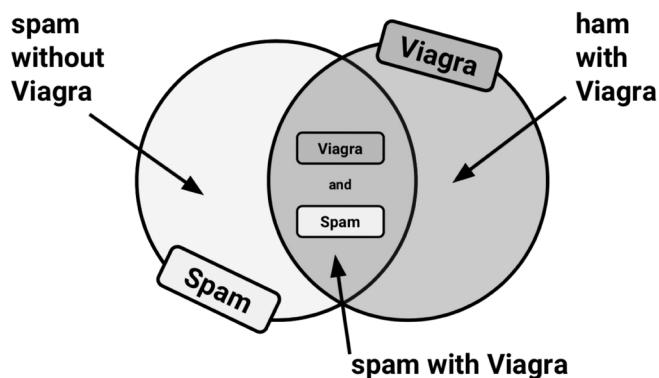


Figure 4.3: A Venn diagram illustrates the overlap of the spam and Viagra events

We know that 20 percent of all messages were spam (the left circle), and 5 percent of all messages contained the word *Viagra* (the right circle). We would like to quantify the degree of overlap between these two proportions. In other words, we hope to estimate the probability that both $P(\text{spam})$ and $P(\text{Viagra})$ occur, which can be written as $P(\text{spam} \cap \text{Viagra})$. The \cap symbol signifies the intersection of the two events; the notation $A \cap B$ refers to the event in which both A and B occur.

Calculating $P(\text{spam} \cap \text{Viagra})$ depends on the **joint probability** of the two events, or how the probability of one event is related to the probability of the other. If the two events are totally unrelated, they are called **independent events**. This is not to say that independent events cannot occur at the same time; event independence simply implies that knowing the outcome of one event does not provide any information about the outcome of the other. For instance, the outcome of a heads result on a coin flip is independent of whether the weather is rainy or sunny on any given day.

If all events were independent, it would be impossible to predict one event by observing another. In other words, **dependent events** are the basis of predictive modeling. Just as the presence of clouds is predictive of a rainy day, the appearance of the word *Viagra* is predictive of a spam email.

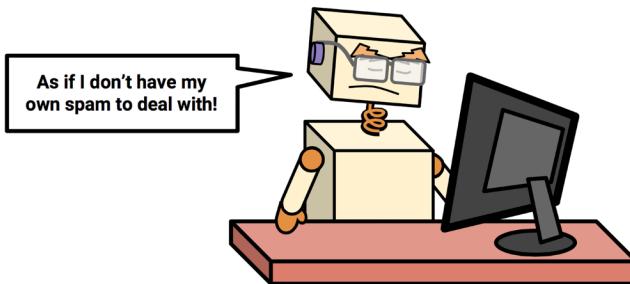


Figure 4.4: Dependent events are required for machines to learn how to identify useful patterns

Calculating the probability of dependent events is a bit more complex than for independent events. If $P(\text{spam})$ and $P(\text{Viagra})$ were independent, we could easily calculate $P(\text{spam} \cap \text{Viagra})$, the probability of both events happening at the same time. Because 20 percent of all messages are spam, and 5 percent of all emails contain the word *Viagra*, we could assume that 1 percent of all messages with the term *Viagra* are spam. This is because $0.05 * 0.20 = 0.01$. More generally, for independent events A and B , the probability of both happening can be computed as $P(A \cap B) = P(A) * P(B)$.

That said, we know that $P(\text{spam})$ and $P(\text{Viagra})$ are likely to be highly dependent, which means that this calculation is incorrect. To obtain a more reasonable estimate, we need to use a more careful formulation of the relationship between these two events, which is based on more advanced Bayesian methods.

Computing conditional probability with Bayes' theorem

The relationships between dependent events can be described using **Bayes' theorem**, which provides a way of thinking about how to revise an estimate of the probability of one event in light of the evidence provided by another. One formulation is as follows:

$$P(A | B) = \frac{P(A \cap B)}{P(B)}$$

The notation $P(A|B)$ is read as the probability of event A given that event B occurred. This is known as **conditional probability** since the probability of A is dependent (that is, conditional) on what happened with event B .

Bayes' theorem states that the best estimate of $P(A|B)$ is the proportion of trials in which A occurred with B , out of all the trials in which B occurred. This implies that the probability of event A is higher if A and B often occur together each time B is observed. Note that this formula adjusts $P(A \cap B)$ for the probability of B occurring. If B is extremely rare, $P(B)$ and $P(A \cap B)$ will always be small; however, if A almost always happens together with B , $P(A|B)$ will still be high in spite of B 's rarity.

By definition, $P(A \cap B) = P(A|B) * P(B)$, a fact that can be easily derived by applying a bit of algebra to the previous formula. Rearranging this formula once more with the knowledge that $P(A \cap B) = P(B \cap A)$ results in the conclusion that $P(A|B) = P(B|A) * P(A)$, which we can then use in the following formulation of Bayes' theorem:

$$P(A | B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B | A) P(A)}{P(B)}$$

In fact, this is the traditional formulation of Bayes' theorem for reasons that will become clear as we apply it to machine learning. First, to better understand how Bayes' theorem works in practice, let's revisit our hypothetical spam filter.

Without knowledge of an incoming message's content, the best estimate of its spam status would be $P(spam)$, the probability that any prior message was spam. This estimate is known as the **prior probability**. We found this previously to be 20 percent.

Suppose that you obtained additional evidence by looking more carefully at the set of previously received messages and examining the frequency with which the term *Viagra* appeared. The probability that the word *Viagra* was used in previous spam messages, or $P(Viagra|spam)$, is called the **likelihood**. The probability that *Viagra* appeared in any message at all, or $P(Viagra)$, is known as the **marginal likelihood**.

By applying Bayes' theorem to this evidence, we can compute a **posterior probability** that measures how likely a message is to be spam. If the posterior probability is greater than 50 percent, the message is more likely to be spam than ham, and it should perhaps be filtered. The following formula shows how Bayes' theorem is applied to the evidence provided by previous email messages:

$$P(\text{spam}|\text{Viagra}) = \frac{P(\text{Viagra}|\text{spam})P(\text{spam})}{P(\text{Viagra})}$$

posterior probability likelihood prior probability
 marginal likelihood

Figure 4.5: Bayes' theorem acting on previously received emails

To calculate the components of Bayes' theorem, it helps to construct a **frequency table** (shown on the left in the tables that follow) recording the number of times *Viagra* appeared in spam and ham messages. Just like a two-way cross-tabulation, one dimension of the table indicates levels of the class variable (spam or ham), while the other dimension indicates levels for features (*Viagra*: yes or no). The cells then indicate the number of instances that have the specified combination of the class value and feature value.

The frequency table can then be used to construct a **likelihood table**, as shown on the right in the following tables. The rows of the likelihood table indicate the conditional probabilities for *Viagra* (yes/no), given that an email was either spam or ham.

	Viagra			
Frequency	Yes	No	Total	
spam	4	16	20	
ham	1	79	80	
Total	5	95	100	

	Viagra		
Likelihood	Yes	No	Total
spam	4 / 20	16 / 20	20
ham	1 / 80	79 / 80	80
Total	5 / 100	95 / 100	100

Figure 4.6: Frequency and likelihood tables are the basis for computing the posterior probability of spam

The likelihood table reveals that $P(Viagra=Yes|spam) = 4 / 20 = 0.20$, indicating that there is a 20 percent probability that a message contains the term *Viagra* given that the message is spam. Additionally, since $P(A \cap B) = P(B|A) * P(A)$, we can calculate $P(spam \cap Viagra)$ as $P(Viagra|spam) * P(spam) = (4/20) * (20/100) = 0.04$. The same result can be found in the frequency table, which notes that 4 out of 100 messages were spam and contained the term *Viagra*. Either way, this is four times greater than the previous estimate of 0.01 we calculated as $P(A \cap B) = P(A) * P(B)$ under the false assumption of independence. This, of course, illustrates the importance of Bayes' theorem for estimating joint probability.

To compute the posterior probability, $P(\text{spam}|\text{Viagra})$, we simply take $P(\text{Viagra}|\text{spam}) * P(\text{spam}) / P(\text{Viagra})$, or $(4/20) * (20/100) / (5/100) = 0.80$. Therefore, the probability that a message is spam is 80 percent given that it contains the word *Viagra*. In light of this finding, any message containing this term should probably be filtered.

This is very much how commercial spam filters work, although they consider a much larger number of words simultaneously when computing the frequency and likelihood tables. In the next section, we'll see how this method can be adapted to accommodate cases when additional features are involved.

The Naive Bayes algorithm

The **Naive Bayes** algorithm defines a simple method for applying Bayes' theorem to classification problems. Although it is not the only machine learning method that utilizes Bayesian methods, it is the most common. It grew in popularity due to its successes in text classification, where it was once the de facto standard. The strengths and weaknesses of this algorithm are as follows:

Strengths	Weaknesses
<ul style="list-style-type: none">• Simple, fast, and very effective• Does well with noisy and missing data and large numbers of features• Requires relatively few examples for training• Easy to obtain the estimated probability for a prediction	<ul style="list-style-type: none">• Relies on an often-faulty assumption of equally important and independent features• Not ideal for datasets with many numeric features• Estimated probabilities are less reliable than the predicted classes

The Naive Bayes algorithm is named as such because it makes some so-called “naive” assumptions about the data. In particular, Naive Bayes assumes that all of the features in the dataset are **equally important and independent**. These assumptions are rarely true in most real-world applications.

For example, when attempting to identify spam by monitoring email messages, it is almost certainly true that some features will be more important than others. For example, the email sender may be a more important indicator of spam than the message text. Additionally, the words in the message body are not independent of one another, since the appearance of some words is a very good indication that other words are also likely to appear. A message with the word *Viagra* will probably also contain the word *prescription* or *drugs*.

However, in most cases, even when these assumptions are violated, Naive Bayes still performs surprisingly well. This is true even in circumstances where strong dependencies are found among the features. Due to the algorithm's versatility and accuracy across many types of conditions, particularly with smaller training datasets, Naive Bayes is often a reasonable baseline candidate for classification learning tasks.



The exact reason why Naive Bayes works well in spite of its faulty assumptions has been the subject of much speculation. One explanation is that it is not important to obtain a precise estimate of probability so long as the predictions are accurate. For instance, if a spam filter correctly identifies spam, does it matter whether the predicted probability of spam was 51 percent or 99 percent? For one discussion of this topic, refer to *On the Optimality of the Simple Bayesian Classifier under Zero-One Loss*, Domingos, P. and Pazzani, M., *Machine Learning*, 1997, Vol. 29, pp. 103-130.

Classification with Naive Bayes

Let's extend our spam filter by adding a few additional terms to be monitored in addition to the term *Viagra*: *money*, *groceries*, and *unsubscribe*. The Naive Bayes learner is trained by constructing a likelihood table for the appearance of these four words (labeled W^1 , W^2 , W^3 , and W^4), as shown in the following diagram for 100 emails:

	Viagra (W_1)		Money (W_2)		Groceries (W_3)		Unsubscribe (W_4)		
Likelihood	Yes	No	Yes	No	Yes	No	Yes	No	Total
spam	4 / 20	16 / 20	10 / 20	10 / 20	0 / 20	20 / 20	12 / 20	8 / 20	20
ham	1 / 80	79 / 80	14 / 80	66 / 80	8 / 80	71 / 80	23 / 80	57 / 80	80
Total	5 / 100	95 / 100	24 / 100	76 / 100	8 / 100	91 / 100	35 / 100	65 / 100	100

Figure 4.7: An expanded table adds likelihoods for additional terms in spam and ham messages

As new messages are received, we need to calculate the posterior probability to determine whether they are more likely spam or ham, given the likelihood of the words being found in the message text. For example, suppose that a message contains the terms *Viagra* and *unsubscribe* but does not contain either *money* or *groceries*.

Using Bayes' theorem, we can define the problem as shown in the following formula. This computes the probability that a message is spam given that *Viagra* = Yes, *Money* = No, *Groceries* = No, and *Unsubscribe* = Yes:

$$P(\text{spam} | W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4) = \frac{P(W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4 | \text{spam})P(\text{spam})}{P(W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4)}$$

For two reasons, this formula is computationally difficult to solve. First, as additional features are added, tremendous amounts of memory are needed to store the probabilities for all possible intersecting events. Imagine the complexity of a Venn diagram for the events for four words, let alone for hundreds or more. Second, many of these potential intersections will never have been observed in past data, which would lead to a joint probability of zero and problems that will become clear later.

The computation becomes more reasonable if we exploit the fact that Naive Bayes makes the naive assumption of independence among events. Specifically, it assumes **class-conditional independence**, which means that events are independent so long as they are conditioned on the same class value. The conditional independence assumption allows us to use the probability rule for independent events, which states that $P(A \cap B) = P(A) * P(B)$. This simplifies the numerator by allowing us to multiply the individual conditional probabilities rather than computing a complex conditional joint probability.

Lastly, because the denominator does not depend on the target class (spam or ham), it is treated as a constant value and can be ignored for the time being. This means that the conditional probability of spam can be expressed as:

$$P(\text{spam} | W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4) \propto P(W_1 | \text{spam})P(\neg W_2 | \text{spam})P(\neg W_3 | \text{spam})P(W_4 | \text{spam})P(\text{spam})$$

And the probability that the message is ham can be expressed as:

$$P(\text{ham} | W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4) \propto P(W_1 | \text{ham})P(\neg W_2 | \text{ham})P(\neg W_3 | \text{ham})P(W_4 | \text{ham})P(\text{ham})$$

Note that the equals symbol has been replaced by the proportional-to symbol (similar to a sideways, open-ended “8”) to indicate the fact that the denominator has been omitted.

Using the values in the likelihood table, we can start filling in numbers in these equations. The overall likelihood of spam is then:

$$(4 / 20) * (10 / 20) * (20 / 20) * (12 / 20) * (20 / 100) = 0.012$$

While the likelihood of ham is:

$$(1 / 80) * (66 / 80) * (71 / 80) * (23 / 80) * (80 / 100) = 0.002$$

Because $0.012 / 0.002 = 6$, we can say that this message is 6 times more likely to be spam than ham. However, to convert these numbers into probabilities, we need one last step to reintroduce the denominator that has been excluded. Essentially, we must re-scale the likelihood of each outcome by dividing it by the total likelihood across all possible outcomes.

In this way, the probability of spam is equal to the likelihood that the message is spam divided by the likelihood that the message is either spam or ham:

$$0.012 / (0.012 + 0.002) = 0.857$$

Similarly, the probability of ham is equal to the likelihood that the message is ham divided by the likelihood that the message is either spam or ham:

$$0.002 / (0.012 + 0.002) = 0.143$$

Given the pattern of words found in this message, we expect that the message is spam with an 85.7 percent probability, and ham with a 14.3 percent probability. Because these are mutually exclusive and exhaustive events, the probabilities sum up to 1.

The Naive Bayes classification algorithm used in the preceding example can be summarized by the following formula. The probability of level L for class C , given the evidence provided by features F_1 through F_n , is equal to the product of the probabilities of each piece of evidence conditioned on the class level, the prior probability of the class level, and a scaling factor $1/Z$, which converts the likelihood values into probabilities. This is formulated as:

$$P(C_L | F_1, \dots, F_n) = \frac{1}{Z} p(C_L) \prod_{i=1}^n p(F_i | C_L)$$

Although this equation seems intimidating, as the spam filtering example illustrated, the series of steps is fairly straightforward. Begin by building a frequency table, use this to build a likelihood table, and multiply out the conditional probabilities with the naive assumption of independence. Finally, divide by the total likelihood to transform each class likelihood into a probability. After attempting this calculation a few times by hand, it will become second nature.

The Laplace estimator

Before we employ Naive Bayes on more complex problems, there are some nuances to consider. Suppose we received another message, this time containing all four terms: *Viagra*, *groceries*, *money*, and *unsubscribe*. Using the Naive Bayes algorithm as before, we can compute the likelihood of spam as:

$$(4/20) * (10/20) * (0/20) * (12/20) * (20/100) = 0$$

And the likelihood of ham is:

$$(1/80) * (14/80) * (8/80) * (23/80) * (80/100) = 0.00005$$

Therefore, the probability of spam is:

$$0 / (0 + 0.00005) = 0$$

And the probability of ham is:

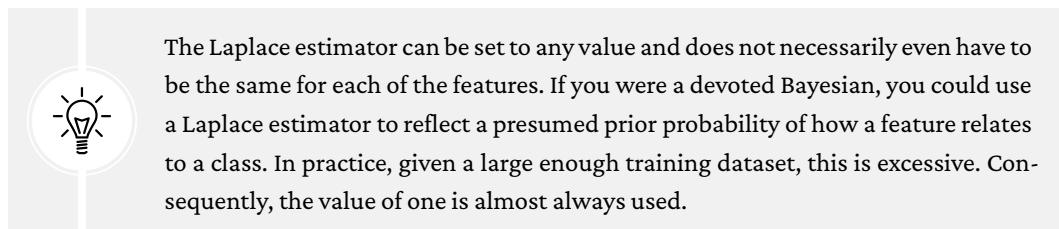
$$0.00005 / (0 + 0.00005) = 1$$

These results suggest that the message is spam with 0 percent probability and ham with 100 percent probability. Does this prediction make sense? Probably not. The message contains several words usually associated with spam, including *Viagra*, which is rarely used in legitimate messages. It is therefore very likely that the message has been incorrectly classified.

This problem arises if an event never occurs for one or more levels of the class and therefore the resulting likelihoods are zero. For instance, the term *groceries* had never previously appeared in a spam message. Consequently, $P(\text{groceries}/\text{spam}) = 0\%$.

Now, because probabilities in the Naive Bayes formula are multiplied in a chain, this zero-percent value causes the posterior probability of spam to be zero, giving the word *groceries* the ability to effectively nullify and overrule all of the other evidence. Even if the email was otherwise overwhelmingly expected to be spam, the absence of the word *groceries* in spam will always veto the other evidence and result in the probability of spam being zero.

A solution to this problem involves using something called the **Laplace estimator**, which is named after the French mathematician Pierre-Simon Laplace. The Laplace estimator adds a small number to each of the counts in the frequency table, which ensures that each feature has a non-zero probability of occurring with each class. Typically, the Laplace estimator is set to one, which ensures that each class-feature combination is found in the data at least once.



The Laplace estimator can be set to any value and does not necessarily even have to be the same for each of the features. If you were a devoted Bayesian, you could use a Laplace estimator to reflect a presumed prior probability of how a feature relates to a class. In practice, given a large enough training dataset, this is excessive. Consequently, the value of one is almost always used.

Let's see how this affects our prediction for this message. Using a Laplace value of 1, we add 1 to each numerator in the likelihood function. Then, we need to add 4 to each conditional probability denominator to compensate for the 4 additional values added to the numerator. The likelihood of spam is therefore:

$$(5/24) * (11/24) * (1/24) * (13/24) * (20/100) = 0.0004$$

And the likelihood of ham is:

$$(2/84) * (15/84) * (9/84) * (24/84) * (80/100) = 0.0001$$

By computing $0.0004 / (0.0004 + 0.0001)$, we find that the probability of spam is 80 percent and therefore the probability of ham is about 20 percent. This is a more plausible result than the $P(\text{spam}) = 0$ computed when the term *groceries* alone determined the result.



Although the Laplace estimator was added to the numerator and denominator of the likelihoods, it was not added to the prior probabilities—the values of 20/100 and 80/100. This is because our best estimate of the overall probability of spam and ham remains at 20% and 80% respectively given what was observed in the data.

Using numeric features with Naive Bayes

Naive Bayes uses frequency tables for learning the data, which means that each feature must be categorical in order to create the combinations of class and feature values comprising the matrix. Since numeric features do not have categories of values, the preceding algorithm does not work directly with numeric data. There are, however, ways that this can be addressed.

One easy and effective solution is to **discretize** numeric features, which simply means that the numbers are put into categories known as **bins**. For this reason, discretization is also sometimes called **binning**. This method works best when there are large amounts of training data.

There are several different ways to discretize a numeric feature. Perhaps the most common is to explore the data for natural categories or **cut points** in the distribution. For example, suppose that you added a feature to the spam dataset that recorded the time of day or night the email was sent, from 0 to 24 hours past midnight. Depicted using a histogram, the time data might look something like the following diagram:

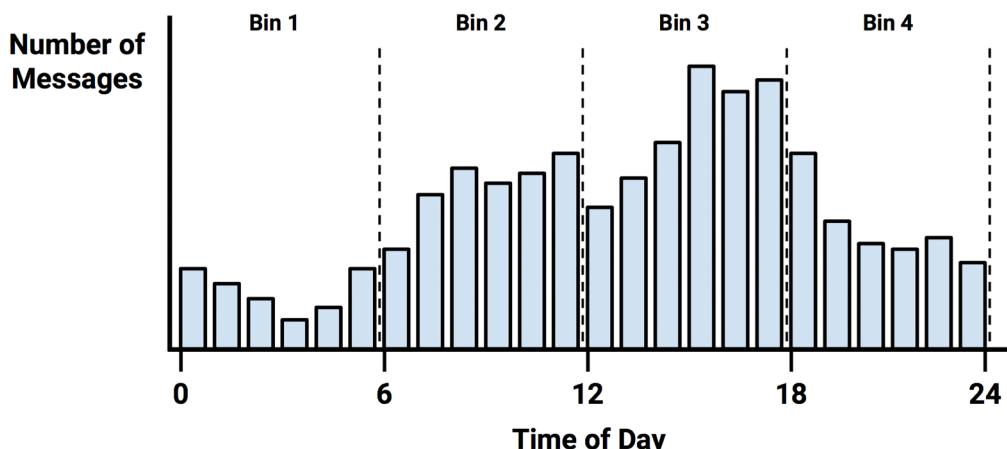


Figure 4.8: A histogram visualizing the distribution of the time emails were received

In the early hours of the morning, message frequency is low. Activity picks up during business hours and tapers off in the evening. This creates four natural bins of activity, as partitioned by the dashed lines. These indicate places where the numeric data could be divided into levels to create a new categorical feature, which could then be used with Naive Bayes.

The choice of four bins was based on the natural distribution of data and a hunch about how the proportion of spam might change throughout the day. We might expect that spammers operate in the late hours of the night, or they may operate during the day when people are likely to check their email. That said, to capture these trends, we could have just as easily used three bins or twelve.



If there are no obvious cut points, one option is to discretize the feature using quantiles, which were introduced in *Chapter 2, Managing and Understanding Data*. You could divide the data into three bins with tertiles, four bins with quartiles, or five bins with quintiles.

One thing to keep in mind is that discretizing a numeric feature always results in a reduction of information, as the feature's original granularity is reduced to a smaller number of categories. It is important to strike a balance. Too few bins can result in important trends being obscured. Too many bins can result in small counts in the Naive Bayes frequency table, which can increase the algorithm's sensitivity to noisy data.

Example – filtering mobile phone spam with the Naive Bayes algorithm

As the worldwide use of mobile phones has grown, a new avenue for electronic junk mail has opened for disreputable marketers. These advertisers utilize SMS text messages to target potential consumers with unwanted advertising known as SMS spam. This type of spam is troublesome because, unlike email spam, an SMS message is particularly disruptive due to the omnipresence of one's mobile phone. Developing a classification algorithm that could filter SMS spam would provide a useful tool for cellular phone providers.

Since Naive Bayes has been used successfully for email spam filtering, it seems likely that it could also be applied to SMS spam. However, relative to email spam, SMS spam poses additional challenges for automated filters. SMS messages are often limited to 160 characters, reducing the amount of text that can be used to identify whether a message is junk. The limit, combined with small mobile phone keyboards, has led many to adopt a form of SMS shorthand lingo, which further blurs the line between legitimate messages and spam. Let's see how a simple Naive Bayes classifier handles these challenges.

Step 1 – collecting data

To develop the Naive Bayes classifier, we will use data adapted from the SMS Spam Collection at <https://www.dt.fee.unicamp.br/~tiago/smsspamcollection/>.



To read more about how the SMS Spam Collection was developed, refer to *On the Validity of a New SMS Spam Collection*, Gómez, J. M., Almeida, T. A., and Yamakami, A., *Proceedings of the 11th IEEE International Conference on Machine Learning and Applications*, 2012.

This dataset includes the text of SMS messages, along with a label indicating whether the message is unwanted. Junk messages are labeled spam, while legitimate messages are labeled ham. Some examples of spam and ham are shown in the following table:

Sample SMS ham	Sample SMS spam
<ul style="list-style-type: none"> • Better. Made up for Friday and stuffed myself like a pig yesterday. Now I feel bleh. But at least its not writhing pain kind of bleh. • If he started searching he will get job in few days. he have great potential and talent. • I got another job! The one at the hospital doing data analysis or something, starts on monday! Not sure when my thesis will got finished 	<ul style="list-style-type: none"> • Congratulations ur awarded 500 of CD vouchers or 125gift guaranteed & Free entry 2 100 wkly draw txt MUSIC to 87066 • December only! Had your mobile 11mths+? You are entitled to update to the latest colour camera mobile for Free! Call The Mobile Update Co FREE on 08002986906 • Valentines Day Special! Win over £1000 in our quiz and take your partner on the trip of a lifetime! Send GO to 83600 now. 150p/msg rcvd.

Looking at the preceding messages, do you notice any distinguishing characteristics of spam? One notable characteristic is that two of the three spam messages use the word *free*, yet this word does not appear in any of the ham messages. On the other hand, two of the ham messages cite specific days of the week, as compared to zero in spam messages.

Our Naive Bayes classifier will take advantage of such patterns in the word frequency to determine whether the SMS messages seem to better fit the profile of spam or ham. While it's not inconceivable that the word *free* would appear outside of a spam SMS, a legitimate message is likely to provide additional words giving context.

For instance, a ham message might ask, “Are you free on Sunday?” whereas a spam message might use the phrase “free ringtones.” The classifier will compute the probability of spam and ham given the evidence provided by all the words in the message.

Step 2 – exploring and preparing the data

The first step toward constructing our classifier involves processing the raw data for analysis. Text data is challenging to prepare because it is necessary to transform the words and sentences into a form that a computer can understand. We will transform our SMS data into a representation known as **bag-of-words**, which provides a binary feature indicating whether each word appears in the given example while ignoring word order or the context in which the word appears. Although this is a relatively simple representation, as we will soon demonstrate, it performs well enough for many classification tasks.



The dataset used here has been modified slightly from the original to make it easier to work with in R. If you plan on following along with the example, download the `sms_spam.csv` file from the Packt website and save it to your R working directory.

We'll begin by importing the CSV data and saving it to a data frame:

```
> sms_raw <- read.csv("sms_spam.csv")
```

Using the `str()` function, we see that the `sms_raw` data frame includes 5,559 total SMS messages with two features: `type` and `text`. The SMS type has been coded as either ham or spam. The `text` element stores the full raw SMS message text:

```
> str(sms_raw)
```

```
'data.frame': 5559 obs. of 2 variables:  
 $ type: chr "ham" "ham" "ham" "spam" ...  
 $ text: chr "Hope you are having a good week. Just checking in"  
 "K..give back my thanks." "Am also doing in cbe only. But have to  
 pay." "complimentary 4 STAR Ibiza Holiday or £10,000 cash needs your  
 URGENT collection. 09066364349 NOW from Landline not to lose out" |  
 __truncated__ ...
```

The `type` element is currently a character vector. Since this is a categorical variable, it would be better to convert it into a factor, as shown in the following code:

```
> sms_raw$type <- factor(sms_raw$type)
```

Examining this with the `str()` and `table()` functions, we see that `type` has now been appropriately recoded as a factor. Additionally, we see that 747 (about 13 percent) of SMS messages in our data were labeled as `spam`, while the others were labeled as `ham`:

```
> str(sms_raw$type)
Factor w/ 2 levels "ham","spam": 1 1 1 2 2 1 1 1 2 1 ...
> table(sms_raw$type)

  ham  spam
4812 747
```

For now, we will leave the message text alone. As you will learn in the next section, processing raw SMS messages will require the use of a new set of powerful tools designed specifically for processing text data.

Data preparation – cleaning and standardizing text data

SMS messages are strings of text composed of words, spaces, numbers, and punctuation. Handling this type of complex data takes a large amount of thought and effort. One needs to consider how to remove numbers and punctuation; handle uninteresting words, such as *and*, *but*, and *or*; and break apart sentences into individual words. Thankfully, this functionality has been provided by members of the R community in a text-mining package titled `tm`.



The `tm` package was originally created by Ingo Feinerer as a dissertation project at the Vienna University of Economics and Business. To learn more, see *Text Mining Infrastructure in R*, Feinerer, I., Hornik, K., and Meyer, D., *Journal of Statistical Software*, 2008, Vol. 25, pp. 1-54.

The `tm` package can be installed via the `install.packages("tm")` command and loaded with the `library(tm)` command. Even if you already have it installed, it may be worth redoing the installation to ensure that your version is up to date, as the `tm` package is still under active development. This occasionally results in changes to its functionality.



This chapter was tested using `tm` version 0.7-11, which was current as of May 2023. If you see differences in the output or if the code does not work, you may be using a different version. The Packt support page for this book, as well as its GitHub repository, will post solutions for future `tm` package versions if significant changes are noted.

The first step in processing text data involves creating a **corpus**, which is a collection of text documents. The documents can be short or long, from individual news articles, pages in a book, pages from the web, or even entire books. In our case, the corpus will be a collection of SMS messages.

To create a corpus, we'll use the `VCorpus()` function in the `tm` package, which refers to a volatile corpus—the term “volatile” meaning that it is stored in memory as opposed to being stored on disk (the `PCorpus()` function is used to access a permanent corpus stored in a database). This function requires us to specify the source of documents for the corpus, which could be a computer's filesystem, a database, the web, or elsewhere.

Since we already loaded the SMS message text into R, we'll use the `VectorSource()` reader function to create a source object from the existing `sms_raw$text` vector, which can then be supplied to `VCorpus()` as follows:

```
> sms_corpus <- VCorpus(VectorSource(sms_raw$text))
```

The resulting corpus object is saved with the name `sms_corpus`.



By specifying an optional `readerControl` parameter, the `VCorpus()` function can be used to import text from sources such as PDFs and Microsoft Word files. To learn more, examine the *Data Import* section in the `tm` package vignette using the `vignette("tm")` command.

By printing the corpus, we see that it contains documents for each of the 5,559 SMS messages in the training data:

```
> print(sms_corpus)

<<VCorpus>>
Metadata: corpus specific: 0, document level (indexed): 0
Content: documents: 5559
```

Now, because the `tm` corpus is essentially a complex list, we can use list operations to select documents in the corpus. The `inspect()` function shows a summary of the result. For example, the following command will view a summary of the first and second SMS messages in the corpus:

```
> inspect(sms_corpus[1:2])

<<VCorpus>>
Metadata: corpus specific: 0, document level (indexed): 0
Content: documents: 2
```

```
[[1]]
<<PlainTextDocument>>
Metadata: 7
Content: chars: 49

[[2]]
<<PlainTextDocument>>
Metadata: 7
Content: chars: 23
```

To view the actual message text, the `as.character()` function must be applied to the desired messages. To view one message, use the `as.character()` function on a single list element, noting that the double-bracket notation is required:

```
> as.character(sms_corpus[[1]])

[1] "Hope you are having a good week. Just checking in"
```

To view multiple documents, we'll need to apply `as.character()` to several items in the `sms_corpus` object. For this, we'll use the `lapply()` function, which is part of a family of R functions that applies a procedure to each element of an R data structure. These functions, which include `apply()` and `sapply()` among others, are one of the key idioms of the R language. Experienced R coders use these much like the way `for` or `while` loops are used in other programming languages, as they result in more readable (and sometimes more efficient) code. The `lapply()` function for applying `as.character()` to a subset of corpus elements is as follows:

```
> lapply(sms_corpus[1:2], as.character)

$'1'
[1] "Hope you are having a good week. Just checking in"

$'2'
[1] "K..give back my thanks."
```

As noted earlier, the corpus contains the raw text of 5,559 text messages. To perform our analysis, we need to divide these messages into individual words. First, we need to clean the text to standardize the words and remove punctuation characters that clutter the result. For example, we would like the strings `Hello!`, `HELLO`, and `hello` to be counted as instances of the same word.

The `tm_map()` function provides a method to apply a transformation (also known as a mapping) to a `tm` corpus. We will use this function to clean up our corpus using a series of transformations and save the result in a new object called `corpus_clean`.

Our first transformation will standardize the messages to use only lowercase characters. To this end, R provides a `tolower()` function that returns a lowercase version of text strings. In order to apply this function to the corpus, we need to use the `tm` wrapper function `content_transformer()` to treat `tolower()` as a transformation function that can be used to access the corpus. The full command is as follows:

```
> sms_corpus_clean <- tm_map(sms_corpus,  
+ content_transformer(tolower))
```

To check whether the command worked as expected, let's inspect the first message in the original corpus and compare it to the same in the transformed corpus:

```
> as.character(sms_corpus[[1]])  
[1] "Hope you are having a good week. Just checking in"  
  
> as.character(sms_corpus_clean[[1]])  
[1] "hope you are having a good week. just checking in"
```

As expected, uppercase letters in the clean corpus have been replaced with lowercase versions.



The `content_transformer()` function can be used to apply more sophisticated text processing and cleanup processes like `grep` pattern matching and replacement. Simply write a custom function and wrap it before applying the `tm_map()` function.

Let's continue our cleanup by removing numbers from the SMS messages. Although some numbers may provide useful information, the majority are likely to be unique to individual senders and thus will not provide useful patterns across all messages. With this in mind, we'll strip all numbers from the corpus as follows:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, removeNumbers)
```

Note that the preceding code did not use the `content_transformer()` function. This is because `removeNumbers()` is included with `tm` along with several other mapping functions that do not need to be wrapped. To see the other built-in transformations, simply type `getTransformations()`.

Our next task is to remove filler words such as *to*, *and*, *but*, and *or* from the SMS messages. These terms are known as **stop words** and are typically removed prior to text mining. This is due to the fact that although they appear very frequently, they do not provide much useful information for our model as they are unlikely to distinguish between spam and ham.

Rather than define a list of stop words ourselves, we'll use the `stopwords()` function provided by the `tm` package. This function allows us to access sets of stop words from various languages. By default, common English language stop words are used. To see the default list, type `stopwords()` at the R command prompt. To see the other languages and options available, type `?stopwords` for the documentation page.



Even within a single language, there is no single definitive list of stop words. For example, the default English list in `tm` includes about 174 words, while another option includes 571 words. You can even specify your own list of stop words. Regardless of the list you choose, keep in mind the goal of this transformation, which is to eliminate useless data while keeping as much useful information as possible.

Defining the stop words alone is not a transformation. What we need is a way to remove any words that appear in the stop words list. The solution lies in the `removeWords()` function, which is a transformation included with the `tm` package. As we have done before, we'll use the `tm_map()` function to apply this mapping to the data, providing the `stopwords()` function as a parameter to indicate the words we would like to remove. The full command is as follows:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean,
  removeWords, stopwords())
```

Since `stopwords()` simply returns a vector of stop words, if we had so chosen, we could have replaced this function call with our own vector of words to remove. In this way, we could expand or reduce the list of stop words to our liking or remove a different set of words entirely.

Continuing our cleanup process, we can also eliminate any punctuation from the text messages using the built-in `removePunctuation()` transformation:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, removePunctuation)
```

The `removePunctuation()` transformation completely strips punctuation characters from the text, which can lead to unintended consequences. For example, consider what happens when it is applied as follows:

```
> removePunctuation("hello...world")
```

```
[1] "helloworld"
```

As shown, the lack of a blank space after the ellipses caused the words `hello` and `world` to be joined as a single word. While this is not a substantial problem right now, it is worth noting for the future.

To work around the default behavior of `removePunctuation()`, it is possible to create a custom function that replaces rather than removes punctuation characters:

```
> replacePunctuation <- function(x) {  
  gsub("[[:punct:]]+", " ", x)  
}
```



This uses R's `gsub()` function to substitute any punctuation characters in `x` with a blank space. This `replacePunctuation()` function can then be used with `tm_map()` as with other transformations. The odd syntax of the `gsub()` command here is due to the use of a **regular expression**, which specifies a pattern that matches text characters. Regular expressions are covered in more detail in *Chapter 12, Advanced Data Preparation*.

Another common standardization for text data involves reducing words to their root form in a process called **stemming**. The stemming process takes words like *learned*, *learning*, and *learns* and strips the suffix in order to transform them into the base form, *learn*. This allows machine learning algorithms to treat the related terms as a single concept rather than attempting to learn a pattern for each variant.

The `tm` package provides stemming functionality via integration with the `SnowballC` package. At the time of writing, `SnowballC` is not installed by default with `tm`, so do so with `install.packages("SnowballC")` if you have not done so already.



The `SnowballC` package is maintained by Milan Bouchet-Valat and provides an R interface for the C-based `libstemmer` library, itself based on M. F. Porter's "Snowball" word-stemming algorithm, a widely used open-source stemming method. For more details, see <http://snowballstem.org>.

The `SnowballC` package provides a `wordStem()` function, which for a character vector returns the same vector of terms in its root form. For example, the function correctly stems the variants of the word *learn* as described previously:

```
> library(SnowballC)  
> wordStem(c("learn", "learned", "learning", "learns"))  
  
[1] "learn"   "learn"   "learn"   "learn"
```

To apply the `wordStem()` function to an entire corpus of text documents, the `tm` package includes a `stemDocument()` transformation. We apply this to our corpus with the `tm_map()` function exactly as before:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, stemDocument)
```



If you receive an error message when applying the `stemDocument()` transformation, please confirm that you have the `SnowballC` package installed.

After removing numbers, stop words, and punctuation, then also performing stemming, the text messages are left with the blank spaces that once separated the now-missing pieces. Therefore, the final step in our text cleanup process is to remove additional whitespace using the built-in `stripWhitespace()` transformation:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, stripWhitespace)
```

The following table shows the first three messages in the SMS corpus before and after the cleaning process. The messages have been limited to the most interesting words, and punctuation and capitalization have been removed:

SMS messages before cleaning	SMS messages after cleaning
> as.character(sms_corpus[1:3])	> as.character(sms_corpus_clean[1:3])
[[1]] Hope you are having a good week. Just checking in	[[1]] hope good week just check
[[2]] K..give back my thanks.	[[2]] kgive back thank
[[3]] Am also doing in cbe only. But have to pay.	[[3]] also cbe pay

Data preparation – splitting text documents into words

Now that the data is processed to our liking, the final step is to split the messages into individual terms through a process called **tokenization**. A token is a single element of a text string; in this case, the tokens are words.

As you might assume, the `tm` package provides functionality to tokenize the SMS message corpus. The `DocumentTermMatrix()` function takes a corpus and creates a data structure called a **document-term matrix (DTM)** in which rows indicate documents (SMS messages) and columns indicate terms (words).



The `tm` package also provides a data structure for a **term-document matrix (TDM)**, which is simply a transposed DTM in which the rows are terms and the columns are documents. Why the need for both? Sometimes, it is more convenient to work with one or the other. For example, if the number of documents is small, while the word list is large, it may make sense to use a TDM because it is usually easier to display many rows than to display many columns. That said, machine learning algorithms will generally require a DTM, as the columns are the features and the rows are the examples.

Each cell in the matrix stores a number indicating a count of the times the word represented by the column appears in the document represented by the row. The following figure depicts only a small portion of the DTM for the SMS corpus, as the complete matrix has 5,559 rows and over 7,000 columns:

message #	balloon	balls	bam	bambling	band
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

Figure 4.9: The DTM for the SMS messages is filled with mostly zeros

The fact that each cell in the table is zero implies that none of the words listed at the top of the columns appear in any of the first five messages in the corpus. This highlights the reason why this data structure is called a **sparse matrix**; the vast majority of cells in the matrix are filled with zeros. Stated in real-world terms, although each message must contain at least one word, the probability of any one word appearing in a given message is small.

Creating a DTM sparse matrix from a `tm` corpus involves a single command:

```
> sms_dtm <- DocumentTermMatrix(sms_corpus_clean)
```

This will create an `sms_dtm` object that contains the tokenized corpus using the default settings, which apply minimal additional processing. The default settings are appropriate because we have already prepared the corpus manually.

On the other hand, if we hadn't already performed the preprocessing, we could do so here by providing a list of control parameter options to override the defaults. For example, to create a DTM directly from the raw, unprocessed SMS corpus, we can use the following command:

```
> sms_dtm2 <- DocumentTermMatrix(sms_corpus, control = list(
  tolower = TRUE,
  removeNumbers = TRUE,
  stopwords = TRUE,
  removePunctuation = TRUE,
  stemming = TRUE
))
```

This applies the same preprocessing steps to the SMS corpus in the same order as done earlier. However, comparing `sms_dtm` to `sms_dtm2`, we see a slight difference in the number of terms in the matrix:

```
> sms_dtm

<<DocumentTermMatrix (documents: 5559, terms: 6559)>>
Non-/sparse entries: 42147/36419334
Sparsity           : 100%
Maximal term length: 40
Weighting          : term frequency (tf)
```

```
> sms_dtm2

<<DocumentTermMatrix (documents: 5559, terms: 6961)>>
Non-/sparse entries: 43221/38652978
Sparsity           : 100%
Maximal term length: 40
Weighting          : term frequency (tf)
```

The reason for this discrepancy has to do with a minor difference in the ordering of the preprocessing steps. The `DocumentTermMatrix()` function applies its cleanup functions to the text strings only after they have been split apart into words. Thus, it uses a slightly different stop word removal function. Consequently, some words are split differently than when they are cleaned before tokenization.



To force the two prior DTM^s to be identical, we can override the default stop words function with our own that uses the original replacement function. Simply replace `stopwords = TRUE` with the following:

```
stopwords = function(x) { removeWords(x, stopwords()) }
```

The code file for this chapter includes the full set of steps to create an identical DTM using a single function call.

The differences between these bring up an important principle of cleaning text data: the order of operations matters. With this in mind, it is very important to think through how early steps in the process are going to affect later ones. The order presented here will work in many cases, but when the process is tailored more carefully to specific datasets and use cases, it may require rethinking. For example, if there are certain terms you hope to exclude from the matrix, consider whether to search for them before or after stemming. Also, consider how the removal of punctuation—and whether the punctuation is eliminated or replaced by blank space—affects these steps.

Data preparation – creating training and test datasets

With our data prepared for analysis, we now need to split the data into training and test datasets so that after our spam classifier is built, it can be evaluated on data it has not previously seen. However, even though we need to keep the classifier blinded as to the contents of the test dataset, it is important that the split occurs after the data has been cleaned and processed. We need exactly the same preparation steps to have occurred on both the training and test datasets.

We'll divide the data into two portions: 75 percent for training and 25 percent for testing. Since the SMS messages are sorted in a random order, we can simply take the first 4,169 for training and leave the remaining 1,390 for testing. Thankfully, the DTM object acts very much like a data frame and can be split using the standard `[row, col]` operations. As our DTM stores SMS messages as rows and words as columns, we must request a specific range of rows and all columns for each:

```
> sms_dtm_train <- sms_dtm[1:4169, ]  
> sms_dtm_test <- sms_dtm[4170:5559, ]
```

For convenience later, it is also helpful to save a pair of vectors with the labels for each of the rows in the training and testing matrices. These labels are not stored in the DTM, so we need to pull them from the original `sms_raw` data frame:

```
> sms_train_labels <- sms_raw[1:4169, ]$type  
> sms_test_labels <- sms_raw[4170:5559, ]$type
```

To confirm that the subsets are representative of the complete set of SMS data, let's compare the proportion of spam in the training and test data frames:

```
> prop.table(table(sms_train_labels))
```

ham	spam
0.8647158	0.1352842

```
> prop.table(table(sms_test_labels))
```

ham	spam
0.8683453	0.1316547

Both the training data and test data contain about 13 percent spam. This suggests that the spam messages were divided evenly between the two datasets.

Visualizing text data – word clouds

A **word cloud** is a way to visually depict the frequency at which words appear in text data. The cloud is composed of words scattered somewhat randomly around the figure. Words appearing more often in the text are shown in a larger font, while less common terms are shown in smaller fonts. This type of figure grew in popularity as a way to observe trending topics on social media websites.

The `wordcloud` package provides a simple R function to create this type of diagram. We'll use it to visualize the words in SMS messages. Comparing the clouds for spam and ham messages will help us gauge whether our Naive Bayes spam filter is likely to be successful. If you haven't already done so, install and load the package by typing `install.packages("wordcloud")` and `library(wordcloud)` at the R command line.



The `wordcloud` package was written by Ian Fellows. For more information about this package, visit his blog at <http://blog.fellstat.com/?cat=11>.

A word cloud can be created directly from a *tm* corpus object using the syntax:

```
> wordcloud(sms_corpus_clean, min.freq = 50, random.order = FALSE)
```

This will create a word cloud from our prepared SMS corpus. Since we specified `random.order = FALSE`, the cloud will be arranged in a non-random order, with higher-frequency words placed closer to the center. If we do not specify `random.order`, the cloud will be arranged randomly by default.

The `min.freq` parameter specifies the number of times a word must appear in the corpus before it will be displayed in the cloud. Since a frequency of 50 is about 1 percent of the corpus, this means that a word must be found in at least 1 percent of the SMS messages to be included in the cloud.



You might get a warning message noting that R was unable to fit all the words in the figure. If so, try increasing the `min.freq` to reduce the number of words in the cloud. It might also help to use the `scale` parameter to reduce the font size.

The resulting word cloud should appear similar to the following:

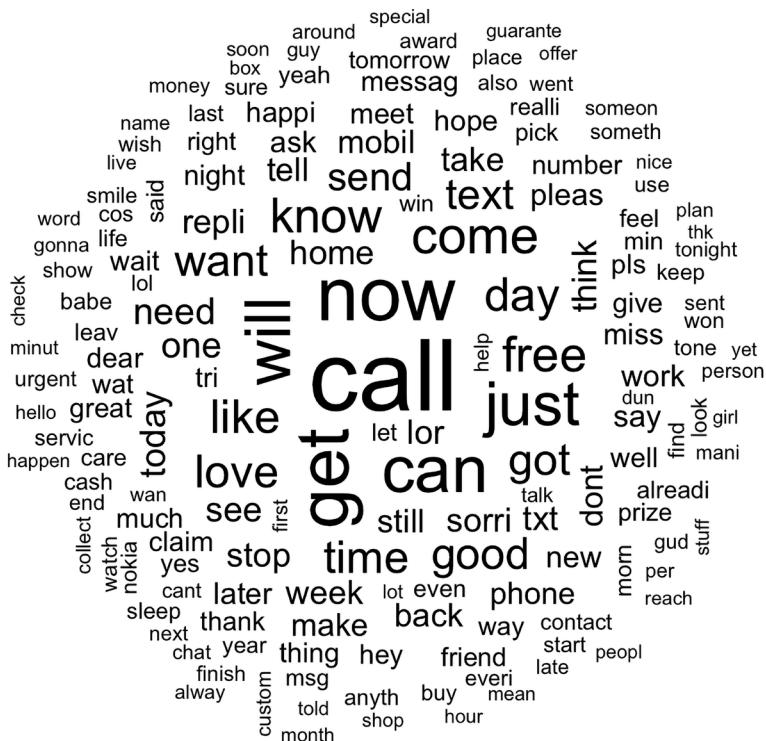


Figure 4.10: A word cloud depicting words appearing in all SMS messages

A perhaps more interesting visualization involves comparing the clouds for SMS spam and ham. Since we did not construct separate corpora for spam and ham, this is an appropriate time to note a very helpful feature of the `wordcloud()` function. Given a vector of raw text strings, it will automatically apply common text preparation processes before displaying the cloud.

Let's use R's `subset()` function to take a subset of the `sms_raw` data by the SMS type. First, we'll create a subset where the type is `spam`:

```
> spam <- subset(sms_raw, type == "spam")
```

Next, we'll do the same thing for the `ham` subset:

```
> ham <- subset(sms_raw, type == "ham")
```



Be careful to note the double equals sign. Like many programming languages, R uses `==` to test equality. If you accidentally use a single equals sign, you'll end up with a subset much larger than you expected!

We now have two data frames, `spam` and `ham`, each with a `text` feature containing the raw text strings for SMS messages. Creating word clouds is as simple as before. This time, we'll use the `max.words` parameter to look at the 40 most common words in each of the 2 sets. The `scale` parameter adjusts the maximum and minimum font sizes for words in the cloud. Feel free to change these parameters as you see fit. This is illustrated in the following code:

```
> wordcloud(spam$text, max.words = 40, scale = c(3, 0.5))  
> wordcloud(ham$text, max.words = 40, scale = c(3, 0.5))
```



Note that R provides warning messages when running this code that the “transformation drops documents.” The warnings are related to the `removePunctuation()` and `removeWords()` procedures that `wordcloud()` performs by default when given raw text data rather than a term matrix. Basically, there are some messages that are excluded from the result because there is no remaining message text after cleaning. For example, the `ham` message with the text `:)` representing the smiley emoji is removed from the set after cleaning. This is not a problem for the word clouds and the warnings can be ignored.

The resulting word clouds should appear similar to those that follow. Do you have a hunch on which one is the `spam` cloud, and which represents `ham`?



Figure 4.11: Side-by-side word clouds depicting SMS spam and ham messages

As you probably guessed, the spam cloud is on the left. Spam messages include words such as *call*, *free*, *mobile*, *claim*, and *stop*; these terms do not appear in the ham cloud at all. Instead, ham messages use words such as *can*, *sorry*, *love*, and *time*. These stark differences suggest that our Naive Bayes model will have some strong keywords to differentiate between the classes.

Data preparation – creating indicator features for frequent words

The final step in the data preparation process is to transform the sparse matrix into a data structure that can be used to train a Naive Bayes classifier. Currently, the sparse matrix includes over 6,500 features; this is a feature for every word that appears in at least one SMS message. It's unlikely that all of these are useful for classification. To reduce the number of features, we'll eliminate any word that appears in less than 5 messages, or in less than about 0.1 percent of records in the training data.

Finding frequent words requires the use of the `findFreqTerms()` function in the `tm` package. This function takes a DTM and returns a character vector containing words that appear at least a minimum number of times. For instance, the following command displays the words appearing at least five times in the `sms_dtm_train` matrix:

```
> findFreqTerms(sms_dtm_train, 5)
```

The result of the function is a character vector, so let's save our frequent words for later:

```
> sms_freq_words <- findFreqTerms(sms_dtm_train, 5)
```

A peek into the contents of the vector shows us that there are 1,139 terms appearing in at least 5 SMS messages:

```
> str(sms_freq_words)
chr [1:1137] "fkw" "abiola" "abl" "abt" "accept" "access" "account"
"across" "act" "activ" ...
```

We now need to filter our DTM to include only the terms appearing in the frequent word vector. As before, we'll use data frame-style `[row, col]` operations to request specific sections of the DTM, noting that the DTM column names are based on the words the DTM contains. We can take advantage of this fact to limit the DTM to specific words. Since we want all rows but only the columns representing the words in the `sms_freq_words` vector, our commands are:

```
> sms_dtm_freq_train <- sms_dtm_train[ , sms_freq_words]
> sms_dtm_freq_test <- sms_dtm_test[ , sms_freq_words]
```

The training and test datasets now include 1,137 features, which correspond to words appearing in at least 5 messages.

The Naive Bayes classifier is usually trained on data with categorical features. This poses a problem since the cells in the sparse matrix are numeric and measure the number of times a word appears in a message. We need to change this to a categorical variable that simply indicates yes or no, depending on whether the word appears at all.

The following defines a `convert_counts()` function to convert counts into Yes or No strings:

```
> convert_counts <- function(x) {
  x <- ifelse(x > 0, "Yes", "No")
}
```

By now, some of the pieces of the preceding function should look familiar. The first line defines the function. The statement `ifelse(x > 0, "Yes", "No")` transforms the values in `x` such that if the value is greater than 0, then it will be replaced with "Yes"; otherwise, it will be replaced with a "No" string. Lastly, the newly transformed vector `x` is returned.

We now need to apply `convert_counts()` to each of the columns in our sparse matrix. You may be able to guess the name of the R function that does exactly this. The function is simply called `apply()` and is used much like `lapply()` was used previously.

The `apply()` function allows a function to be used on each of the rows or columns in a matrix. It uses a `MARGIN` parameter to specify either rows or columns. Here, we'll use `MARGIN = 2` since we're interested in the columns (`MARGIN = 1` is used for rows). The commands to convert the training and test matrices are as follows:

```
> sms_train <- apply(sms_dtm_freq_train, MARGIN = 2,  
+ convert_counts)  
> sms_test <- apply(sms_dtm_freq_test, MARGIN = 2,  
+ convert_counts)
```

The result will be two character-type matrices, each with cells indicating "Yes" or "No" for whether the word represented by the column appears at any point in the message represented by the row.

Step 3 – training a model on the data

Now that we have transformed the raw SMS messages into a format that can be represented by a statistical model, it is time to apply the Naive Bayes algorithm. The algorithm will use the presence or absence of words to estimate the probability that a given SMS message is spam.

The Naive Bayes implementation we will employ is in the `naivebayes` package. This package is maintained by Michal Majka and is a modern and efficient R implementation. If you have not done so already, be sure to install and load the package using the `install.packages("naivebayes")` and `library(naivebayes)` commands before continuing.



Many machine learning approaches are implemented in more than one R package, and Naive Bayes is no exception. Another option is `naiveBayes()` in the `e1071` package, which was used in older editions of this book but is otherwise nearly identical to `naive_bayes()` in usage. The `naivebayes` package used in this edition offers better performance and more advanced functionality, which is described at its website: <https://majkamichal.github.io/naivebayes/>.

Unlike the k-NN algorithm we used for classification in the previous chapter, training a Naive Bayes learner and using it for classification occur in separate stages. Still, as shown in the following table, these steps are fairly straightforward:

Naive Bayes classification syntax

Using the `naive_bayes()` function in the `naivebayes` package

Building the classifier:

```
m <- naive_bayes(x, y, laplace = 0)
```

- `x` is a data frame or matrix containing training data
- `y` is a factor vector with the class for each row in the training data
- `laplace` is a number to control the Laplace estimator (by default, 0)

The function will return a Naive Bayes model object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "class")
```

- `m` is a model trained by the `naive_bayes()` function
- `test` is a data frame or matrix containing test data with the same features as the training data used to build the classifier
- `type` is either "class" or "prob" and specifies whether the predictions should be the most likely class value or the raw predicted probabilities

The function will return a vector of predicted class values or raw predicted probabilities depending upon the value of the `type` parameter.

Example:

```
sms_classifier <- naive_bayes(sms_train, sms_type)
sms_predictions <- predict(sms_classifier, sms_test)
```

Figure 4.12: Naive Bayes classification syntax

Using the `sms_train` matrix, the following command trains a `naive_bayes` classifier object that can be used to make predictions:

```
> sms_classifier <- naiveBayes(sms_train, sms_train_labels)
```

After running the previous command, you may notice the following output:

```
There were 50 or more warnings (use warnings() to see the first 50)
```

This is nothing to be alarmed about for now; typing the `warnings()` command reveals the cause of this issue:

```
> warnings()
```

```
Warning messages:
```

```
1: naive_bayes(): Feature fwk - zero probabilities are present. Consider
Laplace smoothing.
```

```
2: naive_bayes(): Feature biola - zero probabilities are present.  
Consider Laplace smoothing.  
3: naive_bayes(): Feature abl - zero probabilities are present. Consider  
Laplace smoothing.  
4: naive_bayes(): Feature abt - zero probabilities are present. Consider  
Laplace smoothing.  
5: naive_bayes(): Feature accept - zero probabilities are present.  
Consider Laplace smoothing.
```

These warnings are caused by words that appeared in zero spam or zero ham messages and have veto power over the classification process due to their associated zero probabilities. For instance, because the word *accept* only appeared in ham messages in the training data, it does not mean that every future message with this word should be automatically classified as ham.

There is an easy solution to this problem using the Laplace estimator described earlier, but for now, we will evaluate this model using `laplace = 0`, which is the model's default setting.

Step 4 – evaluating model performance

To evaluate the SMS classifier, we need to test its predictions on the unseen messages in the test data. Recall that the unseen message features are stored in a matrix named `sms_test`, while the class labels (spam or ham) are stored in a vector named `sms_test_labels`. The classifier that we trained has been named `sms_classifier`. We will use this classifier to generate predictions and then compare the predicted values to the true values.

The `predict()` function is used to make the predictions. We will store these in a vector named `sms_test_pred`. We simply supply this function with the names of our classifier and test dataset as shown:

```
> sms_test_pred <- predict(sms_classifier, sms_test)
```

To compare the predictions to the true values, we'll use the `CrossTable()` function in the `gmodels` package, which we used in previous chapters. This time, we'll add some additional parameters to eliminate unnecessary cell proportions, and use the `dnn` parameter (dimension names) to relabel the rows and columns as shown in the following code:

```
> library(gmodels)  
> CrossTable(sms_test_pred, sms_test_labels,  
prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,  
dnn = c('predicted', 'actual'))
```

This produces the following table:

Total Observations in Table: 1390				
		actual		
predicted	ham	spam	Row Total	
ham	1201	30	1231	
	0.864	0.022		
spam	6	153	159	
	0.004	0.110		
Column Total	1207	183	1390	

Looking at the table, we can see that a total of only $6 + 30 = 36$ of 1,390 SMS messages were incorrectly classified (2.6 percent). Among the errors were 6 out of 1,207 ham messages that were misidentified as spam and 30 of 183 spam messages that were incorrectly labeled as ham. Considering the little effort that we put into the project, this level of performance seems quite impressive. This case study exemplifies the reason why Naive Bayes is so often used for text classification: directly out of the box, it performs surprisingly well.

On the other hand, the six legitimate messages that were incorrectly classified as spam could cause significant problems for the deployment of our filtering algorithm because the filter could cause a person to miss an important text message. We should try to see whether we can slightly tweak the model to achieve better performance.

Step 5 – improving model performance

You may recall that we didn't set a value for the Laplace estimator when training our model; in fact, it was hard to miss the message from R warning us about more than 50 features with zero probabilities! To address this issue, we'll build a Naive Bayes model as before, but this time set `laplace = 1`:

```
> sms_classifier2 <- naiveBayes(sms_train, sms_train_labels,
  laplace = 1)
```

Next, we'll make predictions as before:

```
> sms_test_pred2 <- predict(sms_classifier2, sms_test)
```

Finally, we'll compare the predicted classes to the actual classifications using cross-tabulation:

```
> CrossTable(sms_test_pred2, sms_test_labels,
  prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
  dnn = c('predicted', 'actual'))
```

This produces the following table:

Total Observations in Table: 1390				
		actual		
predicted		ham	spam	Row Total
ham	ham	1202	28	1230
		0.865	0.020	
spam	spam	5	155	160
		0.004	0.112	
Column Total		1207	183	1390

Adding a Laplace estimator by setting `laplace = 1` reduced the number of false positives (ham messages erroneously classified as spam) from 6 to 5, and the number of false negatives from 30 to 28. Although this seems like a small change, it's substantial considering that the model's accuracy was already quite impressive. We'd need to be careful before tweaking the model too much more, as it is important to maintain a balance between being overly aggressive and overly passive when filtering spam. Users prefer that a small number of spam messages slip through the filter rather than the alternative, in which ham messages are filtered too aggressively.

Summary

In this chapter, we learned about classification using Naive Bayes. This algorithm constructs tables of probabilities that are used to estimate the likelihood that new examples belong to various classes. The probabilities are calculated using a formula known as Bayes' theorem, which specifies how dependent events are related. Although Bayes' theorem can be computationally expensive, a simplified version that makes so-called "naive" assumptions about the independence of features is capable of handling much larger datasets.

The Naive Bayes classifier is often used for text classification. To illustrate its effectiveness, we employed Naive Bayes on a classification task involving spam SMS messages. Preparing the text data for analysis required the use of specialized R packages for text processing and visualization. Ultimately, the model was able to classify over 97 percent of all the SMS messages correctly as spam or ham.

In the next chapter, we will examine two more machine learning methods. Each performs classification by partitioning data into groups of similar values. As you will discover shortly, these methods are quite useful on their own. Yet, looking further ahead, these basic algorithms also serve as an important foundation for some of the most powerful machine learning methods known today, which will be introduced later in *Chapter 14, Building Better Learners*.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



5

Divide and Conquer – Classification Using Decision Trees and Rules

When deciding between job offers, many people begin by making lists of pros and cons, then eliminate options using simple rules. For instance, they may decide, “If I have to commute for more than an hour, I will be unhappy,” or “If I make less than \$50K, I can’t support my family.” In this way, the complex decision of predicting one’s future career happiness can be reduced to a series of simple decisions.

This chapter covers decision trees and rule learners—two machine learning methods that also make complex decisions from sets of simple choices. These methods present their knowledge in the form of logical structures that can be understood with no statistical knowledge. This aspect makes these models particularly useful for business strategy and process improvement.

By the end of this chapter, you will have learned:

- How trees and rules “greedily” partition data into interesting segments
- The most common decision tree and classification rule learners, including the C5.0, 1R, and RIPPER algorithms
- How to use these algorithms for performing real-world classification tasks, such as identifying risky bank loans and poisonous mushrooms

We will begin by examining decision trees and follow that with a look at classification rules. Then, we will summarize what we've learned by previewing later chapters, which discuss methods that use trees and rules as a foundation for more advanced machine learning techniques.

Understanding decision trees

Decision tree learners are powerful classifiers that utilize a **tree structure** to model the relationships among the features and the potential outcomes. As illustrated in the following figure, this structure earned its name because it mirrors the way a literal tree begins, with a wide trunk at the base that splits off into narrower and narrower branches as it works its way upward. In much the same way, a decision tree classifier uses a structure of branching decisions to channel examples into a final predicted class value.

To better understand how this works in practice, let's consider the following tree, which predicts whether a job offer should be accepted. A job offer under consideration begins at the **root node**, from where it then passes through the **decision nodes**, which require choices to be made based on the attributes of the job. These choices split the data across **branches** that indicate the potential outcomes of a decision. They are depicted here as yes or no outcomes, but in other cases, there may be more than two possibilities.

If a final decision can be made, the tree terminates in **leaf nodes** (also known as **terminal nodes**) that denote the action to be taken as the result of the series of decisions. In the case of a predictive model, the leaf nodes provide the expected result given the series of events in the tree.

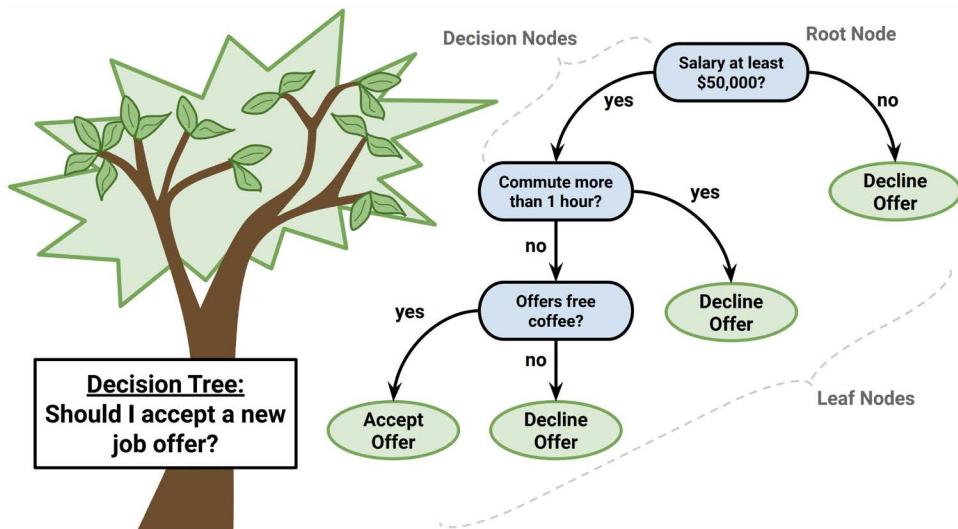


Figure 5.1: A decision tree depicting the process of determining whether to accept a new job offer

A great benefit of decision tree algorithms is that the flowchart-like tree structure is not only for the machine's internal use. After the model is created, many decision tree algorithms output the resulting structure in a human-readable format. This provides insight into how and why the model works or doesn't work well for a particular task. This also makes decision trees particularly appropriate for applications in which the classification mechanism needs to be transparent for legal reasons, or if the results need to be shared with others to inform future business practices. With this in mind, some potential uses include the following:

- Credit scoring models in which the criteria that cause an applicant to be rejected need to be clearly documented and free from bias
- Marketing studies of customer behavior, such as satisfaction or churn, which will be shared with management or advertising agencies
- Diagnosis of medical conditions based on laboratory measurements, symptoms, or rates of disease progression

Although the previous applications illustrate the value of trees in informing decision-making processes, this is not to suggest that their utility ends here. In fact, decision trees are one of the single most widely used machine learning techniques, and can be applied to model almost any type of data—often with excellent out-of-the-box performance.

That said, despite their wide applicability, it is worth noting that there are some scenarios where trees may not be an ideal fit. This includes tasks where the data has many nominal features with many levels or a large number of numeric features. These cases may result in a very large number of decisions and an overly complex tree. They may also contribute to the tendency of decision trees to overfit data, though as we will soon see, even this weakness can be overcome by adjusting some simple parameters.

Divide and conquer

Decision trees are built using a heuristic called **recursive partitioning**. This approach is also commonly known as **divide and conquer** because it splits the data into subsets, which are then split repeatedly into even smaller subsets, and so on and so forth, until the process stops when the algorithm determines the data within the subsets are sufficiently homogenous, or another stopping criterion has been met.

To see how splitting a dataset can create a decision tree, imagine a root node that will grow into a mature tree. At first, the root node represents the entire dataset, since no splitting has transpired. Here, the decision tree algorithm must choose a feature to split upon; ideally, it chooses the feature most predictive of the target class.

The examples are then partitioned into groups according to the distinct values of this feature, and the first set of tree branches is formed.

Working down each branch, the algorithm continues to divide and conquer the data, choosing the best candidate feature each time to create another decision node until a stopping criterion is reached. Divide and conquer might stop at a node if:

- All (or nearly all) of the examples at the node have the same class
- There are no remaining features to distinguish among the examples
- The tree has grown to a predefined size limit

To illustrate the tree-building process, let's consider a simple example. Imagine that you work for a Hollywood studio, where your role is to decide whether the studio should move forward with producing the screenplays pitched by promising new authors. After returning from a vacation, your desk is piled high with proposals. Without the time to read each proposal cover-to-cover, you decide to develop a decision tree algorithm to predict whether a potential movie would fall into one of three categories: *Critical Success*, *Mainstream Hit*, or *Box Office Bust*.

To source data to create the decision tree model, you turn to the studio archives to examine the factors leading to the success or failure of the company's 30 most recent releases. You quickly notice a relationship between the film's estimated shooting budget, the number of A-list celebrities lined up for starring roles, and the film's level of success. Excited about this finding, you produce a scatterplot to illustrate the pattern:

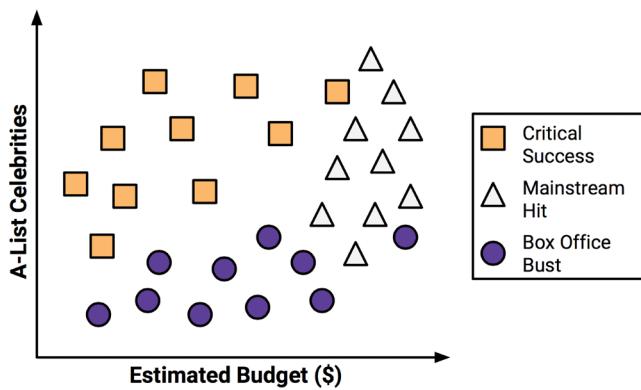


Figure 5.2: A scatterplot depicting the relationship between a movie's budget and celebrity count

Using the divide and conquer strategy, you can build a simple decision tree from this data. First, to create the tree's root node, you split the feature indicating the number of celebrities, partitioning the movies into groups with and without a significant number of A-list stars:

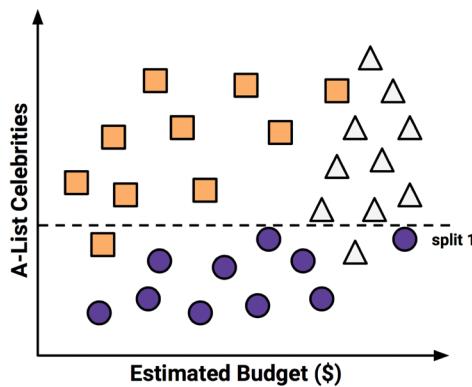


Figure 5.3: The decision tree's first split divides the data into films with high and low celebrity counts

Next, among the group of movies with a larger number of celebrities, you make another split between movies with and without a high budget:

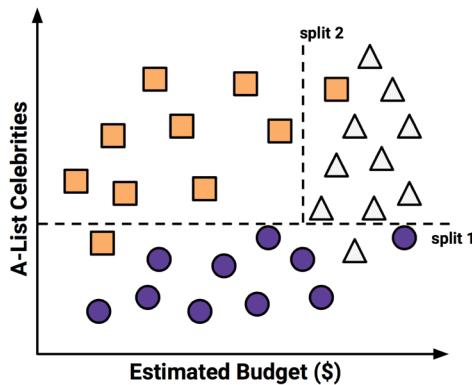


Figure 5.4: The decision tree's second split further divides the films with a high celebrity count into those with low and high budgets

At this point, you've partitioned the data into three groups. The group in the top-left corner of the diagram is composed entirely of critically acclaimed films. This group is distinguished by a high number of celebrities and a relatively low budget. In the top-right corner, nearly all movies are box office hits with high budgets and many celebrities. The final group, which has little star power but budgets ranging from small to large, contains the flops.

If desired, you could continue to divide and conquer the data by splitting it on increasingly specific ranges of budget and celebrity count until each of the currently misclassified values is correctly classified in its own tiny partition. However, it is not advisable to overfit a decision tree in this way. Although there is nothing stopping the algorithm from splitting the data indefinitely, overly specific decisions do not always generalize more broadly. Thus, you choose to avoid the problem of overfitting by stopping the algorithm here, since more than 80 percent of the examples in each group are from a single class. This is the stopping criterion for the decision tree model.



You might have noticed that diagonal lines might have split the data even more cleanly. This is one limitation of the decision tree's knowledge representation, which uses **axis-parallel splits**. The fact that each split considers one feature at a time prevents the decision tree from forming more complex decision boundaries. For example, a diagonal line could be created by a decision that asks, "Is the number of celebrities greater than the estimated budget?" If so, then "it will be a critical success."

The model for predicting the future success of movies can be represented in a simple tree, as shown in the following diagram. Each step in the tree shows the fraction of examples falling into each class, which shows how the data becomes more homogeneous as the branches get closer to a leaf. To evaluate a new movie script, follow the branches through each decision until the script's success or failure has been predicted. Using this approach, you will be able to quickly identify the most promising options among the backlog of scripts and get back to more important work, such as writing an Academy Awards acceptance speech!

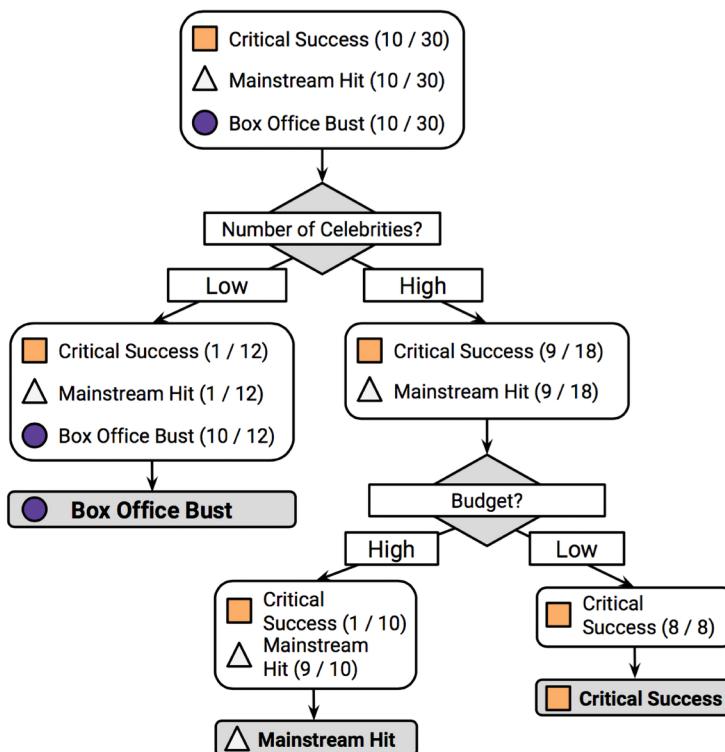


Figure 5.5: A decision tree built on historical movie data can forecast the performance of future movies

Since real-world data contains more than two features, decision trees quickly become far more complex than this, with many more nodes, branches, and leaves. In the next section, you will learn about a popular algorithm to build decision tree models automatically.

The C5.0 decision tree algorithm

There are numerous implementations of decision trees, but one of the most well known is the C5.0 algorithm. This algorithm was developed by computer scientist J. Ross Quinlan as an improved version of his prior algorithm, C4.5, which itself is an improvement over his **Iterative Dichotomiser 3 (ID3)** algorithm. Although Quinlan markets C5.0 to commercial clients (see <http://www.rulequest.com/> for details), the source code for a single-threaded version of the algorithm was made public, and has therefore been incorporated into programs such as R.



To further confuse matters, a popular Java-based open-source alternative to C4.5, titled **J48**, is included in R's **RWeka** package (introduced later in this chapter). As the differences between C5.0, C4.5, and J48 are minor, the principles in this chapter apply to any of these three methods and the algorithms should be considered synonymous.

The C5.0 algorithm has become the industry standard for producing decision trees because it does well for most types of problems directly out of the box. Compared to other advanced machine learning models, such as those described in *Chapter 7, Black-Box Methods – Neural Networks and Support Vector Machines*, the decision trees built by C5.0 generally perform nearly as well but are much easier to understand and deploy. Additionally, as shown in the following table, the algorithm's weaknesses are relatively minor and can be largely avoided.

Strengths	Weaknesses
<ul style="list-style-type: none"> An all-purpose classifier that does well on many types of problems Highly automatic learning process, which can handle numeric or nominal features, as well as missing data Excludes unimportant features Can be used on both small and large datasets Results in a model that can be interpreted without a mathematical background (for relatively small trees) More efficient than other complex models 	<ul style="list-style-type: none"> Decision tree models are often biased toward splits on features having a large number of levels It is easy to overfit or underfit the model Can have trouble modeling some relationships due to reliance on axis-parallel splits Small changes in training data can result in large changes to decision logic Large trees can be difficult to interpret and the decisions they make may seem counterintuitive

To keep things simple, our earlier decision tree example ignored the mathematics involved with how a machine would employ a divide and conquer strategy. Let's explore this in more detail to examine how this heuristic works in practice.

Choosing the best split

The first challenge that a decision tree will face is to identify which feature to split upon. In the previous example, we looked for a way to split the data such that the resulting partitions contained examples primarily of a single class.

The degree to which a subset of examples contains only a single class is known as **purity**, and any subset composed of only a single class is called **pure**.

There are various measurements of purity that can be used to identify the best decision tree splitting candidate. C5.0 uses **entropy**, a concept borrowed from information theory that quantifies the randomness, or disorder, within a set of class values. Sets with high entropy are very diverse and provide little information about other items that may also belong in the set, as there is no apparent commonality. The decision tree hopes to find splits that reduce entropy, ultimately increasing homogeneity within the groups.

Typically, entropy is measured in **bits**. If there are only two possible classes, entropy values can range from 0 to 1. For n classes, entropy ranges from 0 to $\log_2(n)$. In each case, the minimum value indicates that the sample is completely homogenous, while the maximum value indicates that the data are as diverse as possible, and no group has even a small plurality.

In mathematical notion, entropy is specified as:

$$\text{Entropy}(S) = \sum_{i=1}^c -p_i \log_2(p_i)$$

In this formula, for a given segment of data (S), the term c refers to the number of class levels, and p_i refers to the proportion of values falling into the i th class level.

For example, suppose we have a partition of data with two classes: red (60 percent) and white (40 percent). We can calculate the entropy as:

```
> -0.60 * log2(0.60) - 0.40 * log2(0.40)
```

```
[1] 0.9709506
```

We can visualize the entropy for all possible two-class arrangements. If we know the proportion of examples in one class is x , then the proportion in the other class is $(1 - x)$. Using the `curve()` function, we can then plot the entropy for all possible values of x :

```
> curve(-x * log2(x) - (1 - x) * log2(1 - x),
       col = "red", xlab = "x", ylab = "Entropy", lwd = 4)
```

This results in the following graph:

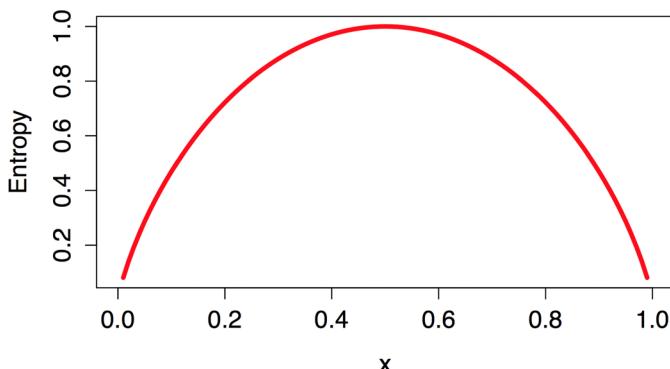


Figure 5.6: The total entropy as the proportion of one class varies in a two-class outcome

As illustrated by the peak at $x = 0.50$, a 50-50 split results in the maximum entropy. As one class increasingly dominates the other, the entropy reduces to zero.

To use entropy to determine the optimal feature to split upon, the algorithm calculates the change in homogeneity that would result from a split on each possible feature, a measure known as **information gain**. The information gain for a feature F is calculated as the difference between the entropy in the segment before the split (S_1) and the partitions resulting from the split (S_2):

$$\text{InfoGain}(F) = \text{Entropy}(S_1) - \text{Entropy}(S_2)$$

One complication is that after a split, the data is divided into more than one partition. Therefore, the function to calculate $\text{Entropy}(S_2)$ needs to consider the total entropy across all partitions resulting from the split. It does this by weighting each partition's entropy according to the proportion of all records falling into that partition. This can be stated in a formula as:

$$\text{Entropy}(S) = \sum_{i=1}^n w_i \text{Entropy}(P_i)$$

In simple terms, the total entropy resulting from a split is the sum of entropy of each of the n partitions weighted by the proportion of examples falling in the partition (w_i).

The higher the information gain, the better a feature is at creating homogeneous groups after a split on that feature. If the information gain is zero, there is no reduction in entropy for splitting on this feature. On the other hand, the maximum information gain is equal to the entropy prior to the split. This would imply the entropy after the split is zero, which means that the split results in completely homogeneous groups.

The previous formulas assume nominal features, but decision trees use information gain for splitting on numeric features as well. To do so, a common practice is to test various splits that divide the values into groups greater than or less than a threshold. This reduces the numeric feature into a two-level categorical feature that allows information gain to be calculated as usual. The numeric cut point yielding the largest information gain is chosen for the split.



Though it is used by C5.0, information gain is not the only splitting criterion that can be used to build decision trees. Other commonly used criteria are the **Gini index**, **chi-squared statistic**, and **gain ratio**. For a review of these (and many more) criteria, refer to *An Empirical Comparison of Selection Measures for Decision-Tree Induction*, Mingers, J, *Machine Learning*, 1989, Vol. 3, pp. 319-342.

Pruning the decision tree

As mentioned earlier, a decision tree can continue to grow indefinitely, choosing splitting features and dividing into smaller and smaller partitions until each example is perfectly classified or the algorithm runs out of features to split on. However, if the tree grows overly large, many of the decisions it makes will be overly specific and the model will be overfitted to the training data. The process of **pruning** a decision tree involves reducing its size such that it generalizes better to unseen data.

One solution to this problem is to stop the tree from growing once it reaches a certain number of decisions or when the decision nodes contain only a small number of examples. This is called **early stopping** or **pre-pruning** the decision tree. As the tree avoids doing needless work, this is an appealing strategy. However, one downside to this approach is that there is no way to know whether the tree will miss subtle but important patterns that it would have learned had it grown to a larger size.

An alternative, called **post-pruning**, involves growing a tree that is intentionally too large and pruning leaf nodes to reduce the size of the tree to a more appropriate level. This is often a more effective approach than pre-pruning because it is quite difficult to determine the optimal depth of a decision tree without growing it first. Pruning the tree later allows the algorithm to be certain that all the important data structures were discovered.



The implementation details of pruning operations are very technical and beyond the scope of this book. For a comparison of some of the available methods, see *A Comparative Analysis of Methods for Pruning Decision Trees, Esposito, F, Malerba, D, Semeraro, G, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1997, Vol. 19, pp. 476-491.*

One of the benefits of the C5.0 algorithm is that it is opinionated about pruning—it takes care of many of the decisions automatically using reasonable defaults. Its overall strategy is to post-prune the tree. It first grows a large tree that overfits the training data. Later, the nodes and branches that have little effect on the classification errors are removed. In some cases, entire branches are moved further up the tree or replaced by simpler decisions. These processes of grafting branches are known as **subtree raising** and **subtree replacement**, respectively.

Getting the right balance of overfitting and underfitting is a bit of an art, but if model accuracy is vital, it may be worth investing some time with various pruning options to see if it improves the test dataset performance. As you will soon see, one of the strengths of the C5.0 algorithm is that it is very easy to adjust the training options.

Example – identifying risky bank loans using C5.0 decision trees

The global financial crisis of 2007-2008 highlighted the importance of transparency and rigor in banking practices. As the availability of credit was limited, banks tightened their lending systems and turned to machine learning to more accurately identify risky loans.

Decision trees are widely used in the banking industry due to their high accuracy and ability to formulate a statistical model in plain language. Since governments in many countries carefully monitor the fairness of lending practices, executives must be able to explain why one applicant was rejected for a loan while another was approved. This information is also useful for customers hoping to determine why their credit rating is unsatisfactory.

It is likely that automated credit scoring models are used for credit card mailings and instant online approval processes. In this section, we will develop a simple credit approval model using C5.0 decision trees. We will also see how the model results can be tuned to minimize errors that result in a financial loss.

Step 1 – collecting data

The motivation for our credit model is to identify factors that are linked to a higher risk of loan default. To do this, we must obtain data on past bank loans as well as information about the loan applicants that would have been available at the time of credit application.

Data with these characteristics are available in a dataset donated to the UCI Machine Learning Repository (<http://archive.ics.uci.edu/ml>) by Hans Hofmann of the University of Hamburg. The dataset contains information on loans obtained from a credit agency in Germany.



The dataset presented in this chapter has been modified slightly from the original in order to eliminate some preprocessing steps. To follow along with the examples, download the `credit.csv` file from the Packt Publishing GitHub repository for this chapter and save it to your R working directory.

The credit dataset includes 1,000 examples of loans, plus a set of numeric and nominal features indicating characteristics of the loan and the loan applicant. A class variable indicates whether the loan went into default. Let's see if we can identify any patterns that predict this outcome.

Step 2 – exploring and preparing the data

As we have done previously, we will import the data using the `read.csv()` function. Now, because the character data is entirely categorical, we can set the `stringsAsFactors = TRUE` to automatically convert all character type columns to factors in the resulting data frame:

```
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
```

We can check the resulting object by examining the first few lines of output from the `str()` function:

```
> str(credit)

'data.frame': 1000 obs. of 17 variables:
 $ checking_balance : Factor w/ 4 levels "< 0 DM","> 200 DM",...
 $ months_loan_duration: int 6 48 12 42 24 36 24 36 12 30 ...
 $ credit_history : Factor w/ 5 levels "critical","good",...
 $ purpose : Factor w/ 6 levels "business","car",...
 $ amount : int 1169 5951 2096 7882 4870 9055 2835 6948 ...
```

We see the expected 1,000 observations and 17 features, which are a combination of factor and integer data types.

Let's take a look at the `table()` output for a couple of loan features that seem likely to predict a default. The applicant's checking and savings account balances are recorded as categorical variables:

```
> table(credit$checking_balance)
```

< 0 DM	> 200 DM	1 - 200 DM	unknown
274	63	269	394

```
> table(credit$savings_balance)
```

< 100 DM	> 1000 DM	100 - 500 DM	500 - 1000 DM	unknown
603	48	103	63	183

The checking and savings account balances may prove to be important predictors of loan default status. Note that since the loan data was obtained from Germany, the values use the **Deutsche Mark (DM)**, which was the currency used in Germany prior to the adoption of the Euro.

Some of the loan's features are numeric, such as its duration and the amount of credit requested:

```
> summary(credit$months_loan_duration)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
4.0	12.0	18.0	20.9	24.0	72.0

```
> summary(credit$amount)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
250	1366	2320	3271	3972	18424

The loan amounts ranged from 250 DM to 18,420 DM across terms of 4 to 72 months. They had a median amount of 2,320 DM and a median duration of 18 months.

The `default` vector indicates whether the loan applicant was able to meet the agreed payment terms or if they went into default. A total of 30 percent of the loans in this dataset went into default:

```
> table(credit$default)
```

no	yes
700	300

A high rate of default is undesirable for a bank because it means that the bank is unlikely to fully recover its investment. If we are successful, our model will identify applicants who are at high risk of default, allowing the bank to refuse the credit request before the money is given.

Data preparation – creating random training and test datasets

As we have done in previous chapters, we will split our data into two portions: a training dataset to build the decision tree and a test dataset to evaluate its performance on new data. We will use 90 percent of the data for training and 10 percent for testing, which will provide us with 100 records to simulate new applicants. A 90-10 split is used here rather than the more common 75-25 split due to the relatively small size of the credit dataset; given that predicting loan defaults is a challenging learning task, we need as much training data as possible while still holding out a sufficient test sample.



More sophisticated approaches for training and evaluating models with relatively small datasets are introduced in *Chapter 10, Evaluating Model Performance*.

As prior chapters used data that had been sorted in a random order, we simply divided the dataset into two portions by taking the first subset of records for training and the remaining subset for testing. In contrast, the credit dataset is not randomly ordered, making the prior approach unwise. Suppose that the bank had sorted the data by the loan amount, with the largest loans at the end of the file. If we used the first 90 percent for training and the remaining 10 percent for testing, we would be training a model on only the small loans and testing the model on the big loans. Obviously, this could be problematic.

We'll solve this problem by training the model on a **random sample** of the credit data. A random sample is simply a process that selects a subset of records at random. In R, the `sample()` function is used to perform random sampling. However, before putting it in action, a common practice is to set a **seed** value, which causes the randomization process to follow a sequence that can be replicated later. It may seem that this defeats the purpose of generating random numbers, but there is a good reason for doing it this way. Providing a seed value via the `set.seed()` function ensures that if the analysis is repeated in the future, an identical result is obtained.



You may wonder how a so-called random process can be seeded to produce an identical result. This is because computers use a mathematical function called a **pseudorandom number generator** to create random number sequences that appear to act very random, but are actually quite predictable given knowledge of the previous values in the sequence. In practice, modern pseudorandom number sequences are virtually indistinguishable from true random sequences, but have the benefit that computers can generate them quickly and easily.

The following commands use `sample()` with a seed value. Note that the `set.seed()` function uses the arbitrary value 9829. Omitting this seed will cause your training and testing splits to differ from those shown in the remainder of this chapter. The following commands select 900 values at random out of the sequence of integers from 1 to 1,000:

```
> set.seed(9829)
> train_sample <- sample(1000, 900)
```

As expected, the resulting `train_sample` object is a vector of 900 random integers:

```
> str(train_sample)
int [1:900] 653 866 119 152 6 617 250 343 367 138 ...
```

By using this vector to select rows from the credit data, we can split it into the 90 percent training and 10 percent test datasets we desired. Recall that the negation operator (the - character) used in the selection of the test records tells R to select records that are not in the specified rows; in other words, the test data includes only the rows that are not in the training sample:

```
> credit_train <- credit[train_sample, ]
> credit_test <- credit[-train_sample, ]
```

If randomization was done correctly, we should have about 30 percent of loans with default in each of the datasets:

```
> prop.table(table(credit_train$default))
```

no	yes
0.7055556	0.2944444

```
> prop.table(table(credit_test$default))
```

no	yes
0.65	0.35

Both the training and test datasets have roughly similar distributions of loan defaults, so we can now build our decision tree. In the case that the proportions differ greatly, we may decide to resample the dataset, or attempt a more sophisticated sampling approach, such as those covered in *Chapter 10, Evaluating Model Performance*.



If your results do not match exactly, ensure that you ran the command `set.seed(9829)` immediately prior to creating the `train_sample` vector. Note that R's default random number generator changed in R version 3.6.0 and your results will differ if this code is run on earlier versions. This also means that the results here are slightly different from those in prior editions of this book.

Step 3 – training a model on the data

We will use the C5.0 algorithm in the `C50` package for training our decision tree model. If you have not done so already, install the package with `install.packages("C50")` and load it to your R session using `library(C50)`.

The following syntax box lists some of the most common parameters used when building decision trees. Compared to the machine learning approaches we have used previously, the C5.0 algorithm offers many more ways to tailor the model to a particular learning problem.

C5.0 decision tree syntax	
Using the <code>C5.0()</code> function in the <code>C50</code> package	
Building the classifier:	
<pre>m <- C5.0(class ~ predictors, data = mydata, trials = 1, costs = NULL)</pre> <ul style="list-style-type: none"> • <code>class</code> is the column in the <code>mydata</code> data frame to be predicted • <code>predictors</code> is an R formula specifying the features in the <code>mydata</code> data frame to use for prediction • <code>trials</code> is an optional number to control the number of boosting iterations (set to 1 by default) • <code>costs</code> is an optional matrix specifying costs associated with various types of errors 	
The function will return a C5.0 model object that can be used to make predictions.	
Making predictions:	
<pre>p <- predict(m, test, type = "class")</pre> <ul style="list-style-type: none"> • <code>m</code> is a model trained by the <code>C5.0()</code> function • <code>test</code> is a data frame containing test data with the same features as the training data used to build the classifier • <code>type</code> is either "<code>class</code>" or "<code>prob</code>" and specifies whether the predictions should be the most probable class value or the raw predicted probabilities 	
The function will return a vector of predicted class values or raw predicted probabilities depending upon the value of the <code>type</code> parameter.	
Example:	
<pre>credit_model <- C5.0(default ~ ., data = credit_train) credit_prediction <- predict(credit_model, credit_test)</pre>	

Figure 5.7: C5.0 decision tree syntax

The `C5.0()` function uses a new syntax known as the **R formula interface** to specify the model to be trained. The formula syntax uses the `~` operator (known as the tilde) to express the relationship between a target variable and its predictors. The class variable to be learned goes to the left of the tilde and the predictor features are written on the right, separated by `+` operators.

If you would like to model the relationship between the target `y` and predictors `x1` and `x2`, you would write the formula as `y ~ x1 + x2`. To include all variables in the model, the period character is used. For example, `y ~ .` specifies the relationship between `y` and all other features in the dataset.



The R formula interface is used across many R functions and offers some powerful features to describe the relationships among predictor variables. We will explore some of these features in later chapters. However, if you're eager for a preview, feel free to read the documentation using the `?formula` command.

For the first iteration of the credit approval model, we'll use the default C5.0 settings, as shown in the following code. The target class is named `default`, so we put it on the left-hand side of the tilde, which is followed by a period indicating that all other columns in the `credit_train` data frame are to be used as predictors:

```
> credit_model <- C5.0(default ~ ., data = credit_train)
```

The `credit_model` object now contains a C5.0 decision tree. We can see some basic data about the tree by typing its name:

```
> credit_model
```

```
Call:  
C5.0.formula(formula = default ~ ., data = credit_train)
```

```
Classification Tree  
Number of samples: 900  
Number of predictors: 16
```

```
Tree size: 67
```

```
Non-standard options: attempt to group attributes
```

The output shows some simple facts about the tree, including the function call that generated it, the number of features (labeled `predictors`), and examples (labeled `samples`) used to grow the tree. Also listed is the tree size of 67, which indicates that the tree is 67 decisions deep—quite a bit larger than the example trees we've considered so far!

To see the tree's decisions, we can call the `summary()` function on the model:

```
> summary(credit_model)
```

This results in the following output, which has been truncated to show only the first few lines:

```
> summary(credit_model)
```

```
Call:  
C5.0.formula(formula = default ~ ., data = credit_train)  
  
C5.0 [Release 2.07 GPL Edition]  
-----  
  
Class specified by attribute `outcome'  
  
Read 900 cases (17 attributes) from undefined.data  
  
Decision tree:  
  
  checking_balance in {> 200 DM,unknown}: no (415/55)  
  checking_balance in {< 0 DM,1 - 200 DM}:  
    ....credit_history in {perfect,very good}: yes (59/16)  
      credit_history in {critical,good,poor}:  
        ....months_loan_duration > 27:  
          ....dependents > 1:  
            ....age <= 45: no (12/2)  
            ....age > 45: yes (2)
```

The preceding output shows some of the first branches in the decision tree. The first three lines could be represented in plain language as:

1. If the checking account balance is unknown or greater than 200 DM, then classify as “not likely to default”
2. Otherwise, if the checking account balance is less than zero DM or between one and 200 DM...
3. ...and the credit history is perfect or very good, then classify as “likely to default”

The numbers in parentheses indicate the number of examples meeting the criteria for that decision and the number incorrectly classified by the decision. For instance, on the first line, 415/55 indicates that, of the 415 examples reaching the decision, 55 were incorrectly classified as “not likely to default.” In other words, 55 out of 415 applicants actually defaulted in spite of the model’s prediction to the contrary.



Sometimes a tree results in decisions that make little logical sense. For example, why would an applicant whose credit history is perfect or very good be likely to default, while those whose checking balance is unknown are not likely to default? Contradictory rules like this occur sometimes. They might reflect a real pattern in the data, or they may be a statistical anomaly. In either case, it is important to investigate such strange decisions to see whether the tree’s logic makes sense for business use.

After the tree, the `summary(credit_model)` output displays a confusion matrix, which is a cross-tabulation that indicates the model’s incorrectly classified records in the training data:

```
Evaluation on training data (900 cases):
```

```
Decision Tree
-----
Size      Errors

66  118(13.1%)  <<

(a)  (b)  <-classified as
-----
604    31    (a): class no
     87    178   (b): class yes
```

The Errors heading shows that the model correctly classified all but 118 of the 900 training instances for an error rate of 13.1 percent. A total of 31 actual no values were incorrectly classified as yes (false positives), while 87 yes values were misclassified as no (false negatives). Given the tendency of decision trees to overfit to the training data, the error rate reported here, which is based on training data performance, may be overly optimistic. Therefore, it is especially important to apply the decision tree to an unseen test dataset, which we will do shortly.

The output also includes a section labeled **Attribute usage**, which provides a general sense of the most important predictors used in the decision tree model. The first few lines of this output are as follows:

Attribute usage:

100.00%	checking_balance
53.89%	credit_history
47.33%	months_loan_duration
26.11%	purpose
24.33%	savings_balance
18.22%	job
12.56%	dependents
12.11%	age

The **attribute usage** statistics in decision tree output refer to the percentage of rows in the training data that use the listed feature to make a final prediction. For example, 100 percent of rows require the `checking_balance` feature, because the checking account balance is used at the very first split in the tree. The second split uses `credit_history`, but 46.11 percent of rows were already classified as non-default based on the checking account balance. This leaves only 53.89 percent of rows that need to consider the applicant's credit history. At the bottom of this list, only 12.11 percent of examples require the applicant's age to make a prediction, which suggests that the applicant's age is less important than their checking account balance or credit history.

This information, along with the tree structure itself, provides insight into how the model works. Both are readily understood, even without a statistics background. Of course, a model that cannot make accurate predictions is useless even if it is easy to understand, so we will now perform a more formal evaluation of its performance.



C5.0 decision tree models can be visualized using the `plot()` function, which relies on functionality in the `partykit` package. Unfortunately, this is useful only for relatively small decision trees. For example, our decision tree can be visualized by typing `plot(credit_model)`, but unless you have a very large display, the resulting plot will likely appear as a jumbled mess due to the large number of nodes and splits in the tree.

Step 4 – evaluating model performance

To apply our decision tree to the test dataset, we use the `predict()` function as shown in the following line of code:

```
> credit_pred <- predict(credit_model, credit_test)
```

This creates a vector of predicted class values, which we can compare to the actual class values using the `CrossTable()` function in the `gmodels` package. Setting the `prop.c` and `prop.r` parameters to `FALSE` removes the column and row percentages from the table. The remaining percentage (`prop.t`) indicates the proportion of records in the cell out of the total number of records:

```
> library(gmodels)
> CrossTable(credit_test$default, credit_pred,
  prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
  dnn = c('actual default', 'predicted default'))
```

This results in the following table:

		predicted default		Row Total
actual default	no	yes		
no	56	9	65	
	0.560	0.090		
yes	24	11	35	
	0.240	0.110		
Column Total	80	20	100	

Out of the 100 loan applications in the test set, our model correctly predicted that 56 did not default and 11 did default, resulting in an accuracy of 67 percent and an error rate of 33 percent. This is somewhat worse than its performance on the training data, but not unexpected, given that a model's performance is often worse on unseen data. Also note that the model only correctly predicted 11 of the 35 actual loan defaults in the test data, or 31 percent. Unfortunately, this type of error is potentially a very costly mistake, as the bank loses money on each default. Let's see if we can improve the result with a bit more effort.

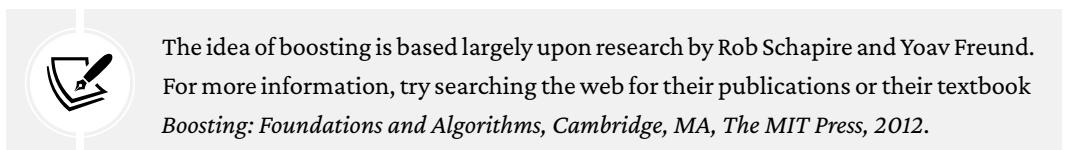
Step 5 – improving model performance

Our model’s error rate is likely to be too high to deploy it in a real-time credit scoring application. In fact, if the model had predicted “no default” for every test case, it would have been correct 65 percent of the time—a result barely worse than our model but requiring much less effort! Predicting loan defaults using only 900 training examples seems to be a challenging problem.

Making matters even worse, our model performed especially poorly at identifying applicants who do default on their loans. Luckily, there are a couple of simple ways to adjust the C5.0 algorithm that may help to improve the performance of the model, both overall and for the costlier type of mistakes.

Boosting the accuracy of decision trees

One way the C5.0 algorithm improved upon the C4.5 algorithm was through the addition of **adaptive boosting**. This is a process in which many decision trees are built, and the trees vote on the best class for each example.



The idea of boosting is based largely upon research by Rob Schapire and Yoav Freund. For more information, try searching the web for their publications or their textbook *Boosting: Foundations and Algorithms*, Cambridge, MA, The MIT Press, 2012.

As boosting can be applied more generally to any machine learning algorithm, it is covered in more detail later in this book in *Chapter 14, Building Better Learners*. For now, it suffices to say that boosting is rooted in the notion that by combining several weak-performing learners, you can create a team that is much stronger than any of the learners alone. Each of the models has a unique set of strengths and weaknesses, and may be better or worse at certain problems. Using a combination of several learners with complementary strengths and weaknesses can therefore dramatically improve the accuracy of a classifier.

The `C5.0()` function makes it easy to add boosting to our decision tree. We simply need to add an additional `trials` parameter indicating the number of separate decision trees to use in the boosted team. The `trials` parameter sets an upper limit; the algorithm will stop adding trees if it recognizes that additional trials do not seem to be improving the accuracy. We’ll start with 10 trials, a number that has become the de facto standard, as research suggests that this reduces error rates on test data by about 25 percent.

Aside from the new parameter, the command is similar to before:

```
> credit_boost10 <- C5.0(default ~ ., data = credit_train,  
                           trials = 10)
```

While examining the resulting model, we can see that the output now indicates the addition of boosting:

```
> credit_boost10  
  
Number of boosting iterations: 10  
Average tree size: 57.3
```

The new output shows that across the 10 iterations, our tree size shrunk. If you would like, you can see all 10 trees by typing `summary(credit_boost10)` at the command prompt. Note that some of these trees, including the tree built for the first trial, have one or more subtrees such as the one denoted [S1] in the following excerpt of the output:

```
dependents > 1: yes (8.8/0.8)  
dependents <= 1:  
  ....years_at_residence <= 1: no (13.4/1.6)  
    years_at_residence > 1:  
      ....age <= 23: yes (11.9/1.6)  
        age > 23: [S1]
```

Notice the line that says that if `age > 23`, the result is [S1]. To determine what this means, we must match [S1] to the corresponding subtree slightly further down in the output, where we see that the final decision requires several more steps:

```
SubTree [S1]  
  
employment_duration in {< 1 year,> 7 years,4 - 7 years,  
  ....unemployed}: no (27.7/6.3)  
employment_duration = 1 - 4 years:  
  ....months_loan_duration > 30: yes (7.2)  
    months_loan_duration <= 30:  
      ....other_credit = bank: yes (2.4)  
        other_credit in {none,store}: no (16.6/5.6)
```

Such subtrees are the result of post-pruning options like subtree raising and subtree replacement, as mentioned earlier in this chapter.

The tree's `summary()` output also shows the tree's performance on the training data:

```
> summary(credit_boost10)

(a)   (b)    <-classified as
-----
633     2    (a): class no
17    248    (b): class yes
```

The classifier made 19 mistakes on 900 training examples for an error rate of 2.1 percent. This is quite an improvement over the 13.1 percent training error rate we noted before boosting! However, it remains to be seen whether we see a similar improvement on the test data. Let's take a look:

```
> credit_boost_pred10 <- predict(credit_boost10, credit_test)
> CrossTable(credit_test$default, credit_boost_pred10,
  prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
  dnn = c('actual default', 'predicted default'))
```

The resulting table is as follows:

		predicted default		Row Total
actual default		no	yes	
no		58	7	65
		0.580	0.070	
yes		19	16	35
		0.190	0.160	
Column Total		77	23	100

Here, we reduced the total error rate from 33 percent prior to boosting to 26 percent in the boosted model. This may not seem like a large improvement, but it is not too far away from the 25 percent reduction we expected. That said, if boosting can be added this easily, why not apply it by default to every decision tree? The reason is twofold. First, if building a decision tree once takes a great deal of computation time, building many trees may be computationally impractical. Secondly, if the training data is very noisy, then boosting might not result in an improvement at all. Still, if greater accuracy is needed, it's worth giving boosting a try.

On the other hand, the model is still doing poorly at identifying the true defaults, predicting only 46 percent correctly (16 out of 35) compared to 31 percent (11 of 35) in the simpler model. Let's investigate one more option to see if we can reduce these types of costly errors.

Making some mistakes cost more than others

Giving a loan to an applicant who is likely to default can be an expensive mistake. One solution to reduce the number of false negatives may be to reject a larger number of borderline applicants under the assumption that the interest that the bank would earn from a risky loan is far outweighed by the massive loss it would incur if the money is not paid back at all.

The C5.0 algorithm allows us to assign a penalty to different types of errors in order to discourage a tree from making more costly mistakes. The penalties are designated in a **cost matrix**, which specifies how many times more costly each error is relative to any other.

To begin constructing the cost matrix, we need to start by specifying the dimensions. Since the predicted and actual values can both take two values, yes or no, we need to describe a 2x2 matrix using a list of two vectors, each with two values. At the same time, we'll also name the matrix dimensions to avoid confusion later:

```
> matrix_dimensions <- list(c("no", "yes"), c("no", "yes"))
> names(matrix_dimensions) <- c("predicted", "actual")
```

Examining the new object shows that our dimensions have been set up correctly:

```
> matrix_dimensions
```

```
$predicted
[1] "no"   "yes"
$actual
[1] "no"   "yes"
```

Next, we need to assign the penalty for the various types of errors by supplying four values to fill the matrix. Since R fills a matrix by filling columns one by one from top to bottom, we need to supply the values in a specific order:

1. Predicted no, actual no
2. Predicted yes, actual no
3. Predicted no, actual yes
4. Predicted yes, actual yes

Suppose we believe that a loan default costs the bank four times as much as a missed opportunity. Our penalty values then could be defined as:

```
> error_cost <- matrix(c(0, 1, 4, 0), nrow = 2,
                           dimnames = matrix_dimensions)
```

This creates the following matrix:

```
> error_cost

      actual
predicted no yes
  no    0    4
  yes   1    0
```

As defined by this matrix, there is no cost assigned when the algorithm classifies a no or yes correctly, but a false negative has a cost of 4 versus a false positive's cost of 1. To see how this impacts classification, let's apply it to our decision tree using the `costs` parameter of the `C5.0()` function. We'll otherwise use the same steps as before:

```
> credit_cost <- C5.0(default ~ ., data = credit_train,
                           costs = error_cost)
> credit_cost_pred <- predict(credit_cost, credit_test)
> CrossTable(credit_test$default, credit_cost_pred,
              prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
              dnn = c('actual default', 'predicted default'))
```

This produces the following confusion matrix:

		predicted default		Row Total
actual default	no	yes		
no	34	31	65	
	0.340	0.310		
yes	5	30	35	
	0.050	0.300		
Column Total	39	61	100	

Compared to our boosted model, this version makes more mistakes overall: 36 percent error here versus 26 percent in the boosted case. However, the types of mistakes are very different. Where the previous models classified only 31 and 46 percent of defaults correctly, in this model, $30 / 35 = 86\%$ of the actual defaults were correctly predicted to be defaults. This trade-off resulting in a reduction of false negatives at the expense of increasing false positives may be acceptable if our cost estimates were accurate.

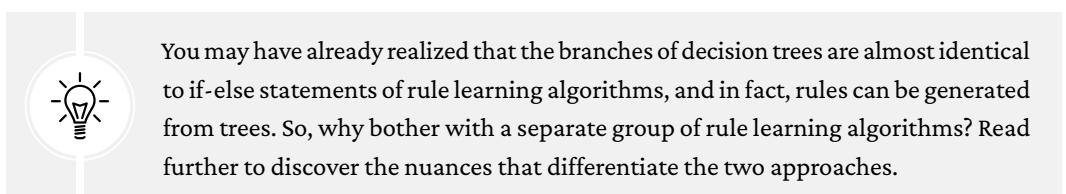
Understanding classification rules

Classification rules represent knowledge in the form of logical if-else statements that assign a class to unlabeled examples. They are specified in terms of an **antecedent** and a **consequent**, which form a statement stating that “*if this happens, then that happens.*” The antecedent comprises certain combinations of feature values, while the consequent specifies the class value to assign if the rule’s conditions are met. A simple rule might state, “*if the computer is making a clicking sound, then it is about to fail.*”

Rule learners are closely related siblings of decision tree learners and are often used for similar types of tasks. Like decision trees, they can be used for applications that generate knowledge for future action, such as:

- Identifying conditions that lead to hardware failure in mechanical devices
- Describing the key characteristics of groups of people for customer segmentation
- Finding conditions that precede large drops or increases in the prices of shares on the stock market

Rule learners do have some distinct contrasts relative to decision trees. Unlike a tree, which must be followed through a series of branching decisions, rules are propositions that can be read much like independent statements of fact. Additionally, for reasons that will be discussed later, the results of a rule learner can be more simple, direct, and easier to understand than a decision tree built on the same data.



Rule learners are generally applied to problems where the features are primarily or entirely nominal. They do well at identifying rare events, even if the rare event occurs only for a very specific interaction among feature values.

Separate and conquer

Rule learning classification algorithms utilize a heuristic known as **separate and conquer**. The process involves identifying a rule that covers a subset of examples in the training data and then separating this partition from the remaining data. As rules are added, additional subsets of data are separated until the entire dataset has been covered and no more examples remain. Although separate and conquer is in many ways similar to the divide and conquer heuristic covered earlier, it differs in subtle ways that will become clear soon.

One way to imagine the rule learning process of separate and conquer is to imagine drilling down into the data by creating increasingly specific rules to identify class values. Suppose you were tasked with creating rules to identify whether or not an animal is a mammal. You could depict the set of all animals as a large space, as shown in the following diagram:

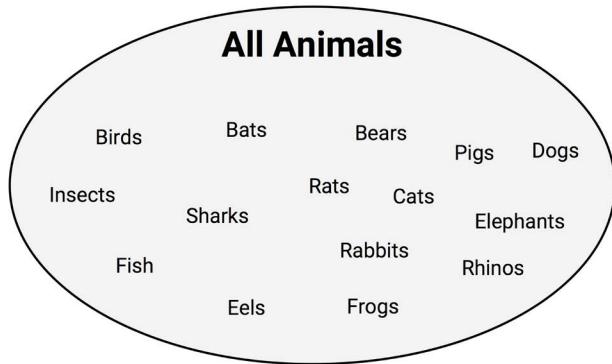


Figure 5.8: A rule learning algorithm may help divide animals into groups of mammals and non-mammals

A rule learner begins by using the available features to find homogeneous groups. For example, using a feature that indicates whether the species travels via land, sea, or air, the first rule might suggest that any land-based animals are mammals:

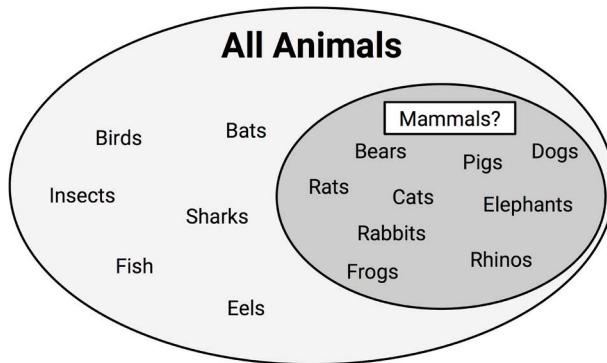


Figure 5.9: A potential rule considers animals that travel on land to be mammals

Do you notice any problems with this rule? If you're an animal lover, you might have realized that frogs are amphibians, not mammals. Therefore, our rule needs to be a bit more specific. Let's drill down further by suggesting that mammals must walk on land and have a tail:

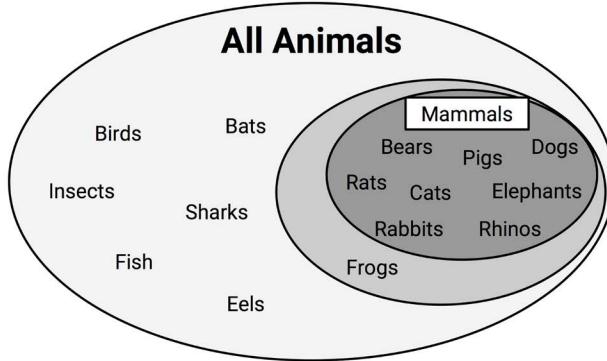


Figure 5.10: A more specific rule suggests that animals that walk on land and have tails are mammals

As shown in the previous figure, the new rule results in a subset of animals that are entirely mammals. Thus, the subset of mammals can be separated from the other data and the frogs are returned to the pool of remaining animals—no pun intended!

An additional rule can be defined to separate out the bats, the only remaining mammal. A potential feature distinguishing bats from the remaining animals would be the presence of fur. Using a rule built around this feature, we have then correctly identified all the animals:

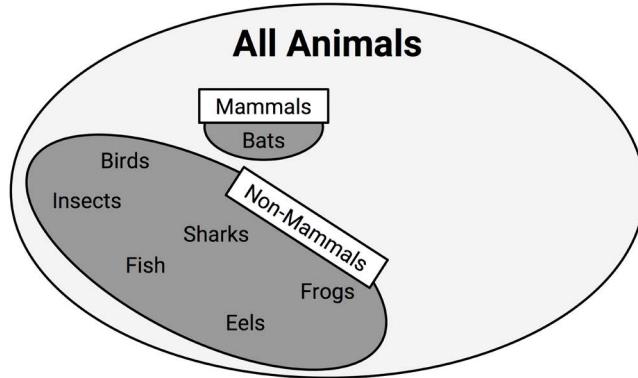


Figure 5.11: A rule stating that animals with fur are mammals perfectly classifies the remaining animals

At this point, since all training instances have been classified, the rule learning process would stop. We learned a total of three rules:

- Animals that walk on land and have tails are mammals
- If the animal does not have fur, it is not a mammal
- Otherwise, the animal is a mammal

The previous example illustrates how rules gradually consume larger and larger segments of data to eventually classify all instances. As the rules seem to cover portions of the data, separate and conquer algorithms are also known as **covering algorithms**, and the resulting rules are called covering rules. In the next section, we will learn how covering rules are applied in practice by examining a simple rule learning algorithm. We will then examine a more complex rule learner and apply both algorithms to a real-world problem.

The 1R algorithm

Suppose a television game show has an animal hidden behind a large curtain. You are asked to guess whether it is a mammal and if correct, you win a large cash prize. You are not given any clues about the animal's characteristics, but you know that a very small portion of the world's animals are mammals. Consequently, you guess "non-mammal." What do you think about your chances of winning?

Choosing this, of course, maximizes your odds of winning the prize, as it is the most likely outcome under the assumption the animal was chosen at random. Clearly, this game show is a bit ridiculous, but it demonstrates the simplest classifier, **ZeroR**, which is a rule learner that considers no features and literally learns no rules (hence the name). For every unlabeled example, regardless of the values of its features, it predicts the most common class. This algorithm has very little real-world utility, except that it provides a simple baseline for comparison to other, more sophisticated, rule learners.

The **1R algorithm** (also known as **One Rule** or **OneR**), improves over ZeroR by selecting a single rule. Although this may seem overly simplistic, it tends to perform better than you might expect. As demonstrated in empirical studies, the accuracy of this algorithm can approach that of much more sophisticated algorithms for many real-world tasks.



For an in-depth look at the surprising performance of 1R, see *Very Simple Classification Rules Perform Well on Most Commonly Used Datasets*, Holte, RC, *Machine Learning*, 1993, Vol. 11, pp. 63-91.

The strengths and weaknesses of the 1R algorithm are shown in the following table:

Strengths	Weaknesses
<ul style="list-style-type: none">Generates a single, easy-to-understand, human-readable ruleOften performs surprisingly wellCan serve as a benchmark for more complex algorithms	<ul style="list-style-type: none">Uses only a single featureProbably overly simplistic

The way this algorithm works is simple. For each feature, 1R divides the data into groups with similar values of the feature. Then, for each segment, the algorithm predicts the majority class. The error rate for the rule based on each feature is calculated and the rule with the fewest errors is chosen as the one rule.

The following tables show how this would work for the animal data we looked at earlier:

Animal	Travels By	Has Fur	Mammal
Bats	Air	Yes	Yes
Bears	Land	Yes	Yes
Birds	Air	No	No
Cats	Land	Yes	Yes
Dogs	Land	Yes	Yes
Eels	Sea	No	No
Elephants	Land	No	Yes
Fish	Sea	No	No
Frogs	Land	No	No
Insects	Air	No	No
Pigs	Land	No	Yes
Rabbits	Land	Yes	Yes
Rats	Land	Yes	Yes
Rhinos	Land	No	Yes
Sharks	Sea	No	No

Full Dataset

Travels By	Predicted	Mammal
Air	No	Yes
Air	No	No
Air	No	No
Land	Yes	Yes
Land	Yes	No
Land	Yes	Yes
Land	Yes	Yes
Land	Yes	Yes
Sea	No	No
Sea	No	No
Sea	No	No

Rule for "Travels By"
Error Rate = 2 / 15

Has Fur	Predicted	Mammal
No	No	No
No	No	No
No	No	Yes
No	No	No
No	No	No
No	No	No
No	No	Yes
No	No	Yes
No	No	No
Yes	Yes	Yes

Rule for "Has Fur"
Error Rate = 3 / 15

Figure 5.12: The 1R algorithm chooses the single rule with the lowest misclassification rate

For the *Travels By* feature, the dataset was divided into three groups: *Air*, *Land*, and *Sea*. Animals in the *Air* and *Sea* groups were predicted to be non-mammal, while animals in the *Land* group were predicted to be mammals. This resulted in two errors: bats and frogs.

The *Has Fur* feature divided animals into two groups. Those with fur were predicted to be mammals, while those without fur were not. Three errors were counted: pigs, elephants, and rhinos. As the *Travels By* feature resulted in fewer errors, the 1R algorithm would return the following:

- If the animal travels by air, it is not a mammal
- If the animal travels by land, it is a mammal
- If the animal travels by sea, it is not a mammal

The algorithm stops here, having found the single most important rule.

Obviously, this rule learning algorithm may be too basic for some tasks. Would you want a medical diagnosis system to consider only a single symptom, or an automated driving system to stop or accelerate your car based on only a single factor? For these types of tasks, a more sophisticated rule learner might be useful. We'll learn about one in the following section.

The RIPPER algorithm

Early rule learning algorithms were plagued by a couple of problems. First, they were notorious for being slow, which made them ineffective for the increasing number of large datasets. Second, they were often prone to being inaccurate on noisy data.

A first step toward solving these problems was proposed in 1994 by Johannes Furnkranz and Gerhard Widmer. Their **incremental reduced error pruning (IREP) algorithm** uses a combination of pre-pruning and post-pruning methods that grow very complex rules and prune them before separating the instances from the full dataset. Although this strategy helped the performance of rule learners, decision trees often still performed better.



For more information on IREP, see *Incremental Reduced Error Pruning, Furnkranz, J and Widmer, G, Proceedings of the 11th International Conference on Machine Learning, 1994, pp. 70-77.*

Rule learners took another step forward in 1995 when William W. Cohen introduced the **repeated incremental pruning to produce error reduction (RIPPER) algorithm**, which improved upon IREP to generate rules that match or exceed the performance of decision trees.



For more detail on RIPPER, see *Fast Effective Rule Induction, Cohen, WW, Proceedings of the 12th International Conference on Machine Learning, 1995, pp. 115-123.*

As outlined in the following table, the strengths and weaknesses of RIPPER are generally comparable to decision trees. The chief benefit is that they may result in a slightly more parsimonious model.

Strengths	Weaknesses
<ul style="list-style-type: none">Generates easy-to-understand, human-readable rulesEfficient on large and noisy datasetsGenerally, produces a simpler model than a comparable decision tree	<ul style="list-style-type: none">May result in rules that seem to defy common sense or expert knowledgeNot ideal for working with numeric dataMight not perform as well as more complex models

Having evolved from several iterations of rule learning algorithms, the RIPPER algorithm is a patchwork of efficient heuristics for rule learning. Due to its complexity, a discussion of the implementation details is beyond the scope of this book. However, it can be understood in general terms as a three-step process:

1. Grow
2. Prune
3. Optimize

The growing phase uses the separate and conquer technique to greedily add conditions to a rule until it perfectly classifies a subset of data or runs out of attributes for splitting. Like decision trees, the information gain criterion is used to identify the next splitting attribute. When increasing a rule's specificity no longer reduces entropy, the rule is immediately pruned. Steps one and two are repeated until reaching a stopping criterion, at which point the entire set of rules is optimized using a variety of heuristics.

The RIPPER algorithm can create much more complex rules than the 1R algorithm can, as it can consider more than one feature. This means that it can create rules with multiple antecedents such as “if an animal flies and has fur, then it is a mammal.” This improves the algorithm’s ability to model complex data, but just like decision trees, it means that the rules can quickly become difficult to comprehend.



The evolution of classification rule learners didn’t stop with RIPPER. New rule learning algorithms are being proposed rapidly. A survey of literature shows algorithms called IREP++, SLIPPER, TRIPPER, among many others.

Rules from decision trees

Classification rules can also be obtained directly from decision trees. Beginning at a leaf node and following the branches back to the root, you obtain a series of decisions. These can be combined into a single rule. The following figure shows how rules could be constructed from the decision tree for predicting movie success:

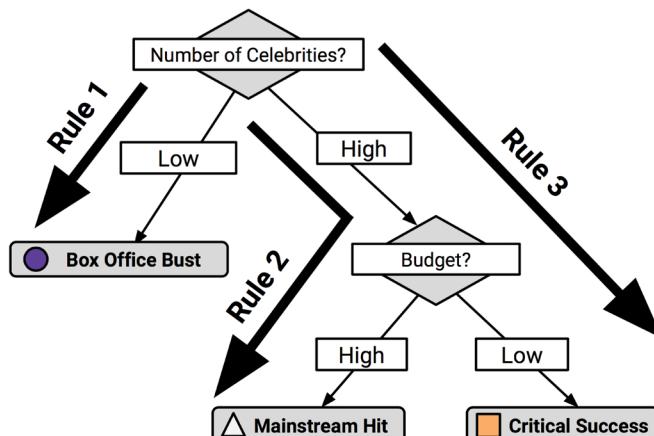


Figure 5.13: Rules can be generated from decision trees by following paths from the root node to each leaf node

Following the paths from the root node down to each leaf, the rules would be:

1. If the number of celebrities is low, then the movie will be a *Box Office Bust*
2. If the number of celebrities is high and the budget is high, then the movie will be a *Main-stream Hit*
3. If the number of celebrities is high and the budget is low, then the movie will be a *Critical Success*

For reasons that will be made clear in the following section, the chief downside to using a decision tree to generate rules is that the resulting rules are often more complex than those learned by a rule learning algorithm. The divide and conquer strategy employed by decision trees biases the results differently to that of a rule learner. On the other hand, it is sometimes more computationally efficient to generate rules from trees.



The `C5.0()` function in the `C50` package will generate a model using classification rules if you specify `rules = TRUE` when training the model.

What makes trees and rules greedy?

Decision trees and rule learners are known as **greedy learners** because they use data on a first-come, first-served basis. Both the divide and conquer heuristic used by decision trees and the separate and conquer heuristic used by rule learners attempt to make partitions one at a time, finding the most homogeneous partition first, followed by the next best, and so on, until all examples have been classified.

The downside to the greedy approach is that greedy algorithms are not guaranteed to generate the optimal, most accurate, or smallest number of rules for a particular dataset. By taking the low-hanging fruit early, a greedy learner may quickly find a single rule that is accurate for one subset of data; however, in doing so, the learner may miss the opportunity to develop a more nuanced set of rules with better overall accuracy on the entire set of data. However, without using the greedy approach to rule learning, it is likely that for all but the smallest of datasets, rule learning would be computationally infeasible.



Figure 5.14: Both decision trees and classification rule learners are greedy algorithms

Though both trees and rules employ greedy learning heuristics, there are subtle differences in how they build rules. Perhaps the best way to distinguish them is to note that once divide and conquer splits on a feature, the partitions created by the split may not be re-conquered, only further subdivided. In this way, a tree is permanently limited by its history of past decisions. In contrast, once separate and conquer finds a rule, any examples not covered by all the rule's conditions may be re-conquered.

To illustrate this contrast, consider the previous case in which we built a rule learner to determine whether an animal was a mammal. The rule learner identified three rules that perfectly classify the example animals:

1. Animals that walk on land and have tails are mammals (bears, cats, dogs, elephants, pigs, rabbits, rats, and rhinos)
2. If the animal does not have fur, it is not a mammal (birds, eels, fish, frogs, insects, and sharks)
3. Otherwise, the animal is a mammal (bats)

In contrast, a decision tree built on the same data might have come up with four rules to achieve the same perfect classification:

1. If an animal walks on land and has a tail, then it is a mammal (bears, cats, dogs, elephants, pigs, rabbits, rats, and rhinos)

2. If an animal walks on land and does not have a tail, then it is not a mammal (frogs)
3. If the animal does not walk on land and has fur, then it is a mammal (bats)
4. If the animal does not walk on land and does not have fur, then it is not a mammal (birds, insects, sharks, fish, and eels)

The different result across these two approaches has to do with what happens to the frogs after they are separated by the “walk on land” decision. Where the rule learner allows frogs to be re-conquered by the “do not have fur” decision, the decision tree cannot modify the existing partitions and therefore must place the frog into its own rule.

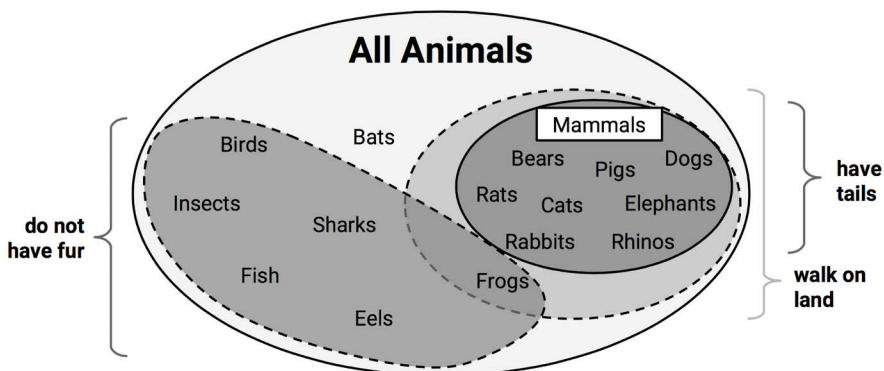


Figure 5.15: The handling of frogs distinguishes the divide and conquer and separate and conquer heuristics. The latter approach allows the frogs to be re-conquered by later rules.

On the one hand, because rule learners can reexamine cases that were considered but ultimately not covered as part of prior rules, rule learners often find a more parsimonious set of rules than those generated by decision trees. On the other hand, this reuse of data means that the computational cost of rule learners may be somewhat higher than for decision trees.

Example – identifying poisonous mushrooms with rule learners

Each year, many people fall ill and some even die from ingesting poisonous wild mushrooms. Since many mushrooms are very similar to each other in appearance, occasionally, even experienced mushroom gatherers are poisoned.

Unlike the identification of harmful plants, such as poison oak or poison ivy, there are no clear rules like “leaves of three, let them be” for identifying whether a wild mushroom is poisonous or edible.

Complicating matters, many traditional rules such as “poisonous mushrooms are brightly colored” provide dangerous or misleading information. If simple, clear, and consistent rules were available for identifying poisonous mushrooms, they could save the lives of foragers.

As one of the strengths of rule learning algorithms is the fact that they generate easy-to-understand rules, they seem like an appropriate fit for this classification task. However, the rules will only be as useful as they are accurate.

Step 1 – collecting data

To identify rules for distinguishing poisonous mushrooms, we will use the Mushroom dataset by Jeff Schlimmer of Carnegie Mellon University. The raw dataset is available freely from the UCI Machine Learning Repository (<http://archive.ics.uci.edu/ml>).

The dataset includes information on 8,124 mushroom samples from 23 species of gilled mushrooms listed in the *Audubon Society Field Guide to North American Mushrooms* (1981). In the field guide, each of the mushroom species is identified as “definitely edible,” “definitely poisonous,” or “likely poisonous, and not recommended to be eaten.” For the purposes of this dataset, the latter group was combined with the “definitely poisonous” group to make two classes: poisonous and non-poisonous. The data dictionary available on the UCI website describes the 22 features of the mushroom samples, including characteristics such as cap shape, cap color, odor, gill size and color, stalk shape, and habitat.



This chapter uses a slightly modified version of the mushroom data. If you plan on following along with the example, download the `mushrooms.csv` file from the Packt Publishing GitHub repository for this chapter and save it to your R working directory.

Step 2 – exploring and preparing the data

We begin by using `read.csv()` to import the data for our analysis. Since all 22 features and the target class are nominal, we will set `stringsAsFactors = TRUE` to take advantage of the automatic factor conversion:

```
> mushrooms <- read.csv("mushrooms.csv", stringsAsFactors = TRUE)
```

The output of the `str(mushrooms)` command notes that the data contains 8,124 observations of 23 variables as the data dictionary had described. While most of the `str()` output is unremarkable, one feature is worth mentioning. Do you notice anything peculiar about the `veil_type` variable in the following line?

```
$ veil_type : Factor w/ 1 level "partial": 1 1 1 1 1 1 ...
```

If you found it to be strange that a factor has only one level, you are correct. The data dictionary lists two levels for this feature: `partial` and `universal`; however, all examples in our data are classified as `partial`. It is likely that this data element was somehow coded incorrectly. In any case, since the veil type does not vary across samples, it does not provide any useful information for prediction. We will drop this variable from our analysis using the following command:

```
> mushrooms$veil_type <- NULL
```

By assigning `NULL` to the `veil_type` vector, R eliminates the feature from the mushrooms data frame.

Before going much further, we should take a quick look at the distribution of the mushroom type variable in our dataset:

```
> table(mushrooms$type)
```

edible	poisonous
4208	3916

About 52 percent of the mushroom samples are edible, while 48 percent are poisonous.

For the purposes of this experiment, we will consider the 8,214 samples in the mushroom data to be an exhaustive set of all the possible wild mushrooms. This is an important assumption because it means that we do not need to hold some samples out of the training data for testing purposes. We are not trying to develop rules that cover unforeseen types of mushrooms; we are merely trying to find rules that accurately depict the complete set of known mushroom types. Therefore, we can build and test the model on the same data.

Step 3 – training a model on the data

If we trained a hypothetical ZeroR classifier on this data, what would it predict? Since ZeroR ignores all of the features and simply predicts the target's mode, in plain language, its rule would state that “all mushrooms are edible.” Obviously, this is not a very helpful classifier because it would leave a mushroom gatherer sick or dead for nearly half of the mushroom samples! Our rules will need to do much better than this benchmark in order to provide safe advice that can be published. At the same time, we need simple rules that are easy to remember.

Since simple rules can still be useful, let’s see how a very simple rule learner performs on the mushroom data. Toward this end, we will apply the 1R classifier, which will identify the single feature that is the most predictive of the target class and use this feature to construct a rule.

We will use the 1R implementation found in the OneR package by Holger von Jouanne-Diedrich at the Aschaffenburg University of Applied Sciences. This is a relatively new package, which implements 1R in native R code for speed and ease of use. If you don't already have this package, it can be installed using the `install.packages("OneR")` command and loaded by typing `library(OneR)`.

1R classification rule syntax
Using the <code>OneR()</code> function in the <code>OneR</code> package
Building the classifier:
<pre>m <- OneR(class ~ predictors, data = mydata)</pre> <ul style="list-style-type: none"> • <code>class</code> is the column in the <code>mydata</code> data frame to be predicted • <code>predictors</code> is an R formula specifying the features in the <code>mydata</code> data frame to use for prediction • <code>data</code> is the data frame in which <code>class</code> and <code>predictors</code> can be found
The function will return a OneR model object that can be used to make predictions.
Making predictions:
<pre>p <- predict(m, test)</pre> <ul style="list-style-type: none"> • <code>m</code> is a model trained by the <code>OneR()</code> function • <code>test</code> is a data frame containing test data with the same features as the training data used to build the classifier
The function will return a vector of predicted class values.
Example:
<pre>mushroom_classifier <- OneR(type ~ odor + cap_color, data = mushroom_train) mushroom_prediction <- predict(mushroom_classifier, mushroom_test)</pre>

Figure 5.16: 1R classification rule syntax

Like C5.0, the `OneR()` function uses the R formula syntax to specify the model to be trained. Using the formula `type ~ .` with `OneR()` allows our first rule learner to consider all possible features in the mushroom data when predicting the mushroom type:

```
> mushroom_1R <- OneR(type ~ ., data = mushrooms)
```

To examine the rules it created, we can type the name of the classifier object:

```
> mushroom_1R

Call:
OneR.formula(formula = type ~ ., data = mushrooms)

Rules:

If odor = almond    then type = edible
If odor = anise     then type = edible
If odor = creosote  then type = poisonous
If odor = fishy     then type = poisonous
If odor = foul      then type = poisonous
If odor = musty    then type = poisonous
If odor = none      then type = edible
If odor = pungent   then type = poisonous
If odor = spicy     then type = poisonous

Accuracy:
8004 of 8124 instances classified correctly (98.52%)
```

Examining the output, we see that the `odor` feature was selected for rule generation. The categories of `odor`, such as `almond`, `anise`, and so on, specify rules for whether the mushroom is likely to be `edible` or `poisonous`. For instance, if the mushroom smells `fishy`, `foul`, `musty`, `pungent`, `spicy`, or like `creosote`, the mushroom is likely to be `poisonous`. On the other hand, mushrooms with more pleasant smells, like `almond` and `anise`, and those with no smell at all, are predicted to be `edible`. For the purposes of a field guide for mushroom gathering, these rules could be summarized in a simple rule of thumb: “if the mushroom smells unappetizing, then it is likely to be `poisonous`.”

Step 4 – evaluating model performance

The last line of the output notes that the rules correctly predict the edibility for 8,004 of the 8,124 mushroom samples, or nearly 99 percent. Anything short of perfection, however, runs the risk of poisoning someone if the model were to classify a `poisonous` mushroom as `edible`.

To determine whether this occurred, let's examine a confusion matrix of the predicted versus actual values. This requires us to first generate the 1R model's predictions, then compare the predictions to the actual values:

```
> mushroom_1R_pred <- predict(mushroom_1R, mushrooms)
> table(actual = mushrooms$type, predicted = mushroom_1R_pred)
```

		predicted	
actual	edible		poisonous
	edible	4208	0
poisonous	120	3796	

Here, we can see where our rules went wrong. The table's columns indicate the predicted edibility of the mushroom while the table's rows divide the 4,208 actually edible mushrooms and the 3,916 actually poisonous mushrooms. Examining the table, we can see that although the 1R classifier did not classify any edible mushrooms as poisonous, it did classify 120 poisonous mushrooms as edible, which makes for an incredibly dangerous mistake!

Considering that the learner utilized only a single feature, it did reasonably well; if you avoid unappetizing smells when foraging for mushrooms, you will almost always avoid a trip to the hospital. That said, close does not cut it when lives are involved, not to mention the field guide publisher might not be happy about the prospect of a lawsuit when its readers fall ill. Let's see if we can add a few more rules and develop an even better classifier.

Step 5 – improving model performance

For a more sophisticated rule learner, we will use `JRip()`, a Java-based implementation of the RIPPER algorithm. The `JRip()` function is included in the `RWeka` package, which gives R access to the machine learning algorithms in the Java-based Weka software application by Ian H. Witten and Eibe Frank.



Weka is a popular open source and full-featured graphical application for performing data mining and machine learning tasks—one of the earliest such tools. For more information on Weka, see <http://www.cs.waikato.ac.nz/~ml/weka/>.

The `RWeka` package depends on the `rJava` package, which itself requires the **Java development kit (JDK)** to be installed on the host computer before installation. This can be downloaded from <https://www.java.com/> and installed using the instructions specific to your platform. After installing Java, use the `install.packages("RWeka")` command to install `RWeka` and its dependencies, then load the `RWeka` package using the `library(RWeka)` command.



Java is a set of programming tools available at no cost, which allows the development and use of cross-platform applications such as Weka. Although it was once included by default with many computers, this is no longer true. Unfortunately, it can be tricky to install, especially on Apple computers. If you are having trouble, be sure you have the latest Java version. Additionally, on Microsoft Windows, you may need to set your environment variables like `JAVA_HOME` correctly, and check your PATH settings (search the web for details). On macOS or Linux computers, you may also try executing R CMD javareconf from a terminal window then install the R package `rJava` from source using the R command `install.packages("rJava", type = "source")`. If all else fails, you may try a free Posit Cloud account (<https://posit.cloud/>), which offers an RStudio environment in which Java is already installed.

With `rJava` and `RWeka` installed, the process of training a `JRip()` model is very similar to training a `OneR()` model, as shown in the syntax box that follows. This is one of the pleasant benefits of the R formula interface: the syntax is consistent across algorithms, which makes it simple to compare a variety of models.

RIPPER classification rule syntax

Using the `JRip()` function in the `RWeka` package

Building the classifier:

```
m <- JRip(class ~ predictors, data = mydata)
```

- `class` is the column in the `mydata` data frame to be predicted
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` is the data frame in which `class` and `predictors` can be found

The function will return a RIPPER model object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test)
```

- `m` is a model trained by the `JRip()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier

The function will return a vector of predicted class values.

Example:

```
mushroom_classifier <- JRip(type ~ odor + cap_color,
                               data = mushroom_train)
mushroom_prediction <- predict(mushroom_classifier,
                                mushroom_test)
```

Figure 5.17: RIPPER classification rule syntax

Let's train the `JRip()` rule learner as we did with `OneR()`, allowing it to find rules among all of the available features:

```
> mushroom_JRip <- JRip(type ~ ., data = mushrooms)
```

To examine the rules, type the name of the classifier:

```
> mushroom_JRip
```

```
JRIP rules:
```

```
=====
```

```
(odor = foul) => type=poisonous (2160.0/0.0)
(gill_size = narrow) and (gill_color = buff)
=> type=poisonous (1152.0/0.0)
(gill_size = narrow) and (odor = pungent)
=> type=poisonous (256.0/0.0)
(odor = creosote) => type=poisonous (192.0/0.0)
```

```
(spore_print_color = green) => type=poisonous (72.0/0.0)
(stalk_surface_below_ring = scaly)
    and (stalk_surface_above_ring = silky)
        => type=poisonous (68.0/0.0)
(habitat = leaves) and (gill_attachment = free)
    and (population = clustered)
        => type=poisonous (16.0/0.0)
=> type=edible (4208.0/0.0)
Number of Rules : 8
```

The JRip() classifier learned a total of eight rules from the mushroom data.

An easy way to read these rules is to think of them as a list of if-else statements, similar to programming logic. The first three rules could be expressed as:

- If the odor is foul, then the mushroom type is poisonous
- If the gill size is narrow and the gill color is buff, then the mushroom type is poisonous
- If the gill size is narrow and the odor is pungent, then the mushroom type is poisonous

Finally, the eighth rule implies that any mushroom sample that was not covered by the preceding seven rules is edible. Following the example of our programming logic, this can be read as:

- Else, the mushroom is edible

The numbers next to each rule indicate the number of instances covered by the rule and a count of misclassified instances. Notably, there were no misclassified mushroom samples using these eight rules. As a result, the number of instances covered by the last rule is exactly equal to the number of edible mushrooms in the data ($N = 4,208$).

The following figure provides a rough illustration of how the rules are applied to the mushroom data. If you imagine the large oval as containing all mushroom species, the rule learner identifies features, or sets of features, that separate homogeneous segments from the larger group. First, the algorithm found a large group of poisonous mushrooms uniquely distinguished by their foul odor. Next, it found smaller and more specific groups of poisonous mushrooms. By identifying covering rules for each of the varieties of poisonous mushrooms, all remaining mushrooms were edible.

Thanks to Mother Nature, each variety of mushrooms was unique enough that the classifier was able to achieve 100 percent accuracy.

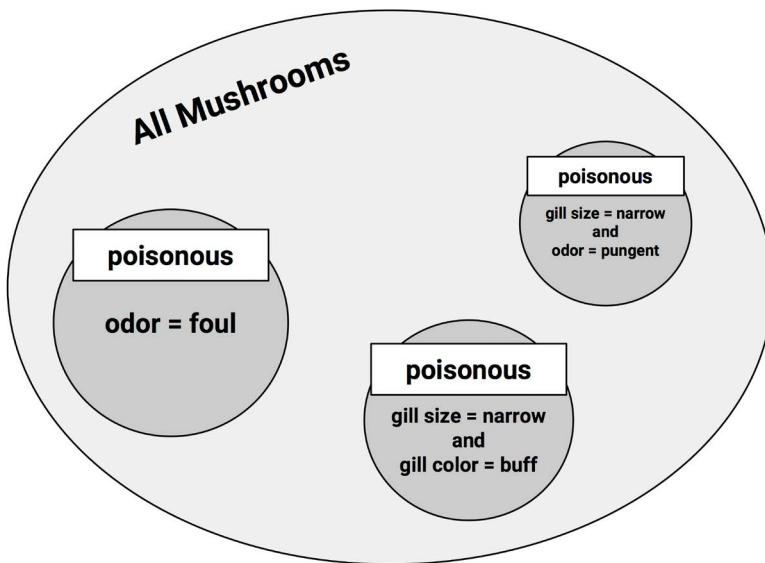


Figure 5.18: A sophisticated rule learning algorithm identified rules to perfectly cover all types of poisonous mushrooms

Summary

This chapter covered two classification methods that use so-called “greedy” algorithms to partition the data according to feature values. Decision trees use a divide and conquer strategy to create flowchart-like structures, while rule learners separate and conquer data to identify logical if-else rules. Both methods produce models that can be interpreted without a statistical background.

One popular and highly configurable decision tree algorithm is C5.0. We used the C5.0 algorithm to create a tree to predict whether a loan applicant will default. Using options for boosting and cost-sensitive errors, we were able to improve our accuracy and avoid risky loans that could cost the bank more money.

We also used two rule learners, 1R and RIPPER, to develop rules for identifying poisonous mushrooms. The 1R algorithm used a single feature to achieve 99 percent accuracy in identifying potentially fatal mushroom samples. On the other hand, the set of eight rules generated by the more sophisticated RIPPER algorithm correctly identified the edibility of every mushroom.

This merely scratches the surface of how trees and rules can be used. The next chapter, *Chapter 6, Forecasting Numeric Data – Regression Methods*, describes techniques known as regression trees and model trees, which use decision trees for numeric prediction rather than classification. In *Chapter 8, Finding Patterns – Market Basket Analysis Using Association Rules*, we will see how association rules—a close relative of classification rules—can be used to identify groups of items in transactional data. Lastly, in *Chapter 14, Building Better Learners*, we will discover how the performance of decision trees can be improved by grouping them together in a model known as a random forest, in addition to other advanced modeling techniques that rely on decision trees.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



6

Forecasting Numeric Data – Regression Methods

Mathematical relationships help us to make sense of many aspects of everyday life. For example, body weight is a function of one's calorie intake; income is often related to education and job experience; and poll numbers help to estimate a presidential candidate's odds of being re-elected.

When such patterns are formulated with numbers, we gain additional clarity. For example, an additional 250 kilocalories consumed daily may result in nearly a kilogram of weight gain per month; each year of job experience may be worth an additional \$1,000 in yearly salary; and a president is more likely to be re-elected when the economy is strong. Obviously, these equations do not perfectly fit every situation, but we expect that they are reasonably correct most of the time.

This chapter extends our machine learning toolkit by going beyond the classification methods covered previously and introducing techniques for estimating relationships within numeric data. While examining several real-world numeric prediction tasks, you will learn:

- The basic statistical principles used in regression, a technique that models the size and strength of numeric relationships
- How to prepare data for regression analysis, estimate and interpret regression models, and apply regression variants such as generalized linear models
- A pair of hybrid techniques known as regression trees and model trees, which adapt decision tree classifiers for numeric prediction tasks

Based on a large body of work in the field of statistics, the methods used in this chapter are a bit heavier on math than those covered previously, but don't worry! Even if your algebra skills are a bit rusty, R takes care of the heavy lifting.

Understanding regression

Regression involves specifying the relationship between a single numeric **dependent variable** (the value to be predicted) and one or more numeric **independent variables** (the predictors). As the name implies, the dependent variable depends upon the value of the independent variable or variables. The simplest forms of regression assume that the relationship between the independent and dependent variables follows a straight line.



The origin of the term “regression” to describe the process of fitting lines to data is rooted in a study of genetics by Sir Francis Galton in the late 19th century. He discovered that fathers who were extremely short or tall tended to have sons whose heights were closer to the average height. He called this phenomenon “regression to the mean.”

You might recall from basic algebra that lines can be defined in a **slope-intercept form** similar to $y = a + bx$. In this form, the letter y indicates the dependent variable and x indicates the independent variable. The **slope** term b specifies how much the line rises for each increase in x . Positive values define lines that slope upward while negative values define lines that slope downward. The term a is known as the **intercept** because it specifies the point where the line crosses, or intercepts, the vertical y axis. It indicates the value of y when $x = 0$.

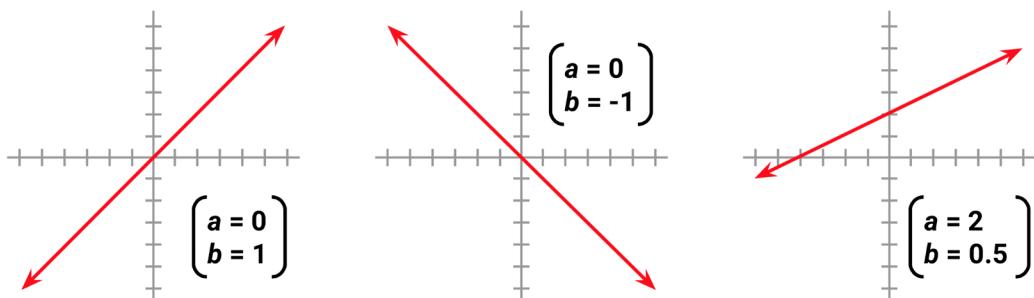


Figure 6.1: Examples of lines with various slopes and intercepts

Regression equations model data using a similar slope-intercept format. The machine's job is to identify values of a and b such that the specified line is best able to relate the supplied x values to the values of y .

There may not always be a single set of a and b parameters that perfectly relates the values, so the machine must also have some way to quantify the margin of error and choose the best fit. We'll discuss this in depth shortly.

Regression analysis is used for a huge variety of tasks—it is almost surely the most widely used machine learning method. It can be used both for explaining the past and extrapolating into the future and can be applied to nearly any task. Some specific use cases include:

- Examining how populations and individuals vary by their measured characteristics, in scientific studies in the fields of economics, sociology, psychology, physics, and ecology
- Quantifying the causal relationship between an event and its response, in cases such as clinical drug trials, engineering safety tests, or market research
- Identifying patterns that can be used to forecast future behavior given known criteria, such as for predicting insurance claims, natural disaster damage, election results, and crime rates

Regression methods are also used for **statistical hypothesis testing**, which determines whether a premise is likely to be true or false in light of observed data. The regression model's estimates of the strength and consistency of a relationship provide information that can be used to assess whether the observations are due to chance alone.



Hypothesis testing is extremely nuanced and falls outside the scope of machine learning. If you are interested in this topic, an introductory statistics textbook is a good place to get started, for instance, *Intuitive Introductory Statistics*, Wolfe, D. A. and Schneider, G., Springer, 2017.

Regression analysis is not synonymous with a single algorithm. Rather, it is an umbrella term for many methods, which can be adapted to nearly any machine learning task. If you were limited to choosing only a single machine learning method to study, regression would be a good choice. One could devote an entire career to nothing else and perhaps still have much to learn.

In this chapter, we'll start with the most basic **linear regression** models—those that use straight lines. The case when there is only a single independent variable is known as **simple linear regression**. In the case of two or more independent variables, it is known as **multiple linear regression**, or simply **multiple regression**. Both techniques assume a single dependent variable, which is measured on a continuous scale.

Regression can also be used for other types of dependent variables and even for some classification tasks. For instance, **logistic regression** is used to model a binary categorical outcome, while **Poisson regression**—named after the French mathematician Siméon Poisson—models integer count data. The method known as **multinomial logistic regression** models a categorical outcome and can therefore be used for classification.

These specialized regression methods fall into the class of **generalized linear models (GLMs)**, which adapt the straight lines of traditional regression models to allow the modeling of other forms of data. These will be described later in this chapter.

Because similar statistical principles apply across all regression methods, once you understand the linear case, learning about the other variants is straightforward. We'll begin with the basic case of simple linear regression. Despite the name, this method is not too simple to address complex problems. In the next section, we'll see how the use of a simple linear regression model might have averted a tragic engineering disaster.

Simple linear regression

On January 28th, 1986, seven crew members of the United States space shuttle *Challenger* were killed when a rocket booster failed, causing a catastrophic disintegration. In the aftermath, experts quickly focused on the launch temperature as a potential culprit. The rubber O-rings responsible for sealing the rocket joints had never been tested below 40°F (4°C), and the weather on launch day was unusually cold and below freezing.

With the benefit of hindsight, the accident has been a case study for the importance of data analysis and visualization. Although it is unclear what information was available to the rocket engineers and decision-makers leading up to the launch, it is undeniable that better data, utilized carefully, might very well have averted this disaster.



This section's analysis is based on data presented in *Risk Analysis of the Space Shuttle: Pre-Challenger Prediction of Failure*, Dalal, S. R., Fowlkes, E. B., and Hoadley, B., *Journal of the American Statistical Association*, 1989, Vol. 84, pp. 945-957. For one perspective on how data may have changed the result, see *Visual Explanations: Images And Quantities, Evidence And Narrative*, Tufte, E. R., Cheshire, C. T.: Graphics Press, 1997. For a counterpoint, see *Representation and misrepresentation: Tufte and the Morton Thiokol engineers on the Challenger*, Robison, W., Boisjoly, R., Hoeker, D., and Young, S., *Science and Engineering Ethics*, 2002, Vol. 8, pp. 59-81.

The rocket engineers almost certainly knew that cold temperatures could make the components more brittle and less able to seal properly, which would result in a higher chance of a dangerous fuel leak. However, given the political pressure to continue with the launch, they needed data to support this hypothesis. A regression model that demonstrated a link between temperature and O-ring failures, and could forecast the chance of failure given the expected temperature at launch, might have been very helpful.

To build the regression model, the scientists might have used the data on launch temperature and component distress recorded during the 23 previous successful shuttle launches. Component distress indicates one of two types of problems. The first problem, called erosion, occurs when excessive heat burns up the O-ring. The second problem, called blow-by, occurs when hot gasses leak through or “blow by” a poorly sealed O-ring. Since the shuttle had a total of six primary O-rings, up to six distresses could occur per flight. Though the rocket could survive one or more distress events or be destroyed with as few as one, each additional distress increased the probability of a catastrophic failure. The following scatterplot shows a plot of primary O-ring distresses detected for the previous 23 launches, as compared to the temperature at launch:

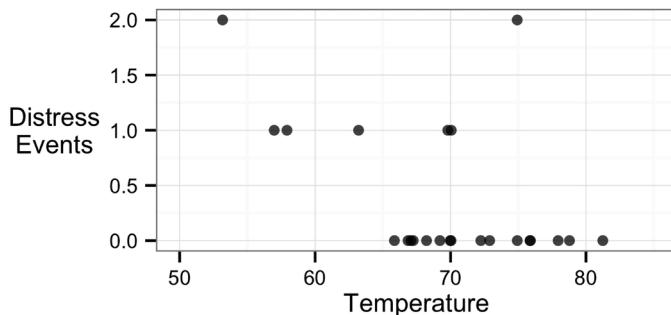


Figure 6.2: A visualization of space shuttle O-ring distresses versus launch temperature

Examining the plot, there is an apparent trend: launches occurring at higher temperatures tend to have fewer O-ring distress events. Additionally, the coldest launch (53°F) had two distress events, a level that had only been reached in one other launch. With this information in mind, the fact that the Challenger was scheduled to launch in conditions more than 20 degrees colder seems concerning. But exactly how concerned should they have been? To answer this question, we can turn to simple linear regression.

A simple linear regression model defines the relationship between a dependent variable and a single independent predictor variable using a line defined by an equation in the following form:

$$y = \alpha + \beta x$$

Aside from the Greek characters, this equation is virtually identical to the slope-intercept form described previously. The intercept, α (alpha), describes where the line crosses the y axis, while the slope, β (beta), describes the change in y given an increase of x . For the shuttle launch data, the slope would tell us the expected change in O-ring failures for each degree the launch temperature increases.



Greek characters are often used in the field of statistics to indicate variables that are parameters of a statistical function. Therefore, performing a regression analysis involves finding **parameter estimates** for α and β . The parameter estimates for alpha and beta are typically denoted using a and b , although you may find that some of this terminology and notation is used interchangeably.

Suppose we know that the estimated regression parameters in the equation for the shuttle launch data are $a = 3.70$ and $b = -0.048$. Consequently, the full linear equation is $y = 3.70 - 0.048x$. Ignoring for a moment how these numbers were obtained, we can plot the line on the scatterplot like this:

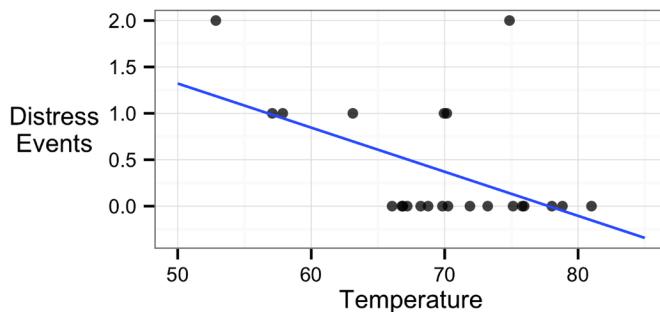


Figure 6.3: A regression line modeling the relationship between distress events and launch temperature

As the line shows, at 60 degrees Fahrenheit, we predict less than one O-ring distress event. At 50 degrees Fahrenheit, we expect around 1.3 failures. If we use the model to extrapolate all the way out to 31 degrees—the forecasted temperature for the Challenger launch—we would expect about $3.70 - 0.048 * 31 = 2.21$ O-ring distress events.

Assuming that each O-ring failure is equally likely to cause a catastrophic fuel leak, this means that the Challenger launch at 31 degrees was nearly three times riskier than the typical launch at 60 degrees, and over eight times riskier than a launch at 70 degrees.

Notice that the line doesn't pass through each data point exactly. Instead, it cuts through the data somewhat evenly, with some predictions lower or higher than the line. In the next section, we will learn why this particular line was chosen.

Ordinary least squares estimation

To determine the optimal estimates of α and β an estimation method known as **ordinary least squares (OLS)** is used. In OLS regression, the slope and intercept are chosen such that they minimize the **sum of the squared errors (SSE)**. The errors, also known as **residuals**, are the vertical distance between the predicted y value and the actual y value. Because the errors can be over-estimates or under-estimates, they can be positive or negative values; squaring them makes the errors positive regardless of direction. The residuals are illustrated for several points in the following diagram:

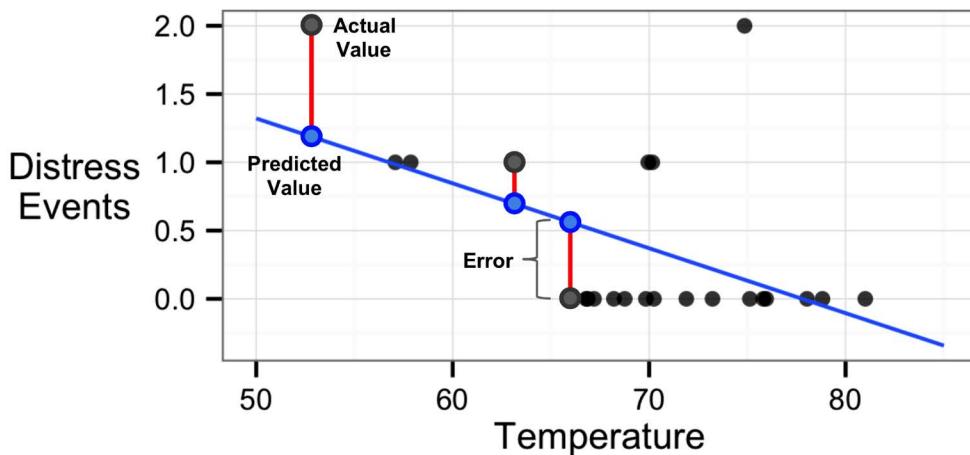


Figure 6.4: The regression line predictions differ from the actual values by a residual amount

In mathematical terms, the goal of OLS regression can be expressed as the task of minimizing the following equation:

$$\sum (y_i - \hat{y}_i)^2 = \sum e_i^2$$

In plain language, this equation defines e (the error) as the difference between the actual y value and the predicted y value. The error values are squared to eliminate the negative values and summed across all points in the data.



The caret character (^) above the y term is a commonly used feature of statistical notation. It indicates that the term is an estimate for the true y value. This is referred to as the y hat.

The solution for a depends on the value of b . It can be obtained using the following formula:

$$a = \bar{y} - b\bar{x}$$



To understand these equations, you'll need to know another bit of statistical notation. The horizontal bar appearing over the x and y terms indicates the mean value of x or y . This is referred to as the x bar or y bar.

Though the proof is beyond the scope of this book, it can be shown using calculus that the value of b that results in the minimum squared error is:

$$b = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$$

If we break this equation apart into its component pieces, we can simplify it somewhat. The denominator for b should look familiar; it is very similar to the variance of x , which is denoted as $\text{Var}(x)$. As we learned in *Chapter 2, Managing and Understanding Data*, the variance involves finding the average squared deviation from the mean of x . This can be expressed as:

$$\text{Var}(x) = \frac{\sum(x_i - \bar{x})^2}{n}$$

The numerator involves taking the sum of each data point's deviation from the mean x value multiplied by that point's deviation away from the mean y value. This is similar to the covariance function for x and y , denoted as $\text{Cov}(x, y)$. The covariance formula is:

$$\text{Cov}(x, y) = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{n}$$

If we divide the covariance function by the variance function, the n terms in the numerator and denominator cancel each other out and we can rewrite the formula for b as:

$$b = \frac{\text{Cov}(x, y)}{\text{Var}(x)}$$

Given this restatement, it is easy to calculate the value of b using built-in R functions. Let's apply them to the shuttle launch data to estimate the regression line.



If you would like to follow along with these examples, download the `challenger.csv` file from the Packt Publishing website and load it to a data frame using the `launch <- read.csv("challenger.csv")` command.

If the shuttle launch data is stored in a data frame named `launch`, the independent variable x is named `temperature`, and the dependent variable y is named `distress_ct`, we can then use the R functions `cov()` and `var()` to estimate b :

```
> b <- cov(launch$temperature, launch$distress_ct) /  
      var(launch$temperature)  
> b  
[1] -0.04753968
```

We can then estimate a by using the computed b value and applying the `mean()` function:

```
> a <- mean(launch$distress_ct) - b * mean(launch$temperature)  
> a  
[1] 3.698413
```

Estimating the regression equation by hand is obviously less than ideal, so R predictably provides a function for fitting regression models automatically. We will use this function shortly. Before that, it is important to expand your understanding of the regression model's fit by first learning a method for measuring the strength of a linear relationship. Additionally, you will soon learn how to apply multiple linear regression to problems with more than one independent variable.

Correlations

The **correlation** between two variables is a number that indicates how closely their relationship follows a straight line. Without additional qualification, correlation typically refers to the **Pearson correlation coefficient**, which was developed by the 20th-century mathematician Karl Pearson. A correlation falls in the range between -1 to +1. The maximum and minimum values indicate a perfectly linear relationship, while a correlation close to zero indicates the absence of a linear relationship.

The following formula defines Pearson's correlation:

$$\rho_{x,y} = \text{Corr}(x, y) = \frac{\text{Cov}(x, y)}{\sigma_x \sigma_y}$$



More Greek notation has been introduced here: the first symbol (which looks like a lowercase p) is *rho*, and it is used to denote the Pearson correlation statistic. The symbols that look like q characters rotated counterclockwise are the lowercase Greek letter *sigma*, and they indicate the standard deviation of x or y .

Using this formula, we can calculate the correlation between the launch temperature and the number of O-ring distress events. Recall that the covariance function is `cov()` and the standard deviation function is `sd()`. We'll store the result in `r`, a letter that is commonly used to indicate the estimated correlation:

```
> r <- cov(launch$temperature, launch$distress_ct) /  
  (sd(launch$temperature) * sd(launch$distress_ct))  
> r
```

```
[1] -0.5111264
```

Alternatively, we can obtain the same result with the `cor()` correlation function:

```
> cor(launch$temperature, launch$distress_ct)
```

```
[1] -0.5111264
```

The correlation between the temperature and the number of distressed O-rings is -0.51 . The negative correlation implies that increases in temperature are related to decreases in the number of distressed O-rings. To the NASA engineers studying the O-ring data, this would have been a very clear indicator that a low-temperature launch could be problematic. The correlation also tells us about the relative strength of the relationship between temperature and O-ring distress. Because -0.51 is halfway to the maximum negative correlation of -1 , this implies that there is a moderately strong negative linear association.

There are various rules of thumb used to interpret correlation strength. One method assigns a status of “weak” to values between 0.1 and 0.3 ; “moderate” to the range of 0.3 to 0.5 ; and “strong” to values above 0.5 (these also apply to similar ranges of negative correlations). However, these thresholds may be too strict or too lax for certain purposes. Often, the correlation must be interpreted in context.

For data involving human beings, a correlation of 0.5 may be considered very high; for data generated by mechanical processes, a correlation of 0.5 may be very weak.



You have probably heard the expression “correlation does not imply causation.” This is rooted in the fact that a correlation only describes the association between a pair of variables, yet there could be other explanations that are unaccounted for and responsible for the observed relationship. For example, there may be a strong correlation between life expectancy and time per day spent watching movies, but before doctors recommend that we all watch more movies, we need to rule out another explanation: younger people watch more movies and younger people are (in general) less likely to die.

Measuring the correlation between two variables gives us a way to quickly check for linear relationships between independent variables and the dependent variable. This will be increasingly important as we start defining regression models with a larger number of predictors.

Multiple linear regression

Most real-world analyses have more than one independent variable. Therefore, it is likely that you will be using multiple linear regression for most numeric prediction tasks. The strengths and weaknesses of multiple linear regression are shown in the following table:

Strengths	Weaknesses
<ul style="list-style-type: none">• By far the most common approach to modeling numeric data• Can be adapted to model almost any modeling task• Provides estimates of both the size and strength of the relationships among features and the outcome	<ul style="list-style-type: none">• Makes strong assumptions about the data• The model’s form must be specified by the user in advance• Does not handle missing data• Only works with numeric features, so categorical data requires additional preparation• Requires some knowledge of statistics to understand the model

We can understand multiple regression as an extension of simple linear regression. The goal in both cases is similar—to find values of slope coefficients that minimize the prediction error of a linear equation. The key difference is that there are additional terms for the additional independent variables.

Multiple regression models take the form of the following equation. The dependent variable y is specified as the sum of an intercept term α plus the product of the estimated β value and the x variable for each of i features. An error term ε (denoted by the Greek letter *epsilon*) has been added here as a reminder that the predictions are not perfect. This represents the residual term noted previously:

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_i x_i + \varepsilon$$

Let's consider for a moment the interpretation of the estimated regression parameters. You will note that in the preceding equation, a coefficient is provided for each feature. This allows each feature to have a separate estimated effect on the value of y . In other words, y changes by the amount β_i for each unit increase in feature x_i . The intercept α is then the expected value of y when the independent variables are all zero.

Since the intercept term α is really no different than any other regression parameter, it is also sometimes denoted as β_0 (pronounced *beta naught*) as shown in the following equation:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_i x_i + \varepsilon$$

Just like before, the intercept is unrelated to any of the independent x variables. However, for reasons that will become clear shortly, it helps to imagine β_0 as if it were being multiplied by a term x_0 . We assign x_0 to be a constant with the value of 1:

$$y = \beta_0 x_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_i x_i + \varepsilon$$

To estimate the regression parameters, each observed value of the dependent variable y must be related to observed values of the independent x variables using the regression equation in the previous form. The following figure is a graphical representation of the setup of a multiple regression task:

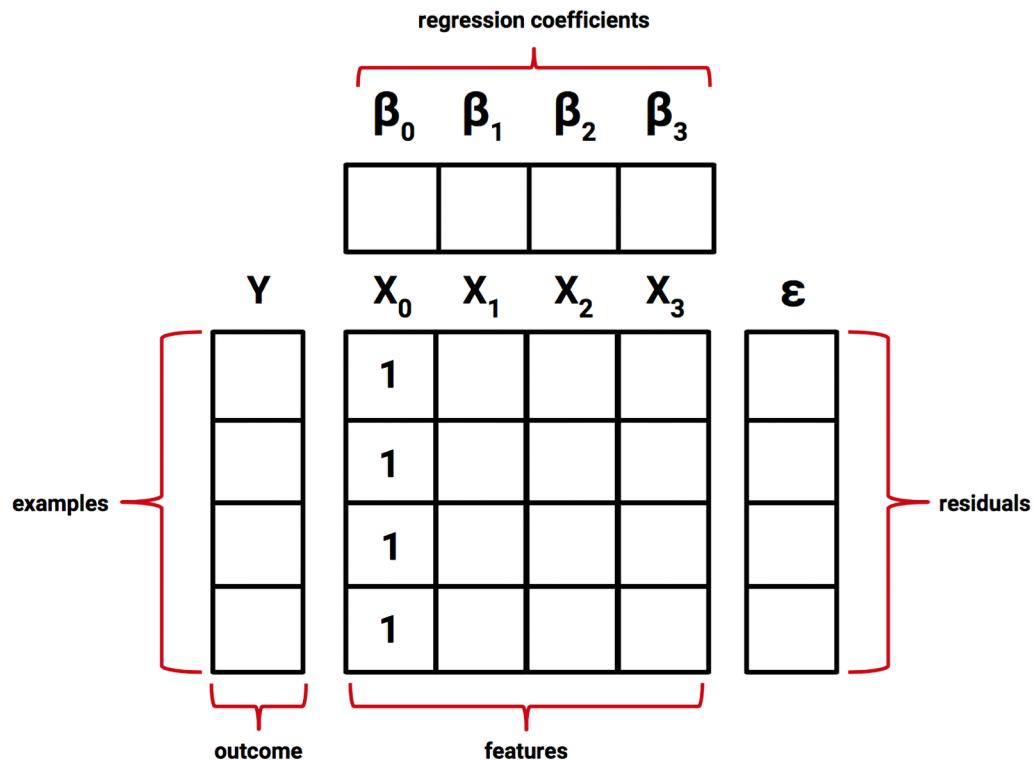


Figure 6.5: Multiple regression seeks to find the β values that relate the X values to Y while minimizing ϵ

The many rows and columns of data illustrated in the preceding figure can be described in a condensed formulation using **matrix notation** in bold font to indicate that each of the terms represents multiple values. Simplified in this way, the formula is as follows:

$$\mathbf{Y} = \boldsymbol{\beta}\mathbf{X} + \boldsymbol{\epsilon}$$

In matrix notation, the dependent variable is a vector, \mathbf{Y} , with a row for every example. The independent variables have been combined into a matrix, \mathbf{X} , with a column for each feature plus an additional column of 1 values for the intercept. Each column has a row for every example. The regression coefficients $\boldsymbol{\beta}$ and residual errors $\boldsymbol{\varepsilon}$ are also now vectors.

The goal now is to solve for $\boldsymbol{\beta}$, the vector of regression coefficients that minimizes the sum of the squared errors between the predicted and actual \mathbf{Y} values. Finding the optimal solution requires the use of matrix algebra; therefore, the derivation deserves more careful attention than can be provided in this text. However, if you’re willing to trust the work of others, the best estimate of the vector $\boldsymbol{\beta}$ can be computed as:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y}$$

This solution uses a pair of matrix operations: the T indicates the transpose of matrix \mathbf{X} , while the negative exponent indicates the **matrix inverse**. Using R’s built-in matrix operations, we can thus implement a simple multiple regression learner. Let’s apply this formula to the Challenger launch data.



If you are unfamiliar with the preceding matrix operations, the Wolfram MathWorld pages for transpose (<http://mathworld.wolfram.com/Transpose.html>) and matrix inverse (<http://mathworld.wolfram.com/MatrixInverse.html>) provide a thorough introduction and are understandable even without an advanced mathematics background.

Using the following code, we can create a basic regression function named `reg()`, which takes a parameter `y` and a parameter `x`, and returns a vector of estimated beta coefficients:

```
> reg <- function(y, x) {
  x <- as.matrix(x)
  x <- cbind(Intercept = 1, x)
  b <- solve(t(x) %*% x) %*% t(x) %*% y
  colnames(b) <- "estimate"
  print(b)
}
```

The `reg()` function created here uses several R commands that we have not used previously. First, since we will be using the function with sets of columns from a data frame, the `as.matrix()` function converts the data frame into matrix form.

Next, the `cbind()` function binds an additional column onto the `x` matrix; the command `Intercept = 1` instructs R to name the new column `Intercept` and to fill the column with repeating 1 values. Then, a series of matrix operations are performed on the `x` and `y` objects:

- `solve()` takes the inverse of a matrix
- `t()` is used to transpose a matrix
- `%%*` multiplies two matrices

By combining these R functions as shown, our function will return a vector `b`, which contains estimated parameters for the linear model relating `x` to `y`. The final two lines in the function give the `b` vector a name and print the result on screen.

Let's apply this function to the shuttle launch data. As shown in the following code, the dataset includes three features and the distress count (`distress_ct`), which is the outcome of interest:

```
> str(launch)
```

```
'data.frame': 23 obs. of 4 variables:  
 $ distress_ct      : int  0 1 0 0 0 0 0 0 1 1 ...  
 $ temperature      : int  66 70 69 68 67 72 73 70 57 63 ...  
 $ field_check_pressure: int  50 50 50 50 50 50 100 100 200 200 ...  
 $ flight_num        : int  1 2 3 4 5 6 7 8 9 10 ...
```

We can confirm that our function is working correctly by comparing its result for the simple linear regression model of O-ring failures versus temperature, which we found earlier to have parameters $a = 3.70$ and $b = -0.048$. Since temperature is in the second column of the launch data, we can run the `reg()` function as follows:

```
> reg(y = launch$distress_ct, x = launch[2])
```

	estimate
Intercept	3.69841270
temperature	-0.04753968

These values exactly match our prior result, so let's use the function to build a multiple regression model. We'll apply it just as before, but this time we will specify columns two through four for the `x` parameter to add two additional predictors:

```
> reg(y = launch$distress_ct, x = launch[2:4])
```

	estimate
Intercept	3.527093383

temperature	-0.051385940
field_check_pressure	0.001757009
flight_num	0.014292843

This model predicts the number of O-ring distress events using temperature, field check pressure, and the launch ID number. Notably, the inclusion of the two new predictors did not change our finding from the simple linear regression model. Just as before, the coefficient for the temperature variable is negative, which suggests that as temperature increases, the number of expected O-ring events decreases. The magnitude of the effect is also approximately the same: roughly 0.05 fewer distress events are expected for each degree increase in launch temperature.

The two new predictors also contribute to the predicted distress events. The field check pressure refers to the amount of pressure applied to the O-ring during pre-launch testing. Although the check pressure was originally 50 psi, it was raised to 100 and 200 psi for some launches, which led some to believe that it may be responsible for O-ring erosion. The coefficient is positive, but small, providing at least a little evidence for this hypothesis. The flight number accounts for the shuttle's age. With each flight, it gets older, and parts may be more brittle or prone to fail. The small positive association between flight number and distress count may reflect this fact.

Overall, our retrospective analysis of the space shuttle data suggests that there was reason to believe that the Challenger launch was highly risky given the weather conditions. Perhaps if the engineers had applied linear regression beforehand, a disaster could have been averted. Of course, the reality of the situation, and the political implications involved, were surely not as simple then as they now appear in hindsight.

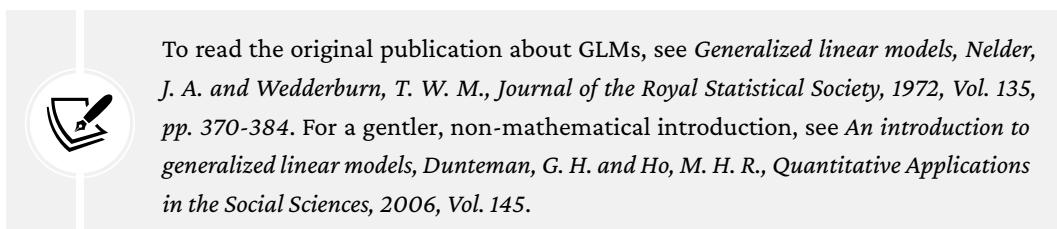
Generalized linear models and logistic regression

As demonstrated in the analysis of the Challenger space shuttle launch data, standard linear regression is a useful method for modeling the relationship between a numeric outcome and one or more predictors. It is no wonder that regression has stood the test of time. Even after over a hundred years, it remains one of the most important techniques in our toolkit, even though it is no more sophisticated than finding the best straight line to fit the data.

However, not every problem is well suited to being modeled by a line, and moreover, the statistical assumptions made by regression models are violated in many real-world tasks. Even the Challenger data is less than ideal for linear regression as it violates the regression assumption that the target variable is measured on a continuous scale. As the number of O-ring failures can only take countable values, it makes no sense that the model might predict exactly 2.21 distress events rather than either two or three.

For modeling counting values, for categorical or binary outcomes, as well as other cases where the target is not a normally distributed continuous variable, standard linear regression is not the best tool for the job—even though many still apply it to these types of problems, and it often performs surprisingly well.

To address these shortcomings, linear regression can be adapted to other use cases using the aptly named GLM, which was first described in 1972 by statisticians John Nelder and Robert Wedderburn. The GLM loosens two assumptions of traditional regression modeling. First, it allows the target to be a non-normally distributed, non-continuous variable. Second, it allows the variance of the target variable to be related to its mean. The former property opens the door for modeling categorical data or counting data, or even cases where there is a limited range of values to predict, such as probability values falling on a range between 0 to 1. The latter property allows the model to better fit cases where the predictors relate to the predictions in a nonlinear way, such as exponential growth in which an increase of one unit of time leads to greater and greater increases in the outcome.



To read the original publication about GLMs, see *Generalized linear models*, Nelder, J. A. and Wedderburn, T. W. M., *Journal of the Royal Statistical Society*, 1972, Vol. 135, pp. 370-384. For a gentler, non-mathematical introduction, see *An introduction to generalized linear models*, Duntzman, G. H. and Ho, M. H. R., *Quantitative Applications in the Social Sciences*, 2006, Vol. 145.

These two generalizations of linear regression are reflected in the two key components of any GLM:

1. The **family** refers to the distribution of the target feature, which must be chosen from members of the **exponential family** of distributions, which includes the normal Gaussian distribution as well as others like Poisson, Binomial, and Gamma. The chosen distribution may be discrete or continuous, and it may span different ranges of values, such as only positive values or only values between zero and one.
2. The **link function** transforms the relationship between the predictors and the target such that it can be modeled using a linear equation, despite the original relationship being nonlinear. There is always a **canonical link function**, which is determined by the chosen family and used by default, but in some cases, one may choose a different link to vary how the model is interpreted or to obtain a better model fit.

Varying the family and link functions gives the GLM approach tremendous flexibility to adapt to many different real-world use cases and to conform to the natural distribution of the target variable. Knowing which combination to use depends on both how the model will be applied as well as the theoretical distribution of the target. Understanding these factors in detail requires knowledge of the various distributions in the exponential family and a background in statistical theory. Thankfully, most GLM use cases conform to a few common combinations of family and link, which are listed in the table that follows:

Family	Canonical Link Function	Target Range	Notes and Applications
Gaussian (normal)	Identity	- ∞ to ∞	Used for linear response modeling; reduces the GLM to standard linear regression.
Poisson	Log	Integers 0 to ∞	Known as Poisson regression; models the count of an event occurring (such as the total number of O-ring failures) by estimating the frequency at which the event happens.
Binomial	Logit	0 to 1	Known as logistic regression; used for modeling a binary outcome (such as whether any O-ring failed) by estimating the probability that the outcome occurs.
Gamma	Negative Inverse	0 to ∞	One of many possibilities for modeling right-skewed data; could be used for modeling the time to an event (such as seconds to an O-ring failure) or cost data (such as insurance claims costs for a car accident).

Multinomial	Logit	1 of K categories	Known as multinomial logistic regression; used for modeling a categorical outcome (such as a successful, unsuccessful, or aborted shuttle launch) by estimating the probability the example falls into each of the categories. Generally uses specialized packages rather than a GLM function to aid interpretation.
-------------	-------	---------------------	--

Due to the nuances of interpreting GLMs, it takes much practice and careful study to be adept at applying just one, and few people can claim to be experts at using all of them. Entire textbooks are devoted to each GLM variant. Fortunately, in the domain of machine learning, interpreting and understanding are less important than being able to apply the correct GLM form to practical problems and produce useful predictions. While this chapter cannot cover each of the listed methods, an introduction to the key details will allow you to later pursue the GLM variants most relevant to your own work.

Beginning with the simplest variant listed in the table, standard linear regression can be thought of as a special type of GLM that uses the Gaussian family and the identity link function. The **identity link** implies that the relationship between the target y and a predictor x_i is not transformed in any way. Thus, as with standard regression, an estimated regression parameter β_i can be interpreted quite simply as the increase in y given a one-unit increase in x_i , assuming all other factors are held equal.

GLM forms that use other link functions are not as simple to interpret and fully understanding the impact of individual predictors requires much more careful analysis. This is because the regression parameters must be interpreted as the additive increase in y for a one-unit increase in x_i , but only after being transformed through the link function.

For instance, the Poisson family that uses the **log link** function to model the expected count of events relates y to the predictors x_i through the natural logarithm; consequently, the additive effect of $\log(\beta_1 x_1) + \log(\beta_2 x_2)$ on y becomes multiplicative on the original scale of the response variable. This is because using the properties of logarithms, we know $\log(\beta_1 x_1) + \log(\beta_2 x_2) = \log(\beta_1 x_1 * \beta_2 x_2)$, and this becomes $\beta_1 x_1 * \beta_2 x_2$ after exponentiating to remove the logarithm.

Due to this multiplicative impact, the parameter estimates are understood as relative rates of increase rather than the absolute increase in y as with linear regression.

To see this in practice, suppose we built a Poisson regression model of the count of O-ring failures versus launch temperature. If x_i is temperature and the estimated $\beta_1 = -0.103$, then we can determine that there are about 9.8 percent fewer O-ring failures on average per each additional degree of temperature at launch. This is because $\exp(-0.103) = 0.902$, or 90.2 percent of the failures per degree, which implies that we would expect 9.8 percent fewer failures per degree increase. Applying this to the Challenger shuttle launch temperature at 36 degrees Fahrenheit, we can extrapolate that a launch 17 degrees warmer (53 degrees Fahrenheit was the previous coldest launch) would have had about $(0.902)^{17} = 17.2$ percent of the expected number of failures, equivalent to a drop of 82.8 percent.

The GLM variant that uses a binomial family distribution with a logit link function is known as **logistic regression**, and is perhaps the single most important form as it allows regression to be adapted to binary classification tasks. The **logit link** is a function in the form $\log(p / (1 - p))$, where p is a probability; the inner portion $(p / (1 - p))$ expresses the probability as **odds**, exactly like the odds used in gambling and sports betting in phrases like “the team has a 2:1 chance of winning.” After taking the natural logarithm, the estimated regression coefficients are interpreted as log odds. Because we understand odds more intuitively than log odds, we usually exponentiate the estimated logistic regression coefficients to convert log odds into odds for interpretation. However, because logistic regression coefficients indicate the difference in the odds of y due to a one-unit increase in x , the exponentiated odds become **odds ratios**, which express the relative increase or decrease in the chances that y happens.

In the context of the space shuttle data, suppose we built a logistic regression model for the binary classification task of predicting whether or not a launch would have one or more O-ring failures. A factor that does not change the probability of an O-ring failure would keep the odds balanced at 1:1 (50-50 probability), which translates to log odds of $\log(0.5 / (1 - 0.5)) = 0$ and the estimated regression coefficient $\beta = 0$ for this feature. Finding the odds ratio as $\exp(0) = 1$ shows that the odds remain unchanged regardless of the value of this factor. Now, suppose a factor like temperature reduces the chance that the outcome occurs, and that in the logistic regression model with x_i as temperature, then the estimated $\beta_1 = -0.232$. By exponentiating this, we find the odds ratio $\exp(-0.232) = 0.793$, which means that the odds of a failure drop by about 20 percent for a one-degree increase in temperature, assuming all else is held equal. It is very important to note that this does not imply that the *probability* of a failure drops by 20 percent for each degree increase.

Because the relationship between odds and probability is nonlinear, the impact of a temperature change on the failure probability depends on the context in which the temperature change is occurring!

Odds and probabilities are related via the inverse connection between the logit and logistic functions. The logistic function has the convenient property that for any input x value, the output is a value in the range between 0 to 1—exactly the same range as a probability. Additionally, the logistic function creates an S-shaped curve when graphed, as illustrated in *Figure 6.6*, which shows a hypothetical logistic regression model of O-ring failure probability versus launch temperature. The probability of failure on the y axis is changed most strongly in the middle of the temperature range; at temperature extremes, the predicted failure probability changes very little for each additional degree of temperature added or removed.

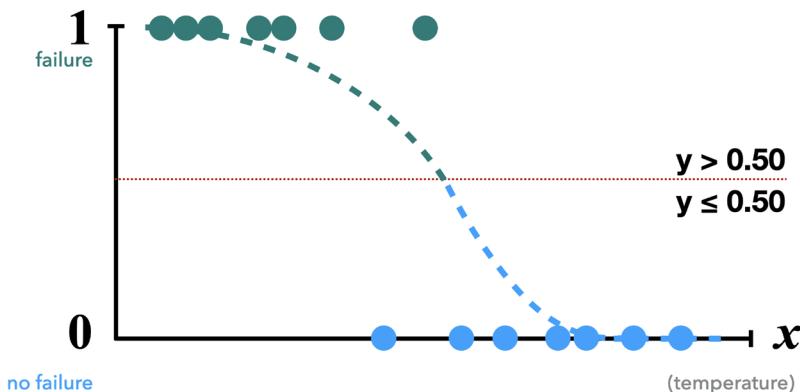


Figure 6.6: Hypothetical logistic regression curve representing space shuttle launch data

The fitted logistic regression model creates a curve representing a probability estimation on a continuous scale anywhere in the range between 0 to 1, despite the target outcome (represented by circles in the figure) only taking the value $y = 0$ or $y = 1$. To obtain the binary prediction, simply define the probability threshold within which the target outcome is to be predicted. For example, if the predicted probability of an O-ring failure is greater than 0.50, then predict “failure,” and otherwise, predict “no failure.” Using a 50 percent threshold is common, but a higher or lower threshold can be used to adjust the cost sensitivity of the model.

Examining the logistic curve in *Figure 6.6* leads to another question: how does the modeling algorithm determine the curve that best fits the data? After all, given that this is not a straight line, the OLS algorithm used in standard linear regression no longer seems to apply.

Indeed, generalized linear models use a different technique called **maximum likelihood estimation (MLE)**, which finds the parameter values for the specified distribution that are most likely to have generated the observed data.

Because OLS estimation is a special case of maximum likelihood estimation, using OLS or MLE for a linear model makes no difference given that the assumptions of OLS modeling are met. For applications outside of linear modeling, the MLE technique will produce different results and must be used instead of OLS. The MLE technique is built into GLM modeling software, and usually applies analytical techniques to identify the optimal model parameters by iterating repeatedly over the data rather than finding the correct solution directly. Fortunately, as we will see shortly, building a GLM in R is hardly more challenging than training a simpler linear model.

This introduction only scratched the surface of what is possible with linear regression and GLM. Although theory and simple examples like the Challenger dataset are helpful for understanding how regression models work, there is more involved in building a useful model than what we've seen so far. R's built-in regression functions include the additional functionality needed to fit more sophisticated models while providing additional diagnostic output to aid model interpretation and assess the fit. Let's apply these functions and expand our knowledge of regression by attempting a real-world learning task.

Example – predicting auto insurance claims costs using linear regression

For an automobile insurance company to make money, it needs to collect more in membership premiums than it spends on claims paid to its beneficiaries in case of vehicle theft, damages, or loss of life in accidents. Consequently, insurers invest time and money to develop models that accurately forecast claims costs for the insured population. This is the field known as **actuarial science**, which uses sophisticated statistical techniques to estimate risk across insured populations.

Insurance expenses are difficult to predict accurately for individuals because accidents, and especially fatal accidents, are thankfully relatively rare—a bit over one fatality per 100 million vehicle miles traveled in the United States—yet, when they do happen, they are extremely costly. Moreover, the specific conditions leading to any given accident are based on factors that are so hard to measure that they appear to be random. An excellent driver with a clean driving record could have bad luck and be hit by a drunk driver, while another person can drive distracted by their cellular phone and, due to good fortune, never cause an accident.

Because of the near impossibility of predicting expenses for a single person, insurance companies apply the law of averages and compute the average cost to insure segments of people with similar risk profiles. If the expense estimates for each risk segment are correct, the insurance company can price the insurance premiums lower for segments with less risk, and potentially attract new, low-risk customers from competing insurance companies. We will simulate this scenario in the analysis that follows.

Step 1 – collecting data

The dataset for this example is a simulation created for this book based on demographics and traffic statistics from the United States government. It is intended to approximate the real-world conditions of automobile insurance companies in the U.S. state of Michigan, which is home to about ten million residents and seven million licensed drivers.



If you would like to follow along interactively, download the `autoinsurance.csv` file from the Packt Publishing GitHub repository for this book and save it to your R working folder.

The insurance dataset includes 20,000 examples of beneficiaries enrolled in the hypothetical automobile vehicle insurance plan. This is much smaller than the typical datasets used by practicing actuaries, especially for very rare outcomes, but the size has been reduced to allow analysis even on computers with limited memory. Each example represents an insured individual's characteristics and total insurance claims costs (expenses) charged to the plan for the calendar year. The features available at the time of enrollment are:

- `age`: The age of the driver, from 16 to 89
- `geo_area`: The geographic area of the vehicle owner's primary residence, and where the vehicle will be used most often; zip codes were bucketed into urban, suburban, and rural categories
- `est_value`: The estimated market value of the vehicle(s), based on age and depreciation; capped at \$125,000—the maximum allowed insured value
- `vehicle_type`: The type of passenger vehicle, either a car, truck, minivan, or sport utility vehicle (SUV)
- `miles_driven`: The distance driven (in miles) in the calendar year
- `college_grad_ind`: A binary indicator set to 1 if the beneficiary has a college education or higher

- `speeding_ticket_ind`: A binary indicator set to 1 if a speeding ticket or infraction was received during the past five years
- `clean_driving_ind`: A binary indicator set to 1 if no at-fault insurance claims were paid during the past five years

In this example scenario, each of the 20,000 beneficiaries enrolled in a “safe driving discount” program, which required the use of a device or mobile phone application that uses location tracking to monitor safe driving conditions throughout the year. This helped validate the accuracy of `miles_driven` and created the following two additional predictors, which are intended to reflect more risky driving behaviors:

- `hard_braking_ind`: A binary indicator set to 1 if the vehicle frequently applies “hard brakes” (as in the case of stopping suddenly)
- `late_driving_ind`: A binary indicator set to 1 if the vehicle is driven regularly after midnight

It is important to give some thought to how these variables may relate to billed insurance expenses—some in more obvious ways than others. For instance, we would clearly expect cars driven more often to be at a higher risk of an accident than those that stay home in the garage. On the other hand, it isn’t as clear whether urban, rural, or suburban drivers would be riskier; rural drivers may drive farther, but urban driving involves more traffic and may be more at risk of vehicle theft. A regression model will help us disentangle these relationships, but requires us to specify the connections between the features ourselves rather than detecting them automatically, which is unlike many other machine learning methods. We’ll explore some of the potential relationships in the next section.



It may also be interesting to consider which potentially useful predictors are not included in the training dataset. Gender is often used for automobile insurance pricing (and it varies whether males or females are more costly!) but the state of Michigan banned the use of gender and credit scoring for this purpose in 2020. Such features may be highly predictive, but may lead to systematic biases against protected groups, as discussed in *Chapter 1, Introducing Machine Learning*.

Step 2 – exploring and preparing the data

As we have done before, we will use the `read.csv()` function to load the data for analysis. We can safely use `stringsAsFactors = TRUE` because it is appropriate to convert the three nominal variables into factors:

```
> insurance <- read.csv("insurance.csv", stringsAsFactors = TRUE)
```

The `str()` function confirms that the data is formatted as we expected:

```
> str(insurance)
```

```
'data.frame': 20000 obs. of 11 variables:  
 $ age : int 19 30 39 64 33 27 62 39 67 38 ...  
 $ geo_area : Factor w/ 3 levels "rural", "suburban", ...  
 $ vehicle_type : Factor w/ 4 levels "car", "minivan", ...  
 $ est_value : int 28811 52603 113870 35228 ...  
 $ miles_driven : int 11700 12811 9784 17400 ...  
 $ college_grad_ind : int 0 1 1 0 0 1 1 0 1 1 ...  
 $ speeding_ticket_ind : int 1 0 0 0 0 0 0 0 0 0 ...  
 $ hard_braking_ind : int 1 0 0 0 0 0 0 0 0 0 ...  
 $ late_driving_ind : int 0 0 0 0 0 0 0 0 0 0 ...  
 $ clean_driving_ind : int 0 1 0 1 1 0 1 1 0 1 ...  
 $ expenses : num 0 6311 49684 0 0 ...
```

Our model's dependent variable is `expenses`, which measures the loss or damages each person claimed under the insurance plan for the year. Prior to building a linear regression model, it is often helpful to check for normality. Although linear regression will not fail without a normally distributed dependent variable, the model often fits better when this is true. Let's look at the summary statistics:

```
> summary(insurance$expenses)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0	0	0	1709	0	232797

The minimum, first quartile, median, and third quartile are all zero, which implies that at least 75% of the beneficiaries had no expenses in the calendar year. The fact that the mean value is greater than the median gives us the sense that the distribution of insurance expenses is right-skewed, but the skew is likely to be quite extreme as the average expense was \$1,709 while the maximum was \$232,797. We can confirm this visually using a histogram:

```
> hist(insurance$expenses)
```

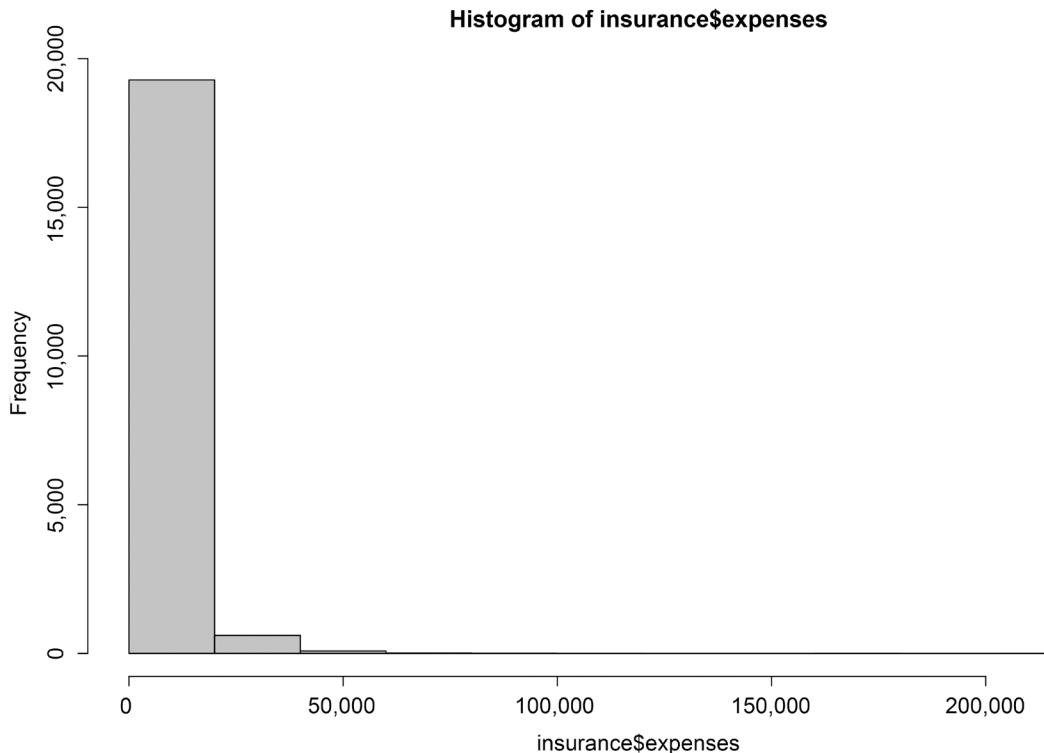


Figure 6.7: The distribution of annual insurance claims costs

As expected, the figure shows a right-skewed distribution with a huge spike at zero, reflecting the fact that only a small portion (about 8.6%) made an insurance claim. Among those that did claim a vehicle loss or damages, the tail end of the distribution extends far to the right, beyond \$200,000 worth of expenses for the costliest injuries. Although this distribution is not ideal for linear regression, knowing this weakness ahead of time may help us design a better-fitting model later. For now, using only the distribution of expenses, we can say that the average beneficiary should be charged an annual premium of \$1,709 for the insurer to break even, or about \$150 a month per subscriber for a slight profit. Of course, this assumes that the risk and costs are shared evenly. An improved insurance model will push greater costs onto riskier drivers and provide safe drivers with financial savings.

Before we add additional predictors, it is important to note that regression models require that every feature is numeric, yet we have two factor-type features in our data frame. For instance, the `geo_area` variable is divided into `urban`, `suburban`, and `rural` levels, while `vehicle_type` has categories for `car`, `truck`, `suv`, and `minivan`.

Let's take a closer look to see how they are distributed:

```
> table(insurance$geo_area)
```

	rural	suburban	urban
	3622	8727	7651


```
> table(insurance$vehicle_type)
```

	car	minivan	suv	truck
	5801	726	9838	3635

Here, we see that the data has been divided nearly evenly between urban and suburban areas, but rural is a much smaller portion of the data. Additionally, SUVs are the most popular vehicle type, followed by cars and trucks, with minivans in a distant fourth place. We will see how R's linear regression function handles these factor variables shortly.

Exploring relationships between features – the correlation matrix

Before fitting a regression model to data, it can be useful to determine how the independent variables are related to the dependent variable and each other. A **correlation matrix** provides a quick overview of these relationships. Given a set of variables, it provides a correlation for each pairwise relationship.

To create a correlation matrix for the four numeric, non-binary variables in the insurance data frame, use the `cor()` command:

```
> cor(insurance[c("age", "est_value", "miles_driven", "expenses")])
```

	age	est_value	miles_driven	expenses
age	1.000000000	-0.05990552	0.04812638	-0.009121269
est_value	-0.059905524	1.000000000	-0.01804807	0.088100468
miles_driven	0.048126376	-0.01804807	1.000000000	0.062146507
expenses	-0.009121269	0.08810047	0.06214651	1.000000000

At the intersection of each row and column pair, the correlation is listed for the variables indicated by that row and column. The diagonal is always **1.0000000** since there is always a perfect correlation between a variable and itself. The values above and below the diagonal are identical since correlations are symmetrical. In other words, `cor(x, y)` is equal to `cor(y, x)`.

None of the correlations in the matrix are very strong, but the associations do match with common sense. For instance, age and expenses appear to have a weak negative correlation, meaning that as someone ages, their expected insurance cost goes down slightly—probably reflecting the greater driving experience. There are also positive correlations between est_value and expenses and miles_driven and expenses, which indicate that more valuable cars and more extensive driving lead to greater expenses. We'll try to tease out these types of relationships more clearly when we build our final regression model.

Visualizing relationships between features – the scatterplot matrix

It can also be helpful to visualize the relationships between numeric features with scatterplots. Although we could create a scatterplot for each possible relationship, doing so for a large set of features quickly becomes tedious.

An alternative is to create a **scatterplot matrix** (sometimes abbreviated as **SPLOM**), which is simply a collection of scatterplots arranged in a grid. It is used to detect patterns among three or more variables. The scatterplot matrix is not a true multi-dimensional visualization because only two features are examined at a time. Still, it provides a general sense of how the data may be interrelated.

We can use R's graphical capabilities to create a scatterplot matrix for the four non-binary numeric features: age, est_value, miles_driven, and expenses. The pairs() function is provided in the default R installation and provides basic functionality for producing scatterplot matrices. To invoke the function, simply provide it with a subset of the data frame to plot. Given the relatively large size of our insurance dataset, we'll set the plot character parameter pch = "." to a dot to make the visualization easier to read, then limit the columns to the four variables of interest:

```
> pairs(insurance[c("age", "est_value", "miles_driven",
                    "expenses")], pch = ".")
```

This produces the following scatterplot matrix:

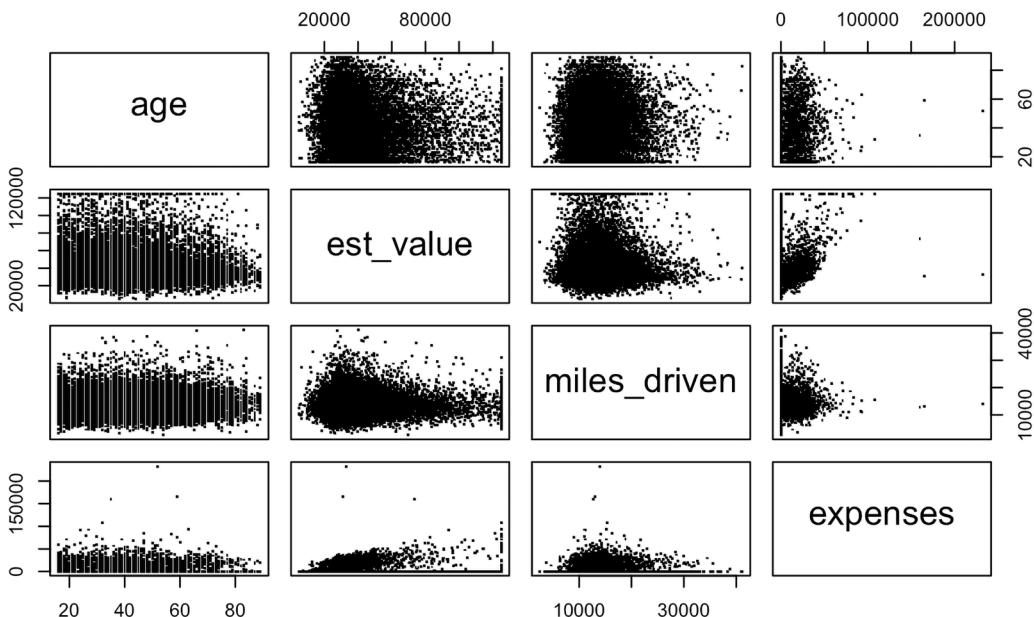


Figure 6.8: A scatterplot matrix of the numeric features in the insurance dataset

In the scatterplot matrix, the intersection of each row and column holds the scatterplot of the variables indicated by the row and column pair. The diagrams above and below the diagonal are transpositions because the *x* axis and *y* axis have been swapped. Do you notice any patterns in these plots? Although they mostly look like random clouds of points, a couple seem to display some subtle trends. The relationships of both *est_value* and *miles_driven* with *expenses* seem to display a slight upward trend, which confirms visually what we already learned from the correlation matrix.

By adding more information to the plot, it can be made even more useful. An enhanced scatterplot matrix can be created with the *pairs.panels()* function in the *psych* package. If you do not have this package installed, type *install.packages("psych")* to install it on your system and load it using the *library(psych)* command. Then, we can create a scatterplot matrix using the *pch* parameter to set the plotting character as we did previously:

```
> library(psych)
> pairs.panels(insurance[c("age", "est_value", "miles_driven",
+ "expenses")], pch = ".")
```

This produces a more informative scatterplot matrix, as follows:

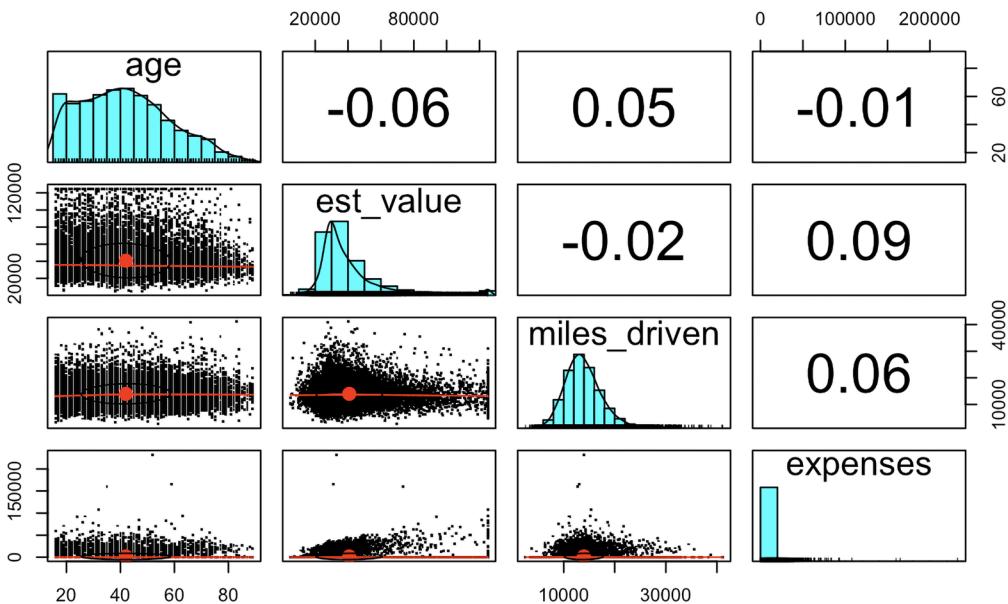


Figure 6.9: The `pairs.panels()` function adds detail to the scatterplot matrix

In the `pairs.panels()` output, the scatterplots above the diagonal are replaced with a correlation matrix. The diagonal now contains histograms depicting the distribution of values for each feature. Finally, the scatterplots below the diagonal are presented with additional visual information.

The oval-shaped object on each scatterplot (which may be difficult to see in print due to the mass of black points but can be seen more easily on a computer screen) is a **correlation ellipse**. It provides a simple visual indicator of correlation strength. In this dataset, there are no strong correlations, so the ovals are mostly flat; with stronger correlations, the ovals would be tilted upward or downward to indicate a positive or negative correlation. The dot at the center of the ellipse is a point reflecting the means of the x - and y -axis variables.

The line superimposed across the scatterplot (which appears in red on a computer screen) is called a **loess curve**. It indicates the general relationship between the x -axis and y -axis variables. It is best understood by example. Although the small size of the plot makes this trend difficult to see, the curve for `age` and `miles_driven` slopes upward very slightly until reaching middle age, then levels off. This means that driving tends to increase with age up to the point at which it remains roughly constant over time.

Although not observed here, the loess curve can sometimes be quite dramatic with V- or U-shaped curves as well as staircase patterns. Recognizing such patterns can assist later with developing a better-fitting regression model.

Step 3 – training a model on the data

To fit a linear regression model to data with R, the `lm()` function can be used. This is part of the `stats` package, which should be included and loaded by default with your R installation. The `lm()` syntax is as follows:

Multiple regression modeling syntax	
using the <code>lm()</code> function in the <code>stats</code> package	
Building the model:	
<pre>m <- lm(dv ~ iv, data = mydata)</pre> <ul style="list-style-type: none"> • <code>dv</code> is the dependent variable in the <code>mydata</code> data frame to be modeled • <code>iv</code> is an R formula specifying the independent variables in the <code>mydata</code> data frame to use in the model • <code>data</code> specifies the data frame in which the <code>dv</code> and <code>iv</code> variables can be found 	
The function will return a regression model object, which can be used to make predictions. Interactions between independent variables can be specified using the <code>*</code> operator.	
Making predictions:	
<pre>p <- predict(m, test)</pre> <ul style="list-style-type: none"> • <code>m</code> is a model trained by the <code>lm()</code> function • <code>test</code> is a data frame containing test data with the same features as the training data used to build the model 	
The function will return a vector of predicted values.	
Example:	
<pre>ins_model <- lm(charges ~ age + est_value + miles_driven, data = insurance) ins_pred <- predict(ins_model, insurance_test)</pre>	

Figure 6.10: Multiple regression syntax

The following command fits a linear regression model, which relates the ten independent variables to the total insurance expenses. The R formula syntax uses the tilde character (`~`) to describe the model; the dependent variable `expenses` is written to the left of the tilde while the independent variables go to the right, separated by `+` signs.

There is no need to specify the regression model's intercept term, as it is included by default:

```
> ins_model <- lm(expenses ~ age + geo_area + vehicle_type +
+ est_value + miles_driven +
+ college_grad_ind + speeding_ticket_ind +
+ hard_braking_ind + late_driving_ind +
+ clean_driving_ind,
+ data = insurance)
```

Because the period character (.) can be used to specify all features (excluding those already specified in the formula), the following command is equivalent to the prior command:

```
> ins_model <- lm(expenses ~ ., data = insurance)
```

After building the model, simply type the name of the model object to see the estimated beta coefficients. Note that the options(scipen = 999) command turns off scientific notation to make the output easier to read:

```
> options(scipen = 999)
> ins_model
```

```
Call:
lm(formula = expenses ~ ., data = insurance)

Coefficients:
(Intercept)           age      geo_arearurban
-1154.91486       -1.88603        191.07895
geo_areaurban  vehicle_typeminivan   vehicle_typesuv
    169.11426       115.27862       -19.69500
vehicle_typetruck      est_value      miles_driven
     21.56836        0.03115        0.11899
college_grad_ind  speeding_ticket_ind  hard_braking_ind
    -25.04030       155.82410        11.84522
late_driving_ind    clean_driving_ind
    362.48550       -239.04740
```

Understanding the regression coefficients for a linear regression model is fairly straightforward. The intercept is the predicted value of expenses when the independent variables are equal to zero. However, in many cases, the intercept is of little explanatory value by itself, as it is often impossible to have values of zero for all features.

This is the case here, where no insured person can exist with zero age or no miles driven, and consequently, the intercept has no real-world interpretation. For this reason, in practice, the intercept is often ignored.

The beta coefficients indicate the estimated increase in insurance claims costs for an increase of one unit in each feature, assuming all other values are held constant. For instance, for each additional year of age, we would expect \$1.89 lower insurance claims costs on average, assuming everything else is held equal. Similarly, we would expect \$0.12 higher claims for each additional mile driven and \$0.03 higher per dollar of insured value, all else equal.

You might notice that although we only specified 10 features in our model formula, there are 13 coefficients reported in addition to the intercept. This happened because the `lm()` function automatically applies dummy coding to each of the factor-type variables included in the model.

As explained in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, dummy coding allows a nominal feature to be treated as numeric by creating a binary variable for each category of the feature except one, which serves as the reference category. Each dummy variable is set to 1 if the observation falls into the specified category or 0 otherwise. For example, the `geo_area` feature has three categories: urban, suburban, and rural. Thus, two dummy variables were used, which are named `geo_areaurban` and `geo_areasuburban`. For the observations where the `geo_area = "rural"`, `geo_areaurban` and `geo_areasuburban` will both be set to zero. Similarly, for the four-category `vehicle_type` feature, R created three dummy variables named `vehicle_typeminivan`, `vehicle_typesuv`, and `vehicle_typetruck`. This left `vehicle_type = "car"` to serve as the reference category when the three dummy variables are all zero.

When dummy coded features are used in a regression model, the regression coefficients are interpreted relative to the categories that were omitted. In our model, R automatically held out the `geo_arearural` and `vehicle_typecar` variables, making rural car owners the reference group. Thus, urban dwellers have \$169.11 more claims costs each year relative to rural areas and trucks cost the insurer an average of \$21.57 more than cars per year. To be clear, these differences assume all other features are held equal, so they are independent of the fact that rural drivers may drive more miles or less expensive vehicles. We would expect two people who are otherwise identical, except that one lives in a rural area and one lives in an urban area, to differ by about \$170, on average.



By default, R uses the first level of the factor variable as the reference. If you would prefer to use another level, the `relevel()` function can be used to specify the reference group manually. Use the `?relevel` command in R for more information.

In general, the results of the linear regression model make logical sense; however, we currently have no sense of how well the model is fitting the data. We'll answer this question in the next section.

Step 4 – evaluating model performance

The parameter estimates we obtained by typing `ins_model` tell us about how the independent variables are related to the dependent variable, but they tell us nothing about how well the model fits our data. To evaluate the model performance, we can use the `summary()` command on the stored model:

```
> summary(ins_model)
```

This produces the following output, which has been annotated for illustrative purposes:

Call:
`lm(formula = expenses ~ ., data = insurance)`

Residuals:

Min	1Q	Median	3Q	Max
-6707	-1989	-1492	-1057	231252

1

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-1154.914856	351.370732	-3.287	0.00101 **
age	-1.886027	3.144556	-0.600	0.54866
geo_areasuburban	191.078953	143.199245	1.334	0.18210
geo_areaurban	169.114255	157.850516	1.071	0.28402
vehicle_typeminivan	115.278619	276.579845	0.417	0.67683
vehicle_typesuv	-19.695001	117.990151	-0.167	0.86743
vehicle_typetruck	21.568360	153.630939	0.140	0.88835
est_value	0.031145	0.002497	12.475 < 0.0000000000000002 ***	
miles_driven	0.118986	0.014327	8.305 < 0.0000000000000002 ***	
college_grad_ind	-25.040302	115.578315	-0.217	0.82848
speeding_ticket_ind	155.824097	140.155213	1.112	0.26624
hard_braking_ind	11.845220	106.912005	0.111	0.91178
late_driving_ind	362.485502	224.655385	1.614	0.10665
clean_driving_ind	-239.047399	111.076229	-2.152	0.03140 *

Signif. codes:	0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1			

2

Residual standard error: 6995 on 19986 degrees of freedom
 Multiple R-squared: 0.01241, Adjusted R-squared: 0.01176
 F-statistic: 19.31 on 13 and 19986 DF, p-value: < 0.000000000000022

3

Figure 6.11: The summary output from a regression model can be divided into three main components, which are annotated in this figure

The `summary()` output may seem overwhelming at first, but the basics are easy to pick up. As indicated by the numbered labels in the preceding output, there are three main ways to evaluate the performance, or fit, of our model:

1. The **Residuals** section provides summary statistics for the prediction errors, some of which are apparently quite substantial. Since a residual is equal to the true value minus the predicted value, the maximum error of 231252 suggests that the model under-predicted expenses by more than \$230,000 for at least one observation. On the other hand, the majority of errors are relatively small negative values, which means that we are over-estimating expenses for most enrollees. This is exactly how the insurance company can afford to cover the expenses for costly accidents.
2. For each estimated regression coefficient, the **p-value**, denoted by $\Pr(|t|)$, provides an estimate of the probability that the true coefficient is zero given the value of the estimate. Small p-values suggest that the true coefficient is very unlikely to be zero, which means that the feature is extremely unlikely to have no relationship with the dependent variable. Note that some of the p-values have stars (**), which correspond to the footnotes that specify the **significance level** met by the estimate. This level is a threshold, chosen prior to building the model, which will be used to indicate “real” findings, as opposed to those due to chance alone; p-values less than the significance level are considered **statistically significant**. If the model had few such terms, it may be a cause for concern, since this would indicate that the features used are not very predictive of the outcome. Here, our model has a few highly significant variables, and they seem to be related to the outcome in expected ways.
3. The **Multiple R-squared** value (also called the coefficient of determination) provides a measure of how well our model as a whole explains the values of the dependent variable. It is similar to the correlation coefficient in that the closer the value is to 1.0, the better the model perfectly explains the data. Since the R-squared value is 0.01241, we know that the model explains about 1.2 percent of the variation in the dependent variable. Because models with more features always explain more variation, the **Adjusted R-squared** value corrects R-squared by penalizing models with a large number of independent variables. This is useful for comparing the performance of models with different numbers of explanatory variables.

Given the preceding three performance indicators, our model is performing well enough. The size of some of the errors is a bit concerning, but not surprising given the nature of insurance expense data.

Additionally, it is not uncommon for regression models of real-world data to have low R-squared values. Although a value of 0.01241 is especially low, it reflects the fact that we have no proximate predictors of automobile accidents; to truly predict accidents, we would need real-time driving data, or at least some measure of true driving skill. This being said, as we will see in the next section, we still may be able to improve the model's performance by specifying the model in a slightly different way.

Step 5 – improving model performance

As mentioned previously, a key difference between regression modeling and other machine learning approaches is that regression typically leaves feature selection and model specification to the user. Consequently, if we have subject-matter knowledge about how a feature is related to the outcome, we can use this information to inform the model specification and potentially improve the model's performance.

Model specification – adding nonlinear relationships

In linear regression, the relationship between an independent variable and the dependent variable is assumed to be linear, yet this may not necessarily be true. For example, the effect of age on insurance expenditures may not be constant across all age values; the treatment may become disproportionately expensive for the youngest and oldest populations—a U-shaped curve, if expenses were plotted against age.

If you recall, a typical regression equation follows a form similar to this:

$$y = \alpha + \beta_1 x$$

To account for a nonlinear relationship, we can add a higher-order term to the regression equation, treating the model as a polynomial. In effect, we will be modeling a relationship like this:

$$y = \alpha + \beta_1 x + \beta_2 x^2$$

The difference between these two models is that an additional regression parameter will be estimated, which is intended to capture the effect of the x^2 term. This allows the impact of age to be measured as a function of age and age squared.

To add the nonlinear age to the model, we simply need to create a new variable:

```
> insurance$age2 <- insurance$age^2
```

Then, when we produce our improved model, we'll add both age and age2 to the `lm()` formula using the form `expenses ~ age + age2`. This will allow the model to separate the linear and nonlinear impact of age on medical expenses.

Model specification – adding interaction effects

So far, we have only considered each feature's individual contribution to the outcome. What if certain features have a combined impact on the dependent variable? For instance, a habit of hard braking and late driving may have harmful effects separately, but it is reasonable to assume that their combined effect may be worse than the sum of each one alone.

When two features have a combined effect, this is known as an **interaction**. If we suspect that two variables interact, we can test this hypothesis by adding their interaction to the model. Interaction effects are specified using the R formula syntax. To interact the hard braking indicator (`hard_braking_ind`) with the late driving indicator (`late_driving_ind`), we would write a formula in the form `expenses ~ hard_braking_ind*late_driving_ind`.

The `*` operator is a shorthand that instructs R to model `expenses ~ hard_braking_ind + late_driving_ind + hard_braking_ind:late_driving_ind`. The colon operator (`:`) in the expanded form indicates that `hard_braking_ind:late_driving_ind` is the interaction between the two variables. Note that the expanded form automatically also included the individual `hard_braking_ind` and `late_driving_ind` variables as well as their interaction.



If you have trouble deciding whether to include a variable, a common practice is to include it and examine the p-value. If the variable is not statistically significant, you have a plausible justification for excluding it from the model.

Putting it all together – an improved regression model

Based on a bit of subject-matter knowledge of how insurance costs may be related to enrollee characteristics, we developed what we think is a more accurately specified regression formula. To summarize the improvements, we:

- Added a nonlinear term for age
- Specified an interaction between hard braking and late driving

We'll train the model using the `lm()` function as before, but this time we'll add the interaction term in addition to `age2`, which will be included automatically:

```
> ins_model2 <- lm(expenses ~ . + hard_braking_ind:late_driving_ind,
  data = insurance)
```

Next, we summarize the results:

```
> summary(ins_model2)
```

The output is as follows:

```
Call:
lm(formula = expenses ~ . + hard_barking_ind:late_driving_ind,
  data = insurance)

Residuals:
    Min      1Q  Median      3Q     Max 
-6618    -1996   -1491    -1044   231358 

Coefficients:
              Estimate Std. Error t value Pr(>|z|)    
(Intercept) -535.038171  457.146614 -1.170  0.2419    
age          -33.142400   15.366892 -2.157  0.0310 *  
geo_areasuburban 178.825158  143.305863  1.248  0.2121    
geo_areaurban  132.463265  158.726709  0.835  0.4040    
vehicle_typeminivan 178.825158  143.305863  1.248  0.2121    
vehicle_typesuv   -8.006108  118.116633 -0.068  0.9460    
vehicle_typetruck  26.426396  153.650455  0.172  0.8634    
est_value        0.031179  0.002496  12.489 <0.000000002 ***  
miles_driven     0.118748  0.014327   8.289 <0.000000002 ***  
college_grad_ind 17.248581  117.398583  0.147  0.8832    
speeding_ticket_ind 155.061583  140.143658  1.107  0.2658    
hard_braking_ind -12.442358  109.794208 -0.113  0.9098    
late_driving_ind  183.329848  284.218859  0.645  0.5189    
clean driving_ind -232.843170  111.106714 -2.096  0.0361    
age2             0.343165   0.165340   2.076  0.0380    
hard_braking_ind:late_driving_ind 469.079140  461.685886  1.016  0.3096
```

```
Signif. codes:  0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6995 on 19984 degrees of freedom
Multiple R-squared:  0.01267,  Adjusted R-squared:  0.01193
F-statistic: 17.1 on 15 and 19984 DF,
p-value: <0.0000000000000022
```

Although the R-squared and adjusted R-squared values didn't change much compared to the previous model, the new features present some interesting insights. In particular, the estimate for age is relatively large and negative (lower expenses) but age2 is relatively small and positive (higher expenses). However, because age squared grows faster than age, expenses will begin to rise for very high age groups. The overall effect is a U-shaped expense curve, where the youngest and oldest enrollees are predicted to have higher expenses. The interaction of hard_braking_ind and late_driving_ind is also interesting, as it is a relatively large positive. Although the interaction is not statistically significant, the direction of the effect implies that driving late at night is especially dangerous if you are the type of driver that already drives dangerously.



Strictly speaking, regression modeling makes some strong assumptions about the data. These assumptions are not as important for numeric forecasting, as the model's worth is not based upon whether it truly captures the underlying process—we simply care about the accuracy of its predictions. However, if you would like to make firm inferences from the regression model coefficients, it is necessary to run diagnostic tests to ensure that the regression assumptions have not been violated. For an excellent introduction to this topic, see *Multiple Regression: A Primer*, Allison, P. D., Pine Forge Press, 1998.

Making predictions with a regression model

After examining the estimated regression coefficients and fit statistics, we can also use the model to predict the expenses of future enrollees on the insurance plan. To illustrate the process of making predictions, let's first apply the model to the original training data using the `predict()` function as follows:

```
> insurance$pred <- predict(ins_model2, insurance)
```

This saves the predictions as a new vector named `pred` in the insurance data frame. We can then compute the correlation between the predicted and actual costs of insurance:

```
> cor(insurance$pred, insurance$expenses)
[1] 0.1125714
```

The correlation of 0.11 suggests a relatively weak linear relationship between the predicted and actual values, which is disappointing but not too surprising given the seemingly random nature of motor vehicle accidents. It can also be useful to examine this finding as a scatterplot. The following R commands plot the relationship and then add an identity line with an intercept equal to zero and a slope equal to one. The `col`, `lwd`, and `lty` parameters affect the line color, width, and type, respectively:

```
> plot(insurance$pred, insurance$expenses)
> abline(a = 0, b = 1, col = "red", lwd = 3, lty = 2)
```

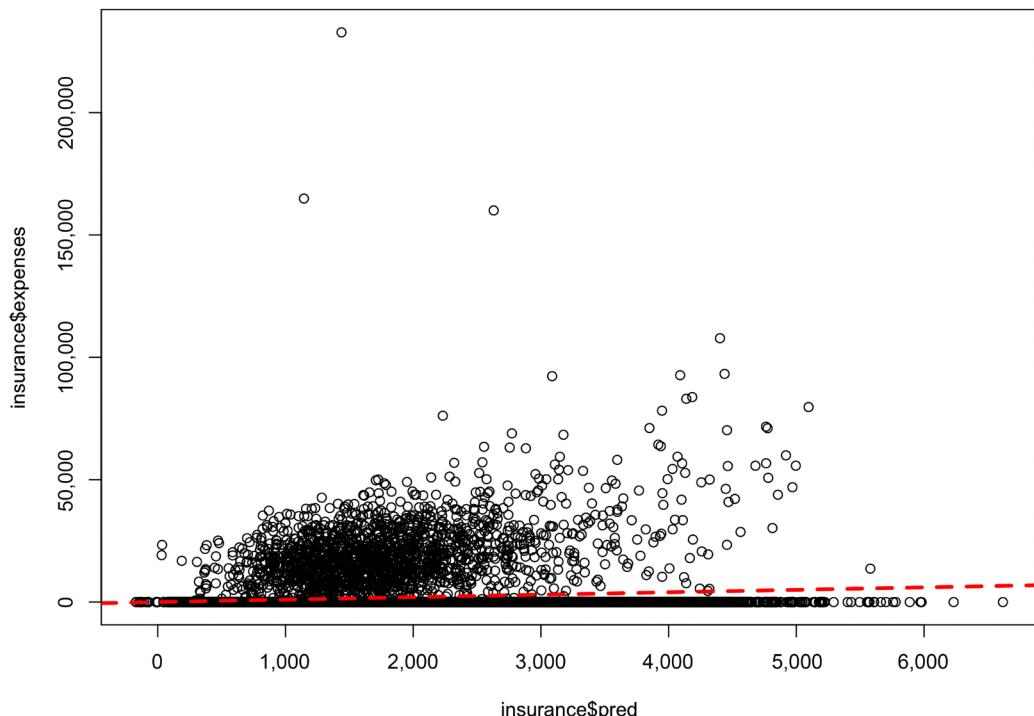


Figure 6.12: In this scatterplot, points falling on or near the diagonal dashed line where $y = x$ indicate the predictions that were very close to the actual values

The off-diagonal points falling above the line are cases where the actual expenses are greater than expected, while cases falling below the line are those less than expected. We can see here that a small number of people with much larger-than-expected expenses are balanced by a huge number of people with slightly smaller-than-expected expenses.

Now, suppose you would like to forecast the expenses for potential new enrollees on the insurance plan. To do this, you must provide the `predict()` function a data frame with the prospective drivers' data. In the case of many drivers, you may consider creating a CSV spreadsheet file to load in R, or for a smaller number, you may simply create a data frame within the `predict()` function itself. For example, to estimate the insurance expenses for a 30-year-old, rural driver of a truck valued at \$25,000 driven for about 14,000 miles annually with a clean driving record:

```
> predict(ins_model2,  
  data.frame(age = 30, age2 = 30^2, geo_area = "rural",  
            vehicle_type = "truck", est_value = 25000,  
            miles_driven = 14000, college_grad_ind = 0,  
            speeding_ticket_ind = 0, hard_braking_ind = 0,  
            late_driving_ind = 0, clean_driving_ind = 1))
```

```
1  
1015.059
```

Using this value, the insurance company would need to charge about \$1,015 annually to break even for this demographic group. To compare the rate for someone who is otherwise similar except for a history of a recent accident, use the `predict()` function in much the same way:

```
> predict(ins_model2,  
  data.frame(age = 30, age2 = 30^2, geo_area = "rural",  
            vehicle_type = "truck", est_value = 25000,  
            miles_driven = 14000, college_grad_ind = 0,  
            speeding_ticket_ind = 0, hard_braking_ind = 0,  
            late_driving_ind = 0, clean_driving_ind = 0))
```

```
1  
1247.903
```

Note that the difference between these two values, $1015.059 - 1247.903 = -232.844$, is the same as the estimated regression model coefficient for `clean_driving_ind`. On average, drivers with a clean history are estimated to have about \$232.84 less in expenses for the plan per year, all else being equal.

This illustrates the more general fact that the predicted expenses are a sum of each of the regression coefficients times their corresponding value in the prediction data frame. For instance, using the model's regression coefficient of 0.118748 for the miles driven, we can predict that adding 10,000 additional miles will result in an increase in expenses of $10,000 * 0.118748 = 1187.48$, which can be confirmed as follows:

```
> predict(ins_model2,
  data.frame(age = 30, age2 = 30^2, geo_area = "rural",
             vehicle_type = "truck", est_value = 25000,
             miles_driven = 14000, college_grad_ind = 0,
             speeding_ticket_ind = 0, hard_braking_ind = 0,
             late_driving_ind = 0, clean_driving_ind = 0))
```

```
1
1247.903
```

```
> 2435.384 - 1247.903
```

```
[1] 1187.481
```

Following similar steps for a number of additional customer risk segments, the insurance company would be able to develop a pricing structure that fairly sets costs according to drivers' estimated risk level while also maintaining a consistent profit across all segments.



Exporting the model's regression coefficients allows you to build your own forecasting function. One potential use case for doing so would be to implement the regression model in a customer database for real-time prediction.

Going further – predicting insurance policyholder churn with logistic regression

Actuarial estimates of claims costs are not the only potential application of machine learning inside an insurance company. Marketing and customer retention teams are likely to be very interested in predicting **churn**, or the customers that leave the company after choosing not to renew their insurance plan. In many businesses, preventing churn is highly valued, as churned customers not only reduce the income stream for one business but also often increase the revenue stream of a direct competitor. Additionally, marketing teams know that the costs of acquiring new customers are generally much higher than the costs of retaining an existing customer.

Therefore, knowing ahead of time which customers are most likely to churn can help direct retention resources to intervene and prevent churn before it happens.

Historically, marketing teams have used a simple model called **recency, frequency, monetary value (RFM)** to identify highly valuable customers as well as those most likely to churn. The RFM analysis considers three characteristics of each customer:

- How recently have they purchased? Customers that haven't purchased in a while may be less valuable and more likely to never return.
- How frequently do they purchase? Do they come back year after year, or are there irregular gaps in their purchasing behavior? Customers that exhibit loyalty may be more valuable and more likely to return.
- How much money do they spend when they purchase? Do they spend more than the average customer or upgrade to premium products? These customers are more valuable financially, but also demonstrate their love of the brand.

Historical customer purchase data is collected with the intention of developing measures of each of these three factors. The measures are then converted into a standard scale (such as a scale from zero to ten) for each of the three areas and summed to create a final RFM score for each customer. A very recent and frequent purchaser that spends an average amount may have a combined score of $10 + 10 + 5 = 25$ while a customer that purchased once a long time ago may have a score of $2 + 1 + 4 = 7$, which places them much lower on the RFM scale.

This type of analysis is a crude but useful tool for understanding a set of customers and helping identify the types of data that may be useful for predicting churn. However, an RFM analysis is not particularly scientific and provides no formal estimation of the probability of churn or the factors that increase its likelihood. In contrast, a logistic regression model predicting a binary churn outcome provides both an estimated probability of churn for each customer as well as the impact of each predictor.

As with the insurance claims cost example, we'll build a churn model using a simulated dataset created for this book, which is intended to approximate the behavior of customers of the automobile insurance company.



If you would like to follow along interactively, download the `insurance_churn.csv` file from the Packt Publishing GitHub repository for this book and save it to your R working folder.

The churn dataset includes 5,000 examples of current and former beneficiaries enrolled in the hypothetical automobile insurance plan. Each example includes features measuring customer behaviors in the plan year, as well as a binary indicator (churn) of whether they churned out of the plan by not renewing at the end of the year. The available features include:

- `member_id`: A randomly assigned customer identification number
- `loyalty_years`: The number of consecutive years enrolled in the insurance plan
- `vehicles_covered`: The number of vehicles covered by the insurance plan
- `premium_plan_ind`: A binary indicator that the member paid for a premium, high-cost version of the plan with additional benefits
- `mobile_app_user`: A binary indicator that the member uses the mobile phone companion application
- `home_auto_bundle`: A binary indicator that the member also holds a homeowner's insurance plan offered by the same company
- `auto_pay_ind`: A binary indicator that the member has automatic payments turned on
- `recent_rate_increase`: A binary indicator that the member's price was raised recently



Notice that many of these factors relate to the three components of RFM in that they are measures of loyalty and monetary value. Thus, even if a more sophisticated model is built later, it can still be helpful to perform an RFM analysis as an earlier step in the process.

To read this dataset into R, type:

```
> churn_data <- read.csv("insurance_churn.csv")
```

Using the `table()` and `prop.table()` functions, we can see the overall churn rate is just over 15 percent:

```
> prop.table(table(churn_data$churn))
```

0	1
0.8492	0.1508

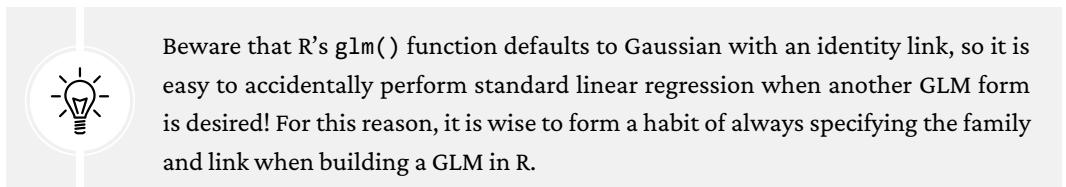
In a more formal analysis, it would be wise to perform more data exploration before going further. Here, we will jump ahead to creating the logistic regression model to predict these churned customers.

The `glm()` function, which is part of R's built-in `stats` package, is used to fit a GLM model such as logistic regression as well as the other variants like Poisson regression described earlier in the chapter. The syntax for logistic regression is shown in the following figure:

Logistic regression modeling syntax	
using the <code>glm()</code> function in the <code>stats</code> package	
Building the model:	
<pre>m <- glm(dv ~ iv, data = mydata, family = binomial(link = "logit")</pre> <ul style="list-style-type: none"> • <code>dv</code> is the dependent variable in the <code>mydata</code> data frame to be modeled • <code>iv</code> is an R formula specifying the independent variables in the <code>mydata</code> data frame to use in the model • <code>data</code> specifies the data frame in which the <code>dv</code> and <code>iv</code> variables can be found • <code>family</code> sets the distribution and link function to be used for the GLM; the above syntax is for logistic regression, but note that linear regression is used by default if <code>family</code> is unspecified (see <code>?family</code> for help with other GLM variants) 	
The function will return a GLM model object, which can be used to make predictions. Interactions between independent variables can be specified using the <code>*</code> operator.	
Making predictions:	
<pre>p <- predict(m, test, type = "link")</pre> <ul style="list-style-type: none"> • <code>m</code> is a model trained by the <code>glm()</code> function • <code>test</code> is a data frame containing test data with the same features as the training data used to build the model • <code>type</code> specifies the type of prediction to return, either <code>"link"</code> for predicted log-odds values (the default) or <code>"response"</code> for predicted probabilities (typically the more useful of the two types) 	
The function will return a vector of predicted values.	
Example:	
<pre>churn_model <- glm(churn ~ ., data = churn_data, family = binomial(link = "logit")) churn_prob <- predict(churn_model, churn_test, type = "response")</pre>	

Figure 6.13: Logistic regression syntax

Note the many similarities between the `glm()` function syntax and the `lm()` function used earlier for standard linear regression. Aside from specifying the family and link function, fitting the model is no more difficult. The key differences are primarily in how the resulting model is interpreted.



To fit the logistic regression churn model, we specify the `binomial` family with the `logit` link function. Here, we model churn as a function of all other features in the dataset, minus `member_id`, which is unique to each member and therefore useless for prediction:

```
> churn_model <- glm(churn ~ . - member_id, data = churn_data,
  family = binomial(link = "logit"))
```

Using `summary()` on the resulting `churn_model` object shows the estimated regression parameters:

```
> summary(churn_model)
```

Call:

```
glm(formula = churn ~ . - member_id,
  family = binomial(link = "logit"), data = ins_churn)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-1.1244	-0.6152	-0.5033	-0.3950	2.4995

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)	
(Intercept)	-0.488893	0.141666	-3.451	0.000558 ***	
loyalty_years	-0.072284	0.007193	-10.050	< 2e-16 ***	
vehicles_covered	-0.212980	0.055237	-3.856	0.000115 ***	
premium_plan_ind	-0.370574	0.148937	-2.488	0.012842 *	
mobile_app_user	-0.292273	0.080651	-3.624	0.000290 ***	
home_auto_bundle	-0.267032	0.093932	-2.843	0.004472 **	
auto_pay_ind	-0.075698	0.106130	-0.713	0.475687	
recent_rate_increase	0.648100	0.102596	6.317	2.67e-10 ***	

Signif. codes:	0 ****	0.001 ***	0.01 **	0.05 *.	0.1 ' '

(Dispersion parameter for binomial family taken to be 1)

```
Null deviance: 4240.9 on 4999 degrees of freedom
Residual deviance: 4059.2 on 4992 degrees of freedom
AIC: 4075.2
```

Number of Fisher Scoring iterations: 5

At a high level, the logistic regression output is fairly similar to a linear regression output. The p-values (labeled $\text{Pr}(>|z|)$) and significance codes (denoted by * characters) indicate whether the variables are statistically significant. All features aside from the `auto_pay_ind` are significant at the 0.05 level or better. The direction of the relationship between the predictors and the target outcome can also be understood simply by looking at the sign (positive or negative) before the `Estimate` value. Nearly all estimates are negative, which implies that these features reduce churn, except for `recent_rate_increase`, which is positive and therefore increases churn. These connections make sense; an increase in the price of the insurance plan would be expected to increase churn, while members that have been loyal for years or pay for premium plan features are less likely to leave.

Interpreting the impact of a specific feature on churn is where logistic regression is trickier than linear regression, as the estimates are shown in log odds. Suppose we want to know how much more likely churn is after a recent increase in the price of the insurance plan. Since the estimate for `recent_rate_increase` is 0.6481, this means that the log odds of churn increase by 0.6481 when the rate increase indicator is 1 versus when it is 0. Exponentiating this to remove the logarithm and find the odds ratio, we find that $\exp(0.6481) = 1.911905$, which implies that churn is almost twice as likely (or 91.2 percent more likely) after a rate increase.

In the opposite direction, members that use the mobile app (`mobile_app_user`) have an estimated difference in log odds of -0.292273 versus those that do not. Finding the odds ratio as $\exp(-0.292273) = 0.7465647$ suggests that the churn of app users is about 75 percent of those that do not use the app, or a decrease of about 25 percent for app users. Similarly, we can find that churn is reduced by about seven percent for each additional year of loyalty, as $\exp(-0.072284) = 0.9302667$. Similar calculations can be performed for all other predictors in the model, including the intercept, which represents the odds of churn when all predictors are zero.

To use this model to prevent churn, we can make predictions on a database of current plan members. Let's begin by loading a dataset containing 1000 subscribers, using the `test` dataset available for this chapter:

```
> churn_test <- read.csv("insurance_churn_test.csv")
```

We'll then use the logistic regression model object with the `predict()` function to add a new column to this data frame, which contains the predictions for each member:

```
> churn_test$churn_prob <- predict(churn_model, churn_test,  
type = "response")
```

Note that the `type = "response"` parameter is set so that the predictions are in probabilities rather than the default `type = "link"` setting, which produces predictions as log odds values.

Summarizing these predicted probabilities, we see that the average churn probability is about 15 percent, but some users are predicted to have very low churn, while others have a churn probability as high as 41 percent:

```
> summary(churn_test$churn_prob)

    Min. 1st Qu. Median     Mean 3rd Qu.     Max.
0.02922 0.09349 0.13489 0.14767 0.18452 0.41604
```

Suppose the customer retention team has the resources to intervene in a limited number of cases. By sorting the members to identify those with the highest predicted churn likelihood, we can provide the team with the direction most likely to make the greatest impact.

First, use the `order()` function to obtain a vector with the row numbers sorted in decreasing order according to their churn probability:

```
> churn_order <- order(churn_test$churn_prob, decreasing = TRUE)
```

Next, after ordering the `churn_test` data frame according to the `churn_order` vector and taking the two columns of interest, use the `head()` function to take the top `n` rows; in this case, we'll set `n = 5` to limit to the five members most likely to churn:

```
> head(churn_test[churn_order,
+                   c("member_id", "churn_prob")], n = 5)

  member_id churn_prob
406 29603520 0.4160438
742 12588881 0.4160438
390 23228258 0.3985958
541 86406649 0.3985958
614 49806111 0.3985958
```

After saving the result to a spreadsheet with `n` set to a higher number, it would be possible to provide the customer retention team with a list of the insurance plan members that are most likely to churn. Focusing retention efforts on these members is likely to be a more fruitful use of the marketing budget than targeting members at random, as most members have a very low churn probability. In this way, machine learning can provide a substantial return on minimal investment, with an impact that is easily quantifiable by comparing the churn rates before and after this intervention.

Estimates of revenue retained as a result of churn prevention can be obtained using simple assumptions about the proportion of customers that will respond to retention efforts. For example, if we assume N of members targeted by the churn model will be retained, this will result in N times $\$X$ retained revenue, where $\$X$ is the average customer spend. Bringing this number to stakeholders helps provide justification for implementing the machine learning project.

The example is only the tip of the iceberg, as churn modeling can become much more sophisticated with additional efforts. For instance, rather than targeting the customers with the highest churn probability, it is also possible to consider the revenue lost if the customer churns; it may be worth prioritizing high-value customers even if they have a lower churn probability than a low-value customer. Additionally, because some customers are assured to churn regardless of intervention while others may be more flexible about staying, it is possible to model retention likelihood in addition to modeling churn probability. In any case, even in a simple form, churn modeling is low-hanging fruit for most businesses and a great first place to implement machine learning.

Understanding regression trees and model trees

If you recall from *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, a decision tree builds a model, much like a flowchart, in which decision nodes, leaf nodes, and branches define a series of decisions, which are used to classify examples. Such trees can also be used for numeric prediction by making only small adjustments to the tree-growing algorithm. In this section, we will consider the ways in which trees for numeric prediction differ from trees used for classification.

Trees for numeric prediction fall into two categories. The first, known as **regression trees**, were introduced in the 1980s as part of the seminal **classification and regression tree (CART)** algorithm. Despite the name, regression trees do not use linear regression methods as described earlier in this chapter; rather, they make predictions based on the average value of examples that reach a leaf.



The CART algorithm is described in detail in *Classification and Regression Trees, Breiman, L., Friedman, J. H., Stone, C. J., Olshen, R. A., Chapman and Hall, 1984*.

The second type of tree for numeric prediction is known as a **model tree**. Introduced several years later than regression trees, they are less widely known, but perhaps more powerful. Model trees are grown in much the same way as regression trees, but at each leaf, a multiple linear regression model is built from the examples reaching that node. Depending on the number of leaf nodes, a model tree may build tens or even hundreds of such models.

This makes model trees more difficult to understand than the equivalent regression tree, with the benefit that they may result in a more accurate model.



The earliest model tree algorithm, **M5**, is described in *Learning with continuous classes*, Quinlan, J. R., *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*, 1992, pp. 343-348.

Adding regression to trees

Trees that can perform numeric prediction offer a compelling, yet often overlooked, alternative to regression modeling. The strengths and weaknesses of regression trees and model trees relative to the more common regression methods are listed in the following table:

Strengths	Weaknesses
<ul style="list-style-type: none"> • Combines the strengths of decision trees with the ability to model numeric data • Does not require the user to specify the model in advance • Uses automatic feature selection, which allows the approach to be used with a very large number of features • May fit some types of data much better than linear regression • Does not require knowledge of statistics to interpret the model 	<ul style="list-style-type: none"> • Not as well known as linear regression • Requires a large amount of training data • Difficult to determine the overall net effect of individual features on the outcome • Large trees can become more difficult to interpret than a regression model

Though traditional regression methods are typically the first choice for numeric prediction tasks, in some cases, numeric decision trees offer distinct advantages. For instance, decision trees may be better suited for tasks with many features or many complex, nonlinear relationships between features and the outcome; these situations present challenges for regression. Regression modeling also makes assumptions about the data that are often violated in real-world data; this is not the case for trees.

Trees for numeric prediction are built in much the same way as they are for classification. Beginning at the root node, the data is partitioned using a divide-and-conquer strategy according to the feature that will result in the greatest increase in homogeneity in the outcome after a split is performed.

In classification trees, you will recall that homogeneity is measured by entropy. This is undefined for numeric data. Instead, for numeric decision trees, homogeneity is measured by statistics such as variance, standard deviation, or absolute deviation from the mean.

One common splitting criterion is called the **standard deviation reduction (SDR)**. It is defined by the following formula:

$$\text{SDR} = \text{sd}(T) - \sum_i \frac{|T_i|}{|T|} \times \text{sd}(T_i)$$

In this formula, the $\text{sd}(T)$ function refers to the standard deviation of the values in set T , while T_1, T_2, \dots, T_n are sets of values resulting from a split on a feature. The $|T|$ term refers to the number of observations in set T . Essentially, the formula measures the reduction in standard deviation by comparing the standard deviation pre-split to the weighted standard deviation post-split.

For example, consider the following case in which a tree is deciding whether to perform a split on binary feature A or a split on binary feature B:

original data	1	1	1	2	2	3	4	5	5	6	6	7	7	7	7
split on feature A	1	1	1	2	2	3	4	5	5	6	6	7	7	7	7
split on feature B	1	1	1	2	2	3	4	5	5	6	6	7	7	7	7

 T_1 T_2

Figure 6.14: The algorithm considers splits on features A and B, which creates different T_1 and T_2 groups

Using the groups that would result from the proposed splits, we can compute the SDR for A and B as follows. The `length()` function used here returns the number of elements in a vector. Note that the overall group T is named `tee` to avoid overwriting R's built-in `T()` and `t()` functions.

```
> tee <- c(1, 1, 1, 2, 2, 3, 4, 5, 5, 6, 6, 7, 7, 7, 7)
> at1 <- c(1, 1, 1, 2, 2, 3, 4, 5, 5)
> at2 <- c(6, 6, 7, 7, 7, 7)
> bt1 <- c(1, 1, 1, 2, 2, 3, 4)
> bt2 <- c(5, 5, 6, 6, 7, 7, 7, 7)
> sdr_a <- sd(tee) - (length(at1) / length(tee) * sd(at1) +
  length(at2) / length(tee) * sd(at2))
> sdr_b <- sd(tee) - (length(bt1) / length(tee) * sd(bt1) +
  length(bt2) / length(tee) * sd(bt2))
```

Let's compare the SDR of A against the SDR of B:

```
> sdr_a  
[1] 1.202815  
  
> sdr_b  
[1] 1.392751
```

The SDR for the split on feature A was about 1.2 versus 1.4 for the split on feature B. Since the standard deviation was reduced more for the split on B, the decision tree would use B first. It results in slightly more homogeneous sets than does A.

Suppose that the tree stopped growing here using this one and only split. A regression tree's work is done. It can make predictions for new examples depending on whether the example's value on feature B places the example into group T_1 or T_2 . If the example ends up in T_1 , the model would predict $\text{mean}(bt1) = 2$, otherwise it would predict $\text{mean}(bt2) = 6.25$.

In contrast, a model tree would go one step further. Using the seven training examples falling in group T_1 and the eight in T_2 , the model tree could build a linear regression model of the outcome versus feature A. Note that feature B is of no help in building the regression model because all examples at the leaf have the same value of B—they were placed into T_1 or T_2 according to their value of B. The model tree can then make predictions for new examples using either of the two linear models.

To further illustrate the differences between these two approaches, let's work through a real-world example.

Example – estimating the quality of wines with regression trees and model trees

Winemaking is a challenging and competitive business, which offers the potential for great profit. However, there are numerous factors that contribute to the profitability of a winery. As an agricultural product, variables as diverse as the weather and the growing environment impact the quality of a varietal. The bottling and manufacturing can also affect the flavor for better or worse. Even the way the product is marketed, from the bottle design to the price point, can affect the customer's perception of the taste.

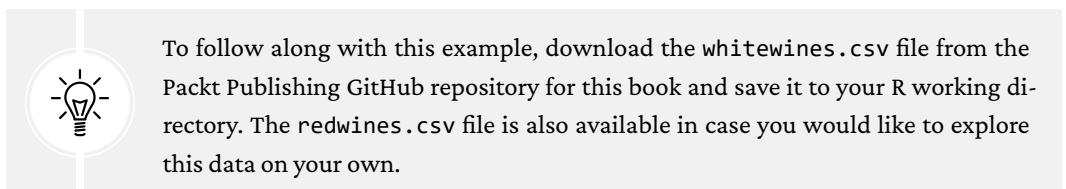
Consequently, the winemaking industry has invested heavily in data collection and machine learning methods that may assist with the decision science of winemaking. For example, machine learning has been used to discover key differences in the chemical composition of wines from different regions, and to identify the chemical factors that lead a wine to taste sweeter.

More recently, machine learning has been employed to assist with rating the quality of wine—a notoriously difficult task. A review written by a renowned wine critic often determines whether the product ends up on the top or bottom shelf, in spite of the fact that even expert judges are inconsistent when rating a wine in a blinded test.

In this case study, we will use regression trees and model trees to create a system capable of mimicking expert ratings of wine. Because trees result in a model that is readily understood, this could allow winemakers to identify key factors that contribute to better-rated wines. Perhaps more importantly, the system does not suffer from the human elements of tasting, such as the rater's mood or palate fatigue. Computer-aided wine testing may therefore result in a better product, as well as more objective, consistent, and fair ratings.

Step 1 – collecting data

To develop the wine rating model, we will use data donated to the UCI Machine Learning Repository (<http://archive.ics.uci.edu/ml>) by P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. Their dataset includes examples of red and white Vinho Verde wines from Portugal—one of the world's leading wine-producing countries. Because the factors that contribute to a highly rated wine may differ between the red and white varieties, for this analysis we will examine only the more popular white wines.



To follow along with this example, download the `whitewines.csv` file from the Packt Publishing GitHub repository for this book and save it to your R working directory. The `redwines.csv` file is also available in case you would like to explore this data on your own.

The white wine data includes information on 11 chemical properties of 4,898 wine samples. For each wine, a laboratory analysis measured characteristics such as acidity, sugar content, chlorides, sulfur, alcohol, pH, and density. The samples were then rated in a blind tasting by panels of no less than three judges on a quality scale ranging from 0 (very bad) to 10 (excellent). In the case that the judges disagreed on the rating, the median value was used.

The study by Cortez evaluated the ability of three machine learning approaches to model the wine data: multiple regression, artificial neural networks, and support vector machines. We covered multiple regression earlier in this chapter, and we will learn about neural networks and support vector machines in *Chapter 7, Black-Box Methods – Neural Networks and Support Vector Machines*. The study found that the support vector machine offered significantly better results than the linear regression model. However, unlike regression, the support vector machine model is difficult to interpret. Using regression trees and model trees, we may be able to improve the regression results while still having a model that is relatively easy to understand.



To read more about the wine study described here, refer to *Modeling wine preferences by data mining from physicochemical properties*, Cortez, P., Cerdeira, A., Almeida, F., Matos, T., and Reis, J., *Decision Support Systems*, 2009, Vol. 47, pp. 547-553.

Step 2 – exploring and preparing the data

As usual, we will use the `read.csv()` function to load the data into R. Since all features are numeric, we can safely ignore the `stringsAsFactors` parameter:

```
> wine <- read.csv("whitewines.csv")
```

The `wine` data includes 11 features and the quality outcome, as follows:

```
> str(wine)
```

```
'data.frame': 4898 obs. of 12 variables:
 $ fixed.acidity      : num  6.7 5.7 5.9 5.3 6.4 7 7.9 ...
 $ volatile.acidity   : num  0.62 0.22 0.19 0.47 0.29 ...
 $ citric.acid        : num  0.24 0.2 0.26 0.1 0.21 0.41 ...
 $ residual.sugar     : num  1.1 16 7.4 1.3 9.65 0.9 ...
 $ chlorides          : num  0.039 0.044 0.034 0.036 0.041 ...
 $ free.sulfur.dioxide: num  6 41 33 11 36 22 33 17 34 40 ...
 $ total.sulfur.dioxide: num  62 113 123 74 119 95 152 ...
 $ density            : num  0.993 0.999 0.995 0.991 0.993 ...
 $ pH                 : num  3.41 3.22 3.49 3.48 2.99 3.25 ...
 $ sulphates          : num  0.32 0.46 0.42 0.54 0.34 0.43 ...
 $ alcohol             : num  10.4 8.9 10.1 11.2 10.9 ...
 $ quality             : int  5 6 6 4 6 6 6 6 6 7 ...
```

Compared with other types of machine learning models, one of the advantages of trees is that they can handle many types of data without preprocessing. This means we do not need to normalize or standardize the features.

However, a bit of effort to examine the distribution of the outcome variable is needed to inform our evaluation of the model's performance. For instance, suppose that there was very little variation in quality from wine to wine, or that wines fell into a bimodal distribution: either very good or very bad. This may impact the way we design the model. To check for such extremes, we can examine the distribution of wine quality using a histogram:

```
> hist(wine$quality)
```

This produces the following figure:



Figure 6.15: The distribution of the quality ratings of white wines

The wine quality values appear to follow a roughly normal, bell-shaped distribution, centered around a value of six. This makes sense intuitively, because most wines are of average quality; few are particularly bad or good. Although the results are not shown here, it is also useful to examine the `summary(wine)` output for outliers or other potential data problems. Even though trees are fairly robust to messy data, it is always prudent to check for severe problems. For now, we'll assume that the data is reliable.

Our last step, then, is to divide the dataset into training and testing sets. Since the wine dataset was already sorted randomly, we can partition it into two sets of contiguous rows as follows:

```
> wine_train <- wine[1:3750, ]  
> wine_test <- wine[3751:4898, ]
```

In order to mirror the conditions used by Cortez, we used sets of 75 percent and 25 percent for training and testing, respectively. We'll evaluate the performance of our tree-based models on the testing data to see if we can obtain results comparable to the prior research study.

Step 3 – training a model on the data

We will begin by training a regression tree model. Although almost any implementation of decision trees can be used to perform regression tree modeling, the `rpart` (recursive partitioning) package offers the most faithful implementation of regression trees as they were described by the CART team. As the classic R implementation of CART, the `rpart` package is also well documented and supported with functions for visualizing and evaluating the `rpart` models.

Install the `rpart` package using the `install.packages("rpart")` command. It can then be loaded into your R session using the `library(rpart)` statement. The following syntax will train a tree using the default settings, which work well most of the time, but not always. If you need more fine-tuned settings, refer to the documentation for the control parameters using the `?rpart.control` command.

Regression tree syntaxusing the `rpart()` function in the `rpart` package**Building the model:**

```
m <- rpart(dv ~ iv, data = mydata)
```

- `dv` is the dependent variable in the `mydata` data frame to be modeled
- `iv` is an R formula specifying the independent variables in the `mydata` data frame to use in the model
- `data` specifies the data frame in which the `dv` and `iv` variables can be found

The function will return a regression tree model object, which can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "vector")
```

- `m` is a model trained by the `rpart()` function
- `test` is a data frame containing test data with the same features as the training data used to build the model
- `type` specifies the type of prediction to return, either "`vector`" (for predicted numeric values), "`class`" for predicted classes, or "`prob`" (for predicted class probabilities)

The function will return a vector of predictions depending on the `type` parameter.

Example:

```
wine_model <- rpart(quality ~ alcohol + sulfates,
                      data = wine_train)
wine_predictions <- predict(wine_model, wine_test)
```

Figure 6.16: Regression tree syntax

Using the R formula interface, we can specify `quality` as the outcome variable and use the dot notation to allow all other columns in the `wine_train` data frame to be used as predictors. The resulting regression tree model object is named `m.rpart` to distinguish it from the model tree we will train later:

```
> m.rpart <- rpart(quality ~ ., data = wine_train)
```

For basic information about the tree, simply type the name of the model object:

```
> m.rpart
n= 3750

node), split, n, deviance, yval
      * denotes terminal node

1) root 3750 2945.53200 5.870933
  2) alcohol< 10.85 2372 1418.86100 5.604975
    4) volatile.acidity>=0.2275 1611  821.30730 5.432030
      8) volatile.acidity>=0.3025 688   278.97670 5.255814 *
      9) volatile.acidity< 0.3025 923   505.04230 5.563380 *
     15) volatile.acidity< 0.2275 761   447.36400 5.971091 *
    3) alcohol>=10.85 1378 1070.08200 6.328737
      6) free.sulfur.dioxide< 10.5 84    95.55952 5.369048 *
      7) free.sulfur.dioxide>=10.5 1294  892.13600 6.391036
        14) alcohol< 11.76667 629   430.11130 6.173291
          28) volatile.acidity>=0.465 11    10.72727 4.545455 *
          29) volatile.acidity< 0.465 618   389.71680 6.202265 *
     15) alcohol>=11.76667 665   403.99400 6.596992 *
```

For each node in the tree, the number of examples reaching the decision point is listed. For instance, all 3,750 examples begin at the root node, of which 2,372 have `alcohol < 10.85` and 1,378 have `alcohol >= 10.85`. Because `alcohol` was used first in the tree, it is the single most important predictor of wine quality.

Nodes indicated by * are terminal or leaf nodes, which means that they result in a prediction (listed here as `yval`). For example, node 5 has a `yval` of 5.971091. When the tree is used for predictions, any wine samples with `alcohol < 10.85` and `volatile.acidity < 0.2275` would therefore be predicted to have a quality value of 5.97.

A more detailed summary of the tree's fit, including the mean squared error for each of the nodes and an overall measure of feature importance, can be obtained using the `summary(m.rpart)` command.

Visualizing decision trees

Although the tree can be understood using only the preceding output, it is often more readily understood using visualization. The `rpart.plot` package by Stephen Milborrow provides an easy-to-use function that produces publication-quality decision trees.



For more information on `rpart.plot`, including additional examples of the types of decision tree diagrams the function can produce, refer to the author's website at <http://www.milbo.org/rpart-plot/>.

After installing the package using the `install.packages("rpart.plot")` command, the `rpart.plot()` function produces a tree diagram from any `rpart` model object. The following commands plot the regression tree we built earlier:

```
> library(rpart.plot)
> rpart.plot(m.rpart, digits = 3)
```

The resulting tree diagram is as follows:

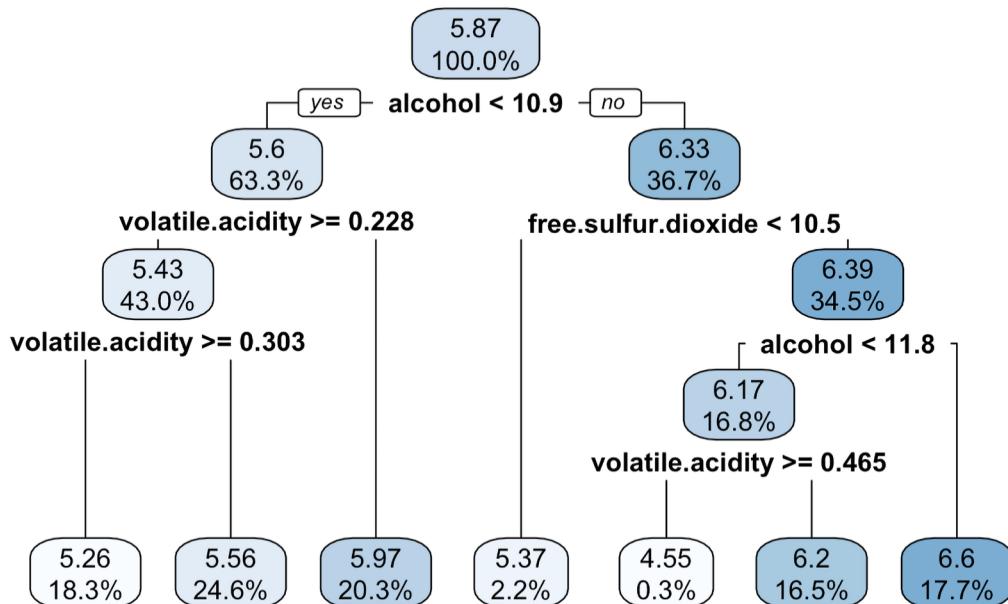


Figure 6.17: A visualization of the wine quality regression tree model

In addition to the `digits` parameter, which controls the number of numeric digits to include in the diagram, many other aspects of the visualization can be adjusted. The following command shows just a few of the useful options:

```
> rpart.plot(m.rpart, digits = 4, fallen.leaves = TRUE,
             type = 3, extra = 101)
```

The `fallen.leaves` parameter forces the leaf nodes to be aligned at the bottom of the plot, while the `type` and `extra` parameters affect the way the decisions and nodes are labeled. The numbers 3 and 101 refer to specific style formats, which can be found in the command's documentation or via experimentation with various numbers.

The result of these changes is a very different-looking tree diagram:

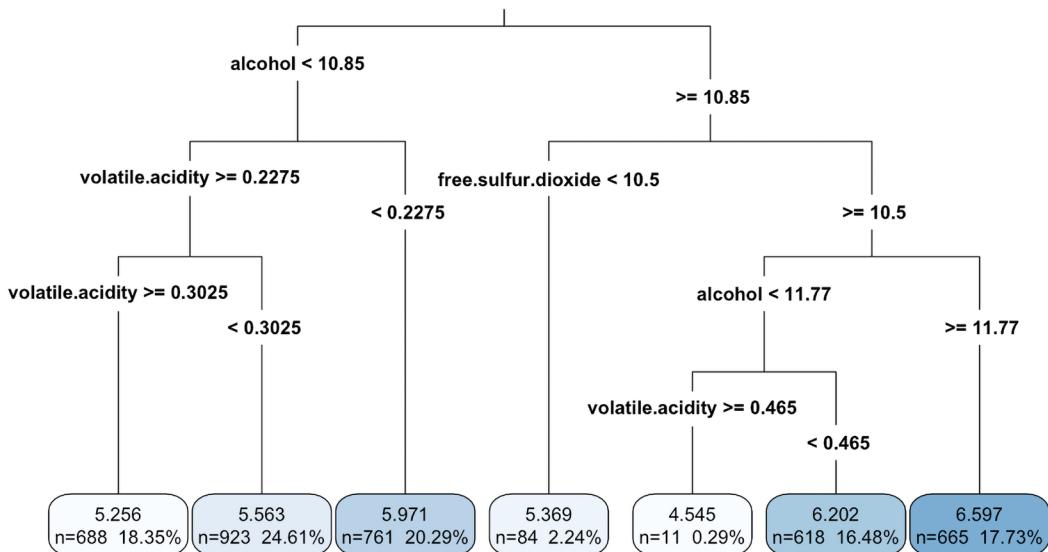


Figure 6.18: Changing the plot function parameters allows customization of the tree visualization

Visualizations like these may assist with the dissemination of regression tree results, as they are readily understood even without a mathematics background. In both cases, the numbers shown in the leaf nodes are the predicted values for the examples reaching that node. Showing the diagram to the wine producers may thus help to identify the key factors involved in predicting higher-rated wines.

Step 4 – evaluating model performance

To use the regression tree model to make predictions on the test data, we use the `predict()` function. By default, this returns the estimated numeric value for the outcome variable, which we'll save in a vector named `p.rpart`:

```
> p.rpart <- predict(m.rpart, wine_test)
```

A quick look at the summary statistics of our predictions suggests a potential problem: the predictions fall into a much narrower range than the true values:

```
> summary(p.rpart)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
4.545	5.563	5.971	5.893	6.202	6.597

```
> summary(wine_test$quality)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
3.000	5.000	6.000	5.901	6.000	9.000

This finding suggests that the model is not correctly identifying the extreme cases, and in particular, the best and worst wines. On the other hand, between the first and third quartile, we may be doing well.

The correlation between the predicted and actual quality values provides a simple way to gauge the model's performance. Recall that the `cor()` function can be used to measure the relationship between two equal-length vectors. We'll use this to compare how well the predicted values correspond to the true values:

```
> cor(p.rpart, wine_test$quality)
```

```
[1] 0.5369525
```

A correlation of 0.54 is certainly acceptable. However, the correlation only measures how strongly the predictions are related to the true value; it is not a measure of how far off the predictions were from the true values.

Measuring performance with the mean absolute error

Another way to think about the model's performance is to consider how far, on average, its prediction was from the true value. This measurement is called the **mean absolute error (MAE)**.

The equation for MAE is as follows, where n indicates the number of predictions and e_i indicates the error for prediction i :

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |e_i|$$

As the name implies, this equation takes the mean of the absolute value of the errors. Since the error is just the difference between the predicted and actual values, we can create a simple `MAE()` function as follows:

```
> MAE <- function(actual, predicted) {
  mean(abs(actual - predicted))
}
```

The MAE for our predictions is then:

```
> MAE(p.rpart, wine_test$quality)
[1] 0.5872652
```

This implies that, on average, the difference between our model's predictions and the true quality score was about 0.59. On a quality scale from 0 to 10, this seems to suggest that our model is doing fairly well.

On the other hand, recall that most wines were neither very good nor very bad; the typical quality score was around 5 to 6. Therefore, a classifier that did nothing but predict the mean value may also do fairly well according to this metric.

The mean quality rating in the training data is as follows:

```
> mean(wine_train$quality)
[1] 5.870933
```

If we predicted the value 5.87 for every wine sample, we would have a MAE of only about 0.67:

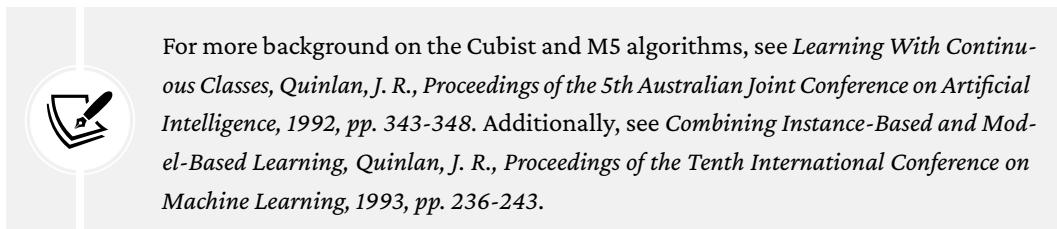
```
> MAE(5.87, wine_test$quality)
[1] 0.6722474
```

Our regression tree ($MAE = 0.59$) comes closer on average to the true quality score than the imputed mean ($MAE = 0.67$), but not by much. In comparison, Cortez reported an MAE of 0.58 for the neural network model and an MAE of 0.45 for the support vector machine. This suggests that there is room for improvement.

Step 5 – improving model performance

To improve the performance of our learner, let's apply a model tree algorithm, which is a more complex application of trees to numeric prediction. Recall that a model tree extends regression trees by replacing the leaf nodes with regression models. This often results in more accurate results than regression trees, which use only a single numeric value for the prediction at the leaf nodes.

The current state-of-the-art in model trees is the **Cubist** algorithm, which itself is an enhancement of the M5 model tree algorithm—both of which were published by J. R. Quinlan in the early 1990s. Though the implementation details are beyond the scope of this book, the Cubist algorithm involves building a decision tree, creating decision rules based on the branches of the tree, and building a regression model at each of the leaf nodes. Additional heuristics, such as pruning and boosting, are used to improve the quality of the predictions and smoothness across the range of predicted values.



The Cubist algorithm is available in R via the **Cubist** package and the associated **cubist()** function. The syntax of this function is shown in the following table:

Model tree syntax
using the cubist() function in the Cubist package
Building the model:
<pre>m <- cubist(train, class)</pre> <ul style="list-style-type: none"> • train is a data frame or matrix containing training data • class is a factor vector with the class for each row in the training data
The function will return a cubist model tree object, which can be used to make predictions.
Making predictions:
<pre>p <- predict(m, test)</pre> <ul style="list-style-type: none"> • m is a model trained by the cubist() function • test is a data frame containing test data with the same features as the training data used to build the model
The function will return a vector of predicted numeric values.
Example:
<pre>wine_model <- cubist(wine_train, wine_quality) wine_predictions <- predict(wine_model, wine_test)</pre>

Figure 6.19: Model tree syntax

We'll fit the Cubist model tree using a slightly different syntax from what was used for the regression tree, as the **cubist()** function does not accept the R formula syntax. Instead, we must specify the data frame columns used for the **x** independent variables and the **y** dependent variable. With the wine quality to be predicted residing in column 12, and using all other columns as predictors, the full command is as follows:

```
> library(Cubist)
> m.cubist <- cubist(x = wine_train[-12], y = wine_train$quality)
```

Basic information about the model tree can be examined by typing its name:

```
> m.cubist
```

```
Call:
cubist.default(x = wine_train[-12], y = wine_train$quality)
Number of samples: 3750
Number of predictors: 11
Number of committees: 1
Number of rules: 25
```

In this output, we see that the algorithm generated 25 rules to model the wine quality. To examine some of these rules, we can apply the `summary()` function to the `model` object. Since the complete tree is very large, only the first few lines of output depicting the first decision rule are included here:

```
> summary(m.cubist)

Rule 1: [21 cases, mean 5.0, range 4 to 6, est err 0.5]
if
  free.sulfur.dioxide > 30
  total.sulfur.dioxide > 195
  total.sulfur.dioxide <= 235
  sulphates > 0.64
  alcohol > 9.1
then
  outcome = 573.6 + 0.0478 total.sulfur.dioxide
            - 573 density - 0.788 alcohol
            + 0.186 residual.sugar - 4.73 volatile.acidity
```

You will note that the `if` portion of the output is somewhat like the regression tree we built earlier. A series of decisions based on the wine properties of sulfur dioxide, sulphates, and alcohol creates a rule culminating in the final prediction. A key difference between this model tree output and the earlier regression tree output, however, is that the nodes here terminate not in a numeric prediction, but rather in a linear model.

The linear model for this rule is shown in the `then` output following the `outcome =` statement. The numbers can be interpreted exactly the same as the multiple regression models we built earlier in this chapter. Each value is the estimated impact of the associated feature, that is, the net effect of one unit increase of that feature on the predicted wine quality. For example, the coefficient of 0.186 for residual sugar implies that for an increase of one unit of residual sugar, the wine quality rating is expected to increase by 0.186.

It is important to note that the regression effects estimated by this model apply only to wine samples reaching this node; an examination of the entirety of the Cubist output reveals that a total of 25 linear models were built in this model tree, one for each decision rule, and each with different parameter estimates of the impact of residual sugar and the 10 other features.

To examine the performance of this model, we'll look at how well it performs on the unseen test data. The `predict()` function gets us a vector of predicted values:

```
> p.cubist <- predict(m.cubist, wine_test)
```

The model tree appears to be predicting a wider range of values than the regression tree:

```
> summary(p.cubist)
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
	3.677	5.416	5.906	5.848	6.238	7.393

The correlation also seems to be substantially higher:

```
> cor(p.cubist, wine_test$quality)
```

[1]	0.6201015
-----	-----------

Furthermore, the model slightly reduced the MAE:

```
> MAE(wine_test$quality, p.cubist)
```

[1]	0.5339725
-----	-----------

Although we did not improve a great deal beyond the regression tree, we surpassed the performance of the neural network model published by Cortez, and we are getting closer to the published MAE value of 0.45 for the support vector machine model, all while using a much simpler learning method.



Not surprisingly, we have confirmed that predicting the quality of wines is a difficult problem; wine tasting, after all, is inherently subjective. If you would like additional practice, you may try revisiting this problem after reading *Chapter 14, Building Better Learners*, which covers additional techniques that may lead to better results.

Summary

In this chapter, we studied two methods for modeling numeric data. The first method, linear regression, involves fitting straight lines to data, but a technique called generalized linear modeling can adapt regression to other contexts as well. The second method uses decision trees for numeric prediction. The latter comes in two forms: regression trees, which use the average value of examples at leaf nodes to make numeric predictions, and model trees, which build a regression model at each leaf node in a hybrid approach that is, in some ways, the best of both worlds.

We began to understand the utility of regression modeling by using it to investigate the causes of the Challenger space shuttle disaster. We then used linear regression modeling to calculate the expected insurance claims costs for various segments of automobile drivers.

Because the relationship between the features and the target variable is well documented by the estimated regression model, we were able to identify certain demographics, such as high-mileage and late-night drivers, who may need to be charged higher insurance rates to cover their higher-than-average claims costs. We then applied logistic regression, a variant of regression used for binary classification, to the task of modeling insurance customer retention. These examples demonstrated the ability of regression to adapt flexibly to many types of real-world problems.

In a somewhat less businesslike application of machine learning, regression trees and model trees were used to model the subjective quality of wines from measurable characteristics. In doing so, we learned how regression trees offer a simple way to explain the relationship between features and a numeric outcome, but the more complex model trees may be more accurate. Along the way, we learned new methods for evaluating the performance of numeric models.

In stark contrast to this chapter, which covered machine learning methods that result in a clear understanding of the relationships between the input and the output, the next chapter covers methods that result in nearly incomprehensible models. The upside is that they are extremely powerful techniques—among the most powerful stock classifiers—which can be applied to both classification and numeric prediction problems.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



7

Black-Box Methods – Neural Networks and Support Vector Machines

The late science fiction author Arthur C. Clarke wrote, “Any sufficiently advanced technology is indistinguishable from magic.” This chapter covers a pair of machine learning methods that may appear at first glance to be magic. Though they are extremely powerful, their inner workings can be difficult to understand.

In engineering, these are referred to as black-box processes because the mechanism that transforms the input into the output is obfuscated by an imaginary box. For instance, the black box of closed-source software intentionally conceals proprietary algorithms, the black box of political lawmaking is rooted in bureaucratic processes, and the black box of sausage-making involves a bit of purposeful (but tasty) ignorance. In the case of machine learning, the black box is due to the complex mathematics allowing them to function.

Although they may not be easy to understand, it is dangerous to apply black-box models blindly. Thus, in this chapter, we’ll peek inside the box and investigate the statistical sausage-making involved in fitting such models. You’ll discover how:

- Neural networks mimic living brains to model mathematic functions
- Support vector machines use multidimensional surfaces to define the relationship between features and outcomes
- Despite their complexity, these can be applied easily to real-world problems

With any luck, you'll realize that you don't need a black belt in statistics to tackle black-box machine learning methods—there's no need to be intimidated!

Understanding neural networks

An **artificial neural network (ANN)** models the relationship between a set of input signals and an output signal using a model derived from our understanding of how a biological brain responds to stimuli from sensory inputs. Just like a brain uses a network of interconnected cells called **neurons** to provide vast learning capability, an ANN uses a network of artificial neurons or **nodes** to solve challenging learning problems.

The human brain is made up of about 85 billion neurons, resulting in a network capable of representing a tremendous amount of knowledge. As you might expect, this dwarfs the brains of other living creatures. For instance, a cat has roughly a billion neurons, a mouse has about 75 million neurons, and a cockroach has only about 1 million neurons. In contrast, many ANNs contain far fewer neurons, typically only hundreds or thousands, so we're in no danger of creating an artificial brain in the near future—even a fruit fly with 100,000 neurons far exceeds a state-of-the-art ANN on standard computing hardware. Some of the largest ANNs ever designed run on clusters of tens of thousands of CPU cores and have millions of nodes, but these are still dwarfed by the brains of small animals, let alone human beings—and the biological brains fit inside a much smaller package!

Though it may be infeasible to completely model a cockroach's brain, a neural network may still provide an adequate heuristic model of certain behaviors. Suppose that we develop an algorithm that can mimic how a roach flees when discovered. If the behavior of the robot roach is convincing, does it matter whether its brain is as sophisticated as the living creature? Similarly, if the written text produced by neural network-based tools like ChatGPT (<https://openai.com/blog/chatgpt/>) can pass for human-produced text most of the time, does it matter if the neural network isn't a perfect model of a human brain? This question is at the heart of the controversial **Turing test**, proposed in 1950 by the pioneering computer scientist Alan Turing, which grades a machine as intelligent if a human being cannot distinguish its behavior from a living creature's.

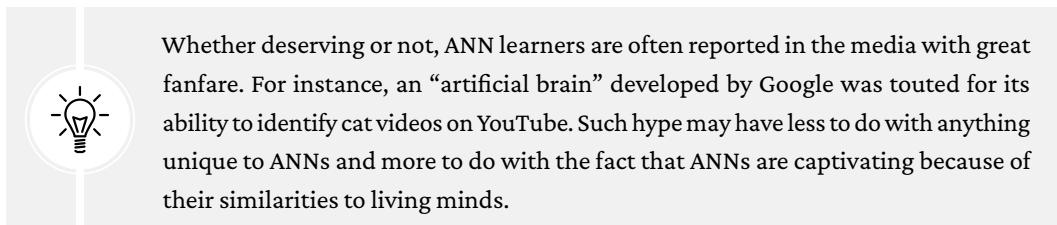


For more about the intrigue and controversy that surrounds the Turing test, refer to the *Stanford Encyclopedia of Philosophy*: <https://plato.stanford.edu/entries/turing-test/>.

Rudimentary ANNs have been used for over 60 years to simulate the brain's approach to problem-solving. At first, this involved learning simple functions like the logical AND function or the logical OR function. These early exercises were used primarily to help scientists understand how biological brains might operate. However, as computers have become increasingly powerful in recent years, the complexity of ANNs has likewise increased so much that they are now frequently applied to more practical problems, including:

- Speech, handwriting, and image recognition programs like those used by smartphone applications, mail-sorting machines, and search engines
- The automation of smart devices, such as an office building's environmental controls, or the control of self-driving cars and self-piloting drones
- Sophisticated models of weather and climate patterns, tensile strength, fluid dynamics, and many other scientific, social, or economic phenomena

Broadly speaking, ANNs are versatile learners that can be applied to nearly any learning task: classification, numeric prediction, and even unsupervised pattern recognition.



Whether deserving or not, ANN learners are often reported in the media with great fanfare. For instance, an “artificial brain” developed by Google was touted for its ability to identify cat videos on YouTube. Such hype may have less to do with anything unique to ANNs and more to do with the fact that ANNs are captivating because of their similarities to living minds.

ANNs are often applied to problems where the input data and output data are well defined, yet the process that relates the input to the output is extremely complex and hard to define. As a black-box method, ANNs work well for these types of black-box problems.

From biological to artificial neurons

Because ANNs were intentionally designed as conceptual models of human brain activity, it is helpful to first understand how biological neurons function. As illustrated in the following figure, incoming signals are received by the cell's **dendrites** through a biochemical process. The process allows the impulse to be weighted according to its relative importance or frequency. As the **cell body** begins to accumulate the incoming signals, a threshold is reached at which the cell fires, and the output signal is transmitted via an electrochemical process down the axon. At the axon's terminals, the electric signal is again processed as a chemical signal to be passed to the neighboring neurons across a tiny gap known as a **synapse**.

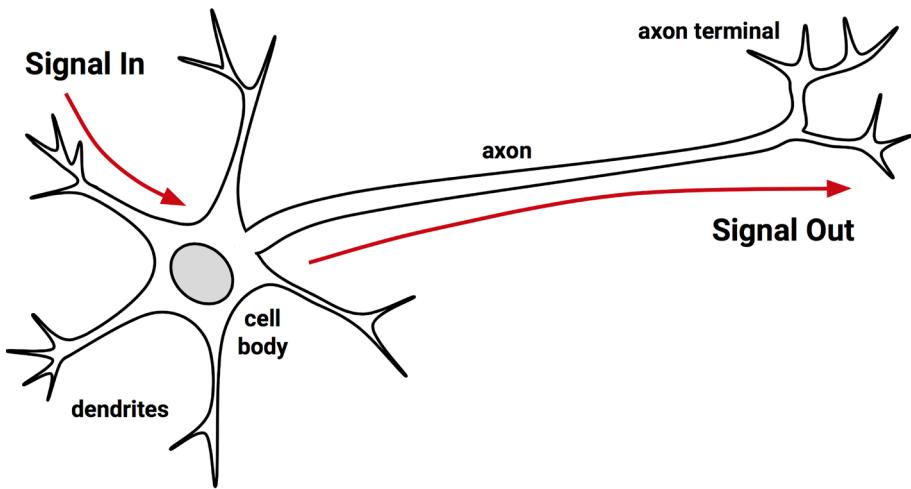


Figure 7.1: An artistic depiction of a biological neuron

The model of a single artificial neuron can be understood in terms very similar to the biological model. As depicted in the following figure, a directed network diagram defines a relationship between the input signals received by the dendrites (x variables) and the output signal (y variable). Just as with the biological neuron, each dendrite's signal is weighted (w values) according to its importance—ignore, for now, how these weights are determined. The input signals are summed by the cell body and the signal is passed on according to an **activation function** denoted by f .

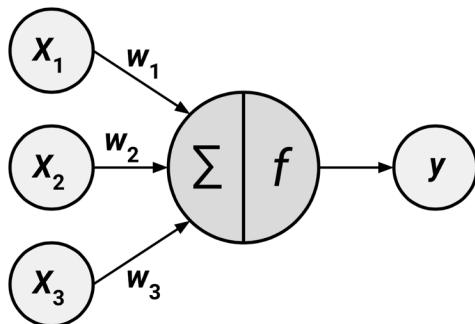


Figure 7.2: An artificial neuron is designed to mimic the structure and function of a biological neuron

A typical artificial neuron with n input dendrites can be represented by the formula that follows. The w weights allow each of the n inputs (denoted by x_i) to contribute a greater or lesser amount to the sum of input signals. The net total is used by the activation function $f(x)$ and the resulting signal, $y(x)$, is the output axon:

$$y(x) = f\left(\sum_{i=1}^n w_i x_i\right)$$

Neural networks use neurons defined in this way as building blocks to construct complex models of data. Although there are numerous variants of neural networks, each can be defined in terms of the following characteristics:

- An **activation function**, which transforms a neuron's net input signal into a single output signal to be broadcasted further in the network
- A **network topology** (or architecture), which describes the number of neurons in the model, as well as the number of layers and the manner in which they are connected
- The **training algorithm**, which specifies how connection weights are set to inhibit or excite neurons in proportion to the input signal

Let's take a look at some of the variations within each of these categories to see how they can be used to construct typical neural network models.

Activation functions

The **activation function** is the mechanism by which the artificial neuron processes incoming information and determines whether to pass the signal to other neurons in the network. Just as an artificial neuron is modeled after the biological version, so too is the activation function modeled after nature's design.

In the biological case, the activation function could be imagined as a process that involves summing the total input signal and determining whether it meets the firing threshold. If so, the neuron passes on the signal; otherwise, it does nothing. In ANN terms, this is known as a **threshold activation function**, as it results in an output signal only once a specified input threshold has been attained.

The following figure depicts a typical threshold function; in this case, the neuron fires when the sum of input signals is at least zero. Because its shape resembles a stair, it is sometimes called a **unit step activation function**.

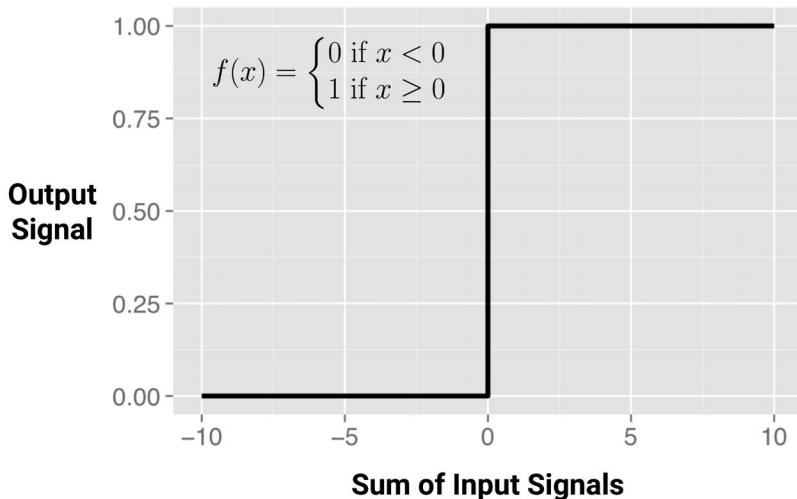


Figure 7.3: The threshold activation function is “on” only after the input signals meet a threshold

Although the threshold activation function is interesting due to its parallels with biology, it is rarely used in ANNs. Freed from the limitations of biochemistry, ANN activation functions can be chosen based on their ability to demonstrate desirable mathematical characteristics and their ability to accurately model relationships among data.

Perhaps the most common alternative is the **sigmoid activation function** (more specifically the *logistic* sigmoid) shown in the following figure. Note that in the formula shown, e is the base of the natural logarithm (a value of approximately 2.72). Although it shares a similar step or “S” shape with the threshold activation function, the output signal is no longer binary; output values can fall anywhere in the range from zero to one. Additionally, the sigmoid is **differentiable**, which means that it is possible to calculate the derivative (the slope of the tangent line for a point on the curve) across the entire range of inputs. As you will learn later, this feature is crucial for creating efficient ANN optimization algorithms.

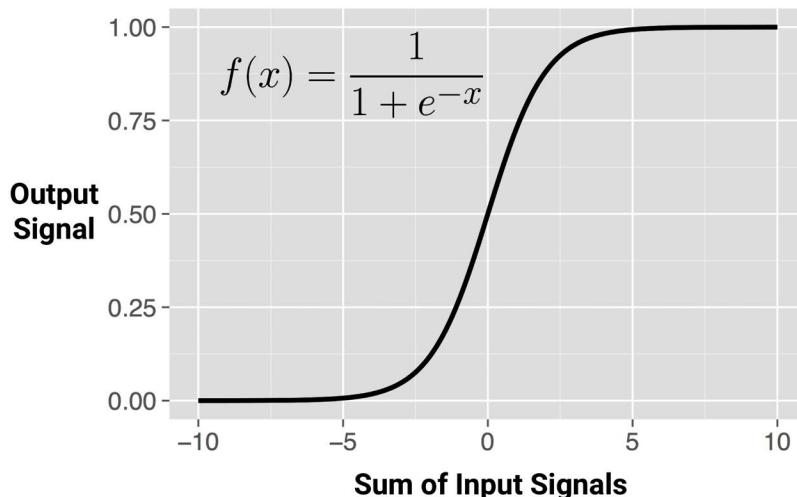


Figure 7.4: The sigmoid activation function uses a smooth curve to mimic the unit step activation function found in nature

Although the sigmoid is perhaps the most commonly used activation function and is often used by default, some neural network algorithms allow a choice of alternatives. A selection of such activation functions is shown in *Figure 7.5*:

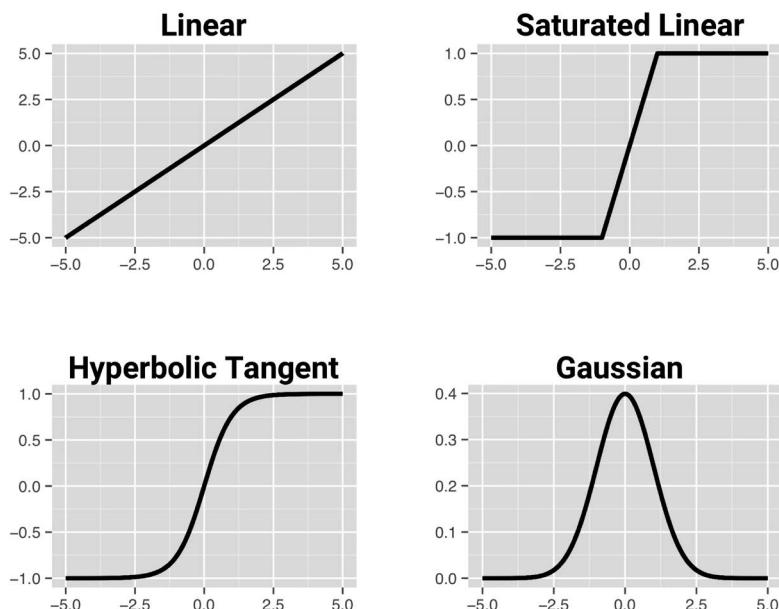


Figure 7.5: Several common neural network activation functions

The primary detail that differentiates these activation functions is the output signal range. Typically, this is one of $(0, 1)$, $(-1, +1)$, or $(-\infty, +\infty)$. The choice of activation function biases the neural network such that it may fit certain types of data more appropriately, allowing the construction of specialized neural networks. For instance, a linear activation function results in a neural network very similar to a linear regression model, while a Gaussian activation function is the basis of a **radial basis function (RBF)** network. Each of these has strengths better suited for certain learning tasks and not others.



Neural networks use nonlinear activation functions almost exclusively since this is what allows the network to become more intelligent as more nodes are added. Limited to linear activation functions only, a network is limited to linear solutions and will perform no better than the much simpler regression methods.

It's important to recognize that for many of the activation functions, the range of input values that affect the output signal is relatively narrow. For example, in the case of the sigmoid, the output signal is very near 0 for an input signal below -5 and very near 1 for an input signal above 5. The compression of the signal in this way results in a saturated signal at the high and low ends of very dynamic inputs, just as turning a guitar amplifier up too high results in a distorted sound due to clipping of the peaks of sound waves. Because this essentially squeezes the input values into a smaller range of outputs, activation functions like the sigmoid are sometimes called **squashing functions**.

One solution to the squashing problem is to transform all neural network inputs such that the feature values fall within a small range around zero. This may involve standardizing or normalizing the features. By restricting the range of input values, the activation function will be able to work across the entire range. A side benefit is that the model may also be faster to train since the algorithm can iterate more quickly through the actionable range of input values.



Although theoretically, a neural network can adapt to a very dynamic feature by adjusting its weight over many iterations, in extreme cases, many algorithms will stop iterating long before this occurs. If your model is failing to converge, double-check that you've correctly standardized the input data. Choosing a different activation function may also be appropriate.

Network topology

The capacity of a neural network to learn is rooted in its topology, or the patterns and structures of interconnected neurons. Although there are countless forms of network architecture, they can be differentiated by three key characteristics:

- The number of layers
- Whether information in the network is allowed to travel backward
- The number of nodes within each layer of the network

The topology determines the complexity of tasks that can be learned by the network. Generally, larger and more complex networks can identify more subtle patterns and more complex decision boundaries. However, the power of a network is not only a function of the network size but also the way units are arranged.

The number of layers

To define topology, we need terminology that distinguishes artificial neurons based on their position in the network. *Figure 7.6* illustrates the topology of a very simple network. A set of neurons called **input nodes** receives unprocessed signals directly from the input data. Each input node is responsible for processing a single feature in the dataset; the feature's value will be transformed by the corresponding node's activation function. The signals sent by the input nodes are received by the **output node**, which uses its own activation function to generate a final prediction (denoted here as p).

The input and output nodes are arranged in groups known as **layers**. Because the input nodes process the incoming data exactly as received, the network has only one set of connection weights (labeled here as w_1 , w_2 , and w_3). It is therefore termed a **single-layer network**. Single-layer networks can be used for basic pattern classification, particularly for patterns that are linearly separable, but more sophisticated networks are required for most learning tasks.

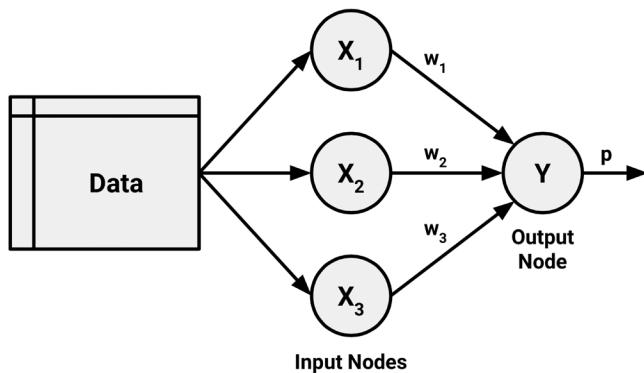


Figure 7.6: A simple single-layer ANN with three input nodes

As you might expect, an obvious way to create more complex networks is by adding additional layers. As depicted in *Figure 7.7*, a **multilayer network** adds one or more **hidden layers** that process the signals from the input nodes prior to reaching the output node. Hidden nodes get their name from the fact that they are obscured in the heart of the network and their relationship to the data and output is much more difficult to understand. The hidden layers are what make ANNs a black-box model; knowing what is happening inside these layers is practically impossible, particularly as the topology becomes more complicated.

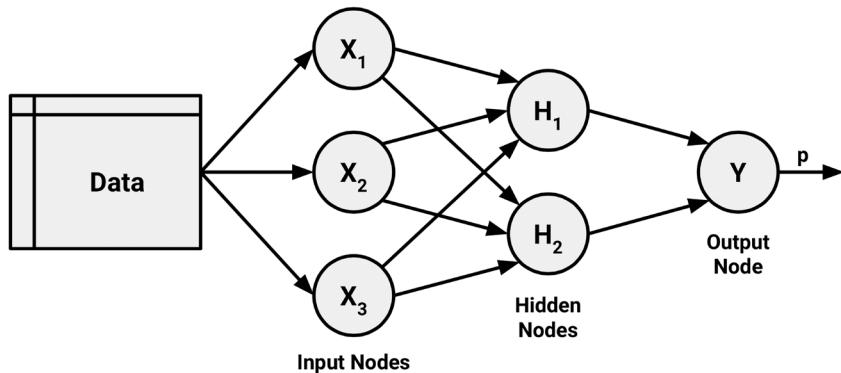


Figure 7.7: A multilayer network with a single two-node hidden layer

Examples of more complex topologies are depicted in *Figure 7.8*. Multiple output nodes can be used to represent outcomes with multiple categories. Multiple hidden layers can be used to allow even more complexity inside the black box, and thus model more challenging problems.

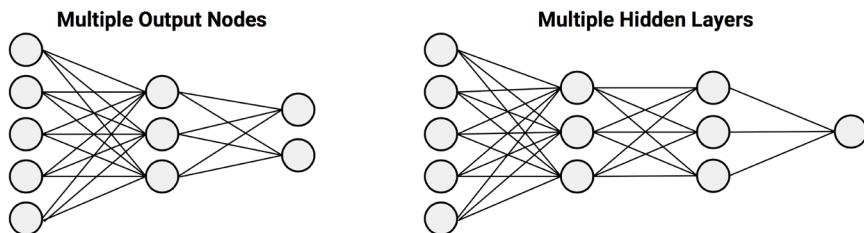


Figure 7.8: Complex ANNs can have multiple output nodes or multiple hidden layers

A neural network with multiple hidden layers is called a **deep neural network (DNN)**, and the practice of training such networks is referred to as **deep learning**. DNNs trained on large datasets are capable of human-like performance on complex tasks like image recognition and text processing. Deep learning has thus been hyped as the next big leap in machine learning, but deep learning is better suited to some tasks than others.

Though deep learning performs quite well on complex learning tasks that conventional models tend to struggle with, it requires much larger datasets with much richer feature sets than found in most projects. Typical learning tasks include modeling unstructured data like images, audio, or text, as well as outcomes measured repeatedly over time, such as stock market prices or energy consumption. Building DNNs on these types of data requires specialized computing software (and sometimes also hardware) that is more challenging to use than a simple R package. *Chapter 15, Making Use of Big Data*, provides details on how to use these tools to perform deep learning and image recognition in R.

The direction of information travel

Simple multilayer networks are usually **fully connected**, which means that every node in one layer is connected to every node in the next layer, but this is not required. Much larger deep learning models, such as the **convolutional neural network (CNN)** for image recognition that will be introduced in *Chapter 15, Making Use of Big Data*, are only partially connected. Removing some connections helps limit the amount of overfitting that can occur inside the numerous hidden layers. However, this is not the only way we can manipulate the topology. In addition to whether nodes are connected at all, we can also dictate the direction of information flow throughout the connections and produce neural networks suited to different types of learning tasks.

You may have noticed that in the prior examples, arrowheads were used to indicate signals traveling in only one direction. Networks in which the input signal is fed continuously in one direction from the input layer to the output layer are called **feedforward networks**. Despite the restriction on information flow, feedforward networks offer a surprising amount of flexibility.

For instance, the number of levels and nodes at each level can be varied, multiple outcomes can be modeled simultaneously, or multiple hidden layers can be applied.

In contrast to feedforward networks, a **recurrent neural network (RNN)** (or **feedback network**) allows signals to travel backward using loops. This property, which more closely mirrors how a biological neural network works, allows extremely complex patterns to be learned. The addition of a short-term memory, or **delay**, increases the power of recurrent networks immensely. Notably, this includes the capability to understand sequences of events over time. This could be used for stock market prediction, speech comprehension, or weather forecasting. A simple recurrent network is depicted as follows:

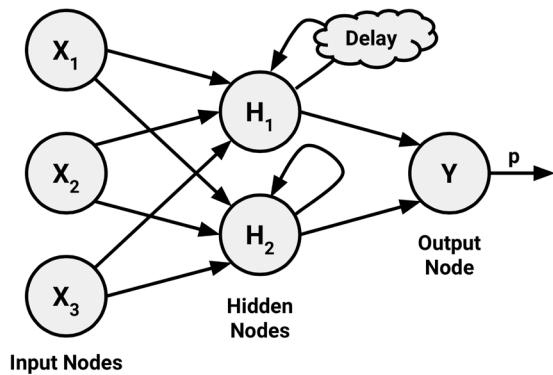


Figure 7.9: Allowing information to travel backward in the network can model a time delay

As the short-term memory of an RNN is, unsurprisingly, short by definition, one form of RNN known as **long short-term memory (LSTM)** adapts the model to have substantially longer recall—much like living creatures that have both short- and long-term memory. While this may seem to be an obvious enhancement, computers have perfect memory and thus need to be explicitly told when to forget and when to remember. The challenge was striking a balance between forgetting too soon and remembering for too long, which is easier said than done because the mathematical functions used to train the model are naturally pulled toward either of these extremes for reasons that will become clearer as you continue in this chapter.



For more information about LSTM neural networks, see *Understanding LSTM—a tutorial into Long Short-Term Memory Recurrent Neural Networks*, Staudemeyer RC and Morris ER, 2019. <https://arxiv.org/abs/1909.09586>.

The development of LSTM neural networks has led to advances in artificial intelligence such as the ability for robots to mimic sequences of human behaviors necessary for controlling machinery, driving, and playing video games. The LSTM model also shows aptitude for speech and text recognition, understanding language semantics and translating between languages, and learning complex strategies. DNNs and recurrent networks are increasingly being used for a variety of high-profile applications and consequently have become much more popular since the first edition of this book was published. However, building such networks uses techniques and software outside the scope of this book, and often requires access to specialized computing hardware or cloud servers.

On the other hand, simpler feedforward networks are also very capable of modeling many real-world tasks, though the tasks may not be quite as exciting as autonomous vehicles and computers playing video games. While deep learning is quickly becoming the status quo, the multilayer feedforward network, also known as the **multilayer perceptron (MLP)**, may still be the de facto standard ANN topology for conventional learning tasks. Furthermore, understanding the MLP topology provides a strong theoretical basis for building more complex deep learning models later.

The number of nodes in each layer

In addition to variations in the number of layers and the direction of information travel, neural networks can also vary in complexity by the number of nodes in each layer. The number of input nodes is predetermined by the number of features in the input data. Similarly, the number of output nodes is predetermined by the number of outcomes to be modeled or the number of class levels in the outcome. However, the number of hidden nodes is left to the user to decide prior to training the model.

Unfortunately, there is no reliable rule for determining the number of neurons in the hidden layer. The appropriate number depends on the number of input nodes, the amount of training data, the amount of noisy data, and the complexity of the learning task, among many other factors.

In general, more complex network topologies with a greater number of network connections allow the learning of more complex problems. A greater number of neurons will result in a model that more closely mirrors the training data, but this runs a risk of overfitting; it may generalize poorly to future data. Large neural networks can also be computationally expensive and slow to train. The best practice is to use the fewest nodes that result in adequate performance on a validation dataset. In most cases, even with only a small number of hidden nodes—often as few as a handful—the neural network can demonstrate a tremendous ability to learn.

It has been proven that a neural network with at least one hidden layer of sufficiently many neurons with nonlinear activation functions is a **universal function approximator**. This means that neural networks can be used to approximate any continuous function to an arbitrary level of precision over a finite interval. This is where a neural network gains the ability to perform the type of “magic” described in the chapter introduction; put a set of inputs into the black box of a neural network and it can learn to produce any set of outputs, no matter how complex the relationships are between the inputs and outputs. Of course, this assumes “sufficiently many neurons” as well as enough data to reasonably train the network—all while avoiding overfitting to noise so that the approximation will generalize beyond the training examples. We’ll peek further into the black box that allows this magic to happen in the next section.



To see a real-time visualization of how changes to network topology allow neural networks to become a universal function approximator, visit the Deep Learning Playground at <http://playground.tensorflow.org/>. The playground allows you to experiment with predictive models where the relationship between the features and target is complex and nonlinear. While methods like regression and decision trees would struggle to find any solution to these problems, you will find that adding more hidden nodes and layers allows the network to come up with a reasonable approximation for each of the examples given enough training time. Note, especially, that choosing the linear activation function rather than a sigmoid or hyperbolic tangent (\tanh) prevents the network from learning a reasonable solution regardless of the complexity of the network topology.

Training neural networks with backpropagation

The network topology is a blank slate that by itself has not learned anything. Like a newborn child, it must be trained with experience. As the neural network processes the input data, connections between the neurons are strengthened or weakened, similar to how a baby’s brain develops as they experience the environment. The network’s connection weights are adjusted to reflect the patterns observed over time.

Training a neural network by adjusting connection weights is very computationally intensive. Consequently, though they had been studied for decades prior, ANNs were rarely applied to real-world learning tasks until the mid-to-late 1980s, when an efficient method of training an ANN was discovered. The algorithm, which used a strategy of back-propagating errors, is now known simply as **backpropagation**.



Coincidentally, several research teams independently discovered and published the backpropagation algorithm around the same time. Among them, perhaps the most often cited work is *Learning representations by back-propagating errors*, Rumelhart, DE, Hinton, GE, Williams, RJ, *Nature*, 1986, Vol. 323, pp. 533-566.

Although still somewhat computationally expensive relative to many other machine learning algorithms, the backpropagation method led to a resurgence of interest in ANNs. As a result, multilayer feedforward networks that use the backpropagation algorithm are now common in the field of data mining. Such models offer the following strengths and weaknesses:

Strengths	Weaknesses
<ul style="list-style-type: none"> • Can be adapted to classification or numeric prediction problems • A “universal approximator” capable of modeling more complex patterns than nearly any algorithm • Makes few assumptions about the data’s underlying relationships 	<ul style="list-style-type: none"> • Extremely computationally intensive and slow to train, particularly if the network topology is complex • Very prone to overfitting training data, leading to poor generalization • Results in a complex black-box model that is difficult, if not impossible, to interpret

In its most general form, the backpropagation algorithm iterates through many cycles of two processes. Each cycle is known as an **epoch**. Because the network contains no *a priori* (existing) knowledge, the starting weights are typically set at random. Then, the algorithm iterates through the processes until a stopping criterion is reached. Each epoch in the backpropagation algorithm includes:

- A **forward phase**, in which the neurons are activated in sequence from the input layer to the output layer, applying each neuron’s weights and activation function along the way. Upon reaching the final layer, an output signal is produced.
- A **backward phase**, in which the network’s output signal resulting from the forward phase is compared to the true target value in the training data. The difference between the network’s output signal and the true value results in an error that is propagated backward in the network to modify the connection weights between neurons and reduce future errors.

Over time, the algorithm uses the information sent backward to reduce the total errors made by the network. Yet one question remains: because the relationship between each neuron’s inputs and outputs is complex, how does the algorithm determine how much a weight should be changed?

The answer to this question involves a technique called **gradient descent**. Conceptually, this works similarly to how an explorer trapped in the jungle might find a path to water. By examining the terrain and continually walking in the direction with the greatest downward slope, the explorer will eventually reach the lowest valley, which is likely to be a riverbed.

In a similar process, the backpropagation algorithm uses the derivative of each neuron's activation function to identify the gradient in the direction of each of the incoming weights—hence the importance of having a differentiable activation function. The gradient suggests how steeply the error will be reduced or increased for a change in the weight. The algorithm will attempt to change the weights that result in the greatest reduction in error by an amount known as the **learning rate**. The greater the learning rate, the faster the algorithm will attempt to descend down the gradients, which could reduce the training time at the risk of overshooting the valley.

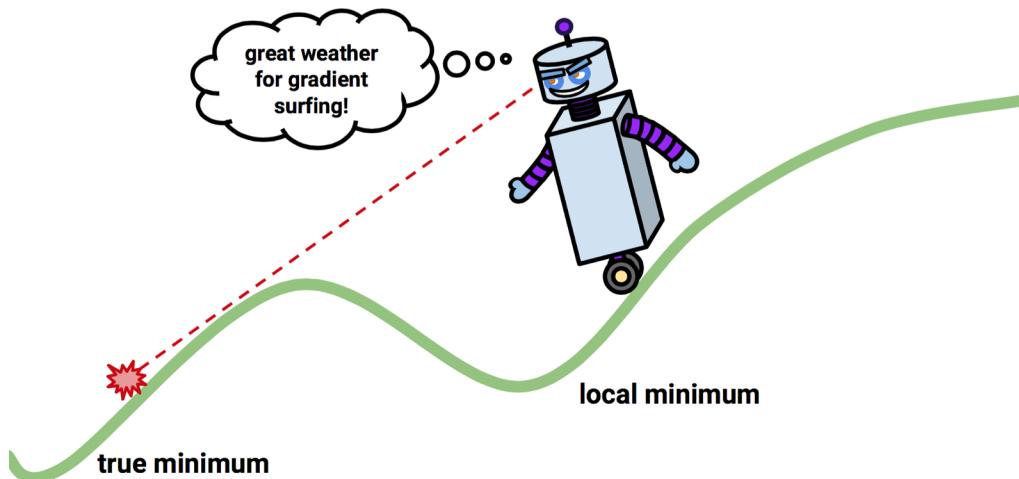


Figure 7.10: The gradient descent algorithm seeks the minimum error but may also find a local minimum

Although the math needed to find the minimum error rate using gradient descent is complex and therefore outside the scope of this book, it is easy to apply in practice via its implementation in neural network algorithms in R. Let's apply our understanding of multilayer feedforward networks to a real-world problem.

Example – modeling the strength of concrete with ANNs

In the field of engineering, it is crucial to have accurate estimates of the performance of building materials. These estimates are required in order to develop safety guidelines governing the materials used in the construction of buildings, bridges, and roadways.

Estimating the strength of concrete is a challenge of particular interest. Although it is used in nearly every construction project, concrete performance varies greatly due to a wide variety of ingredients that interact in complex ways. As a result, it is difficult to accurately predict the strength of the final product. A model that could reliably predict concrete strength given a listing of the composition of the input materials could result in safer construction practices.

Step 1 – collecting data

For this analysis, we will utilize data on the compressive strength of concrete donated to the UCI Machine Learning Repository (<http://archive.ics.uci.edu/ml>) by I-Cheng Yeh. As he found success using neural networks to model this data, we will attempt to replicate Yeh's work using a simple neural network model in R.



For more information on Yeh's approach to this learning task, refer to *Modeling of Strength of High-Performance Concrete Using Artificial Neural Networks*, Yeh, IC, Cement and Concrete Research, 1998, Vol. 28, pp. 1797-1808.

According to the website, the dataset contains 1,030 examples of concrete, with 8 features describing the components used in the mixture. These features are thought to be related to the final compressive strength and include the amount (in kilograms per cubic meter) of cement, slag, ash, water, superplasticizer, coarse aggregate, and fine aggregate used in the product, in addition to the aging time (measured in days).



To follow along with this example, download the `concrete.csv` file from the Packt Publishing website and save it to your R working directory.

Step 2 – exploring and preparing the data

As usual, we'll begin our analysis by loading the data into an R object using the `read.csv()` function and confirming that it matches the expected structure:

```
> concrete <- read.csv("concrete.csv")
> str(concrete)

'data.frame': 1030 obs. of  9 variables:
 $ cement      : num  141 169 250 266 155 ...
 $ slag        : num  212 42.2 0 114 183.4 ...
 $ ash         : num  0 124.3 95.7 0 0 ...
 $ water       : num  204 158 187 228 193 ...
 $ superplastic: num  0 10.8 5.5 0 9.1 0 0 6.4 0 9 ...
 $ coarseagg   : num  972 1081 957 932 1047 ...
 $ fineagg     : num  748 796 861 670 697 ...
 $ age         : int  28 14 28 28 28 90 7 56 28 28 ...
 $ strength    : num  29.9 23.5 29.2 45.9 18.3 ...
```

The nine variables in the data frame correspond to the eight features and one outcome we expected, although a problem has become apparent. Neural networks work best when the input data is scaled to a narrow range around zero, and here we see values ranging anywhere from zero to over a thousand.

Typically, the solution to this problem is to rescale the data with a normalizing or standardization function. If the data follows a bell-shaped curve (a normal distribution, as described in *Chapter 2, Managing and Understanding Data*), then it may make sense to use standardization via R's built-in `scale()` function. On the other hand, if the data follows a uniform distribution or is severely non-normal, then normalization to a zero-to-one range may be more appropriate. In this case, we'll use the latter.

In *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, we defined our own `normalize()` function as:

```
> normalize <- function(x) {
  return((x - min(x)) / (max(x) - min(x)))
}
```

After executing this code, our `normalize()` function can be applied to every column in the concrete data frame using the `lapply()` function as follows:

```
> concrete_norm <- as.data.frame(lapply(concrete, normalize))
```

To confirm that the normalization worked, we can see that the minimum and maximum strength are now zero and one, respectively:

```
> summary(concrete_norm$strength)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.0000	0.2664	0.4001	0.4172	0.5457	1.0000

In comparison, the original minimum and maximum values were 2.33 and 82.60:

```
> summary(concrete$strength)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
2.33	23.71	34.45	35.82	46.13	82.60



Any transformation applied to the data prior to training the model must be applied in reverse to convert it back into the original units of measurement. To facilitate the rescaling, it is wise to save the original data, or at least the summary statistics of the original data.

Following Yeh's precedent in the original publication, we will partition the data into a training set with 75 percent of the examples and a testing set with 25 percent. The CSV file we used was already sorted in random order, so we simply need to divide it into two portions:

```
> concrete_train <- concrete_norm[1:773, ]  
> concrete_test <- concrete_norm[774:1030, ]
```

We'll use the training dataset to build the neural network and the testing dataset to evaluate how well the model generalizes to future results. As it is easy to overfit a neural network, this step is very important.

Step 3 – training a model on the data

To model the relationship between the ingredients used in concrete and the strength of the finished product, we will use a multilayer feedforward neural network. The `neuralnet` package by Stefan Fritsch and Frauke Guenther provides a standard and easy-to-use implementation of such networks. It also offers a function to plot the network topology. For these reasons, the `neuralnet` implementation is a strong choice for learning more about neural networks, though that's not to say that it cannot be used to accomplish real work as well—it's quite a powerful tool, as you will soon see.



There are several other commonly used packages to train simple ANN models in R, each with unique strengths and weaknesses. Because it ships as part of the standard R installation, the `nnet` package is perhaps the most frequently cited ANN implementation. It uses a slightly more sophisticated algorithm than standard backpropagation. Another option is the `RSNNS` package, which offers a complete suite of neural network functionality, with the downside being that it is more difficult to learn. The specialized software for building or using deep learning neural networks is covered in *Chapter 15, Making Use of Big Data*.

As `neuralnet` is not included in base R, you will need to install it by typing `install.packages("neuralnet")` and load it with the `library(neuralnet)` command. The included `neuralnet()` function can be used for training neural networks for numeric prediction using the following syntax:

Neural network syntax

using the **neuralnet()** function in the **neuralnet** package

Building the model:

```
m <- neuralnet(target ~ predictors, data = mydata,  
                 hidden = 1, act.fct = "logistic")
```

- **target** is the outcome in the **mydata** data frame to be modeled
- **predictors** is an R formula specifying the features in the **mydata** data frame to use for prediction
- **data** specifies the data frame where the **target** and **predictors** are found
- **hidden** specifies the number of neurons in the hidden layer (by default, 1)
note: use an integer vector to specify multiple hidden layers, e.g., **c(2, 2)**
- **act.fct** specifies the activation function, either "**logistic**" or "**tanh**"
note: a *differentiable* custom activation function can also be supplied

The function will return a neural network object that can be used to make predictions.

Making predictions:

```
p <- compute(m, test)
```

- **m** is a model trained by the **neuralnet()** function
- **test** is a data frame containing test data with the same features as the training data used to build the classifier

The function will return a list with two components: **\$neurons**, which stores the neurons for each layer in the network, and **\$net.result**, which stores the model's predicted values.

Example:

```
concrete_model <- neuralnet(strength ~ cement + slag + ash,  
                           data = concrete, hidden = c(5, 5), act.fct = "tanh")  
model_results <- compute(concrete_model, concrete_data)  
strength_predictions <- model_results$net.result
```

Figure 7.11: Neural network syntax

We'll begin by training the simplest multilayer feedforward network with the default settings using only a single hidden node. Because the process of training an ANN involves randomization, the `set.seed()` function used here will ensure the same result is produced when the `neuralnet()` function is run:

```
> set.seed(12345)
> concrete_model <- neuralnet(strength ~ cement + slag +
+ ash + water + superplastic + coarseagg + fineagg + age,
+ data = concrete_train)
```

We can then visualize the network topology using the `plot()` function on the resulting model object:

```
> plot(concrete_model)
```

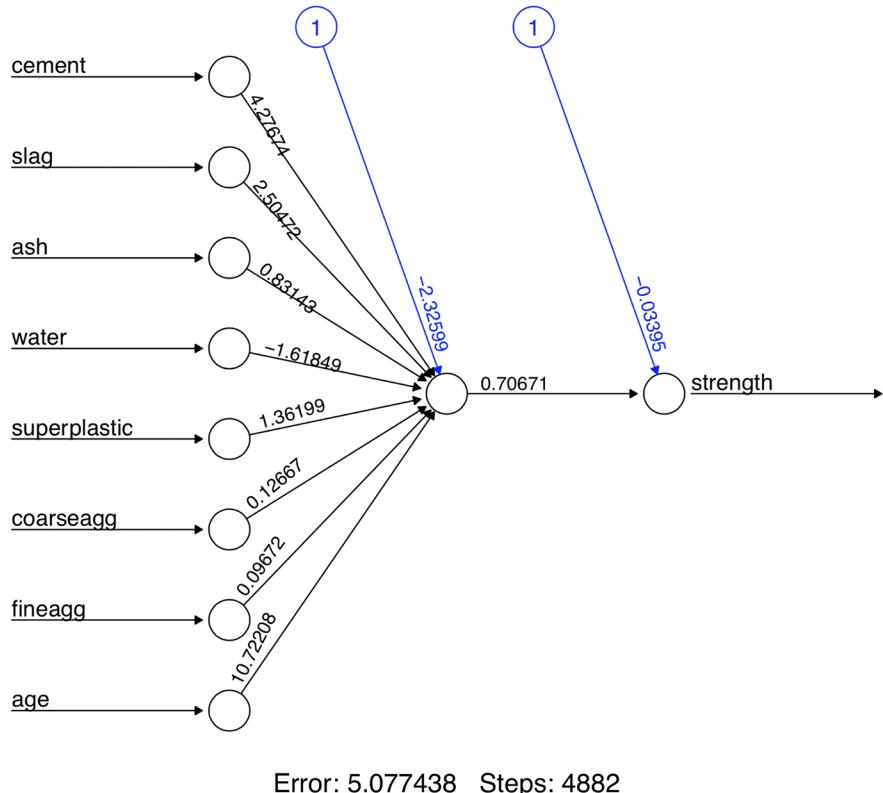


Figure 7.12: Topology visualization of a simple multilayer feedforward network

In this simple model, there is one input node for each of the eight features, followed by a single hidden node and a single output node that predicts the concrete strength. The weights for each of the connections are also depicted, as are the **bias terms** indicated by the nodes labeled with the number 1. The bias terms are numeric constants that allow the value at the indicated nodes to be shifted upward or downward, much like the intercept in a linear equation. In a linear equation of the form $y = ax + b$, the intercept b allows y to have a value other than 0 when $x = 0$. Similarly, the bias terms in a neural network allow the node to pass a value other than zero when the inputs are zero. This gives more flexibility for learning the true patterns found in the data, which in turn helps the model fit better. In the specific case of our concrete model, even though it isn't feasible in the real world to have a case where all inputs to the concrete such as cement, age, and water are all zero, we wouldn't necessarily expect strength to cross the origin exactly at zero even as the values of these factors approach zero. We might expect a model of body weight versus age to have a bias term above zero because the weight at birth ($age = 0$) is greater than zero.



A neural network with a single hidden node can be thought of as a cousin of the linear regression models we studied in *Chapter 6, Forecasting Numeric Data – Regression Methods*. The weight between each input node and the hidden node is similar to the beta coefficients, and the weight for the bias term is similar to the intercept. If a linear activation function is used, the neural network is almost exactly linear regression. A key difference, however, is that the ANN is trained using gradient descent while linear regression typically uses the least squares approach.

At the bottom of the figure, R reports the number of training steps and an error measure called the **sum of squared errors (SSE)**, which, as you might expect, is the sum of the squared differences between the predicted and actual values. The lower the SSE, the more closely the model conforms to the training data, which tells us about performance on the training data but little about how it will perform on unseen data.

Step 4 – evaluating model performance

The network topology diagram gives us a peek into the black box of the ANN, but it doesn't provide much information about how well the model fits future data. To generate predictions on the test dataset, we can use the `compute()` function as follows:

```
> model_results <- compute(concrete_model, concrete_test[1:8])
```

The `compute()` function works a bit differently from the `predict()` functions we've used so far. It returns a list with two components: `$neurons`, which stores the neurons for each layer in the network, and `$net.result`, which stores the predicted values. We'll want the latter:

```
> predicted_strength <- model_results$net.result
```

Because this is a numeric prediction problem rather than a classification problem, we cannot use a confusion matrix to examine model accuracy. Instead, we'll measure the correlation between our predicted concrete strength and the true value. If the predicted and actual values are highly correlated, the model is likely to be a useful gauge of concrete strength.

Recall that the `cor()` function is used to obtain a correlation between two numeric vectors:

```
> cor(predicted_strength, concrete_test$strength)
```

```
[,1]
[1,] 0.8064656
```

Correlations close to one indicate strong linear relationships between two variables. Therefore, the correlation here of about 0.806 indicates a fairly strong relationship. This implies that our model is doing a fairly good job, even with only a single hidden node. Yet, given that we only used one hidden node, it is likely that we can improve the performance of our model. Let's try to do a bit better.

Step 5 – improving model performance

As networks with more complex topologies are capable of learning more difficult concepts, let's see what happens when we increase the number of hidden nodes to five. We use the `neuralnet()` function as before, but add the parameter `hidden = 5`. Note that due to the increased complexity of this neural network, depending on the capabilities of your computer, the new model may take 30 to 60 seconds to train:

```
> set.seed(12345)
> concrete_model2 <- neuralnet(strength ~ cement + slag +
+ ash + water + superplastic +
+ coarseagg + fineagg + age,
+ data = concrete_train, hidden = 5)
```

Plotting the network again, we see a drastic increase in the number of connections. How did this impact performance?

```
> plot(concrete_model2)
```

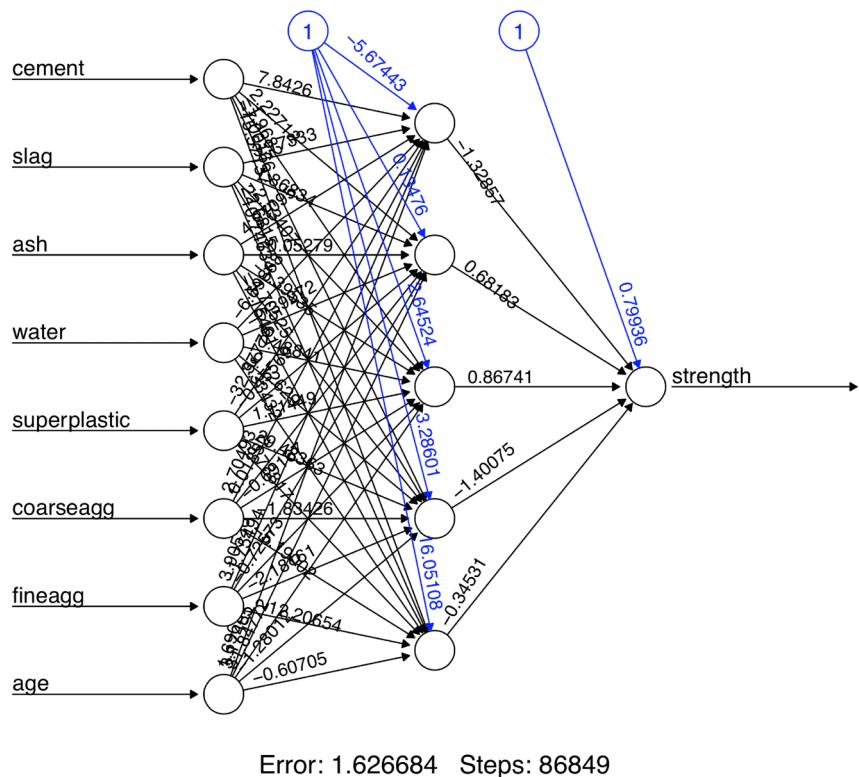


Figure 7.13: Topology visualization of a neural network with additional hidden nodes

Notice that the reported error (measured again by the SSE) has been reduced from 5.08 in the previous model to 1.63 here. Additionally, the number of training steps rose from 4,882 to 86,849, which should come as no surprise given how much more complex the model has become. More complex networks take many more iterations to find the optimal weights.

Applying the same steps to compare the predicted values to the true values, we now obtain a correlation of around 0.92, which is a considerable improvement over the previous result of 0.80 with a single hidden node:

```
> model_results2 <- compute(concrete_model2, concrete_test[1:8])
> predicted_strength2 <- model_results2$net.result
> cor(predicted_strength2, concrete_test$strength)
```

[,1]
[1,] 0.9244533426

Despite these substantial improvements, there is still more we can do to attempt to improve the model’s performance. In particular, we have the ability to add additional hidden layers and change the network’s activation function. In making these changes, we create the foundations of a very simple DNN.

The choice of activation function is usually very important for deep learning. The best function for a particular learning task is typically identified through experimentation, then shared more widely within the machine learning research community. As deep learning has become more studied, an activation function known as a **rectifier** has become extremely popular due to its success in complex tasks such as image recognition. A node in a neural network that uses the rectifier activation function is known as a **rectified linear unit (ReLU)**. As depicted in the following figure, the ReLU activation function is defined such that it returns x if x is at least zero, and zero otherwise. The popularity of this function for deep learning is due to the fact that it is nonlinear yet has simple mathematical properties that make it both computationally inexpensive and highly efficient for gradient descent. Unfortunately, its derivative is undefined at $x = 0$ and therefore cannot be used with the `neuralnet()` function.

Instead, we can use a smooth approximation of the ReLU known as **softplus** or **SmoothReLU**, an activation function defined as $\log(1 + e^x)$. As shown in the following figure, the softplus function is nearly zero for x less than zero and approximately x when x is greater than zero:

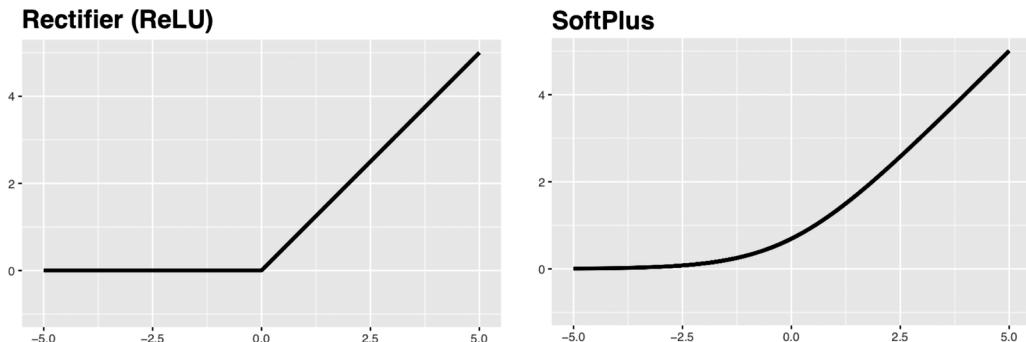


Figure 7.14: The softplus activation function provides a smooth, differentiable approximation of ReLU

To define a `softplus()` function in R, use the following code:

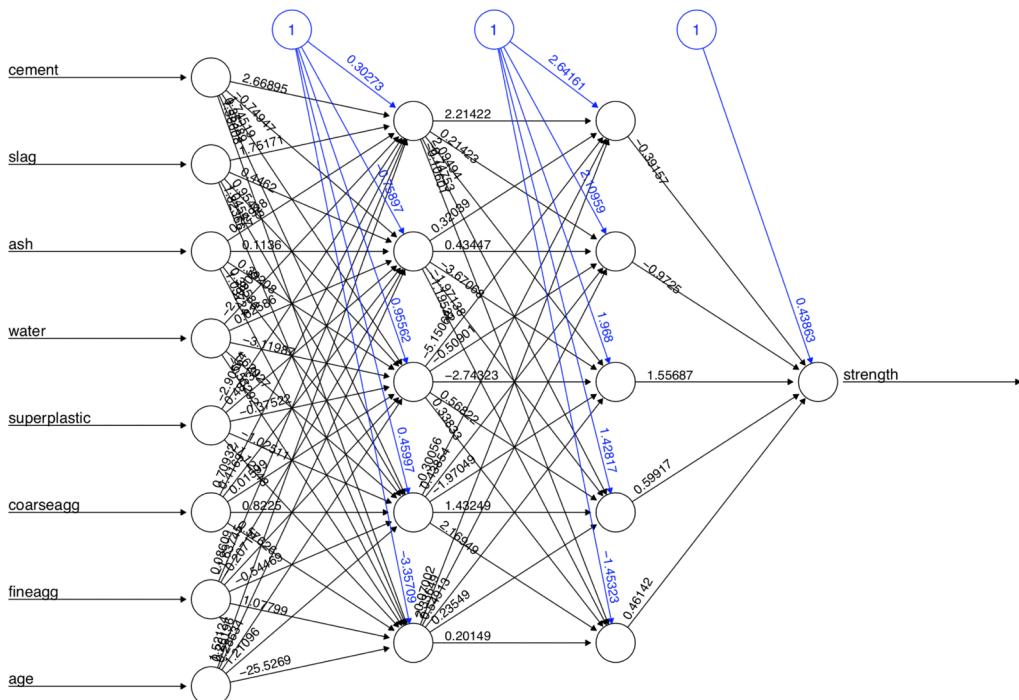
```
> softplus <- function(x) { log(1 + exp(x)) }
```

This activation function can be provided to `neuralnet()` using the `act.fct` parameter. Additionally, we will add a second hidden layer of five nodes by supplying the `hidden` parameter the integer vector `c(5, 5)`. This creates a two-layer network, each having five nodes, all using the softplus activation function. As before, this may take a minute or so to run:

```
> set.seed(12345)
> concrete_model3 <- neuralnet(strength ~ cement + slag +
+                                 ash + water + superplastic +
+                                 coarseagg + fineagg + age,
+                                 data = concrete_train,
+                                 hidden = c(5, 5),
+                                 act.fct = softplus)
```

The network visualization now shows a topology of two hidden layers of five nodes each:

```
> plot(concrete_model3)
```



Error: 1.666068 Steps: 88240

Figure 7.15: Visualizing our network with two layers of hidden nodes using the softplus activation function

Again, let's compute the correlation between the predicted and actual concrete strength:

```
> model_results3 <- compute(concrete_model3, concrete_test[1:8])
> predicted_strength3 <- model_results3$net.result
> cor(predicted_strength3, concrete_test$strength)

[,1]
[1,] 0.9348395359
```

The correlation between the predicted and actual strength was 0.935, which is our best performance yet. Interestingly, in the original publication, Yeh reported a correlation of 0.885. This means that with relatively little effort, we were able to match and even exceed the performance of a subject matter expert. Of course, Yeh's results were published in 1998, giving us the benefit of 25 years of additional neural network research!

One important thing to be aware of is that, because we had normalized the data prior to training the model, the predictions are also on a normalized scale from zero to one. For example, the following code shows a data frame comparing the original dataset's concrete strength values to their corresponding predictions side by side:

```
> strengths <- data.frame(
  actual = concrete$strength[774:1030],
  pred = predicted_strength3
)

> head(strengths, n = 3)
      actual      pred
774  30.14 0.2860639091
775  44.40 0.4777304648
776  24.50 0.2840964250
```

Using correlation as a performance metric, the choice of normalized or unnormalized data does not affect the result. For example, the correlation of 0.935 is the same whether the predicted strengths are compared to the original, unnormalized concrete strength values (`strengths$actual`) or to the normalized values (`concrete_test$strength`):

```
> cor(strengths$pred, strengths$actual)

[1] 0.9348395

> cor(strengths$pred, concrete_test$strength)
```

```
[1] 0.9348395
```

However, if we were to compute a different performance metric, such as the percentage difference between the predicted and actual values, the choice of scale would matter quite a bit.

With this in mind, we can create an `unnormalize()` function that reverses the min-max normalization procedure and allow us to convert the normalized predictions into the original scale:

```
> unnormalize <- function(x) {  
  return(x * (max(concrete$strength) -  
              min(concrete$strength)) + min(concrete$strength))  
}
```

After applying the custom `unnormalize()` function to the predictions, we can see that the new predictions (`pred_new`) are on a similar scale to the original concrete strength values. This allows us to compute a meaningful percentage error value. The resulting `error_pct` is the percentage difference between the true and predicted values:

```
> strengths$pred_new <- unnormalize(strengths$pred)  
> strengths$error_pct <- (strengths$pred_new - strengths$actual) /  
                           strengths$actual  
  
> head(strengths, n = 3)  
    actual      pred pred_new  error_pct  
774  30.14 0.2860639 25.29235 -0.16083776  
775  44.40 0.4777305 40.67742 -0.08384179  
776  24.50 0.2840964 25.13442 -0.02589470
```

Unsurprisingly, the correlation remains the same despite reversing the normalization:

```
> cor(strengths$pred_new, strengths$actual)
```

```
[1] 0.9348395
```

When applying neural networks to your own projects, you will need to perform a similar series of steps to return the data to its original scale.

You may also find that neural networks quickly become much more complicated as they are applied to more challenging learning tasks. For example, you may find that you run into the so-called **vanishing gradient problem** and closely related **exploding gradient problem**, where the backpropagation algorithm fails to find a useful solution due to an inability to converge in a reasonable amount of time.

As a remedy to these problems, one may perhaps try varying the number of hidden nodes, applying different activation functions such as the ReLU, adjusting the learning rate, and so on. The `?neuralnet` help page provides more information on the various parameters that can be adjusted. This leads to another problem, however, in which testing many parameters becomes a bottleneck to building a strong-performing model. This is the tradeoff of ANNs and, even more so, DNNs: harnessing their great potential requires a great investment of time and computing power.



Just as is often the case in life more generally, it is possible to trade time and money in machine learning. Using paid cloud computing resources such as **Amazon Web Services (AWS)** and Microsoft Azure allows one to build more complex models or test many models more quickly.

Understanding support vector machines

A **support vector machine (SVM)** can be imagined as a surface that creates a boundary between points of data plotted in a multidimensional space representing examples and their feature values. The goal of an SVM is to create a flat boundary called a **hyperplane**, which divides the space to create fairly homogeneous partitions on either side. In this way, SVM learning combines aspects of both the instance-based nearest neighbor learning presented in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, and the linear regression modeling described in *Chapter 6, Forecasting Numeric Data – Regression Methods*. The combination is extremely powerful, allowing SVMs to model highly complex relationships.

Although the basic mathematics that drive SVMs has been around for decades, interest in them grew greatly after they were adopted by the machine learning community. Their popularity exploded after high-profile success stories about difficult learning problems, as well as the development of award-winning SVM algorithms that were implemented in well-supported libraries across many programming languages, including R. SVMs have thus been adopted by a wide audience, which might have otherwise been unable to apply the somewhat complex mathematics needed to implement an SVM. The good news is that although the mathematics may be difficult, the basic concepts are understandable.

SVMs can be adapted for use with nearly any type of learning task, including both classification and numeric prediction. Many of the algorithm's key successes have been in pattern recognition. Notable applications include:

- Classification of microarray gene expression data in the field of bioinformatics to identify cancer or other genetic diseases

- Text categorization, such as identification of the language used in a document or classification of documents by subject matter
- The detection of rare yet important events like engine failure, security breaches, or earthquakes

SVMs are most easily understood when used for binary classification, which is how the method has been traditionally applied. Therefore, in the remaining sections, we will focus only on SVM classifiers. Principles like those presented here also apply when adapting SVMs for numeric prediction.

Classification with hyperplanes

As noted previously, SVMs use a boundary called a **hyperplane** to partition data into groups of similar class values. For example, the following figure depicts hyperplanes that separate groups of circles and squares in two and three dimensions. Because the circles and squares can be separated perfectly by a straight line or flat surface, they are said to be **linearly separable**. At first, we'll consider only a simple case where this is true, but SVMs can also be extended to problems where the points are not linearly separable.

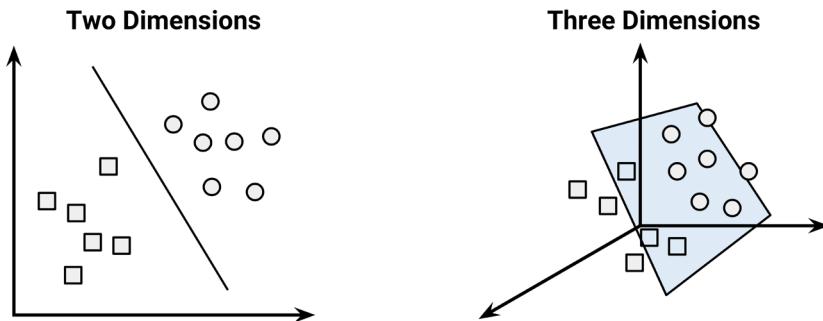


Figure 7.16: The squares and circles are linearly separable in both two and three dimensions



For convenience, the hyperplane is traditionally depicted as a line in 2D space, but this is simply because it is difficult to illustrate space in greater than 2 dimensions. In reality, the hyperplane is a flat surface in a high-dimensional space—a concept that can be difficult to get your mind around.

In two dimensions, the task of the SVM algorithm is to identify a line that separates the two classes. As shown in the following figure, there is more than one choice of dividing line between the groups of circles and squares. Three such possibilities are labeled *a*, *b*, and *c*. How does the algorithm choose?

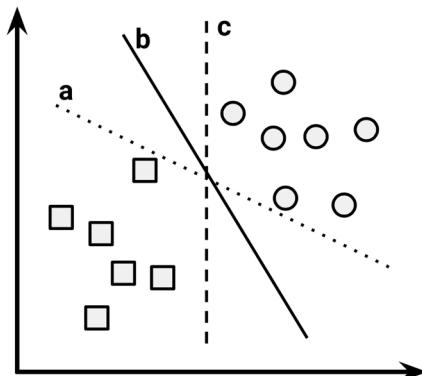


Figure 7.17: Three of many potential lines dividing the squares and circles

The answer to that question involves a search for the **maximum margin hyperplane (MMH)** that creates the greatest separation between the two classes. Although any of the three lines separating the circles and squares would correctly classify all the data points, the line that leads to the greatest separation is expected to generalize the best to future data. The maximum margin will improve the chance that even if random noise is added, each class will remain on its own side of the boundary.

The **support vectors** (indicated by arrows in the figure that follows) are the points from each class that are the closest to the MMH. Each class must have at least one support vector, but it is possible to have more than one. The support vectors alone define the MMH. This is a key feature of SVMs; the support vectors provide a very compact way to store a classification model, even if the number of features is extremely large.

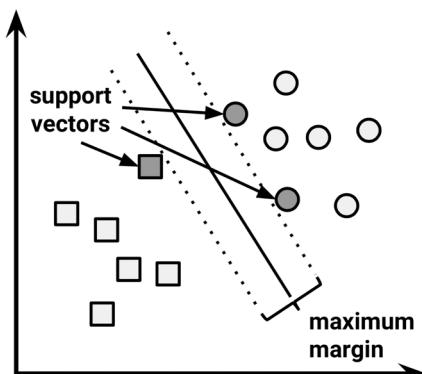


Figure 7.18: The MMH is defined by the support vectors

The algorithm to identify the support vectors relies on vector geometry and involves some tricky mathematics outside the scope of this book. However, the basic principles of the process are straightforward.



More information on the mathematics of SVMs can be found in the classic paper *Support-Vector Networks*, Cortes, C and Vapnik, V, *Machine Learning*, 1995, Vol. 20, pp. 273-297. A beginner-level discussion can be found in *Support Vector Machines: Hype or Hallelujah?*, Bennett, KP and Campbell, C, *SIGKDD Explorations*, 2000, Vol. 2, pp. 1-13. A more in-depth look can be found in *Support Vector Machines*, Steinwart, I and Christmann, A, New York: Springer, 2008.

The case of linearly separable data

Finding the maximum margin is easiest under the assumption that the classes are linearly separable. In this case, the MMH is as far away as possible from the outer boundaries of the two groups of data points. These outer boundaries are known as the **convex hull**. The MMH is then the perpendicular bisector of the shortest line between the two convex hulls. Sophisticated computer algorithms that use a technique known as **quadratic optimization** can find the maximum margin in this way.

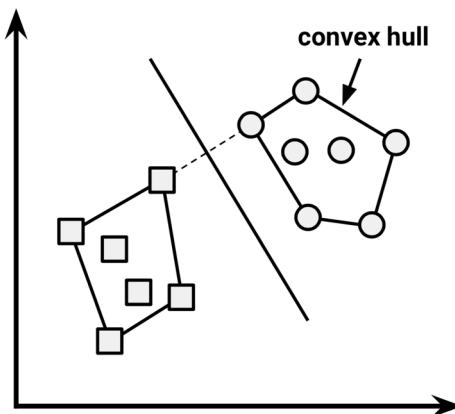


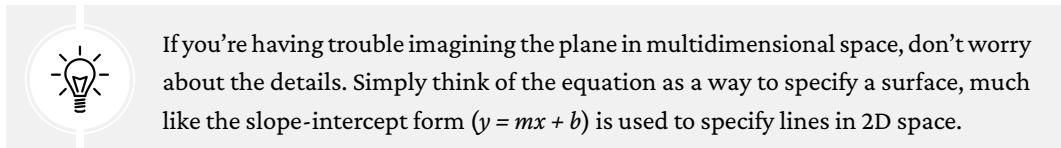
Figure 7.19: The MMH is the perpendicular bisector of the shortest path between convex hulls

An alternative (but equivalent) approach involves a search through the space of every possible hyperplane in order to find a set of two parallel planes that divide the points into homogeneous groups yet themselves are as far apart as possible. To use a metaphor, one can imagine this process as like trying to find the thickest mattress that can fit up a stairwell to your bedroom.

To understand this search process, we'll need to define exactly what we mean by a hyperplane. In n -dimensional space, the following equation is used:

$$\vec{w} \cdot \vec{x} + b = 0$$

If you aren't familiar with this notation, the arrows above the letters indicate that they are vectors rather than single numbers. In particular, w is a vector of n weights, that is, $\{w_1, w_2, \dots, w_n\}$, and b is a single number known as the bias. The bias is conceptually equivalent to the intercept term in the slope-intercept form discussed in *Chapter 6, Forecasting Numeric Data – Regression Methods*.



Using this formula, the goal of the process is to find a set of weights that specify two hyperplanes, as follows:

$$\vec{w} \cdot \vec{x} + b \geq +1$$

$$\vec{w} \cdot \vec{x} + b \leq -1$$

We will also require that these hyperplanes are specified such that all the points of one class fall above the first hyperplane and all the points of the other class fall beneath the second hyperplane. The two planes should create a gap such that there are no points from either class in the space between them. This is possible so long as the data is linearly separable. Vector geometry defines the distance between these two planes—the distance we want as large as possible—as:

$$\frac{2}{\|\vec{w}\|}$$

Here, $\|\vec{w}\|$ indicates the **Euclidean norm** (the distance from the origin to vector w). Because $\|\vec{w}\|$ is the denominator, to maximize distance, we need to minimize $\|\vec{w}\|$. The task is typically re-expressed as a set of constraints, as follows:

$$\min \frac{1}{2} \|\vec{w}\|^2$$

$$s.t. y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1, \forall \vec{x}_i$$

Although this looks messy, it's really not too complicated to understand conceptually. Basically, the first line implies that we need to minimize the Euclidean norm (squared and divided by two to make the calculation easier). The second line notes that this is subject to (*s.t.*) the condition that each of the y_i data points is correctly classified. Note that y indicates the class value (transformed into either +1 or -1) and the upside-down "A" is shorthand for "for all."

As with the other method for finding the maximum margin, finding a solution to this problem is a task best left for quadratic optimization software. Although it can be processor-intensive, specialized algorithms can solve these problems quickly even on large datasets.

The case of nonlinearly separable data

As we've worked through the theory behind SVMs, you may be wondering about the elephant in the room: what happens in a case where that the data is not linearly separable? The solution to this problem is the use of a **slack variable**, which creates a soft margin that allows some points to fall on the incorrect side of the margin. The figure that follows illustrates two points falling on the wrong side of the line with the corresponding slack terms (denoted by the Greek letter ξ_i):

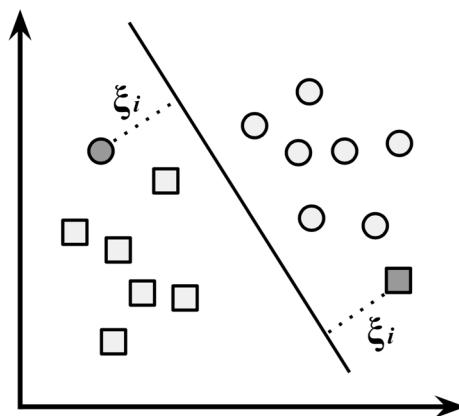


Figure 7.20: Points falling on the wrong side of the boundary come with a cost penalty

A cost value (denoted as C) is applied to all points that violate the constraints and rather than finding the maximum margin, the algorithm attempts to minimize the total cost. We can therefore revise the optimization problem to:

$$\min \frac{1}{2} \|\vec{w}\|^2 + C \sum_{i=1}^n \xi_i$$

$$s. t. \quad y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1 - \xi_i, \forall \vec{x}_i, \xi_i \geq 0$$

If you're confused by now, don't worry, you're not alone. Luckily, SVM packages will happily optimize this for you without you having to understand the technical details. The important piece to understand is the addition of the cost parameter, C . Modifying this value will adjust the penalty for points that fall on the wrong side of the hyperplane. The greater the cost parameter, the harder the optimization will try to achieve 100 percent separation. On the other hand, a lower cost parameter will place the emphasis on a wider overall margin. It is important to strike a balance between these two in order to create a model that generalizes well to future data.

Using kernels for nonlinear spaces

In many real-world datasets, the relationships between variables are nonlinear. As we just discovered, an SVM can still be trained on such data through the addition of a slack variable, which allows some examples to be misclassified. However, this is not the only way to approach the problem of nonlinearity. A key feature of SVMs is their ability to map the problem into a higher dimension space using a process known as the **kernel trick**. In doing so, a nonlinear relationship may suddenly appear to be quite linear.

Though this seems like nonsense, it is actually quite easy to illustrate with an example. In the following figure, the scatterplot on the left depicts a nonlinear relationship between a weather class (Sunny or Snowy) and two features: Latitude and Longitude. The points at the center of the plot are members of the Snowy class, while the points at the margins are all Sunny. Such data could have been generated from a set of weather reports, some of which were obtained from stations near the top of a mountain, while others were obtained from stations around the base of the mountain.

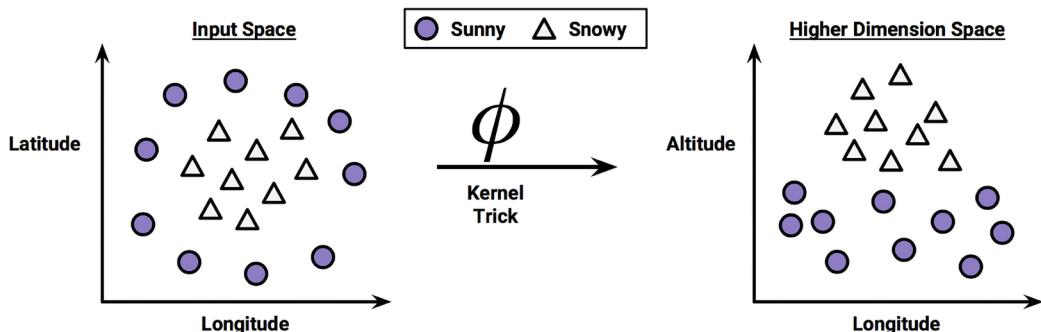


Figure 7.21: The kernel trick can help transform a nonlinear problem into a linear one

On the right side of the figure, after the kernel trick has been applied, we look at the data through the lens of a new dimension: Altitude. With the addition of this feature, the classes are now perfectly linearly separable. This is possible because we have obtained a new perspective on the data. In the left figure, we are viewing the mountain from a bird's-eye view, while on the right, we are viewing the mountain from a distance at ground level. Here, the trend is obvious: snowy weather is found at higher altitudes.

In this way, SVMs with nonlinear kernels add additional dimensions to the data in order to create separation. Essentially, the kernel trick involves a process of constructing new features that express mathematical relationships between measured characteristics. For instance, the Altitude feature can be expressed mathematically as an interaction between Latitude and Longitude—the closer the point is to the center of each of these scales, the greater the Altitude. This allows the SVM to learn concepts that were not explicitly measured in the original data.

SVMs with nonlinear kernels are extremely powerful classifiers, although they do have some downsides, as shown in the following table:

Strengths	Weaknesses
<ul style="list-style-type: none"> • Can be used for classification or numeric prediction problems • Not overly influenced by noisy data and not very prone to overfitting • May be easier to use than neural networks, particularly due to the existence of several well-supported SVM algorithms • Gained popularity due to their high accuracy and high-profile wins in data mining competitions 	<ul style="list-style-type: none"> • Finding the best model requires the testing of various combinations of kernels and model parameters • Can be slow to train, particularly if the input dataset has a large number of features or examples • Results in a complex black-box model that is difficult, if not impossible, to interpret

Kernel functions, in general, are of the following form. The function denoted by the Greek letter phi, that is, $\phi(x)$, is a mapping of the data into another space. Therefore, the general kernel function applies some transformation to the feature vectors x_i and x_j , and combines them using the **dot product**, which takes two vectors and returns a single number:

$$K(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$$

Using this form, kernel functions have been developed for many different domains. A few of the most commonly used kernel functions are listed as follows. Nearly all SVM software packages will include these kernels, among many others.

The **linear kernel** does not transform the data at all. Therefore, it can be expressed simply as the dot product of the features:

$$K(\vec{x}_i, \vec{x}_j) = \vec{x}_i \cdot \vec{x}_j$$

The **polynomial kernel** of degree d adds a simple nonlinear transformation of the data:

$$K(\vec{x}_i, \vec{x}_j) = (\vec{x}_i \cdot \vec{x}_j + 1)^d$$

The **sigmoid kernel** results in an SVM model somewhat analogous to a neural network using a sigmoid activation function. The Greek letters kappa and delta are used as kernel parameters:

$$K(\vec{x}_i, \vec{x}_j) = \tanh(\kappa \vec{x}_i \cdot \vec{x}_j - \delta)$$

The **Gaussian RBF kernel** is similar to an RBF neural network. The RBF kernel performs well on many types of data and is thought to be a reasonable starting point for many learning tasks:

$$K(\vec{x}_i, \vec{x}_j) = e^{\frac{-||\vec{x}_i - \vec{x}_j||^2}{2\sigma^2}}$$

There is no reliable rule for matching a kernel to a particular learning task. The fit depends heavily on the concept to be learned, as well as the amount of training data and the relationships between the features. Often, a bit of trial and error is required by training and evaluating several SVMs on a validation dataset. That said, in many cases, the choice of the kernel is arbitrary, as the performance may vary only slightly. To see how this works in practice, let's apply our understanding of SVM classification to a real-world problem.

Example – performing OCR with SVMs

Image processing is a difficult task for many types of machine learning algorithms. The relationships linking patterns of pixels to higher concepts are extremely complex and hard to define. For instance, it's easy for a human being to recognize a face, a cat, or the letter "A," but defining these patterns in strict rules is difficult. Furthermore, image data is often noisy. There can be many slight variations in how the image was captured depending on the lighting, orientation, and positioning of the subject.

SVMs are well suited to tackle the challenges of image data. Capable of learning complex patterns without being overly sensitive to noise, they can recognize visual patterns with a high degree of accuracy. Moreover, the key weakness of SVMs—the black-box model representation—is less critical for image processing. If an SVM can differentiate a cat from a dog, it does not matter much how it is doing so.

In this section, we will develop a model like those used at the core of the **optical character recognition (OCR)** software often bundled with desktop document scanners or smartphone applications. The purpose of such software is to process paper-based documents by converting printed or handwritten text into an electronic form to be saved in a database.

Of course, this is a difficult problem due to the many variants in handwriting style and printed fonts. Even so, software users expect perfection, as errors or typos can result in embarrassing or costly mistakes in a business environment. Let's see whether our SVM is up to the task.

Step 1 – collecting data

When OCR software first processes a document, it divides the paper into a matrix such that each cell in the grid contains a single **glyph**, which is a term referring to a letter, symbol, or number. Next, for each cell, the software will attempt to match the glyph to a set of all characters it recognizes. Finally, the individual characters can be combined into words, which optionally could be spell-checked against a dictionary in the document's language.

In this exercise, we'll assume that we have already developed the algorithm to partition the document into rectangular regions, each consisting of a single glyph. We will also assume the document contains only alphabetic characters in English. Therefore, we'll simulate a process that involves matching glyphs to 1 of the 26 letters, A to Z.

To this end, we'll use a dataset donated to the UCI Machine Learning Repository (<http://archive.ics.uci.edu/ml>) by W. Frey and D. J. Slate. The dataset contains 20,000 examples of 26 English alphabet capital letters as printed using 20 different randomly reshaped and distorted black-and-white fonts.



For more information about this data, refer to *Letter Recognition Using Holland-Style Adaptive Classifiers*, Slate, DJ and Frey, PW, *Machine Learning*, 1991, Vol. 6, pp. 161-182.

The following figure, published by Frey and Slate, provides an example of some of the printed glyphs. Distorted in this way, the letters are challenging for a computer to identify, yet are easily recognized by a human being:

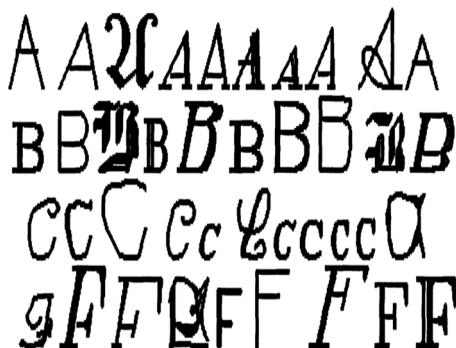


Figure 7.22: Examples of glyphs the SVM algorithm will attempt to identify

Step 2 – exploring and preparing the data

According to the documentation provided by Frey and Slate, when the glyphs are scanned into the computer, they are converted into pixels and 16 statistical attributes are recorded.

The attributes measure characteristics such as the horizontal and vertical dimensions of the glyph; the proportion of black (versus white) pixels; and the average horizontal and vertical position of the pixels. Presumably, differences in the concentration of black pixels across various areas of the box should provide a way to differentiate between the 26 letters of the alphabet.



To follow along with this example, download the `letterdata.csv` file from the Packt Publishing website and save it to your R working directory.

Reading the data into R, we confirm that we have received the data with the 16 features that define each example of the letter class. As expected, it has 26 levels:

```
> letters <- read.csv("letterdata.csv", stringsAsFactors = TRUE)
> str(letters)

'data.frame': 20000 obs. of 17 variables:
 $ letter: Factor w/ 26 levels "A","B","C","D",...: 20 9 4 14 ...
 $ xbox  : int 2 5 4 7 2 4 4 1 2 11 ...
 $ ybox  : int 8 12 11 11 1 11 2 1 2 15 ...
```

```
$ width : int  3 3 6 6 3 5 5 3 4 13 ...
$ height: int  5 7 8 6 1 8 4 2 4 9 ...
$ onpix : int  1 2 6 3 1 3 4 1 2 7 ...
$ xbar  : int  8 10 10 5 8 8 8 8 10 13 ...
$ ybar  : int  13 5 6 9 6 8 7 2 6 2 ...
$ x2bar : int  0 5 2 4 6 6 6 2 2 6 ...
$ y2bar : int  6 4 6 6 6 9 6 2 6 2 ...
$ xybar : int  6 13 10 4 6 5 7 8 12 12 ...
$ x2ybar: int  10 3 3 4 5 6 6 2 4 1 ...
$ xy2bar: int  8 9 7 10 9 6 6 8 8 9 ...
$ xedge : int  0 2 3 6 1 0 2 1 1 8 ...
$ xedgy: int  8 8 7 10 7 8 8 6 6 1 ...
$ yedge : int  0 4 3 2 5 9 7 2 1 1 ...
$ yedgx: int  8 10 9 8 10 7 10 7 7 8 ...
```

SVM learners require all features to be numeric, and moreover, that each feature is scaled to a fairly small interval. In this case, every feature is an integer, so we do not need to convert any factors into numbers. On the other hand, some of the ranges for these integer variables appear wide. This indicates that we need to normalize or standardize the data. However, we can skip this step for now because the R package that we will use for fitting the SVM model will perform rescaling automatically.

Given that there is no data preparation left to perform, we can move directly to the training and testing phases of the machine learning process. In previous analyses, we randomly divided the data between the training and testing sets. Although we could do so here, Frey and Slate have already randomized the data and therefore suggest using the first 16,000 records (80 percent) for building the model and the next 4,000 records (20 percent) for testing. Following their advice, we can create training and testing data frames as follows:

```
> letters_train <- letters[1:16000, ]
> letters_test  <- letters[16001:20000, ]
```

With our data ready to go, let's start building our classifier.

Step 3 – training a model on the data

When it comes to fitting an SVM model in R, there are several outstanding packages to choose from. The **e1071** package from the Department of Statistics at the **Vienna University of Technology (TU Wien)** provides an R interface to the award-winning LIBSVM library, a widely used open-source SVM program written in C++. If you are already familiar with LIBSVM, you may want to start here.



For more information on LIBSVM, refer to the authors' website at <http://www.csie.ntu.edu.tw/~cjlin/libsvm/>.

Similarly, if you're already invested in the SVMlight algorithm, the `klaR` package from the Department of Statistics at the **Dortmund University of Technology (TU Dortmund)** provides functions for working with this SVM implementation directly from R.



For information on SVMlight, see https://www.cs.cornell.edu/people/tj/svm_light/.

Finally, if you are starting from scratch, it is perhaps best to begin with the SVM functions in the `kernlab` package. An interesting advantage of this package is that it was developed natively in R rather than C or C++, which allows it to be easily customized; none of the internals are hidden behind the scenes. Perhaps even more importantly, unlike the other options, `kernlab` can be used with the `caret` package, which allows SVM models to be trained and evaluated using a variety of automated methods (covered in *Chapter 14, Building Better Learners*).



For a more thorough introduction to `kernlab`, please refer to the authors' paper at <http://www.jstatsoft.org/v11/i09/>.

The syntax for training SVM classifiers with `kernlab` is as follows. If you do happen to be using one of the other packages, the commands are largely similar. By default, the `ksvm()` function uses the Gaussian RBF kernel, but a number of other options are provided:

Support vector machine syntax

using the `ksvm()` function in the `kernlab` package

Building the model:

```
m <- ksvm(target ~ predictors, data = mydata,
            kernel = "rbfdot", C = 1)
```

- `target` is the outcome in the `mydata` data frame to be modeled
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` specifies the data frame in which the `target` and `predictors` variables can be found
- `kernel` specifies a nonlinear mapping such as `"rbfdot"` (radial basis), `"polydot"` (polynomial), `"tanhdot"` (hyperbolic tangent sigmoid), or `"vanilladot"` (linear)
- `C` is a number that specifies the cost of violating the constraints, i.e., how big of a penalty there is for the "soft margin." Larger values will result in narrower margins

The function will return a SVM object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "response")
```

- `m` is a model trained by the `ksvm()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier
- `type` specifies whether the predictions should be `"response"` (the predicted class) or `"probabilities"` (the predicted probability, one column per class level).

The function will return a vector (or matrix) of predicted classes (or probabilities) depending on the value of the `type` parameter.

Example:

```
letter_classifier <- ksvm(letter ~ ., data =
  letters_train, kernel = "vanilladot")
letter_prediction <- predict(letter_classifier,
  letters_test)
```

Figure 7.23: SVM syntax

To provide a baseline measure of SVM performance, let's begin by training a simple linear SVM classifier. If you haven't already, install the `kernlab` package to your library using the `install.packages("kernlab")` command. Then, we can call the `ksvm()` function on the training data and specify the linear (that is, "vanilla") kernel using the `vanilladot` option, as follows:

```
> library(kernlab)
> letter_classifier <- ksvm(letter ~ ., data = letters_train,
  kernel = "vanilladot")
```

Depending on the performance of your computer, this operation may take some time to complete. When it finishes, type the name of the stored model to see some basic information about the training parameters and the fit of the model:

```
> letter_classifier

Support Vector Machine object of class "ksvm"

SV type: C-svc (classification)
parameter : cost C = 1

Linear (vanilla) kernel function.

Number of Support Vectors : 7037

Objective Function Value : -14.1746 -20.0072 -23.5628 -6.2009 -7.5524
-32.7694 -49.9786 -18.1824 -62.1111 -32.7284 -16.2209...

Training error : 0.130062
```

This information tells us very little about how well the model will perform in the real world. We'll need to examine its performance on the testing dataset to know whether it generalizes well to unseen data.

Step 4 – evaluating model performance

The `predict()` function allows us to use the letter classification model to make predictions on the testing dataset:

```
> letter_predictions <- predict(letter_classifier, letters_test)
```

Because we didn't specify the `type` parameter, the default `type = "response"` was used. This returns a vector containing a predicted letter for each row of values in the testing data. Using the `head()` function, we can see that the first six predicted letters were U, N, V, X, N, and H:

```
> head(letter_predictions)
```

```
[1] U N V X N H
Levels: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

To examine how well our classifier performed, we need to compare the predicted letter with the true letter in the testing dataset. We'll use the `table()` function for this purpose (only a portion of the full table is shown here):

```
> table(letter_predictions, letters_test$letter)
```

letter_predictions	A	B	C	D	E
A	144	0	0	0	0
B	0	121	0	5	2
C	0	0	120	0	4
D	2	2	0	156	0
E	0	0	5	0	127

The diagonal values of 144, 121, 120, 156, and 127 indicate the total number of records where the predicted letter matches the true value. Similarly, the number of mistakes is also listed. For example, the value of 5 in row B and column D indicates that there were 5 cases where the letter D was misidentified as a B.

Looking at each type of mistake individually may reveal some interesting patterns about the specific types of letters the model has trouble with, but this is time-consuming. We can simplify our evaluation by instead calculating the overall accuracy. This considers only whether the prediction was correct or incorrect and ignores the type of error.

The following command returns a vector of TRUE or FALSE values indicating whether the model's predicted letter agrees with (that is, matches) the actual letter in the test dataset:

```
> agreement <- letter_predictions == letters_test$letter
```

Using the `table()` function, we see that the classifier correctly identified the letter in 3,357 out of the 4,000 test records:

```
> table(agreement)
```

agreement
FALSE
TRUE
643 3357

In percentage terms, the accuracy is about 84 percent:

```
> prop.table(table(agreement))
```

```

agreement
  FALSE    TRUE
0.16075 0.83925

```

Note that when Frey and Slate published the dataset in 1991, they reported a recognition accuracy of about 80 percent. Using just a few lines of R code, we were able to surpass their result, although we also have the benefit of decades of additional machine learning research. With that in mind, it is likely that we can do even better.

Step 5 – improving model performance

Let's take a moment to contextualize the performance of the SVM model we trained to identify letters of the alphabet from image data. With 1 line of R code, the model was able to achieve an accuracy of nearly 84 percent, which slightly surpassed the benchmark percent published by academic researchers in 1991. Although an accuracy of 84 percent is not nearly high enough to be useful for OCR software, the fact that a relatively simple model can reach this level is a remarkable accomplishment in itself. Keep in mind that the probability the model's prediction would match the actual value by dumb luck alone is quite small at under four percent. This implies that our model performs over 20 times better than random chance. As remarkable as this is, perhaps by adjusting the SVM function parameters to train a slightly more complex model, we can also find that the model is useful in the real world.



To calculate the probability of the SVM model's predictions matching the actual values by chance alone, apply the joint probability rule for independent events covered in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. Because there are 26 letters, each appearing at approximately the same rate in the test set, the chance that any one letter is predicted correctly is $(1/26) * (1/26)$. Since there are 26 different letters, the total probability of agreement is $26 * (1/26) * (1/26) = 0.0384$, or 3.84 percent.

Changing the SVM kernel function

Our previous SVM model used the simple linear kernel function. By using a more complex kernel function, we can map the data into a higher dimensional space, and potentially obtain a better model fit.

It can be challenging, however, to choose from the many different kernel functions. A popular convention is to begin with the Gaussian RBF kernel, which has been shown to perform well for many types of data.

We can train an RBF-based SVM using the `ksvm()` function as shown here. Note that, much like other methods used previously, we need to set the random seed to ensure the results are reproducible:

```
> set.seed(12345)
> letter_classifier_rbf <- ksvm(letter ~ ., data = letters_train,
                                 kernel = "rbfdot")
```

Next, we make predictions as before:

```
> letter_predictions_rbf <- predict(letter_classifier_rbf,
                                      letters_test)
```

Finally, we'll compare the accuracy to our linear SVM:

```
> agreement_rbf <- letter_predictions_rbf == letters_test$letter
> table(agreement_rbf)
```

```
agreement_rbf
FALSE  TRUE
278 3722
```

```
> prop.table(table(agreement_rbf))
```

```
agreement_rbf
FALSE      TRUE
0.0695  0.9305
```

Simply by changing the kernel function, we were able to increase the accuracy of our character recognition model from 84 percent to 93 percent.

Identifying the best SVM cost parameter

If this level of performance is still unsatisfactory for the OCR program, it is certainly possible to test additional kernels. However, another fruitful approach is to vary the cost parameter, which modifies the width of the SVM decision boundary. This governs the model's balance between overfitting and underfitting the training data—the larger the cost value, the harder the learner will try to perfectly classify every training instance, as there is a higher penalty for each mistake. On the one hand, a high cost can lead the learner to overfit the training data. On the other hand, a cost parameter set too small can cause the learner to miss important, subtle patterns in the training data and underfit the true pattern.

There is no rule of thumb for knowing the ideal value beforehand, so instead, we will examine how the model performs for various values of C , the cost parameter. Rather than repeating the training and evaluation process repeatedly, we can use the `sapply()` function to apply a custom function to a vector of potential cost values. We begin by using the `seq()` function to generate this vector as a sequence counting from 5 to 40 by 5. Then, as shown in the following code, the custom function trains the model as before, each time using the cost value and making predictions on the test dataset. Each model's accuracy is computed as the number of predictions that match the actual values divided by the total number of predictions. The result is visualized using the `plot()` function. Note that depending on the capabilities of your computer, this may take a few minutes to complete:

```
> cost_values <- c(1, seq(from = 5, to = 40, by = 5))
> accuracy_values <- sapply(cost_values, function(x) {
  set.seed(12345)
  m <- ksvm(letter ~ ., data = letters_train,
             kernel = "rbfdot", C = x)
  pred <- predict(m, letters_test)
  agree <- ifelse(pred == letters_test$letter, 1, 0)
  accuracy <- sum(agree) / nrow(letters_test)
  return (accuracy)
})
> plot(cost_values, accuracy_values, type = "b")
```

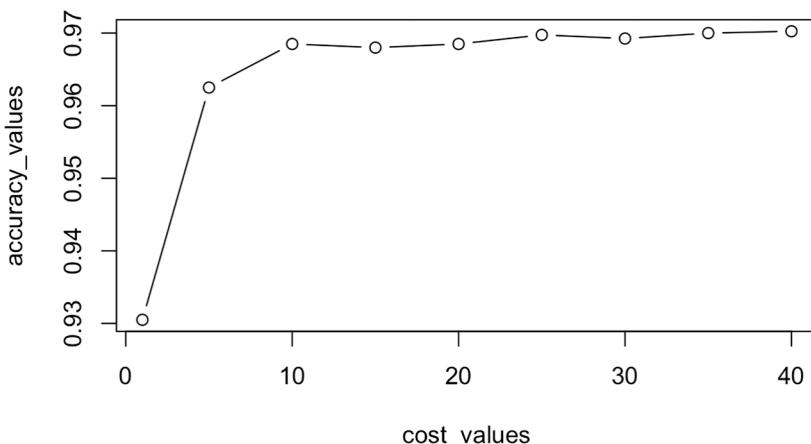


Figure 7.24: Mapping accuracy against the SVM cost for the RBF kernel

As depicted in the visualization, with an accuracy of 93 percent, the default SVM cost parameter of $C = 1$ resulted in by far the least accurate model among the 9 models evaluated. Instead, setting C to a value of 10 or higher results in an accuracy of around 97 percent, which is quite an improvement in performance! Perhaps this is close enough to perfect for the model to be deployed in a real-world environment, though it may still be worth experimenting further with various kernels to see if it is possible to get even closer to 100 percent accuracy. Each additional improvement in accuracy will result in fewer mistakes for the OCR software and a better overall experience for the end user.

Summary

In this chapter, we examined two machine learning methods that offer a great deal of potential but are often overlooked due to their complexity. Hopefully, you now see that this reputation is at least somewhat undeserved. The basic concepts that drive ANNs and SVMs are not too difficult to understand.

On the other hand, because ANNs and SVMs have been around for many decades, each of them has numerous variations. This chapter just scratches the surface of what is possible with these methods. By utilizing the terminology that you learned here, you should be capable of picking up the nuances that distinguish the many advancements that are being developed every day, including the ever-growing field of deep learning. We will revisit deep learning in *Chapter 15, Making Use of Big Data*, to see how it can solve some of machine learning's most challenging problems.

In the past several chapters, we have learned about many different types of predictive models, from those based on simple heuristics like nearest neighbors to sophisticated black-box models and many others. In the next chapter, we will begin to consider methods for another type of learning task. These unsupervised learning techniques will bring to light fascinating patterns within the data as they assist us with finding needles in haystacks.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



8

Finding Patterns – Market Basket Analysis Using Association Rules

Think back to your last impulse purchase. Maybe in the grocery store checkout lane, you bought a pack of chewing gum or a candy bar. Perhaps on a late-night trip for diapers and formula, you picked up a caffeinated beverage or a six-pack of beer. You might have even bought this book on a bookseller's recommendation. These impulse buys are no coincidence, as retailers use sophisticated data analysis techniques to identify useful patterns for marketing promotions and driving upselling via product placement.

In years past, such recommendations were based on the subjective intuition of marketing professionals and inventory managers. Now, barcode scanners, inventory databases, and online shopping carts all generate transactional data that machine learning can use to learn purchasing patterns. The practice is commonly known as market basket analysis because it has been so frequently applied to supermarket data.

Although the technique originated with shopping data, it is also useful in other contexts. By the time you finish this chapter, you will know how to apply market basket analysis techniques to your own tasks, whatever they may be. Generally, the work involves:

- Understanding the peculiarities of transactional data
- Using simple performance measures to find associations in large databases
- Knowing how to identify useful and actionable patterns

Because market basket analysis is able to discover nuggets of insight in many types of large datasets, as we apply the technique, you are likely to identify applications to your work even if you have no affiliation with the retail sector.

Understanding association rules

The building blocks of a market basket analysis are the items that may appear in any given transaction. Groups of one or more items are surrounded by brackets to indicate that they form a set, or more specifically, an **itemset** that appears in the data with some regularity. Transactions are specified in terms of itemsets, such as the following transaction that might be found in a typical grocery store:

```
{bread, peanut butter, jelly}
```

The result of a market basket analysis is a collection of **association rules** that specify patterns found in the relationships among items in the itemsets. Association rules are always composed from subsets of itemsets and are denoted by relating one itemset on the **left-hand side (LHS)** of the rule to another itemset on the **right-hand side (RHS)** of the rule. The LHS is the condition that needs to be met in order to trigger the rule, and the RHS is the expected result of meeting that condition. A rule identified from the preceding example transaction might be expressed in the form:

```
{peanut butter, jelly} → {bread}
```

In plain language, this association rule states that if peanut butter and jelly are purchased together, then bread is also likely to be purchased. In other words, “peanut butter and jelly imply bread.”

Developed in the context of retail transaction databases, association rules are not used for prediction, but rather for unsupervised knowledge discovery in large databases. This is unlike the classification and numeric prediction algorithms presented in previous chapters. Even so, you will find that the result of association rule learning is closely related to and shares many features with the result of classification rule learning as presented in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*.

Because association rule learners are unsupervised, there is no need for the algorithm to be trained and the data does not need to be labeled ahead of time. The program is simply unleashed on a dataset in the hope that interesting associations are found. The downside, of course, is that there isn’t an easy way to objectively measure the performance of a rule learner, aside from evaluating it for qualitative usefulness—typically, an eyeball test of some sort.

Although association rules are most often used for market basket analysis, they are helpful for finding patterns in many different types of data. Other potential applications include:

- Searching for interesting and frequently occurring patterns of DNA and protein sequences in cancer data
- Finding patterns of purchases or medical claims that occur in combination with fraudulent credit card or insurance use
- Identifying combinations of behavior that precede customers dropping their cellular phone service or upgrading their cable television package

Association rule analysis is used to search for interesting connections among a very large number of elements. Human beings are capable of such insight quite intuitively, but it often takes expert-level knowledge or a great deal of experience to do what a rule learning algorithm can do in minutes or even seconds. Additionally, some datasets are simply too large and complex for a human being to find the needle in the haystack.

The Apriori algorithm for association rule learning

Just as large transactional datasets create challenges for humans, these datasets also present challenges for machines. Transactional datasets can be large in both the number of transactions as well as the number of items or features that are recorded. The fundamental problem of searching for interesting itemsets is that the number of potential itemsets grows exponentially with the number of items. Given k items that can appear or not appear in a set, there are 2^k possible itemsets that could be potential rules. A retailer that sells only 100 different items could have on the order of $2^{100} = 1.27e+30$ itemsets that an algorithm must evaluate—a seemingly impossible task.

Rather than evaluating each of these itemsets one by one, a smarter rule learning algorithm takes advantage of the fact that many of the potential combinations of items are rarely, if ever, found in practice. For instance, even if a store sells both automotive items and food products, a set of *{motor oil, bananas}* is likely to be extraordinarily uncommon. By ignoring these rare (and perhaps less important) combinations, it is possible to limit the scope of the search for rules to a more manageable size.

Much work has been done to identify heuristic algorithms for reducing the number of itemsets to search. Perhaps the most widely-known approach for efficiently searching large databases for rules is known as **Apriori**. Introduced in 1994 by Rakesh Agrawal and Ramakrishnan Srikant, the Apriori algorithm has since become somewhat synonymous with association rule learning, despite the invention of newer and faster algorithms. The name is derived from the fact that the algorithm utilizes a simple prior (that is, *a priori*) belief about the properties of frequent itemsets.

Before we discuss that in more depth, it's worth noting that this algorithm, like all learning algorithms, is not without its strengths and weaknesses. Some of these are listed as follows:

Strengths	Weaknesses
<ul style="list-style-type: none"> • Capable of working with large amounts of transactional data • Results in rules that are easy to understand • Useful for data mining and discovering unexpected knowledge in databases 	<ul style="list-style-type: none"> • Not very helpful for relatively small datasets • Takes effort to separate true insight from common sense • Easy to draw spurious conclusions from random patterns

As noted earlier, the Apriori algorithm employs a simple *a priori* belief as a guideline for reducing the association rule search space: all subsets of a frequent itemset must also be frequent. This heuristic is known as the **Apriori property**. Using this astute observation, it is possible to dramatically limit the number of rules to search. For example, the set $\{motor\ oil, bananas\}$ can only be frequent if both $\{motor\ oil\}$ and $\{bananas\}$ occur frequently as well. Consequently, if either $\{motor\ oil\}$ or $\{bananas\}$ are infrequent, then any set containing these items can be excluded from the search.



For additional details on the Apriori algorithm, refer to *Fast Algorithms for Mining Association Rules*, Agrawal, R., Srikant, R., Proceedings of the 20th International Conference on Very Large Databases, 1994, pp. 487-499.

To see how this principle can be applied in a more realistic setting, let's consider a simple transaction database. The following table shows five completed transactions at an imaginary hospital's gift shop:

transaction ID	items purchased
1	{flowers, get-well card, soda}
2	{plush toy bear, flowers, balloons, candy bar}
3	{get-well card, candy bar, flowers}
4	{plush toy bear, balloons, soda}
5	{flowers, get-well card, soda}

Figure 8.1: Itemsets representing five transactions in a hypothetical hospital's gift shop

By looking at the sets of purchases, one can infer that there are a couple of typical buying patterns. A person visiting a sick friend or family member tends to buy a get-well card and flowers, while visitors of new mothers tend to buy plush toy bears and balloons. Such patterns are notable because they appear frequently enough to catch our interest; we simply apply a bit of logic and subject matter experience to explain the rule.

In a similar fashion, the Apriori algorithm uses statistical measures of an itemset's "interestingness" to locate association rules in much larger transaction databases. In the sections that follow, we will discover how Apriori computes such measures of interest, and how they are combined with the Apriori property to reduce the number of rules to be learned.

Measuring rule interest – support and confidence

Whether or not an association rule is deemed interesting is determined by two statistical measures: support and confidence. By providing minimum thresholds for each of these metrics and applying the Apriori principle, it is easy to drastically limit the number of rules reported. If this limit is too strict, it may cause only the most obvious or common-sense rules to be identified. For this reason, it is important to carefully understand the types of rules that are excluded under these criteria so that the right balance can be obtained.

The **support** of an itemset or rule measures how frequently it occurs in the data. For instance, the itemset $\{get\ well\ card, flowers\}$ has the support of $3/5 = 0.6$ in the hospital gift shop data. Similarly, the support for $\{get\ well\ card\} \rightarrow \{flowers\}$ is also 0.6. Support can be calculated for any itemset or even a single item; for instance, the support for $\{candy\ bar\}$ is $2/5 = 0.4$, since candy bars appear in 40 percent of purchases. A function defining support for itemset X could be defined as:

$$\text{support}(X) = \frac{\text{count}(X)}{N}$$

Here, N is the number of transactions in the database, and $\text{count}(X)$ is the number of transactions containing itemset X .

A rule's **confidence** is a measurement of its predictive power or accuracy. It is defined as the support of the itemset containing both X and Y divided by the support of the itemset containing only X :

$$\text{confidence}(X \rightarrow Y) = \frac{\text{support}(X, Y)}{\text{support}(X)}$$

Essentially, the confidence tells us the proportion of transactions where the presence of item or itemset X results in the presence of item or itemset Y . Keep in mind that the confidence that X leads to Y is not the same as the confidence that Y leads to X .

For example, the confidence of $\{flowers\} \rightarrow \{get-well\ card\}$ is $0.6 / 0.8 = 0.75$. In comparison, the confidence of $\{get-well\ card\} \rightarrow \{flowers\}$ is $0.6 / 0.6 = 1.0$. This means that a purchase of flowers also includes the purchase of a get-well card 75 percent of the time, while a purchase of a get-well card also includes flowers 100 percent of the time. This information could be quite useful to the gift shop management.



You may have noticed similarities between support, confidence, and the Bayesian probability rules covered in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. In fact, $support(A, B)$ is the same as $P(A \cap B)$ and $confidence(A \rightarrow B)$ is the same as $P(B | A)$. It is just the context that differs.

Rules like $\{get-well\ card\} \rightarrow \{flowers\}$ are known as **strong rules** because they have both high support and confidence. One way to find more strong rules would be to examine every possible combination of items in the gift shop, measure the support and confidence, and report back only those rules that meet certain levels of interest. However, as noted before, this strategy is generally not feasible for anything but the smallest of datasets.

In the next section, you will see how the Apriori algorithm uses minimum levels of support and confidence with the Apriori principle to find strong rules quickly by reducing the number of rules to a more manageable level.

Building a set of rules with the Apriori principle

Recall that the Apriori principle states that all subsets of a frequent itemset must also be frequent. In other words, if $\{A, B\}$ is frequent, then $\{A\}$ and $\{B\}$ must both be frequent. Recall also that, by definition, the support metric indicates how frequently an itemset appears in the data. Therefore, if we know that $\{A\}$ does not meet a desired support threshold, there is no reason to consider $\{A, B\}$ or any other itemset containing $\{A\}$; these cannot possibly be frequent.

The Apriori algorithm uses this logic to exclude potential association rules prior to evaluating them. The process of creating rules then occurs in two phases:

1. Identifying all the itemsets that meet a minimum support threshold
2. Creating rules from these itemsets using those meeting a minimum confidence threshold

The first phase occurs in multiple iterations. Each successive iteration involves evaluating the support of a set of increasingly large itemsets. For instance, iteration one involves evaluating the set of 1-item itemsets (1-itemsets), iteration two evaluates the 2-itemsets, and so on. The result of each iteration i is a set of all the i -itemsets that meet the minimum support threshold.

All the itemsets from iteration i are combined to generate candidate itemsets for evaluation in iteration $i + 1$. But the Apriori principle can eliminate some of them even before the next round begins. If $\{A\}$, $\{B\}$, and $\{C\}$ are frequent in iteration one, while $\{D\}$ is not frequent, then iteration two will consider only $\{A, B\}$, $\{A, C\}$, and $\{B, C\}$. Thus, the algorithm needs to evaluate only three itemsets rather than the six 2-item itemsets that would have needed to be evaluated if the sets containing D had not been eliminated *a priori*.

Continuing this thought, suppose during iteration two it is discovered that $\{A, B\}$ and $\{B, C\}$ are frequent, but $\{A, C\}$ is not. Although iteration three would normally begin by evaluating the support for the 3-item itemset $\{A, B, C\}$, this step is not necessary. Why not? The Apriori principle states that $\{A, B, C\}$ cannot possibly be frequent, since the subset $\{A, C\}$ is not. Therefore, having generated no new itemsets in iteration three, the algorithm may stop.

iteration	must evaluate	frequent itemsets	infrequent itemsets
1	$\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$	$\{A\}$, $\{B\}$, $\{C\}$	$\{D\}$
2	$\{A, B\}$, $\{A, C\}$, $\{B, C\}$ $\cancel{\{A, D\}}$, $\cancel{\{B, D\}}$, $\cancel{\{C, D\}}$	$\{A, B\}$, $\{B, C\}$	$\{A, C\}$
3	$\cancel{\{A, B, C\}}$, $\cancel{\{A, B, D\}}$ $\cancel{\{A, C, D\}}$, $\cancel{\{B, C, D\}}$		
4	$\cancel{\{A, B, C, D\}}$		

Figure 8.2: In this example, the Apriori algorithm only evaluated 7 of the 15 potential itemsets that can occur in transactional data for four items (the 0-item itemset is not shown)

At this point, the second phase of the Apriori algorithm may begin. Given the set of frequent itemsets, association rules are generated from all possible subsets. For instance, $\{A, B\}$ would result in candidate rules for $\{A\} \rightarrow \{B\}$ and $\{B\} \rightarrow \{A\}$. These are evaluated against a minimum confidence threshold, and any rule that does not meet the desired confidence level is eliminated.

Example – identifying frequently purchased groceries with association rules

As noted in this chapter's introduction, market basket analysis is used behind the scenes for the recommendation systems used in many brick-and-mortar and online retailers. The learned association rules indicate the items that are often purchased together. Knowledge of these patterns provides insight into new ways a grocery chain might optimize the inventory, advertise promotions, or organize the physical layout of the store. For instance, if shoppers frequently purchase coffee or orange juice with a breakfast pastry, it may be possible to increase profit by relocating pastries closer to coffee and juice.

Similarly, online retailers can use the information for dynamic recommendation engines that suggest items related to those you've already viewed or to follow up after a website visit or online purchase with an email that suggests add-on items in a practice called **active after-marketing**.

In this tutorial, we will perform a market basket analysis of transactional data from a grocery store. In doing so, we will see how the Apriori algorithm is able to efficiently evaluate a potentially massive set of association rules. The same techniques could also be applied to many other business tasks, from movie recommendations to dating sites to finding dangerous interactions among medications.

Step 1 – collecting data

Our market basket analysis will utilize purchase data from one month of operation at a real-world grocery store. The data contains 9,835 transactions, or about 327 transactions per day (roughly 30 transactions per hour in a 12-hour business day), suggesting that the retailer is not particularly large, nor is it particularly small.



The dataset used here was adapted from the `Groceries` dataset in the `arules` R package. For more information, see *Implications of Probabilistic Data Modeling for Mining Association Rules*, Hahsler, M., Hornik, K., Reutterer, T., 2005. In *From Data and Information Analysis to Knowledge Engineering*, Gaul, W., Vichi, M., Weihs, C., *Studies in Classification, Data Analysis, and Knowledge Organization*, 2006, pp. 598–605.

A typical grocery store offers a huge variety of items. There might be five brands of milk, a dozen types of laundry detergent, and three brands of coffee. Given the moderate size of the retailer in this example, we will assume that it is not terribly concerned with finding rules that apply only to a specific brand of milk or detergent, and thus all brand names are removed from the purchases. This reduces the number of grocery items to a more manageable 169 types, using broad categories such as chicken, frozen meals, margarine, and soda.



If you hope to identify highly specific association rules—such as whether customers prefer grape or strawberry jelly with their peanut butter—you will need a tremendous amount of transactional data. Large chain retailers use databases of many millions of transactions in order to find associations among particular brands, colors, or flavors of items.

Do you have any guesses about which types of items might be purchased together? Will wine and cheese be a common pairing? Bread and butter? Tea and honey? Let's dig into this data and see if these guesses can be confirmed.

Step 2 – exploring and preparing the data

Transactional data is stored in a slightly different format than we have used previously. Most of our prior analyses utilized data in a matrix format where rows indicated example instances and columns indicated features. As described in *Chapter 1, Introducing Machine Learning*, in matrix format, all examples must have exactly the same set of features.

In comparison, transactional data is more freeform. As usual, each row in the data specifies a single example—in this case, a transaction. However, rather than having a set number of features, each record comprises a comma-separated list of any number of items, from one to many. In essence, the features may differ from example to example.



To follow along with this analysis, download the `groceries.csv` file from the Packt Publishing GitHub repository for this chapter and save it in your R working directory.

The first five rows of the raw `groceries.csv` file are as follows:

```
citrus fruit,semi-finished bread,margarine,ready soups
tropical fruit,yogurt,coffee
whole milk
pip fruit,yogurt,cream cheese,meat spreads
other vegetables,whole milk,condensed milk,long life bakery product
```

These lines indicate five separate grocery store transactions. The first transaction included four items: citrus fruit, semi-finished bread, margarine, and ready soups. In comparison, the third transaction included only one item: whole milk.

Suppose we tried to load the data using the `read.csv()` function as we did in prior analyses. R would happily comply and read the data into a data frame in matrix format as follows:

	V1	V2	V3	V4
1	citrus fruit	semi-finished bread	margarine	ready soups
2	tropical fruit	yogurt	coffee	
3	whole milk			
4	pip fruit	yogurt	cream cheese	meat spreads
5	other vegetables	whole milk	condensed milk	long life bakery product

Figure 8.3: Reading transactional data into R as a data frame will cause problems later on

You will notice that R created four columns to store the items in the transactional data: V1, V2, V3, and V4. Although this may seem reasonable, if we use the data in this form, we will encounter problems later. R chose to create four variables because the first line had exactly four comma-separated values. However, we know that grocery purchases can contain more than four items; in the four-column design, such transactions will be broken across multiple rows in the matrix. We could try to remedy this by putting the transaction with the largest number of items at the top of the file, but this ignores another more problematic issue.

By structuring the data this way, R has constructed a set of features that record not just the items in the transactions but also the order they appear. If we imagine our learning algorithm as an attempt to find a relationship among V1, V2, V3, and V4, then the whole `milk` item in V1 might be treated differently than the whole `milk` item appearing in V2. Instead, we need a dataset that does not treat a transaction as a set of positions to be filled (or not filled) with specific items, but rather as a market basket that either contains or does not contain each item.

Data preparation – creating a sparse matrix for transaction data

The solution to this problem utilizes a data structure called a sparse matrix. You may recall that we used a sparse matrix to process text data in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. Just as with the preceding dataset, each row in the sparse matrix indicates a transaction. However, the sparse matrix has a column (that is, a feature) for every item that could possibly appear in someone's shopping bag. Since there are 169 different items in our grocery store data, our sparse matrix will contain 169 columns.

Why not just store this as a data frame as we did in most of our prior analyses? The reason is that as additional transactions and items are added, a conventional data structure quickly becomes too large to fit in the available memory. Even with the relatively small transactional dataset used here, the matrix contains nearly 1.7 million cells, most of which contain zeros (hence the name "sparse" matrix—there are very few non-zero values).

Since there is no benefit to storing all these zeros, a sparse matrix does not actually store the full matrix in memory; it only stores the cells that are occupied by an item. This allows the structure to be more memory efficient than an equivalently sized matrix or data frame.

In order to create the sparse matrix data structure from transactional data, we can use the functionality provided by the `arules` (association rules) package. Install and load the package using the `install.packages("arules")` and `library(arules)` commands.



For more information on the `arules` package, refer to *arules - A Computational Environment for Mining Association Rules and Frequent Item Sets*, Hahsler, M., Gruen, B., Hornik, K., *Journal of Statistical Software*, 2005, Vol. 14.

Because we're loading transactional data, we cannot simply use the `read.csv()` function used previously. Instead, `arules` provides a `read.transactions()` function that is similar to `read.csv()` with the exception that it results in a sparse matrix suitable for transactional data. The parameter `sep = ","` specifies that items in the input file are separated by a comma. To read the `groceries.csv` data into a sparse matrix named `groceries`, type the following line:

```
> groceries <- read.transactions("groceries.csv", sep = ",")
```

To see some basic information about the `groceries` matrix we just created, use the `summary()` function on the object:

```
> summary(groceries)

transactions as itemMatrix in sparse format with
 9835 rows (elements/itemsets/transactions) and
 169 columns (items) and a density of 0.02609146
```

The first block of information in the output provides a summary of the sparse matrix we created. The output `9835 rows` refers to the number of transactions, and `169 columns` indicates each of the 169 different items that might appear in someone's grocery basket. Each cell in the matrix is a `1` if the item was purchased for the corresponding transaction, or `0` otherwise.

The density value of `0.02609146` (2.6 percent) refers to the proportion of non-zero matrix cells. Since there are $9,835 \times 169 = 1,662,115$ positions in the matrix, we can calculate that a total of $1,662,115 \times 0.02609146 = 43,367$ items were purchased during the store's 30 days of operation (ignoring the fact that duplicates of the same items might have been purchased). With an additional step, we can determine that the average transaction contained $43,367 / 9,835 = 4.409$ distinct grocery items. Of course, if we look a little further down the output, we'll see that the mean number of items per transaction has already been provided.

The next block of `summary()` output lists the items that were most commonly found in the transactional data. Since $2,513 / 9,835 = 0.2555$, we can determine that whole `milk` appeared in 25.6 percent of transactions.

Other vegetables, rolls/buns, soda, and yogurt round out the list of other common items, as follows:

most frequent items:			
	whole milk	other vegetables	rolls/buns
	2513	1903	1809
	soda	yogurt	(Other)
	1715	1372	34055

We are also presented with a set of statistics about the size of the transactions. A total of 2,159 transactions contained only a single item, while one transaction had 32 items. The first quartile and median purchase size are two and three items respectively, implying that 25 percent of the transactions contained two or fewer items and about half contained three items or fewer. The mean of 4.409 items per transaction matches the value we calculated by hand:

```

element (itemset/transaction) length distribution:
sizes
  1   2   3   4   5   6   7   8   9   10  11  12
2159 1643 1299 1005 855 645 545 438 350 246 182 117
  13  14  15  16  17  18  19  20  21  22  23  24
  78  77  55  46  29  14  14   9  11   4   6   1
  26  27  28  29  32
  1   1   1   3   1

  Min. 1st Qu. Median      Mean 3rd Qu.      Max.
1.000  2.000  3.000  4.409  6.000 32.000

```

Finally, the bottom of the output includes additional information about any metadata that may be associated with the item matrix, such as item hierarchies or labels. We have not used these advanced features, but the output still indicates that the data has labels. The `read.transactions()` function added these automatically upon load using the item names in the original CSV file, and the first three labels (in alphabetical order) are shown:

```
includes extended item information - examples:  
    labels  
1 abrasive cleaner  
2 artif. Sweetener  
3 baby cosmetics
```

Note that the arules package represents items internally using numeric item ID numbers with no connection to the item in the real world. By default, most arules functions will decode these numbers using the item labels. However, to illustrate the numeric IDs, we can examine the first two transactions without decoding using the so-called “long” format. In long-format transactional data, each row is a single item from a single transaction rather than each row representing a single transaction with multiple items. For instance, because the first and second transactions had four and three items, respectively, the long format represents these transactions in seven rows:

```
> head(toLongFormat(groceries, decode = FALSE), n = 7)
```

	TID	item
1	1	30
2	1	89
3	1	119
4	1	133
5	2	34
6	2	158
7	2	168

In this representation of the transactional data, the column named TID refers to the transaction ID—that is, the first or second market basket—and the column named item refers to the internal ID number assigned to the item. As the first transaction contained *{citrus fruit, margarine, ready soups, and semi-finished bread}*, we can assume that the item ID of 30 refers to *citrus fruit* and 89 refers to *margarine*.

The arules package, of course, includes features for examining transaction data in more intuitive formats. To look at the contents of the sparse matrix, use the inspect() function in combination with R’s vector operators. The first five transactions can be viewed as follows:

```
> inspect(groceries[1:5])
```

	items
[1]	{citrus fruit, margarine, ready soups, semi-finished bread}
[2]	{coffee, tropical fruit, yogurt}
[3]	{whole milk}

```
[4] {cream cheese,
     meat spreads,
     pip fruit,
     yogurt}
[5] {condensed milk,
     long life bakery product,
     other vegetables,
     whole milk}
```

When formatted using the `inspect()` function, the data does not look very different from what we had seen in the original CSV file.

Because the `groceries` object is stored as a sparse item matrix, the `[row, column]` notation can be used to examine desired items as well as desired transactions. Using this with the `itemFrequency()` function allows us to see the proportion of all transactions that contain the specified item. For instance, to view the support level for the first three items across all rows in the grocery data, use the following command:

```
> itemFrequency(groceries[, 1:3])
abrasive    cleaner    artif.    sweetener    baby    cosmetics
0.0035587189    0.0032536858    0.0006100661
```

Notice that the items in the sparse matrix are arranged in columns in alphabetical order. Abrasive cleaners and artificial sweeteners are found in about 0.3 percent of the transactions, while baby cosmetics are found in about 0.06 percent of the transactions.

Visualizing item support – item frequency plots

To present these statistics visually, use the `itemFrequencyPlot()` function. This creates a bar chart depicting the proportion of transactions containing specified items. Since transactional data contains a very large number of items, you will often need to limit those appearing in the plot in order to produce a legible chart.

If you would like to display items that appear in a minimum proportion of transactions, use `itemFrequencyPlot()` with the `support` parameter:

```
> itemFrequencyPlot(groceries, support = 0.1)
```

As shown in the following plot, this results in a histogram showing the eight items in the `groceries` data with at least 10 percent support:

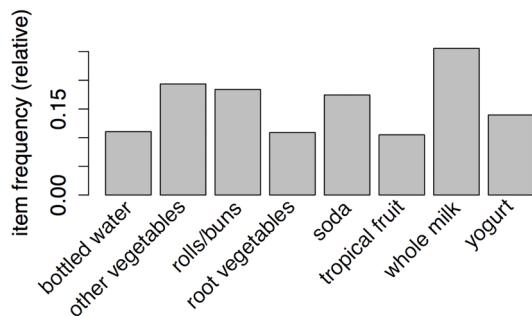


Figure 8.4: Support levels for all grocery items in at least 10 percent of transactions

If you would rather limit the plot to a specific number of items, use the function with the `topN` parameter:

```
> itemFrequencyPlot(groceries, topN = 20)
```

The histogram is then sorted by decreasing support, as shown in the following diagram for the top 20 items in the `groceries` data:

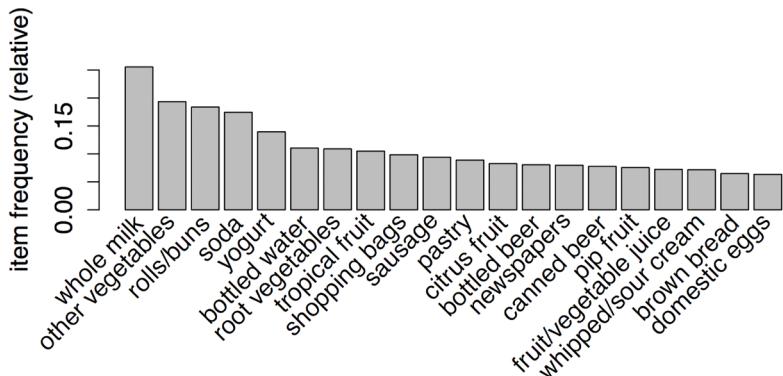


Figure 8.5: Support levels for the top 20 grocery items

Visualizing the transaction data – plotting the sparse matrix

In addition to looking at specific items, it's also possible to obtain a bird's-eye view of the entire sparse matrix using the `image()` function. Of course, because the matrix itself is very large, it is usually best to request a subset of the entire matrix. The command to display the sparse matrix for the first five transactions is as follows:

```
> image(groceries[1:5])
```

The resulting diagram depicts a matrix with 5 rows and 169 columns, indicating the 5 transactions and 169 possible items we requested. Cells in the matrix are filled with black for transactions (rows) where the item (column) was purchased.

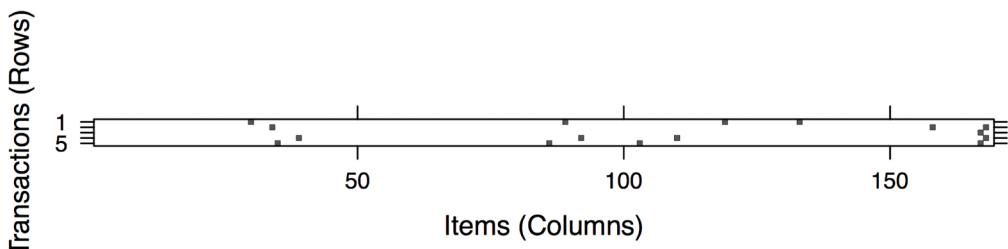


Figure 8.6: A visualization of the sparse matrix for the first five transactions

Although *Figure 8.6* is small and may be slightly hard to read, you can see that the first, fourth, and fifth transactions contained four items each, since their rows have four cells filled in. On the right side of the diagram, you can also see that rows three and five, and rows two and four, share an item in common.

This visualization can be a useful tool for exploring transactional data. For one, it may help with the identification of potential data issues. Columns that are filled all the way down could indicate items that are purchased in every transaction—a problem that could arise, perhaps, if a retailer's name or identification number was inadvertently included in the transaction dataset.

Additionally, patterns in the diagram may help reveal interesting segments of transactions and items, particularly if the data is sorted in interesting ways. For example, if the transactions are sorted by date, patterns in the black dots could reveal seasonal effects in the number or types of items purchased. Perhaps around Christmas or Hanukkah, toys are more common; around Halloween, perhaps candies become popular. This type of visualization could be especially powerful if the items were also sorted into categories. In most cases, however, the plot will look fairly random, like static on a television screen.

Keep in mind that this visualization will not be as useful for extremely large transaction databases because the cells will be too small to discern. Still, by combining it with the `sample()` function, you can view the sparse matrix for a randomly sampled set of transactions. The command to create a random selection of 100 transactions is as follows:

```
> image(sample(groceries, 100))
```

This creates a matrix diagram with 100 rows and 169 columns:

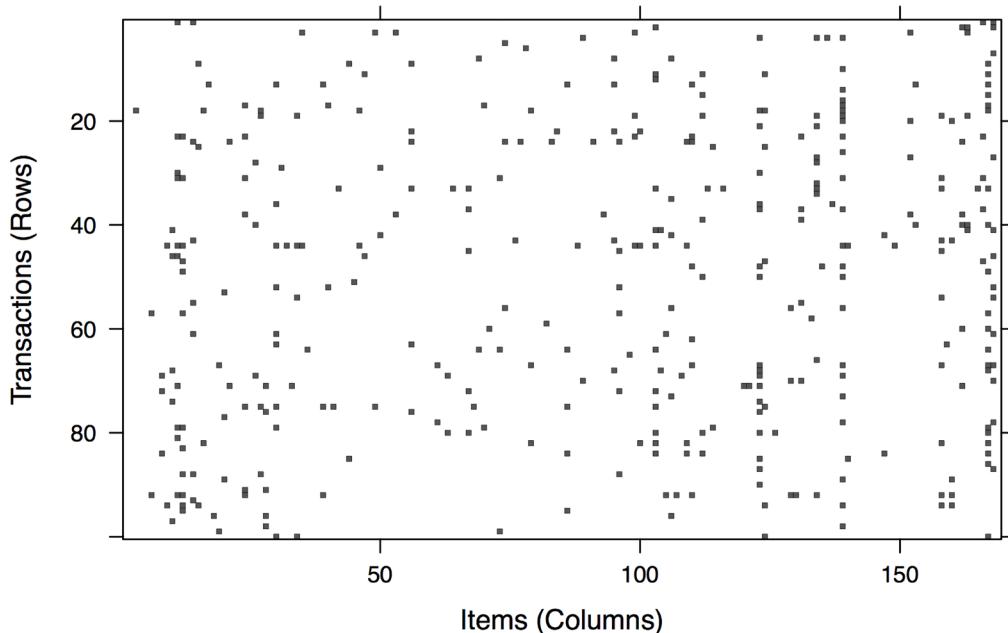


Figure 8.7: A visualization of the sparse matrix for 100 randomly selected transactions

A few columns seem fairly heavily populated, indicating some very popular items at the store. However, the distribution of dots seems fairly random overall. Given nothing else of note, let's continue with our analysis.

Step 3 – training a model on the data

With data preparation complete, we can now work at finding associations among shopping cart items. We will use an implementation of the Apriori algorithm in the `arules` package we've been using for exploring and preparing the `groceries` data. You'll need to install and load this package if you have not done so already.

The following table shows the syntax for creating sets of rules with the `apriori()` function:

Association rule syntax
using the <code>apriori()</code> function in the <code>arules</code> package
Finding association rules:
<pre>myrules <- apriori(data = mydata, parameter = list(support = 0.1, confidence = 0.8, minlen = 1)) • <code>data</code> is a sparse item matrix holding transactional data • <code>support</code> specifies the minimum required rule support • <code>confidence</code> specifies the minimum required rule confidence • <code>minlen</code> specifies the minimum required rule items</pre>
The function will return a rules object storing all rules that meet the minimum criteria.
Examining association rules:
<pre>inspect(myrules) • <code>myrules</code> is a set of association rules from the <code>apriori()</code> function</pre>
This will output the association rules to the screen. Vector operators can be used on <code>myrules</code> to choose a specific rule or rules to view.
Example:
<pre>groceryrules <- apriori(groceries, parameter = list(support = 0.01, confidence = 0.25, minlen = 2)) inspect(groceryrules[1:3])</pre>

Figure 8.8: Apriori association rule learning syntax

Although running the `apriori()` function is straightforward, there can sometimes be a fair amount of trial and error needed to find the `support` and `confidence` parameters that produce a reasonable number of association rules. If you set these levels too high, then you might find no rules, or might find rules that are too generic to be very useful. On the other hand, a threshold too low might result in an unwieldy number of rules. Worse, the operation might take a very long time or run out of memory during the learning phase.

On the `groceries` data, using the default settings of `support = 0.1` and `confidence = 0.8` leads to a disappointing outcome. Although the full output has been omitted for brevity, the end result is a set of zero rules:

```
> apriori(groceries)
```

```
...
```

```
set of 0 rules
```

Obviously, we need to widen the search a bit.



If you think about it, this outcome should not have been terribly surprising. Because `support = 0.1` by default, in order to generate a rule, an item must have appeared in at least $0.1 * 9,385 = 938.5$ transactions. Since only eight items appeared this frequently in our data, it's no wonder we didn't find any rules.

One way to approach the problem of setting a minimum support is to think about the smallest number of transactions needed before a stakeholder would consider a pattern interesting. For instance, one could argue that if an item is purchased twice a day (about 60 times in a month of data) then it may be important. From there, it is possible to calculate the support level needed to find only rules matching at least that many transactions. Since 60 out of 9,835 equals approximately 0.006, we'll try setting the support there first.

Setting the minimum confidence involves a delicate balance. On the one hand, if the confidence is too low, then we might be overwhelmed with many unreliable rules—such as dozens of rules indicating items purchased together frequently by chance alone, like bread and batteries. How would we know where to target our advertising budget then? On the other hand, if we set the confidence too high, then we will be limited to rules that are obvious or inevitable—like the fact that a smoke detector is always purchased in combination with batteries. In this case, moving the smoke detectors closer to the batteries is unlikely to generate additional revenue, since the two items were already almost always purchased together.



The appropriate minimum confidence level depends a great deal on the goals of your analysis. If you start with a conservative value, you can always reduce it to broaden the search if you aren't finding actionable intelligence.

We'll start with a confidence threshold of 0.25, which means that in order to be included in the results, the rule must be correct at least 25 percent of the time. This will eliminate the most unreliable rules while allowing some room for us to modify behavior with targeted promotions.

Now we're ready to generate some rules. In addition to the minimum support and confidence parameters, it is helpful to set `minlen = 2` to eliminate rules that contain fewer than two items. This prevents uninteresting rules from being created simply because the item is purchased frequently, for instance, `{}` => `whole milk`. This rule meets the minimum support and confidence because whole milk is purchased in over 25 percent of transactions, but it isn't a very actionable insight.

The full command for finding a set of association rules using the Apriori algorithm is as follows:

```
> groceryrules <- apriori(groceries, parameter = list(support =
0.006, confidence = 0.25, minlen = 2))
```

The first few lines of output describe the parameter settings we specified, as well as several others that remained set to their defaults; for definitions of these, use the `?APparameter` help command. The second set of lines shows behind-the-scenes algorithmic control parameters that may be helpful for much larger datasets, as they control computing tradeoffs like optimizing for speed or memory use. For information on these parameters, use the `?APcontrol` help command:

```
Apriori
```

```
Parameter specification:
```

confidence	minval	smax	arem	aval	originalSupport	maxtime	support	
0.25	0.1	1	none	FALSE		TRUE	5	0.006
minlen	maxlen	target	ext					
2	10	rules	TRUE					

```
Algorithmic control:
```

filter	tree	heap	memopt	load	sort	verbose
0.1	TRUE	TRUE	FALSE	TRUE	2	TRUE

Next, the output includes information about the steps in the Apriori algorithm execution:

```
Absolute minimum support count: 59
```

```
set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
sorting and recoding items ... [109 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 3 4 done [0.00s].
writing ... [463 rule(s)] done [0.00s].
creating S4 object ... done [0.00s].
```

Given the small size of the transactional dataset, most of the rows show steps that took virtually no time to run—denoted as `[0.00s]` in the output here, but your output may vary slightly depending on computer capability.

The Absolute minimum support count refers to the smallest count of transactions that would meet the support threshold of 0.006 we specified. Since $0.006 * 9,835 = 59.01$, the algorithm requires items to appear in a minimum of 59 transactions. The checking subsets of size 1 2 3 4 output suggests that the algorithm tested *i*-itemsets of 1, 2, 3, and 4 items before stopping the iteration process and writing the final set of 463 rules.

The end result of the `apriori()` function is a `rules` object, which we can peek into by typing its name:

```
> groceryrules
```

```
set of 463 rules
```

Our `groceryrules` object contains quite a large set of association rules! To determine whether any of them are useful, we'll have to dig deeper.

Step 4 – evaluating model performance

To obtain a high-level overview of the association rules, we can use `summary()` as follows. The rule length distribution tells us how many rules have each count of items. In our rule set, 150 rules have only two items, while 297 have three, and 16 have four. The summary statistics associated with this distribution are also provided in the first few lines of output:

```
> summary(groceryrules)
```

```
set of 463 rules
```

```
rule length distribution (lhs + rhs):sizes
```

```
 2   3   4
```

```
150 297 16
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
	2.000	2.000	3.000	2.711	3.000	4.000



As noted in the previous output, the size of the rule is calculated as the total of both the left-hand side (`lhs`) and right-hand side (`rhs`) of the rule. This means that a rule like `{bread} => {butter}` is two items and `{peanut butter, jelly} => {bread}` is three.

Next, we see the summary statistics of the rule quality measures, including support and confidence, as well as coverage, lift, and count:

```
summary of quality measures:
      support      confidence      coverage
Min.    :0.006101   Min.    :0.2500   Min.    :0.009964
1st Qu.:0.007117  1st Qu.:0.2971  1st Qu.:0.018709
Median  :0.008744  Median  :0.3554  Median  :0.024809
Mean    :0.011539  Mean    :0.3786  Mean    :0.032608
3rd Qu.:0.012303  3rd Qu.:0.4495  3rd Qu.:0.035892
Max.    :0.074835  Max.    :0.6600  Max.    :0.255516

      lift      count
Min.    :0.9932   Min.    : 60.0
1st Qu.:1.6229   1st Qu.: 70.0
Median  :1.9332   Median  : 86.0
Mean    :2.0351   Mean    :113.5
3rd Qu.:2.3565   3rd Qu.:121.0
Max.    :3.9565   Max.    :736.0
```

The support and confidence measures should not be very surprising, since we used these as selection criteria for the rules. We might be alarmed if most or all of the rules had support and confidence very near the minimum thresholds, as this would mean that we may have set the bar too high. This is not the case here, as there are many rules with much higher values of each.

The count and coverage measures are closely related to support and confidence. As defined here, **count** is simply the numerator of the support metric or the number (rather than proportion) of transactions that contained the item. Because the absolute minimum support count was 59, it is unsurprising that the minimum observed count of 60 is close to the parameter setting. The maximum count of 736 suggests that an item appeared in 736 out of 9,835 transactions; this relates to the maximum observed support as $736 / 9,835 = 0.074835$.

The **coverage** of an association rule is simply the support of the left-hand side of the rule, but it has a useful real-world interpretation: it can be understood as the chance that a rule applies to any given transaction in the dataset, selected at random. Thus, the minimum coverage of 0.009964 suggests that the least applicable rule covers only about one percent of transactions; the maximum coverage of 0.255516 suggests that at least one rule covers more than 25 percent of transactions. Surely, this rule involves whole milk, as it is the only item that appears in so many transactions.

The final column is a metric we have not considered yet. The **lift** of a rule measures how much more likely one item or itemset is to be purchased relative to its typical rate of purchase, given that you know another item or itemset has been purchased. This is defined by the following equation:

$$\text{lift}(X \rightarrow Y) = \frac{\text{confidence}(X \rightarrow Y)}{\text{support}(Y)}$$



Unlike confidence, where the item order matters, $\text{lift}(X \rightarrow Y)$ is the same as $\text{lift}(Y \rightarrow X)$.

For example, suppose at a grocery store, most people purchase milk and bread. By chance alone, we would expect to find many transactions with both milk and bread. However, if $\text{lift}(\text{milk} \rightarrow \text{bread})$ is greater than 1, this implies that the two items are found together more often than expected by chance alone. In other words, someone who purchases one of the items is more likely to purchase the other. A large lift value is therefore a strong indicator that a rule is important and reflects a true connection between the items, and that the rule will be useful for business purposes. Keep in mind, however, that this is only the case for sufficiently large transactional datasets; lift values can be exaggerated for items with low support.



A pair of authors from the `apriori` package have proposed new metrics called **hyper-lift** and **hyper-confidence** to address the shortcomings of these measures for data with rare items. For more information, see *M. Hahsler and K. Hornik, New Probabilistic Interest Measures for Association Rules (2018)*. <https://arxiv.org/pdf/0803.0966.pdf>.

In the final section of the `summary()` output, we receive mining information, telling us about how the rules were chosen. Here, we see that the `groceries` data, which contained 9,835 transactions, was used to construct rules with a minimum support of 0.006 and minimum confidence of 0.25:

```
mining info:
  data  transactions  support  confidence
  groceries        9835    0.006        0.25
```

We can take a look at specific rules using the `inspect()` function. For instance, the first three rules in the `groceryrules` object can be viewed as follows:

```
> inspect(groceryrules[1:3])
```

```

      lhs          rhs        support
[1] {potted plants} => {whole milk}    0.006914082
[2] {pasta}           => {whole milk}    0.006100661
[3] {herbs}           => {root vegetables} 0.007015760

      confidence coverage   lift      count
[1] 0.4000000  0.01728521 1.565460 68
[2] 0.4054054  0.01504830 1.586614 60
[3] 0.4312500  0.01626843 3.956477 69

```

The first rule can be read in plain language as “if a customer buys potted plants, they will also buy whole milk.” With a support of about 0.007 and a confidence of 0.400, we can determine that this rule covers about 0.7 percent of transactions and is correct in 40 percent of purchases involving potted plants. The lift value tells us how much more likely a customer is to buy whole milk relative to the average customer, given that they bought a potted plant. Since we know that about 25.6 percent of customers bought whole milk (**support**), while 40 percent of customers buying a potted plant bought whole milk (**confidence**), we can compute the lift as $0.40 / 0.256 = 1.56$, which matches the value shown.



Note that the column labeled **support** indicates the support for the rule, not the support for the **lhs** or **rhs** alone. The column labeled **coverage** is the support for the left-hand side.

Although the confidence and lift are high, does $\{potted\ plants\} \rightarrow \{whole\ milk\}$ seem like a very useful rule? Probably not, as there doesn’t seem to be a logical reason why someone would be more likely to buy milk with a potted plant. Yet our data suggests otherwise. How can we make sense of this fact?

A common approach is to take the association rules and divide them into the following three categories:

- Actionable
- Trivial
- Inexplicable

Obviously, the goal of a market basket analysis is to find **actionable** rules that provide a clear and interesting insight. Some rules are clear and others are interesting; it is less common to find a combination of both of these factors.

So-called **trivial** rules include any rules that are so obvious that they are not worth mentioning—they are clear, but not interesting. Suppose you are a marketing consultant being paid large sums of money to identify new opportunities for cross-promoting items. If you report the finding that $\{diapers\} \rightarrow \{formula\}$, you probably won't be invited back for another consulting job.



Trivial rules can also sneak in disguised as more interesting results. For instance, say you found an association between a particular brand of children's cereal and a popular animated movie. This finding is not very insightful if the movie's main character is on the front of the cereal box.

Rules are **inexplicable** if the connection between the items is so unclear that figuring out how to use the information is impossible or nearly impossible. The rule may simply be a random pattern in the data, for instance, a rule stating that $\{pickles\} \rightarrow \{chocolate\ ice\ cream\}$ may be due to a single customer whose pregnant wife had regular cravings for strange combinations of foods.

The best rules are the hidden gems—the undiscovered insights that only seem obvious once discovered. Given enough time, one could evaluate each and every rule to find the gems. However, the data scientists working on the analysis may not be the best judge of whether a rule is actionable, trivial, or inexplicable. Consequently, better rules are likely to arise via collaboration with the domain experts responsible for managing the retail chain, who can help interpret the findings. In the next section, we'll facilitate such sharing by employing methods for sorting and exporting the learned rules so that the most interesting results float to the top.

Step 5 – improving model performance

Subject matter experts may be able to identify useful rules very quickly, but it would be a poor use of their time to ask them to evaluate hundreds or thousands of rules. Therefore, it's useful to be able to sort the rules according to different criteria and get them out of R in a form that can be shared with marketing teams and examined in more depth. In this way, we can improve the performance of our rules by making the results more actionable.

If you are running into memory limitations or if Apriori is taking too long to run, it may also be possible to improve the computational performance of the association rule mining process itself by using a more recent algorithm.

Sorting the set of association rules

Depending upon the objectives of the market basket analysis, the most useful rules might be those with the highest support, confidence, or lift. The arules package works with the R `sort()` function to allow reordering of the list of rules so that those with the highest or lowest values of the quality measure come first.

To reorder the `groceryrules` object, we can `sort()` while specifying a value of "support", "confidence", or "lift" to the `by` parameter. By combining the `sort` with vector operators, we can obtain a specific number of interesting rules. For instance, the best five rules according to the lift statistic can be examined using the following command:

```
> inspect(sort(groceryrules, by = "lift")[1:5])
```

The output is as follows:

lhs	rhs	support	
[1] {herbs}	=> {root vegetables}	0.007015760	
[2] {berries}	=> {whipped/sour cream}	0.009049314	
[3] {other vegetables, tropical fruit, whole milk}	=> {root vegetables}	0.007015760	
[4] {beef, other vegetables}	=> {root vegetables}	0.007930859	
[5] {other vegetables, tropical fruit}	=> {pip fruit}	0.009456024	
confidence	coverage	lift	count
[1] 0.4312500	0.01626843	3.956477	69
[2] 0.2721713	0.03324860	3.796886	89
[3] 0.4107143	0.01708185	3.768074	69
[4] 0.4020619	0.01972547	3.688692	78
[5] 0.2634561	0.03589222	3.482649	93

These rules appear to be more interesting than the ones we looked at previously. The first rule, with a lift of about 3.96, implies that people who buy herbs are nearly four times more likely to buy root vegetables than the typical customer—perhaps for a stew of some sort. Rule two is also interesting. Whipped cream is over three times more likely to be found in a shopping cart with berries versus other carts, suggesting perhaps a dessert pairing.



By default, the sort order is decreasing, meaning the largest values come first. To reverse this order, add an additional parameter, `decreasing = FALSE`.

Taking subsets of association rules

Suppose that, given the preceding rule, the marketing team is excited about the possibility of creating an advertisement to promote berries, which are now in season. Before finalizing the campaign, however, they ask you to investigate whether berries are often purchased with other items. To answer this question, we'll need to find all the rules that include berries in some form.

The `subset()` function provides a method for searching for subsets of transactions, items, or rules. To use it to find any rules with berries appearing in the rule, use the following command. This will store the rules in a new object named `berryrules`:

```
> berryrules <- subset(groceryrules, items %in% "berries")
```

We can then inspect the rules as we had done with the larger set:

```
> inspect(berryrules)
```

The result is the following set of rules:

```
    lhs          rhs           support
[1] {berries} => {whipped/sour cream} 0.009049314
[2] {berries} => {yogurt}            0.010574479
[3] {berries} => {other vegetables} 0.010269446
[4] {berries} => {whole milk}       0.011794611

    confidence coverage   lift      count
[1] 0.2721713  0.0332486 3.796886  89
[2] 0.3180428  0.0332486 2.279848 104
[3] 0.3088685  0.0332486 1.596280 101
[4] 0.3547401  0.0332486 1.388328 116
```

There are four rules involving berries, two of which seem to be interesting enough to be called actionable. In addition to whipped cream, berries are also purchased frequently with yogurt—a pairing that could serve well for breakfast or lunch, as well as dessert.

The `subset()` function is very powerful. The criteria for choosing the subset can be defined with several keywords and operators:

- The keyword `items`, explained previously, matches an item appearing anywhere in the rule. To limit the subset to where the match occurs only on the left-hand side or right-hand side, use `lhs` or `rhs` instead.
- The operator `%in%` means that at least one of the items must be found in the list you defined. If you wanted any rules matching either `berries` or `yogurt`, you could write `items %in% c("berries", "yogurt")`.
- Additional operators are available for partial matching (`%pin%`) and complete matching (`%ain%`). Partial matching allows you to find both citrus fruit and tropical fruit using one search: `items %pin% "fruit"`. Complete matching requires that all listed items are present. For instance, `items %ain% c("berries", "yogurt")` finds only rules with both `berries` and `yogurt`.
- Subsets can also be limited by `support`, `confidence`, or `lift`. For instance, `confidence > 0.50` would limit the rules to those with confidence greater than 50 percent.
- Matching criteria can be combined with standard R logical operators such as `AND (&)`, `OR (|)`, and `NOT (!)`.

Using these options, you can limit the selection of rules to be as specific or general as you would like.

Saving association rules to a file or data frame

To share the results of your market basket analysis, you can save the rules to a CSV file with the `write()` function. This will produce a CSV file that can be used in most spreadsheet programs, including Microsoft Excel:

```
> write(groceryrules, file = "groceryrules.csv",
       sep = ",", quote = TRUE, row.names = FALSE)
```

Sometimes it is also convenient to convert the rules into an R data frame. This can be accomplished using the `as()` function, as follows:

```
> groceryrules_df <- as(groceryrules, "data.frame")
```

This creates a data frame with the rules in character format, and numeric vectors for support, confidence, coverage, lift, and count:

```
> str(groceryrules_df)
'data.frame': 463 obs. of 6 variables:
```

```
$ rules      : chr  "{potted plants} => {whole milk}"
  "{pasta} => {whole milk}"  "{herbs} => {root vegetables}"
  "{herbs} => {other vegetables}" ...
$ support    : num  0.00691 0.0061 0.00702 0.00773 0.00773 ...
$ confidence: num  0.4 0.405 0.431 0.475 0.475 ...
$ coverage   : num  0.0173 0.015 0.0163 0.0163 0.0163 ...
$ lift       : num  1.57 1.59 3.96 2.45 1.86 ...
$ count      : int  68 60 69 76 69 70 67 63 88 ...
```

Saving the rules to a data frame may be useful if you want to perform additional processing on the rules or need to export them to another database.

Using the Eclat algorithm for greater efficiency

The **Eclat algorithm**, which is named for its use of “equivalence class itemset clustering and bottom-up lattice traversal” methods, is a slightly more modern and substantially faster association rule learning algorithm. While the implementation details are outside the scope of this book, it can be understood as a close relative of Apriori; it too assumes all subsets of frequent itemsets are also frequent. However, Eclat is able to search even fewer subsets by utilizing clever tricks that provide shortcuts to identify the potentially maximal frequent itemsets and search only these itemsets’ subsets. Where Apriori is a form of a breadth-first algorithm because it searches wide before it searches deep, Eclat is considered a depth-first algorithm in that it dives to the final endpoint and searches only as wide as needed. For some use cases, this can lead to a performance gain of an order of magnitude and less memory consumed.



For more information on Eclat, refer to *New Algorithms for Fast Discovery of Association Rules*, Zaki, M.J., Parthasarathy, S., Oghara, M., Li, W., KDD-97 Proceedings, 1997.

A key tradeoff with Eclat’s fast searching is that it skips the phase in Apriori in which confidence is calculated. It assumes that once the itemsets with high support are obtained, the most useful associations can be identified later—whether manually via a subjective eyeball test, or via another round of processing to compute metrics like confidence and lift. This being said, the `arules` package makes it just as easy to apply Eclat as Apriori, despite the additional step in the process.

We begin with the `eclat()` function and setting the `support` parameter to 0.006 as before; however, note that the `confidence` is not set at this stage:

```
> groceryitemsets_eclat <- eclat(groceries, support = 0.006)
```

Some of the output is omitted here, but the last few lines are similar to what we obtained from the `apriori()` function, with the key exception that 747 itemsets were written rather than 463 rules:

```
Absolute minimum support count: 59

create itemset ...
set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
sorting and recoding items ... [109 item(s)] done [0.00s].
creating sparse bit matrix ... [109 row(s), 9835 column(s)] done [0.00s].
writing ... [747 set(s)] done [0.02s].
Creating S4 object ... done [0.00s].
```

The resulting Eclat itemset object can be used with the `inspect()` function as we did with the Apriori rules object. The following command shows the first five itemsets:

```
> inspect(groceryitemsets_eclat[1:5])
```

items	support	count
[1] {potted plants, whole milk}	0.006914082	68
[2] {pasta, whole milk}	0.006100661	60
[3] {herbs, whole milk}	0.007727504	76
[4] {herbs, other vegetables}	0.007727504	76
[5] {herbs, root vegetables}	0.007015760	69

To produce rules from the itemsets, use the `ruleInduction()` function with the desired `confidence` parameter value as follows:

```
> groceryrules_eclat <- ruleInduction(groceryitemsets_eclat,
  confidence = 0.25)
```

With `support` and `confidence` set to the earlier values of 0.006 and 0.25, respectively, it is no surprise that the Eclat algorithm produced the same set of 463 rules as Apriori:

```
> groceryrules_eclat
set of 463 rules
```

The resulting `rules` object can be inspected just as before:

```
> inspect(groceryrules_eclat[1:5])

lhs                      rhs                      support
[1] {potted plants} => {whole milk}           0.006914082
```

```
[2] {pasta}      => {whole milk}      0.006100661
[3] {herbs}      => {whole milk}      0.007727504
[4] {herbs}      => {other vegetables} 0.007727504
[5] {herbs}      => {root vegetables}  0.007015760

  confidence lift
[1] 0.4000000 1.565460
[2] 0.4054054 1.586614
[3] 0.4750000 1.858983
[4] 0.4750000 2.454874
[5] 0.4312500 3.956477
```

Given the ease of use with either method, if you have a very large transactional dataset, it may be worth testing Eclat and Apriori on smaller random samples of transactions to see if one outperforms the other.

Summary

Association rules are used to find insight into the massive transaction databases of large retailers. As an unsupervised learning process, association rule learners are capable of extracting knowledge from large databases without any prior knowledge of what patterns to seek. The catch is that it takes some effort to reduce the wealth of information into a smaller and more manageable set of results. The Apriori algorithm, which we studied in this chapter, does so by setting minimum thresholds of interestingness and reporting only the associations meeting these criteria.

We put the Apriori algorithm to work while performing a market basket analysis for a month's worth of transactions at a modestly sized supermarket. Even in this small example, a wealth of associations was identified. Among these, we noted several patterns that may be useful for future marketing campaigns. The same methods we applied are used at much larger retailers on databases many times this size, and can also be applied to projects outside of a retail setting.

In the next chapter, we will examine another unsupervised learning algorithm. Just like association rules, it is intended to find patterns within data. But unlike association rules that seek groups of related items or features, the methods in the next chapter are concerned with finding connections and relationships among the examples.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



9

Finding Groups of Data – Clustering with k-means

Have you ever spent time watching a crowd? If so, you have likely seen some recurring personalities. Perhaps a certain type of person, identified by a freshly pressed suit and a briefcase, comes to typify the “fat cat” business executive. A 20-something wearing skinny jeans, a flannel shirt, and sunglasses might be dubbed a “hipster,” while a woman unloading children from a minivan may be labeled a “soccer mom.”

Of course, these types of stereotypes are dangerous to apply to individuals, as no two people are exactly alike. Yet, understood as a way to describe a collective, the labels capture some underlying aspect of similarity shared among the individuals within the group.

As you will soon learn, the act of clustering, or spotting patterns in data, is not much different from spotting patterns in groups of people. This chapter describes:

- The ways clustering tasks differ from the classification tasks we examined previously
- How clustering defines a group and how such groups are identified by k-means, a classic and easy-to-understand clustering algorithm
- The steps needed to apply clustering to a real-world task of identifying marketing segments among teenage social media users

Before jumping into action, we’ll begin by taking an in-depth look at exactly what clustering entails.

Understanding clustering

Clustering is an unsupervised machine learning task that automatically divides the data into **clusters**, or groups of similar items. It does this without having been told how the groups should look ahead of time. Because we do not tell the machine specifically what we’re looking for, clustering is used for knowledge discovery rather than prediction. It provides an insight into the natural groupings found within data.

Without advanced knowledge of what comprises a cluster, how can a computer possibly know where one group ends and another begins? The answer is simple: clustering is guided by the principle that items inside a cluster should be very similar to each other, but very different from those outside. The definition of similarity might vary across applications, but the basic idea is always the same: group the data such that related elements are placed together.

The resulting clusters can then be used for action. For instance, you might find clustering methods employed in applications such as:

- Segmenting customers into groups with similar demographics or buying patterns for targeted marketing campaigns
- Detecting anomalous behavior, such as unauthorized network intrusions, by identifying patterns of use falling outside known clusters
- Simplifying extremely “wide” datasets—those with a large number of features—by creating a small number of categories to describe rows with relatively homogeneous values of the features

Overall, clustering is useful whenever diverse and varied data can be exemplified by a much smaller number of groups. It results in meaningful and actionable data structures, which reduce complexity and provide insight into patterns of relationships.

Clustering as a machine learning task

Clustering is somewhat different from the classification, numeric prediction, and pattern detection tasks we’ve examined so far. In each of these tasks, the goal was to build a model that relates features to an outcome, or to relate some features to other features. Each of these tasks describes existing patterns within data. In contrast, the goal of clustering is to create new data. In clustering, unlabeled examples are given a new cluster label, which has been inferred entirely from the relationships within the data. For this reason, you will sometimes see a clustering task referred to as **unsupervised classification** because, in a sense, it classifies unlabeled examples.

The catch is that the class labels obtained from an unsupervised classifier are without intrinsic meaning. Clustering will tell you which groups of examples are closely related—for instance, it might return groups A, B, and C—but it's up to you to apply an actionable and meaningful label, and to tell the story of what makes an “A” different from a “B.” To see how this impacts the clustering task, let's consider a simple hypothetical example.

Suppose you were organizing a conference on the topic of data science. To facilitate professional networking and collaboration, you planned to seat people at one of three tables according to their research specialties. Unfortunately, after sending out the conference invitations, you realize that you forgot to include a survey asking which discipline the attendee would prefer to be seated within.

In a stroke of brilliance, you realize that you might be able to infer each scholar's research specialty by examining their publication history. To this end, you begin collecting data on the number of articles each attendee has published in computer science-related journals and the number of articles published in math- or statistics-related journals. Using the data collected for the scholars, you create a scatterplot:

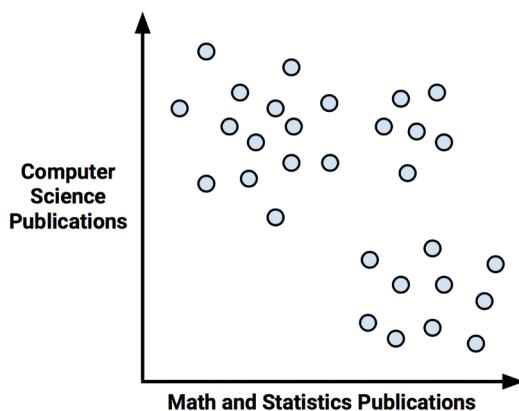


Figure 9.1: Visualizing scholars by their math and computer science publication data

As expected, there seems to be a pattern. We might guess that the upper-left corner, which represents people with many computer science publications but few articles on math, is a cluster of computer scientists. Following this logic, the lower-right corner might be a group of mathematicians or statisticians. Similarly, the upper-right corner, those with both math and computer science experience, may be machine learning experts.

Applying these labels results in the following visualization:

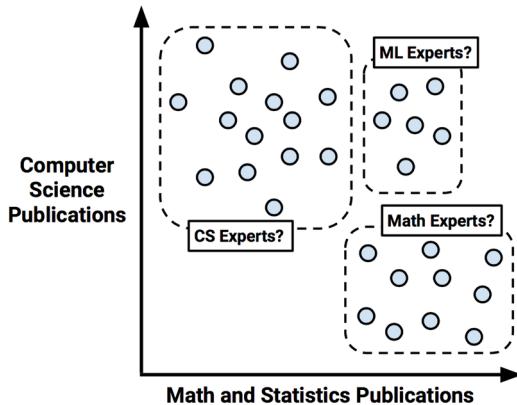


Figure 9.2: Clusters can be identified based on presumptions about the scholars in each group

Our groupings were formed visually; we simply identified clusters as closely grouped data points. Yet, despite the seemingly obvious groupings, without personally asking each scholar about their academic specialty, we have no way to know whether the groups are truly homogeneous. The labels are qualitative, presumptive judgments about the types of people in each group, based on a limited set of quantitative data.

Rather than defining the group boundaries subjectively, it would be nice to use machine learning to define them objectively. Given the axis-parallel splits in the previous figure, our problem seems like an obvious application for decision trees, as described in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*. This would provide us with a clean rule like “if a scholar has few math publications, then they are a computer science expert.” Unfortunately, there’s a problem with this plan. Without data on the true class value for each point, a supervised learning algorithm would have no ability to learn such a pattern, as it would have no way of knowing what splits would result in homogenous groups.

In contrast to supervised learning, clustering algorithms use a process very similar to what we did by visually inspecting the scatterplot. Using a measure of how closely the examples are related, homogeneous groups can be identified. In the next section, we’ll start looking at how clustering algorithms are implemented.



This example highlights an interesting application of clustering. If you begin with unlabeled data, you can use clustering to create class labels. From there, you could apply a supervised learner such as decision trees to find the most important predictors of these classes. This is an example of semi-supervised learning as described in *Chapter 1, Introducing Machine Learning*.

Clusters of clustering algorithms

Just as there are many approaches to building a predictive model, there are multiple approaches to performing the descriptive task of clustering. Many such methods are listed on the following site, the CRAN task view for clustering: <https://cran.r-project.org/view=Cluster>. Here, you will find numerous R packages used for discovering natural groupings in data. The various algorithms are distinguished mainly by two characteristics:

- The **similarity metric**, which provides the quantitative measure of how closely two examples are related
- The **agglomeration function**, which governs the process of assigning examples to clusters based on their similarity

Even though there may be subtle differences between the approaches, they can of course be clustered in various ways. Multiple such typologies exist, but a simple three-part framework helps understand the main distinctions. Using this approach, the three main clusters of clustering algorithms, listed from simplest to most sophisticated, are as follows:

- **Hierarchical methods**, which create a family tree-style hierarchy that positions the most similar examples more closely in the graph structure
- **Partition-based methods**, which treat the examples as points in multidimensional space, and attempt to find boundaries in this space that lead to relatively homogenous groups
- **Model or density-based methods**, which rely on statistical principles and/or the density of points to discover fuzzy boundaries between the clusters; in some cases, examples may be partially assigned to multiple clusters, or even to no cluster at all

Despite **hierarchical clustering** being the simplest of the methods, it is not without a pair of interesting upsides. Firstly, it results in a hierarchical graph visualization called a **dendrogram**, which depicts the associations between examples such that the most similar examples are positioned more closely in the hierarchy.

This can be a useful tool to understand which examples and subsets of examples are the most tightly grouped. Secondly, hierarchical clustering does not require a predefined expectation of how many clusters exist in the dataset. Instead, it implements a process in which, at one extreme, every example is included in a single, giant cluster with all other examples; at the other extreme, every example is found in a tiny cluster containing only itself; and in between, examples may be included in other clusters of varying sizes.

Figure 9.3 illustrates a hypothetical dendrogram for a simple dataset containing eight examples, labeled A through H. Notice that the most closely related examples (depicted via proximity on the x axis) are linked more closely as siblings in the diagram. For instance, examples D and E are the most similar and thus are the first to be grouped. However, all eight examples are eventually linked to one large cluster, or may be included in any number of clusters in between. Slicing the dendrogram horizontally at different positions creates varying numbers of clusters, as shown for three and five clusters:

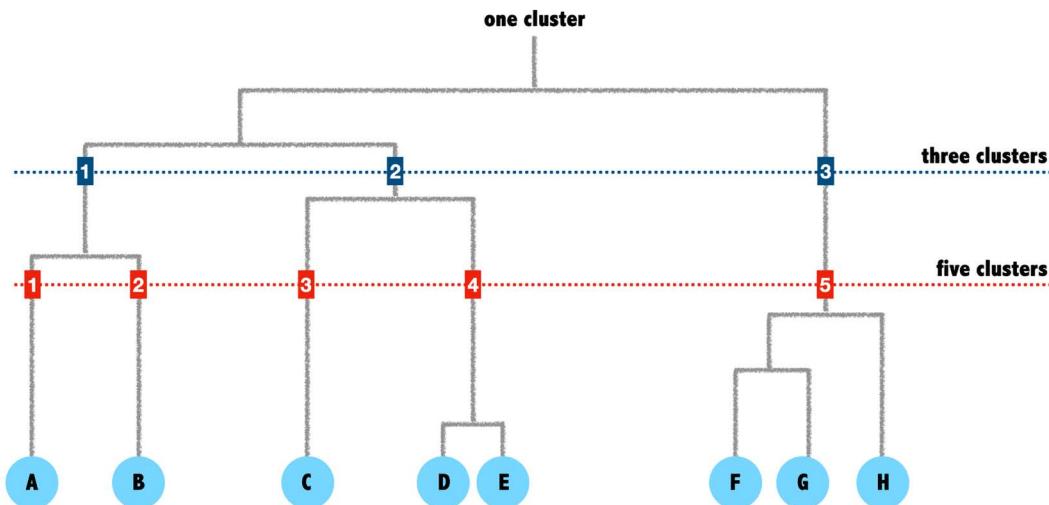


Figure 9.3: Hierarchical clustering produces a dendrogram that depicts the natural groupings for the desired number of clusters

The dendrogram for hierarchical clustering can be grown using “bottom-up” or “top-down” approaches. The former is called **agglomerative clustering**, and begins with each example in its own cluster, then connects the most similar examples first until all examples are connected in a single cluster. The latter is called **divisive clustering**, and begins with a single large cluster and ends with all examples in their own individual clusters.

When connecting examples to groups of examples, different metrics may be used, such as the example's similarity to the most similar, least similar, or average member of the group. A more complex metric known as **Ward's method** does not use similarity between examples, but instead considers a measure of cluster homogeneity to construct the linkages. In any case, the result is a hierarchy that aims to group the most similar examples into any number of subgroups.

The flexibility of the hierarchical clustering technique comes at a cost, which is computational complexity due to the need to compute the similarity between each example and every other. As the number of examples (N) grows, the number of calculations grows as $N^*N = N^2$, as does the memory needed for the similarity matrix that stores the result. For this reason, hierarchical clustering is used only on very small datasets and is not demonstrated in this chapter. However, the `hclust()` function included in R's `stats` package provides a simple implementation, which is installed with R by default.



Clever implementations of divisive clustering have the potential to be slightly more computationally efficient than agglomerative clustering as the algorithm may stop early if it is unnecessary to create larger numbers of clusters. This being said, both agglomerative and divisive clustering are examples of “greedy” algorithms as defined in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, because they use data on a first-come, first-served basis and are thus not guaranteed to produce the overall optimal set of clusters for a given dataset.

Partition-based clustering methods have a distinct efficiency advantage over hierarchical clustering in that they apply heuristic methods to divide the data into clusters without the need to evaluate the similarity between every pair of examples. We'll explore a widely used partition-based method in greater detail shortly, but for now it suffices to understand that this method is concerned with finding boundaries between clusters rather than connecting examples to one another—an approach that requires far fewer comparisons between examples. This heuristic can be quite computationally efficient, but one caveat is that it is somewhat rigid or even arbitrary when it comes to group assignments. For example, if five clusters are requested, it will partition examples into all five clusters; if some examples fall on the boundary between two clusters, these will be placed somewhat arbitrarily yet firmly into one cluster or the other. Similarly, if four or six clusters might have split the data better, this would not be as apparent as it would be with the hierarchical clustering dendrogram.

The more sophisticated **model-based** and **density-based clustering** methods address some of these issues of inflexibility by estimating the probability that an example belongs to each cluster, rather than merely assigning it to a single cluster. Some of them may allow the cluster boundaries to follow the natural patterns identified in the data rather than forcing a strict divide between the groups. Model-based approaches often assume a statistical distribution from which the examples are believed to have been pulled.

One such approach, known as **mixture modeling**, attempts to disentangle datasets composed of examples pulled from a mixture of statistical distributions—typically Gaussian (the normal bell curve). For example, imagine you have a dataset composed of voice data from a mixture of male and female vocal registers, as depicted in *Figure 9.4* (note that the distributions are hypothetical and are not based on real-world data). Although there is some overlap between the two, the average male tends to have a lower register than the average female.

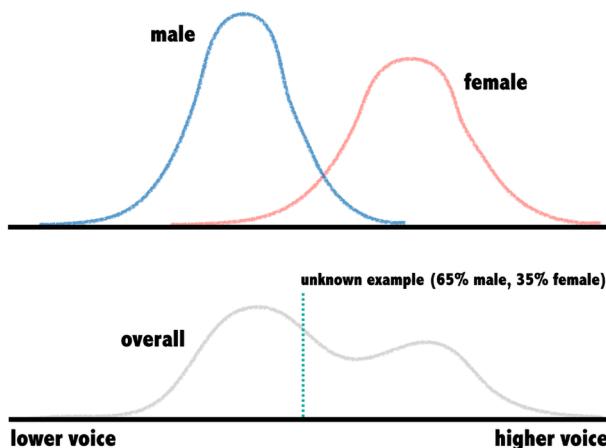


Figure 9.4: Mixture modeling assigns each example a probability of belonging to one of the underlying distributions

Given the unlabeled overall distribution (the bottom part of the figure), a mixture model would be capable of assigning a probability that any given example belongs to the cluster of males or the cluster of females, incredibly, without ever having been trained separately on male or female voices in the top part of the figure! This is possible by discovering the statistical parameters like the mean and standard deviation that are most likely to have generated the observed overall distribution, under the assumption that a specific number of distinct distributions were involved—in this case, two Gaussian distributions.

As an unsupervised method, the mixture model would have no way of knowing that the left distribution is males and the right distribution is females, but this would be readily apparent to a human observer comparing the records with a high likelihood of males being in the left cluster versus the right one. The downside of this technique is that not only does it require knowledge of how many distributions are involved but it also requires an assumption of the types of distributions. This may be too rigid for many real-world clustering tasks.

Another powerful clustering technique called **DBSCAN** is named after the “density-based spatial clustering of applications with noise” approach it uses to identify natural clusters in data. This award-winning technique is incredibly flexible and does well with many of the challenges of clustering, such as adapting to the dataset’s natural number of clusters, being flexible about the boundaries between clusters, and not assuming a particular statistical distribution for the data.

While the implementation details are beyond the scope of this book, the DBSCAN algorithm can be understood intuitively as a process of creating neighborhoods of examples that are all within a given radius of other examples in the cluster. A predefined number of **core points** within a specified radius forms the initial cluster nucleus, and points that are within a specified radius of any of the core points are then added to the cluster and comprise the outermost boundary of the cluster. Unlike many other clustering algorithms, some examples will not be assigned any cluster at all, as any points that are not close enough to a core point will be treated as noise.

Although DBSCAN is powerful and flexible, it may require experimentation to optimize the parameters to fit the data, such as the number of points comprising the core, or the allowed radius between points, which adds time complexity to the machine learning project. Of course, just because model-based methods are more sophisticated does not imply they are the best fit for every clustering project. As we will see throughout the remainder of this chapter, a simpler partition-based method can perform surprisingly well on a challenging real-world clustering task.



Although mixture modeling and DBSCAN are not demonstrated in this chapter, there are R packages that can be used to apply these methods to your own data. The `mclust` package fits a model to mixtures of Gaussian distributions, and the `dbSCAN` package provides a fast implementation of the DBSCAN algorithm.

The k-means clustering algorithm

The **k-means algorithm** is perhaps the most often used clustering method and is an example of a partition-based clustering approach. Having been studied for several decades, it serves as the foundation for many more sophisticated clustering techniques. If you understand the simple principles it uses, you will have the knowledge needed to understand nearly any clustering algorithm in use today.



As k-means has evolved over time, there are many implementations of the algorithm. An early approach is described in *A k-means clustering algorithm*, Hartigan, J.A., Wong, M.A., *Applied Statistics*, 1979, Vol. 28, pp. 100-108.

Even though clustering methods have evolved since the inception of k-means, this is not to imply that k-means is obsolete. In fact, the method may be more popular now than ever. The following table lists some reasons why k-means is still used widely:

Strengths	Weaknesses
<ul style="list-style-type: none"> • Uses simple principles that can be explained in non-statistical terms • Highly flexible and can be adapted with simple adjustments to address many of its shortcomings • Performs well enough under many real-world use cases 	<ul style="list-style-type: none"> • Not as sophisticated as more modern clustering algorithms • Because it uses an element of random chance, it is not guaranteed to find the optimal set of clusters • Requires a reasonable guess as to how many clusters naturally exist in the data • Not ideal for non-spherical clusters or clusters of widely varying density

If the name k-means sounds familiar to you, you may be recalling the **k-nearest neighbors (k-NN)** algorithm presented in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*. As you will soon see, k-means has more in common with k-NN than just the letter k.

The k-means algorithm assigns each of the n examples to one of the k clusters, where k is a number that has been determined ahead of time. The goal is to minimize the differences in feature values of examples within each cluster and maximize the differences between clusters.

Unless k and n are extremely small, it is not feasible to compute the optimal clusters across all possible combinations of examples. Instead, the algorithm uses a heuristic process that finds **locally optimal** solutions. Put simply, this means that it starts with an initial guess for the cluster assignments and then modifies the assignments slightly to see if the changes improve the homogeneity within the clusters.

We will cover the process in depth shortly, but the algorithm essentially involves two phases. First, it assigns examples to an initial set of k clusters. Then, it updates the assignments by adjusting the cluster boundaries according to the examples that currently fall into the cluster. The process of updating and assigning occurs several times until changes no longer improve the cluster fit. At this point, the process stops, and the clusters are finalized.



Due to the heuristic nature of k-means, you may end up with somewhat different results by making only slight changes to the starting conditions. If the results vary dramatically, this could indicate a problem. For instance, the data may not have natural groupings, or the value of k has been poorly chosen. With this in mind, it's a good idea to try a cluster analysis more than once to test the robustness of your findings.

To see how the process of assigning and updating works in practice, let's revisit the case of the hypothetical data science conference. Though this is a simple example, it will illustrate the basics of how k-means operates under the hood.

Using distance to assign and update clusters

As with k-NN, k-means treats feature values as coordinates in a multidimensional feature space. For the conference data, there are only two features, so we can represent the feature space as a two-dimensional scatterplot as depicted previously.

The k-means algorithm begins by choosing k points in the feature space to serve as the cluster centers. These centers are the catalyst that spurs the remaining examples to fall into place. Often, the points are chosen by selecting k random examples from the training dataset. Because we hope to identify three clusters, using this method, $k = 3$ points will be selected at random.

These points are indicated by the star, triangle, and diamond in *Figure 9.5*:

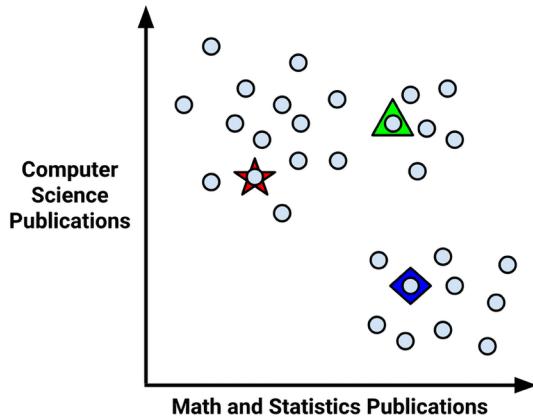


Figure 9.5: k-means clustering begins by selecting k random cluster centers

It's worth noting that although the three cluster centers in the preceding diagram happen to be widely spaced apart, this will not always necessarily be the case. Because the starting points are selected at random, the three centers could have just as easily been three adjacent points. Combined with the fact that the k -means algorithm is highly sensitive to the starting position of the cluster centers, a good or bad set of initial cluster centers may have a substantial impact on the final set of clusters.

To address this problem, k -means can be modified to use different methods for choosing the initial centers. For example, one variant chooses random values occurring anywhere in the feature space rather than only selecting among values observed in the data. Another option is to skip this step altogether; by randomly assigning each example to a cluster, the algorithm can jump ahead immediately to the update phase. Each of these approaches adds a particular bias to the final set of clusters, which you may be able to use to improve your results.



In 2007, an algorithm called **k -means++** was introduced, which proposes an alternative method for selecting the initial cluster centers. It purports to be an efficient way to get much closer to the optimal clustering solution while reducing the impact of random chance. For more information, see *k -means++: The advantages of careful seeding*, Arthur, D, Vassilvitskii, S, *Proceedings of the eighteenth annual ACM-SIAM symposium on discrete algorithms*, 2007, pp. 1,027–1,035.

After choosing the initial cluster centers, the other examples are assigned to the cluster center that is nearest according to a distance function, which is used as a measure of similarity. You may recall that we used distance functions as similarity measures while learning about the k-NN supervised learning algorithm. Like k-NN, k-means traditionally uses Euclidean distance, but other distance functions can be used if desired.



Interestingly, any function that returns a numeric measure of similarity could be used instead of a traditional distance function. In fact, k-means could even be adapted to cluster images or text documents by using a function that measures the similarity of pairs of images or texts.

To apply the distance function, recall that if n indicates the number of features, the formula for Euclidean distance between example x and example y is as follows:

$$\text{dist}(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

For instance, to compare a guest with five computer science publications and one math publication to a guest with zero computer science papers and two math papers, we could compute this in R as:

```
> sqrt((5 - 0)^2 + (1 - 2)^2)
```

```
[1] 5.09902
```

Using the distance function in this way, we find the distance between each example and each cluster center. Each example is then assigned to the nearest cluster center.



Keep in mind that because we are using distance calculations, all the features need to be numeric, and the values should be normalized to a standard range ahead of time. The methods presented in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, will prove helpful for this task.

As shown in the following figure, the three cluster centers partition the examples into three partitions labeled *Cluster A*, *Cluster B*, and *Cluster C*. The dashed lines indicate the boundaries for the **Voronoi diagram** created by the cluster centers. The Voronoi diagram indicates the areas that are closer to one cluster center than any other; the vertex where all three boundaries meet is the maximal distance from all three cluster centers.

Using these boundaries, we can easily see the regions claimed by each of the initial k-means seeds:

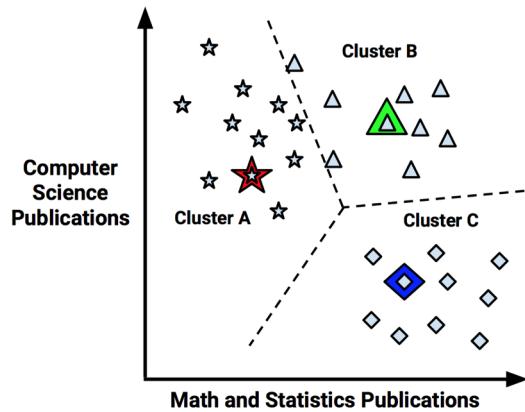


Figure 9.6: The initial cluster centers create three groups of “nearest” points

Now that the initial assignment phase has been completed, the k-means algorithm proceeds to the update phase. The first step of updating the clusters involves shifting the initial centers to a new location, known as the **centroid**, which is calculated as the average position of the points currently assigned to that cluster. The following figure illustrates how as the cluster centers shift to the new centroids, the boundaries in the Voronoi diagram also shift, and a point that was once in *Cluster B* (indicated by an arrow) is added to *Cluster A*:

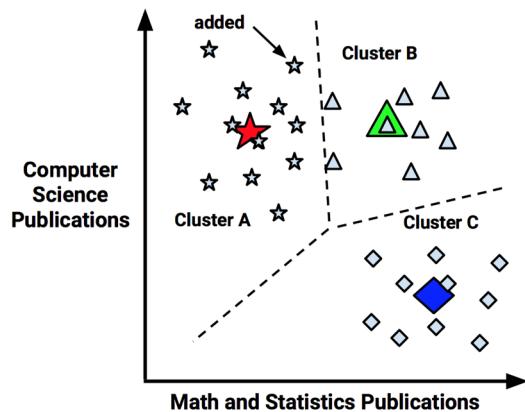


Figure 9.7: The update phase shifts the cluster centers, which causes the reassignment of one point

As a result of this reassignment, the k-means algorithm will continue through another update phase. After shifting the cluster centroids, updating the cluster boundaries, and reassigning points into new clusters (as indicated by arrows), the figure looks like this:

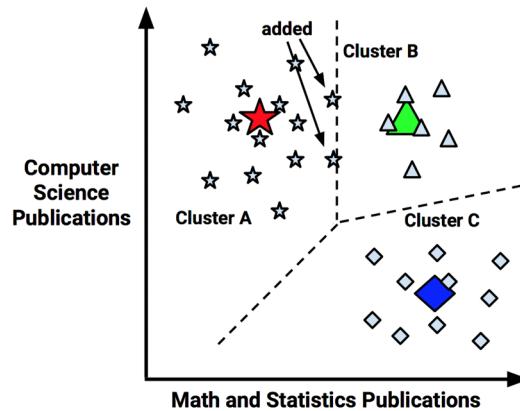


Figure 9.8: After another update, two more points are reassigned to the nearest cluster center

Because two more points were reassigned, another update must occur, which moves the centroids and updates the cluster boundaries. However, because these changes result in no reassessments, the k-means algorithm stops. The cluster assignments are now final:

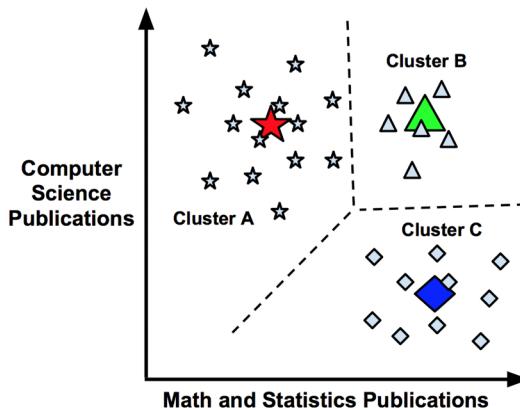


Figure 9.9: Clustering stops after the update phase results in no new cluster assignments

The final clusters can be reported in one of two ways. First, you might simply report the cluster assignments of A, B, or C for each example. Alternatively, you could report the coordinates of the cluster centroids after the final update.

Given either reporting method, you can compute the other; you can calculate the centroids using the coordinates of each cluster's examples, or you can use the centroid coordinates to assign each example to its nearest cluster center.

Choosing the appropriate number of clusters

In the introduction to k-means, we learned that the algorithm is sensitive to the randomly chosen cluster centers. Indeed, if we had selected a different combination of three starting points in the previous example, we may have found clusters that split the data differently from what we had expected. Similarly, k-means is sensitive to the number of clusters; the choice requires a delicate balance. Setting k to be very large will improve the homogeneity of the clusters and, at the same time, it risks overfitting the data.

Ideally, you will have *a priori* knowledge (a prior belief) about the true groupings and you can apply this information to choose the number of clusters. For instance, if you clustered movies, you might begin by setting k equal to the number of genres considered for the Academy Awards. In the data science conference seating problem that we worked through previously, k might reflect the number of academic fields of study that invitees belong to.

Sometimes, the number of clusters is dictated by business requirements or the motivation for the analysis. For example, the number of tables in the meeting hall might dictate how many groups of people should be created from the data science attendee list. Extending this idea to another business case, if the marketing department only has the resources to create three distinct advertising campaigns, it might make sense to set $k = 3$ to assign all the potential customers to one of the three appeals.

Without any prior knowledge, one rule of thumb suggests setting k equal to the square root of $(n/2)$, where n is the number of examples in the dataset. However, this rule of thumb is likely to result in an unwieldy number of clusters for large datasets. Luckily, there are other quantitative methods that can assist in finding a suitable k-means cluster set.

A technique known as the **elbow method** attempts to gauge how the homogeneity or heterogeneity within the clusters changes for various values of k . As illustrated in the following diagrams, the homogeneity within clusters is expected to increase as additional clusters are added; similarly, the heterogeneity within clusters should decrease with more clusters. Because you could continue to see improvements until each example is in its own cluster, the goal is not to maximize homogeneity or minimize heterogeneity endlessly, but rather to find k such that there are diminishing returns beyond that value. This value of k is known as the **elbow point** because it bends like the elbow joint of the human arm.

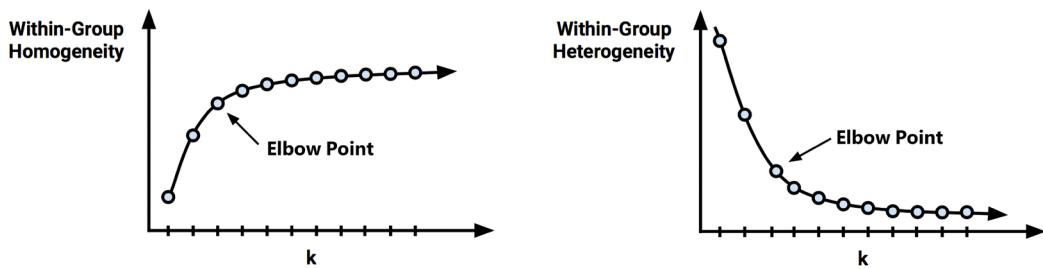


Figure 9.10: The elbow is the point at which increasing k results in relatively small improvements

There are numerous statistics for measuring homogeneity and heterogeneity within clusters that can be used with the elbow method (the information box that follows provides a citation for more detail). Still, in practice, it is not always feasible to iteratively test a large number of k values. This is in part because clustering large datasets can be fairly time-consuming; clustering the data repeatedly is even worse. Furthermore, applications requiring the exact optimal set of clusters are rare. In most clustering applications, it suffices to choose a k value based on convenience rather than the one that creates the most homogenous clusters.



For a thorough review of the vast assortment of cluster performance measures, refer to *On Clustering Validation Techniques*, Halkidi, M, Batistakis, Y, Vazirgiannis, M, *Journal of Intelligent Information Systems*, 2001, Vol. 17, pp. 107-145.

The process of setting k itself can sometimes lead to interesting insights. By observing how the characteristics of the clusters change as k changes, one might infer where the data has naturally defined boundaries. Groups that are more tightly clustered will change very little, while less homogeneous groups will form and disband over time.

In general, it may be wise to spend little time worrying about getting k exactly right. The next example will demonstrate how even a tiny bit of subject-matter knowledge borrowed from a Hollywood film can be used to set k such that actionable and interesting clusters are found. As clustering is unsupervised, the task is really about what you make of it; the value is in the insights you take away from the algorithm's findings.

Finding teen market segments using k-means clustering

Interacting with friends on a **social networking service (SNS)**, such as Facebook, TikTok, and Instagram, has become a rite of passage for teenagers around the world. Having a relatively large amount of disposable income, these adolescents are a coveted demographic for businesses hoping to sell snacks, beverages, electronics, entertainment, and hygiene products.

The many millions of teenage consumers using such sites have attracted the attention of marketers struggling to find an edge in an increasingly competitive market. One way to gain this edge is to identify segments of teenagers who share similar tastes, so that clients can avoid targeting advertisements to teens with no interest in the product being sold. If it costs 10 dollars to display an advertisement to 1,000 website visitors—a measure of **cost per impression**—the advertising budget will stretch further if we are selective about who is targeted. For instance, an advertisement for sporting apparel should be targeted to clusters of individuals more likely to have an interest in sports.

Given the text of teenagers' SNS posts, we can identify groups that share common interests such as sports, religion, or music. Clustering can automate the process of discovering the natural segments in this population. However, it will be up to us to decide whether the clusters are interesting and how to use them for advertising. Let's try this process from start to finish.

Step 1 – collecting data

For this analysis, we will be using a dataset representing a random sample of 30,000 US high school students who had profiles on a well-known SNS in 2006. To protect the users' anonymity, the SNS will remain unnamed. However, at the time the data was collected, the SNS was a popular web destination for US teenagers. Therefore, it is reasonable to assume that the profiles represent a wide cross-section of American adolescents in 2006.



I compiled this dataset while conducting my own sociological research on teenage identities at the University of Notre Dame. If you use the data for research purposes, please cite this book chapter. The full dataset is available in the Packt Publishing GitHub repository for this book with the filename `snsdata.csv`. To follow along interactively, this chapter assumes you have saved this file to your R working directory.

The data was sampled evenly across four high school graduation years (2006 through to 2009) representing the senior, junior, sophomore, and freshman classes at the time of data collection. Using an automated web crawler, the full text of the SNS profiles was downloaded, and each teen's gender, age, and number of SNS friends were recorded.

A text-mining tool was used to divide the remaining SNS page content into words. From the top 500 words appearing across all pages, 36 words were chosen to represent five categories of interests: extracurricular activities, fashion, religion, romance, and antisocial behavior. The 36 words include terms such as *football*, *sexy*, *kissed*, *bible*, *shopping*, *death*, and *drugs*. The final dataset indicates, for each person, how many times each word appeared on the person's SNS profile.

Step 2 – exploring and preparing the data

We'll use `read.csv()` to load the dataset and convert the character data into factor types:

```
> teens <- read.csv("snsdata.csv", stringsAsFactors = TRUE)
```

Let's also take a quick look at the specifics of the data. The first several lines of the `str()` output are as follows:

```
> str(teens)

'data.frame':      30000 obs. of  40 variables:
 $ gradyear     : int  2006 2006 2006 2006 2006 2006 2006 ...
 $ gender       : Factor w/ 2 levels "F","M": 2 1 2 1 NA 1 1 2 ...
 $ age          : num  19 18.8 18.3 18.9 19 ...
 $ friends      : int  7 0 69 0 10 142 72 17 52 39 ...
 $ basketball   : int  0 0 0 0 0 0 0 0 0 0 ...
```

As we had expected, the data includes 30,000 teenagers with four variables indicating personal characteristics and 36 words indicating interests.

Do you notice anything strange around the gender row? If you looked carefully, you may have noticed the `NA` value, which is out of place compared to the `1` and `2` values. The `NA` is R's way of telling us that the record has a **missing value**—we do not know the person's gender. Until now, we haven't dealt with missing data, but it can be a significant problem for many types of analyses.

Let's see how substantial this problem is. One option is to use the `table()` command, as follows:

```
> table(teens$gender)
```

	F	M
22054	5222	

Although this tells us how many F and M values are present, the `table()` function excluded the NA values rather than treating them as a separate category. To include the NA values (if there are any), we simply need to add an additional parameter:

```
> table(teens$gender, useNA = "ifany")
```

F	M	<NA>
22054	5222	2724

Here, we see that 2,724 records (nine percent) have missing gender data. Interestingly, there are over four times as many females as males in the SNS data, suggesting that males are not as inclined to use this social media website as females.

If you examine the other variables in the data frame, you will find that besides gender, only age has missing values. For numeric features, the default output for the `summary()` function includes the count of NA values:

```
> summary(teens$age)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	NA's
3.086	16.310	17.290	17.990	18.260	106.900	5086

A total of 5,086 records (17 percent) have missing ages. Also concerning is the fact that the minimum and maximum values seem to be unreasonable; it is unlikely that a three-year-old or a 106-year-old is attending high school. To ensure that these extreme values don't cause problems for the analysis, we'll need to clean them up before moving on.

A more plausible range of ages for high school students includes those who are at least 13 years old and not yet 20 years old. Any age value falling outside this range should be treated the same as missing data—we cannot trust the age provided. To recode the age variable, we can use the `ifelse()` function, assigning `teen$age` the original value of `teen$age` if the age is at least 13 and less than 20 years; otherwise, it will receive the value NA:

```
> teens$age <- ifelse(teens$age >= 13 & teens$age < 20,
                         teens$age, NA)
```

By rechecking the `summary()` output, we see that the range now follows a distribution that looks much more like an actual high school:

```
> summary(teens$age)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	NA's
13.03	16.30	17.27	17.25	18.22	20.00	5523

Unfortunately, now we've created an even larger missing data problem. We'll need to find a way to deal with these values before continuing with our analysis.

Data preparation – dummy coding missing values

An easy solution for handling missing values is to exclude any record with a missing value. However, if you think through the implications of this practice, you might think twice before doing so—just because it is easy does not mean it is a good idea! The problem with this approach is that even if the missingness is not extensive, you can easily exclude large portions of the data.

For example, suppose that in our data, the people with NA values for gender are completely different from those with missing age data. This would imply that by excluding those missing either gender or age, you would exclude $9\% + 17\% = 26\%$ of the data, or over 7,500 records. And this is for missing data on only two variables! The larger the number of missing values present in a dataset, the more likely it is that any given record will be excluded. Fairly soon, you will be left with a tiny subset of data, or worse, the remaining records will be systematically different or non-representative of the full population.

An alternative solution for categorical data like gender is to treat a missing value as a separate category. For instance, rather than limiting to female and male, we can add an additional category for unknown gender. This allows us to utilize dummy coding, which was covered in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*.

If you recall, dummy coding involves creating a separate binary (1 or 0) valued dummy variable for each level of a nominal feature except one, which is held out to serve as the reference group. The reason one category can be excluded is because its status can be inferred from the other categories. For instance, if someone is not female and not unknown gender, they must be male. Therefore, in this case, we need to only create dummy variables for female and unknown gender:

```
> teens$female <- ifelse(teens$gender == "F" &
+                         !is.na(teens$gender), 1, 0)
> teens$no_gender <- ifelse(is.na(teens$gender), 1, 0)
```

As you might expect, the `is.na()` function tests whether the gender is equal to NA. Therefore, the first statement assigns `teens$female` the value 1 if the gender is equal to F and the gender is not equal to NA; otherwise, it assigns the value 0. In the second statement, if `is.na()` returns TRUE, meaning the gender is missing, then the `teens$no_gender` variable is assigned 1; otherwise, it is assigned the value 0.

To confirm that we did the work correctly, let's compare our constructed dummy variables to the original gender variable:

```
> table(teens$gender, useNA = "ifany")
```

F	M	<NA>
22054	5222	2724

```
> table(teens$female, useNA = "ifany")
```

0	1
7946	22054

```
> table(teens$no_gender, useNA = "ifany")
```

0	1
27276	2724

The number of 1 values for teens\$female and teens\$no_gender matches the number of F and NA values respectively, so the coding has been performed correctly.

Data preparation – imputing the missing values

Next, let's eliminate the 5,523 missing ages. As age is a numeric feature, it doesn't make sense to create an additional category for unknown values—where would you rank “unknown” relative to the other ages? Instead, we'll use a different strategy known as **imputation**, which involves filling in the missing data with a guess as to the true value.

Can you think of a way we might be able to use the SNS data to make an informed guess about a teenager's age? If you are thinking of using the graduation year, you've got the right idea. Most people in a graduation cohort were born within a single calendar year. If we can identify the typical age for each cohort, then we will have a reasonable approximation of the age of a student in that graduation year.

One way to find a typical value is by calculating the average, or mean, value. If we try to apply the `mean()` function as we have done for previous analyses, there's a problem:

```
> mean(teens$age)
```

```
[1] NA
```

The issue is that the mean value is undefined for a vector containing missing data. As our age data contains missing values, `mean(teens$age)` returns a missing value. We can correct this by adding an additional `na.rm` parameter to remove the missing values before calculating the mean:

```
> mean(teens$age, na.rm = TRUE)
```

```
[1] 17.25243
```

This reveals that the average student in our data is about 17 years old. This only gets us part of the way there; we actually need the average age for each graduation year. You might first attempt to calculate the mean four times, but one of the benefits of R is that there's usually a way to avoid repeating oneself. In this case, the `aggregate()` function is the tool for the job. It computes statistics for subgroups of data. Here, it calculates the mean age by graduation year after removing the `NA` values:

```
> aggregate(data = teens, age ~ gradyear, mean, na.rm = TRUE)
```

	gradyear	age
1	2006	18.65586
2	2007	17.70617
3	2008	16.76770
4	2009	15.81957

The `aggregate()` output is in a data frame. This would require extra work to merge back into our original data. As an alternative, we can use the `ave()` function, which returns a vector with the means of each group repeated such that the resulting vector is the same length as the original vector. Where `aggregate()` returns one average age for each graduation year (a total of four values), the `ave()` function returns a value for all 30,000 teenagers reflecting the average age of students in that student's graduation year (the same four values are repeated to reach a total of 30,000 values).

When using the `ave()` function, the first parameter is the numeric vector for which the group averages are to be computed, the second parameter is the categorical vector supplying the group assignments, and the `FUN` parameter is the function to be applied to the numeric vector. In our case, we need to define a new function that computes the mean with the `NA` values removed. The full command is as follows:

```
> ave_age <- ave(teens$age, teens$gradyear, FUN =
  function(x) mean(x, na.rm = TRUE))
```

To impute these means onto the missing values, we need one more `ifelse()` call to use the `ave_age` value only if the original age value was NA:

```
> teens$age <- ifelse(is.na(teens$age), ave_age, teens$age)
```

The `summary()` results show that the missing values have now been eliminated:

```
> summary(teens$age)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
13.03	16.28	17.24	17.24	18.21	20.00

With the data ready for analysis, we are ready to dive into the interesting part of this project. Let's see if our efforts have paid off.

Step 3 – training a model on the data

To cluster the teenagers into marketing segments, we'll use an implementation of k-means in the `stats` package, which should be included in your R installation by default. Although there is no shortage of more sophisticated k-means functions available in other R packages, the `kmeans()` function in the default `stats` package is widely used and provides a simple yet powerful implementation of the algorithm.

Clustering syntax

using the `kmeans()` function in the `stats` package

Finding clusters:

```
myclusters <- kmeans(mydata, k)
```

- `mydata` is a matrix or data frame with the examples to be clustered
- `k` specifies the desired number of clusters

The function will return a cluster object, which stores information about the clusters.

Examining clusters:

- `myclusters$cluster` is a vector of cluster assignments from the `kmeans()` function
- `myclusters$centers` is a matrix indicating the mean values for each feature and cluster combination
- `myclusters$size` lists the number of examples assigned to each cluster

Example:

```
teen_clusters <- kmeans(teens, 5)
teens$cluster_id <- teen_clusters$cluster
```

Figure 9.11: K-means clustering syntax

The `kmeans()` function requires a data frame or matrix containing only numeric data and a parameter specifying k , the desired number of clusters. If you have these two things ready, the actual process of building the model is simple. The trouble is that choosing the right combination of data and clusters can be a bit of an art; sometimes a great deal of trial and error is involved.

We'll start our cluster analysis by considering only the 36 features that measure the number of times various interest-based keywords appeared in the text of the teenagers' social media profiles. In other words, we will not cluster based on age, graduation year, gender, or number of friends. Of course, we *could* use these four features if desired, but *choose* not to, since any clusters built upon them would be less insightful than those built upon interests. This is primarily because age and gender are already de facto clusters whereas the interest-based clusters are yet to be discovered in our data. Secondarily, what will be more interesting later is to see whether the interest clusters are associated with the gender and popularity features held out from the clustering process. If the interest-based clusters are predictive of these individual characteristics, this provides evidence that the clusters may be useful.

To avoid the chance of accidentally including the other features, let's make a data frame called `interests`, by subsetting the data frame to include only the 36 keyword columns:

```
> interests <- teens[5:40]
```

If you recall from *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, a common practice employed prior to any analysis using distance calculations is to normalize or z-score-standardize the features such that each utilizes the same range. By doing so, you can avoid a problem in which some features dominate solely because they have a larger range of values than the others.

The process of z-score standardization rescales features such that they have a mean of zero and a standard deviation of one. This transformation changes the interpretation of the data in a way that may be useful here. Specifically, if someone mentions basketball three times on their profile, without additional information, we have no idea whether this implies they like basketball more or less than their peers. On the other hand, if the z-score is three, we know that they mentioned basketball many more times than the average teenager.

To apply z-score standardization to the `interests` data frame, we can use the `scale()` function with `lapply()`. Since `lapply()` returns a list object, it must be coerced back to data frame form using the `as.data.frame()` function, as follows:

```
> interests_z <- as.data.frame(lapply(interests, scale))
```

To confirm that the transformation worked correctly, we can compare the summary statistics of the basketball column in the old and new interests data:

```
> summary(interests$basketball)
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
basketball	0.0000	0.0000	0.0000	0.2673	0.0000	24.0000


```
> summary(interests_z$basketball)
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
basketball	-0.3322	-0.3322	-0.3322	0.0000	-0.3322	29.4923

As expected, the `interests_z` dataset transformed the basketball feature to have a mean of zero and a range that spans above and below zero. Now, a value less than zero can be interpreted as a person having fewer-than-average mentions of basketball in their profile. A value greater than zero implies that the person mentioned basketball more frequently than the average.

Our last decision involves deciding how many clusters to use for segmenting the data. If we use too many clusters, we may find them too specific to be useful; conversely, choosing too few may result in heterogeneous groupings. You should feel comfortable experimenting with the value of k . If you don't like the result, you can easily try another value and start over.



Choosing the number of clusters is easier if you are familiar with the analysis population. Having a hunch about the true number of natural groupings can save some trial and error.

To help choose the number of clusters in the data, I'll defer to one of my favorite films, *The Breakfast Club*, a coming-of-age comedy released in 1985 and directed by John Hughes. The teenage characters in this movie are self-described in terms of five identities:

- A *brain* – also commonly known as a “nerd” or “geek”
- An *athlete* – sometimes also known as a “jock” or “prep”
- A *basket case* – slang terminology for an anxious or neurotic individual, and depicted in the film as an anti-social outcast
- A *princess* – portrayed as a popular, affluent, and stereotypically feminine girl
- A *criminal* – represents the traditional “burnout” identity described in sociological research as engaging in rebellious anti-school and anti-authority behaviors

Even though the movie depicts five specific identity groups, they have been described throughout popular teenage fiction for many years, and although the stereotypes have evolved over time, American teenagers are likely to understand them intuitively. Thus, five seems like a reasonable starting point for k , though admittedly, it is unlikely to capture the full spectrum of high-school identities.

To use the k-means algorithm to divide the teenagers' interest data into five clusters, we use the `kmeans()` function on the `interests` data frame. Note that because k-means utilizes random starting points, the `set.seed()` function is used to ensure that the results match the output in the examples that follow. If you recall from previous chapters, this command initializes R's random number generator to a specific sequence. In the absence of this statement, the results may vary each time the k-means algorithm is run. Running the k-means clustering process as follows creates a list named `teen_clusters`, which stores the properties of each of the five clusters:

```
> set.seed(2345)
> teen_clusters <- kmeans(interests_z, 5)
```

Let's dig in and see how well the algorithm has divided the teenagers' interest data.



If you find that your results differ from those shown in the sections that follow, ensure that the `set.seed(2345)` command is run immediately prior to the `kmeans()` function. Additionally, because the behavior of R's random number generator changed with R version 3.6, your results may also vary slightly from those shown here if you are using an older version of R.

Step 4 – evaluating model performance

Evaluating clustering results can be somewhat subjective. Ultimately, the success or failure of the model hinges on whether the clusters are useful for their intended purpose. As the goal of this analysis was to identify clusters of teenagers with similar interests for marketing purposes, we will largely measure our success in qualitative terms. For other clustering applications, more quantitative measures of success may be needed.

One of the most basic ways to evaluate the utility of a set of clusters is to examine the number of examples falling in each of the groups. If some groups are too large or too small, then they are less likely to be very useful.

To obtain the size of the `kmeans()` clusters, simply examine the `teen_clusters$size` component as follows:

```
> teen_clusters$size
[1] 1038   601  4066  2696 21599
```

Here we see the five clusters we requested. The smallest cluster has 601 teenagers (2 percent) while the largest has 21,599 (72 percent). Although the large gap between the number of people in the largest and smallest clusters is slightly concerning, without examining these groups more carefully, we will not know whether this indicates a problem. It may be the case that the clusters' size disparity indicates something real, such as a big group of teenagers who share similar interests, or it may be a random fluke caused by the initial k-means cluster centers. We'll know more as we start to look at each cluster's characteristics.



Sometimes, k-means may find extremely small clusters—occasionally as small as a single point. This can happen if one of the initial cluster centers happen to fall on outliers far from the rest of the data. It is not always clear whether to treat such small clusters as a true finding that represents a cluster of extreme cases, or a problem caused by random chance. If you encounter this issue, it may be worth re-running the k-means algorithm with a different random seed to see whether the small cluster is robust to different starting points.

For a more in-depth look at the clusters, we can examine the coordinates of the cluster centroids using the `teen_clusters$centers` component, which is as follows for the first four interests:

```
> teen_clusters$centers
      basketball    football      soccer    softball
1  0.362160730  0.37985213  0.13734997  0.1272107
2 -0.094426312  0.06691768 -0.09956009 -0.0379725
3  0.003980104  0.09524062  0.05342109 -0.0496864
4  1.372334818  1.19570343  0.55621097  1.1304527
5 -0.186822093 -0.18729427 -0.08331351 -0.1368072
```

The rows of the output (labeled 1 to 5) refer to the five clusters, while the numbers across each row indicate the cluster's average value for the interest listed at the top of the column. Because the values are z-score-standardized, positive values are above the overall mean level for all teenagers and negative values are below the overall mean.

For example, the fourth row has the highest value in the basketball column, which means that cluster 4 has the highest average interest in basketball among all the clusters.

By examining whether clusters fall above or below the mean level for each interest category, we can discover patterns that distinguish the clusters from one another. In practice, this involves printing the cluster centers and searching through them for any patterns or extreme values, much like a word search puzzle but with numbers. The following annotated screenshot shows a highlighted pattern for each of the five clusters, for 18 of the 36 teenager interests:

```
> teen_clusters$centers
  basketball   football      soccer    softball   volleyball   swimming
1  0.362160730  0.37985213  0.13734997  0.1272107  0.09247518  0.26180286
2 -0.094426312  0.06691768 -0.09956009 -0.0379725 -0.07286202  0.04578401
3  0.003980104  0.09524062  0.05342109 -0.0496864 -0.01459648  0.32944934
4  1.372334818  1.19570343  0.55621097  1.1304527  1.07177211  0.08513210
5 -0.186822093 -0.18729427 -0.08331351 -0.1368072 -0.13344819 -0.08650052
  cheerleading   baseball      tennis     sports      cute       sex
1   0.2159945  0.25312305  0.11991682  0.77040675  0.475265034  2.043945661
2  -0.1070370 -0.11182941  0.04027335 -0.10638613 -0.027044898 -0.042725567
3  0.5142451 -0.04933628  0.06703386 -0.05435093  0.796948359 -0.003156716
4  0.0400367  1.09279737  0.13887184  1.08316097 -0.005291962 -0.033193640
5 -0.1092056 -0.13616893 -0.03683671 -0.15903307 -0.171452198 -0.092301138
  sexy        hot     kissed     dance     band   marching
1  0.547956598  0.314845390  3.02610259  0.455501275  0.39009330 -0.0105463
2 -0.027913348 -0.035027022 -0.04581067  0.050772118  4.09723438  5.2196105
3  0.266741598  0.623263396 -0.01284964  0.650572336 -0.03301257 -0.1131486
4  0.003036966  0.009046774 -0.08755418 -0.001993853 -0.07317758 -0.1039509
5 -0.076149916 -0.132614350 -0.13080557 -0.145524147 -0.11740538 -0.1104553
```

Figure 9.12: To distinguish clusters, it can be helpful to highlight patterns in the coordinates of their centroids

Given this snapshot of the interest data, we can already infer some characteristics of the clusters. Cluster four is substantially above the mean interest level on nearly all the sports, which suggests that this may be a group of *athletes* per *The Breakfast Club* stereotype. Cluster three includes the most mentions of cheerleading, dancing, and the word “hot.” Are these the so-called princesses?

By continuing to examine the clusters in this way, it is possible to construct a table listing the dominant interests of each of the groups. In the following table, each cluster is shown with the features that most distinguish it from the other clusters, and *The Breakfast Club* identity that seems to most accurately capture the group’s characteristics.

Interestingly, cluster five is distinguished by the fact that it is unexceptional: its members had lower-than-average levels of interest in every measured activity. It is also the single largest group in terms of the number of members. How can we reconcile these apparent contradictions? One potential explanation is that these users created a profile on the website but never posted any interests.

Cluster 1 (N = 1,038)	Cluster 2 (N = 601)	Cluster 3 (N = 4,066)	Cluster 4 (N = 2,696)	Cluster 5 (N = 21,599)
sex sexy kissed music rock god hair clothes die death drunk drugs	band marching music	cheerleading cute hot dance church Jesus Bible dress mall shopping Hollister Abercrombie	basketball football soccer softball volleyball baseball tennis sports	???
Criminals	Brains	Princesses	Athletes	Basket Cases

Figure 9.13: A table can be used to list important dimensions of each cluster

When sharing the results of a segmentation analysis with stakeholders, it is often helpful to apply memorable and informative labels known as **personas**, which simplify and capture the essence of the groups, such as *The Breakfast Club* typology applied here. The risk in adding such labels is that they can obscure the groups' nuances and possibly even offend the group members if negative stereotypes are used. For wider dissemination, provocative labels like "Criminals" and "Princesses" might be replaced by more neutral terminology like "Edgy Adolescents" and "Trendy Teenagers." Additionally, because even relatively harmless labels can bias our thinking, important patterns can be missed if labels are understood as the whole truth rather than a simplification of complexity.

Given memorable labels and a table as depicted in *Figure 9.13*, a marketing executive would have a clear mental picture of five types of teenage visitors to the social networking website. Based on these personas, the executive could sell targeted advertising impressions to businesses with products relevant to one or more of the clusters. In the next section, we will see how the cluster labels can be applied back to the original population for such uses.



It is possible to visualize the results of a cluster analysis using techniques that flatten the multidimensional feature data into two dimensions, then color the points according to cluster assignment. The `fviz_cluster()` function in the `factoextra` package allows such visualizations to be constructed quite easily. If this is of interest to you, load the package and try the command `fviz_cluster(teen_clusters, interests_z, geom = "point")` to see such a visualization for the teenage SNS clusters. Although the visual is of limited use for the SNS example due to the large number of overlapping points, sometimes, it can be a helpful tool for presentation purposes. To better understand how to create and understand these plots, see *Chapter 15, Making Use of Big Data*.

Step 5 – improving model performance

Because clustering creates new information, the performance of a clustering algorithm depends at least somewhat on both the quality of the clusters themselves and what is done with that information. In the preceding section, we demonstrated that the five clusters provided useful and novel insights into the interests of teenagers. By that measure, the algorithm appears to be performing quite well. Therefore, we can now focus our effort on turning these insights into action.

We'll begin by applying the clusters back to the full dataset. The `teen_clusters` object created by the `kmeans()` function includes a component named `cluster`, which contains the cluster assignments for all 30,000 individuals in the sample. We can add this as a column to the `teens` data frame with the following command:

```
> teens$cluster <- teen_clusters$cluster
```

Given this new data, we can start to examine how the cluster assignment relates to individual characteristics. For example, here's the personal information for the first five teenagers in the SNS data:

```
> teens[1:5, c("cluster", "gender", "age", "friends")]
```

	cluster	gender	age	friends
1	5	M	18.982	7
2	3	F	18.801	0
3	5	M	18.335	69
4	5	F	18.875	0
5	1	<NA>	18.995	10

Using the `aggregate()` function, we can also look at the demographic characteristics of the clusters. The mean age does not vary much by cluster, which is not too surprising, as teen identities are often set well before high school. This is depicted as follows:

```
> aggregate(data = teens, age ~ cluster, mean)
```

	cluster	age
1	1	17.09319
2	2	17.38488
3	3	17.03773
4	4	17.03759
5	5	17.30265

On the other hand, there are some substantial differences in the proportion of females by cluster. This is a very interesting finding, as we didn't use gender data to create the clusters, yet the clusters are still predictive of gender:

```
> aggregate(data = teens, female ~ cluster, mean)
```

	cluster	female
1	1	0.8025048
2	2	0.7237937
3	3	0.8866208
4	4	0.6984421
5	5	0.7082735

Recall that overall, about 74 percent of the SNS users are female. Cluster three, the so-called *princesses*, is nearly 89 percent female, while clusters four and five are only about 70 percent female. These disparities imply that there are differences in the interests that teenage boys and girls discuss on their social networking pages.

Given our success in predicting gender, you might suspect that the clusters are also predictive of the number of friends the users have. This hypothesis seems to be supported by the data, which is as follows:

```
> aggregate(data = teens, friends ~ cluster, mean)
```

	cluster	friends
1	1	30.66570
2	2	32.79368
3	3	38.54575

4	4	35.91728
5	5	27.79221

On average, *princesses* have the most friends (38.5), followed by *athletes* (35.9) and *brains* (32.8). On the low end are *criminals* (30.7) and *basket cases* (27.8). As with gender, the connection between a teenager's number of friends and their predicted cluster is remarkable given that we did not use the friendship data as an input to the clustering algorithm. Also interesting is the fact that the number of friends seems to be related to the stereotype of each cluster's high-school popularity: the stereotypically popular groups tend to have more friends in reality.

The association between group membership, gender, and number of friends suggests that the clusters can be useful predictors of behavior. Validating their predictive ability in this way may make the clusters an easier sell when they are pitched to the marketing team, ultimately improving the performance of the algorithm.



Just as the characters in *The Breakfast Club* ultimately come to realize that “each one of us is a brain, an athlete, a basket case, a princess, and a criminal,” it is important for data scientists to realize that the labels or personas we attribute to each cluster are stereotypes, and individuals may embody the stereotype to a greater or lesser degree. Keep this caveat in mind when acting on the results of a cluster analysis; a group may be relatively homogenous, but each member is still unique.

Summary

Our findings support the popular adage that “birds of a feather flock together.” By using machine learning methods to cluster teenagers with others who have similar interests, we were able to develop a typology of teenage identities, which was predictive of personal characteristics such as gender and number of friends. These same methods can be applied to other contexts with similar results.

This chapter covered only the fundamentals of clustering. There are many variants of the k-means algorithm, as well as many other clustering algorithms, which bring unique biases and heuristics to the task. Based on the foundation in this chapter, you will be able to understand these clustering methods and apply them to new problems.

In the next chapter, we will begin to look at methods for measuring the success of a learning algorithm that are applicable across many machine learning tasks. While our process has always devoted some effort to evaluating the success of learning, in order to obtain the highest degree of performance, it is crucial to be able to define and measure it in the strictest terms.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



10

Evaluating Model Performance

When only the wealthy could afford education, tests and exams were not used to evaluate students. Instead, tests evaluated the teachers for parents who wanted to know whether their children learned enough to justify the instructors' wages. Obviously, this is different today. Now, such evaluations are used to distinguish between high-achieving and low-achieving students, filtering them into careers and other opportunities.

Given the significance of this process, a great deal of effort is invested in developing accurate student assessments. Fair assessments have a large number of questions that cover a wide breadth of topics and reward true knowledge over lucky guesses. A good assessment also requires students to think about problems they have never faced before. Correct responses, therefore, reflect an ability to generalize knowledge more broadly.

The process of evaluating machine learning algorithms is very similar to the process of evaluating students. Since algorithms have varying strengths and weaknesses, tests should distinguish between learners. It is also important to understand how a learner will perform on future data.

This chapter provides the information needed to assess machine learners, such as:

- The reasons why predictive accuracy is not sufficient to measure performance, and the performance measures you might use instead
- Methods to ensure that the performance measures reasonably reflect a model's ability to predict or forecast unseen cases
- How to use R to apply these more useful measures and methods to the predictive models covered in previous chapters

Just as the best way to learn a topic is to attempt to teach it to someone else, the process of teaching and evaluating machine learners will provide you with greater insight into the methods you've learned so far.

Measuring performance for classification

In the previous chapters, we measured classifier accuracy by dividing the number of correct predictions by the total number of predictions. This finds the proportion of cases in which the learner is correct, and the proportion of incorrect cases follows directly. For example, suppose that a classifier correctly predicted whether newborn babies were a carrier of a treatable but potentially fatal genetic defect in 99,990 out of 100,000 cases. This would imply an accuracy of 99.99 percent and an error rate of only 0.01 percent.

At first glance, this appears to be an extremely valuable classifier. However, it would be wise to collect additional information before trusting a child's life to the test. What if the genetic defect is found in only 10 out of every 100,000 babies? A test that invariably predicts no defect will be correct for 99.99 percent of all cases, but incorrect for 100 percent of the cases that matter most. In other words, even though the classifier is extremely accurate, it is not very useful for preventing treatable birth defects.



This is one consequence of the **class imbalance problem**, which refers to the trouble associated with data having a large majority of records belonging to a single class.

Though there are many ways to measure a classifier's performance, the best measure is always that which captures whether the classifier is successful at its intended purpose. It is crucial to define performance measures in terms of utility rather than raw accuracy. To this end, we will explore a variety of alternative performance measures derived from the confusion matrix. Before we get started, however, we need to consider how to prepare a classifier for evaluation.

Understanding a classifier's predictions

The goal of evaluating a classification model is to better understand how its performance will extrapolate to future cases. Since it is usually infeasible to test an unproven model in a live environment, we typically simulate future conditions by asking the model to classify cases in a dataset made of cases that resemble what it will be asked to do in the future. By observing the learner's responses to this examination, we can learn about its strengths and weaknesses.

Though we've evaluated classifiers in prior chapters, it's worth reflecting on the types of data at our disposal:

- Actual class values
- Predicted class values
- The estimated probability of the prediction

The actual and predicted class values may be self-evident, but they are the key to the evaluation. Just like a teacher uses an answer key—a list of correct answers—to assess the student's answers, we need to know the correct answer for a machine learner's predictions. The goal is to maintain two vectors of data: one holding the correct or actual class values, and the other holding the predicted class values. Both vectors must have the same number of values stored in the same order. The predicted and actual values may be stored as separate R vectors or as columns in a single R data frame.

Obtaining this data is easy. The actual class values come directly from the target in the test data-set. Predicted class values are obtained from the classifier built upon the training data, which is then applied to the test data. For most machine learning packages, this involves applying the `predict()` function to a model object and a data frame of test data, such as `predictions <- predict(model, test_data)`.

Until now, we have only examined classification predictions using these two vectors of data, but most models can supply another piece of useful information. Even though the classifier makes a single prediction about each example, it may be more confident about some decisions than others.

For instance, a classifier may be 99 percent certain that an SMS with the words “free” and “ring-tones” is spam, but only 51 percent certain that an SMS with the word “tonight” is spam. In both cases, the classifier classifies the message as spam, but it is far more certain about one decision than the other.

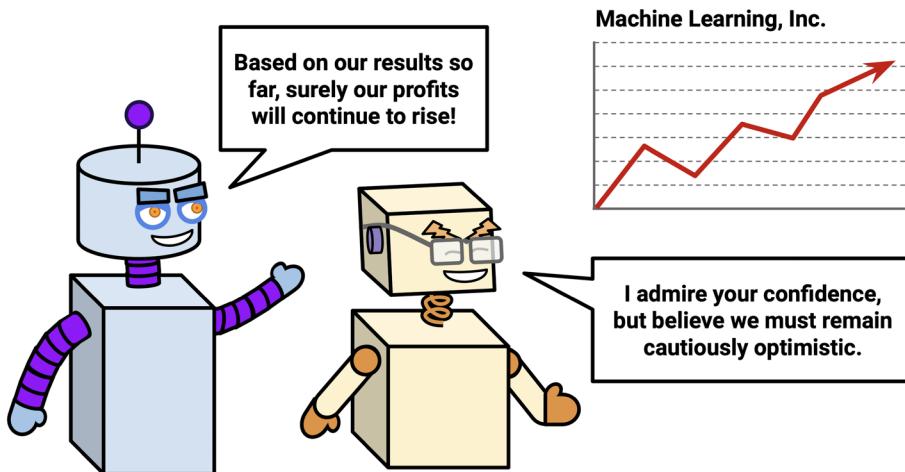


Figure 10.1: Learners may differ in their prediction confidence even when trained on the same data

Studying these internal prediction probabilities provides useful data for evaluating a model’s performance. If two models make the same number of mistakes, but one is more able to accurately assess its uncertainty, then it is a smarter model. It’s ideal to find a learner that is extremely confident when making a correct prediction, but timid in the face of doubt. The balance between confidence and caution is a key part of model evaluation.

The function call to obtain the internal prediction probabilities varies across R packages. For most classifiers, the `predict()` function allows an additional parameter to specify the desired type of prediction. To obtain a single predicted class, such as spam or ham, you typically set the `type = "class"` parameter. To obtain the prediction probability, the `type` parameter should be set to one of `"prob"`, `"posterior"`, `"raw"`, or `"probability"`, depending on the classifier used.



All classifiers presented in this book can provide prediction probabilities. The correct setting for the `type` parameter is included in the syntax box introducing each model.

For example, to output the predicted probabilities for the C5.0 classifier built in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, use the `predict()` function with `type = "prob"` as follows:

```
> predicted_prob <- predict(credit_model, credit_test, type = "prob")
```

To output the Naive Bayes predicted probabilities for the SMS spam classification model developed in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*, use `predict()` with `type = "raw"` as follows:

```
> sms_test_prob <- predict(sms_classifier, sms_test, type = "raw")
```

In most cases, the `predict()` function returns a probability for each category of the outcome. For example, in the case of a two-outcome model like the SMS classifier, the predicted probabilities might be stored in a matrix or data frame, as shown here:

```
> head(sms_test_prob)
```

	ham	spam
[1,]	9.999995e-01	4.565938e-07
[2,]	9.999995e-01	4.540489e-07
[3,]	9.998418e-01	1.582360e-04
[4,]	9.999578e-01	4.223125e-05
[5,]	4.816137e-10	1.000000e+00
[6,]	9.997970e-01	2.030033e-04

Each line in this output shows the classifier's predicted probability of spam and ham. According to probability rules, the sum of the probabilities across each row is 1 because these are mutually exclusive and exhaustive outcomes. For convenience, during the evaluation process, it can be helpful to construct a data frame collecting the predicted class, the actual class, and the predicted probability of the class level (or levels) of interest.

The `sms_results.csv` file available in the GitHub repository for this chapter is an example of a data frame in exactly this format and is built from the predictions of the SMS classifier built in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. The steps required to construct this evaluation dataset have been omitted for brevity, so to follow along with the example here, simply download the file and load it into a data frame using the following command:

```
> sms_results <- read.csv("sms_results.csv", stringsAsFactors = TRUE)
```

The resulting `sms_results` data frame is simple. It contains four vectors of 1,390 values. One column contains values indicating the actual type of SMS message (spam or ham), another indicates the Naive Bayes model's predicted message type, and the third and fourth columns indicate the probability that the message was spam or ham, respectively:

```
> head(sms_results)
```

	actual_type	predict_type	prob_spam	prob_ham
1	ham	ham	0.00000	1.00000
2	ham	ham	0.00000	1.00000
3	ham	ham	0.00016	0.99984
4	ham	ham	0.00004	0.99996
5	spam	spam	1.00000	0.00000
6	ham	ham	0.00020	0.99980

For these six test cases, the predicted and actual SMS message types agree; the model predicted their statuses correctly. Furthermore, the prediction probabilities suggest that the model was extremely confident about these predictions because they all fall close to or are exactly 0 or 1.

What happens when the predicted and actual values are further from 0 and 1? Using the `subset()` function, we can identify a few of these records. The following output shows test cases where the model estimated the probability of spam as being between 40 and 60 percent:

```
> head(subset(sms_results, prob_spam > 0.40 & prob_spam < 0.60))
```

	actual_type	predict_type	prob_spam	prob_ham
377	spam	ham	0.47536	0.52464
717	ham	spam	0.56188	0.43812
1311	ham	spam	0.57917	0.42083

By the model's own estimation, these were cases in which a correct prediction was virtually a coin flip. Yet all three predictions were wrong—an unlucky result. Let's look at a few more cases where the model was wrong:

```
> head(subset(sms_results, actual_type != predict_type))
```

	actual_type	predict_type	prob_spam	prob_ham
53	spam	ham	0.00071	0.99929
59	spam	ham	0.00156	0.99844
73	spam	ham	0.01708	0.98292
76	spam	ham	0.00851	0.99149

184	spam	ham	0.01243	0.98757
332	spam	ham	0.00003	0.99997

These cases illustrate the important fact that a model can be extremely confident and yet it can still be extremely wrong. All six of these test cases were spam messages that the classifier believed to have no less than a 98 percent chance of being ham.

Despite such mistakes, is the model still useful? We can answer this question by applying various error metrics to this evaluation data. In fact, many such metrics are based on a tool we've already used extensively in previous chapters.

A closer look at confusion matrices

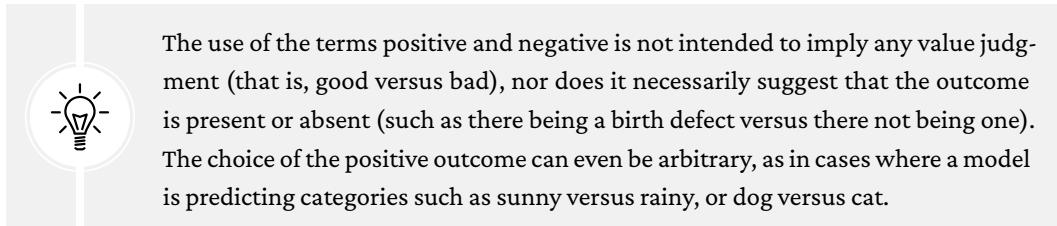
A **confusion matrix** is a table that categorizes predictions according to whether they match the actual value. One of the table's dimensions indicates the possible categories of predicted values, while the other dimension indicates the same thing for actual values. Although we have mostly worked with 2x2 confusion matrices so far, a matrix can be created for models that predict any number of class values. The following figure depicts the familiar confusion matrix for a two-class binary model, as well as the 3x3 confusion matrix for a three-class model.

When the predicted value is the same as the actual value, this is a correct classification. Correct predictions fall on the diagonal in the confusion matrix (denoted by O). The off-diagonal matrix cells (denoted by X) indicate the cases where the predicted value differs from the actual value. These are incorrect predictions. Performance measures for classification models are based on the counts of predictions falling on and off the diagonal in these tables:

		Two Classes		Three Classes		
		Predicted Class		Predicted Class		
		A	B	A	B	C
Actual Class	A			Actual Class		
	B					
Actual Class	A					
	B					
	C					

Figure 10.2: Confusion matrices count cases where the predicted class agrees or disagrees with the actual value

The most common performance measures consider the model's ability to discern one class versus all others. The class of interest is known as the **positive** class, while all others are known as **negative**.



The relationship between positive class and negative class predictions can be depicted as a 2x2 confusion matrix that tabulates whether predictions fall into one of four categories:

- **True positive (TP):** Correctly classified as the class of interest
- **True negative (TN):** Correctly classified as not the class of interest
- **False positive (FP):** Incorrectly classified as the class of interest
- **False negative (FN):** Incorrectly classified as not the class of interest

For the spam classifier, the positive class is spam, as this is the outcome we hope to detect. We then can imagine the confusion matrix as shown in *Figure 10.3*:

		Predicted to be Spam	
		no	yes
Actually Spam	no	TN True Negative	FP False Positive
	yes	FN False Negative	TP True Positive

Figure 10.3: Distinguishing between positive and negative classes adds detail to the confusion matrix

The confusion matrix presented in this way is the basis for many of the most important measures of model performance. In the next section, we'll use this matrix to better understand exactly what is meant by accuracy.

Using confusion matrices to measure performance

With the 2x2 confusion matrix, we can formalize our definition of **prediction accuracy** (sometimes called the **success rate**) as:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

In this formula, the terms *TP*, *TN*, *FP*, and *FN* refer to the number of times the model's predictions fell into each of these categories. The accuracy is therefore a proportion that represents the number of true positives and true negatives divided by the total number of predictions.

The **error rate**, or the proportion of incorrectly classified examples, is specified as:

$$\text{error rate} = \frac{\text{FP} + \text{FN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = 1 - \text{accuracy}$$

Notice that the error rate can be calculated as 1 minus the accuracy. Intuitively, this makes sense; a model that is correct 95 percent of the time is incorrect five percent of the time.

An easy way to tabulate a classifier's predictions into a confusion matrix is to use R's `table()` function. The command for creating a confusion matrix for the SMS data is shown as follows. The counts in this table could then be used to calculate accuracy and other statistics:

```
> table(sms_results$actual_type, sms_results$predict_type)
```

	ham	spam
ham	1203	4
spam	31	152

If you would like to create a confusion matrix with more informative output, the `CrossTable()` function in the `gmodels` package offers a customizable solution. If you recall, we first used this function in *Chapter 2, Managing and Understanding Data*. If you didn't install the package at that time, you will need to do so using the `install.packages("gmodels")` command.

By default, the `CrossTable()` output includes proportions in each cell that indicate the cell count as a percentage of the table's row, column, and overall total counts. The output also includes row and column totals. As shown in the following code, the syntax is similar to the `table()` function:

```
> library(gmodels)
> CrossTable(sms_results$actual_type, sms_results$predict_type)
```

The result is a confusion matrix with a wealth of additional detail:

Cell Contents								
	N							
Chi-square contribution								
	N / Row Total							
	N / Col Total							
	N / Table Total							
----- ----- ----- ----- -----								
Total Observations in Table: 1390								
----- ----- ----- ----- -----								
sms_results\$predict_type								
sms_results\$actual_type	ham	spam	Row Total					
----- ----- ----- ----- -----								
ham	1203	4	1207					
	16.128	127.580						
	0.997	0.003	0.868					
	0.975	0.026						
	0.865	0.003						
----- ----- ----- ----- -----								
spam	31	152	183					
	106.377	841.470						
	0.169	0.831	0.132					
	0.025	0.974						
	0.022	0.109						
----- ----- ----- ----- -----								
Column Total	1234	156	1390					
	0.888	0.112						
----- ----- ----- ----- -----								

We've used `CrossTable()` in several previous chapters, so by now, you should be familiar with its output. If you ever forget how to interpret the output, simply refer to the key (labeled `Cell Contents`), which provides the definition of each number in the table cells.

We can use the confusion matrix to obtain the accuracy and error rate. Since accuracy is $(TP + TN) / (TP + TN + FP + FN)$, we can calculate it as follows:

```
> (152 + 1203) / (152 + 1203 + 4 + 31)
```

```
[1] 0.9748201
```

We can also calculate the error rate $(FP + FN) / (TP + TN + FP + FN)$ as:

```
> (4 + 31) / (152 + 1203 + 4 + 31)
```

```
[1] 0.02517986
```

This is the same as 1 minus accuracy:

```
> 1 - 0.9748201
```

```
[1] 0.0251799
```

Although these calculations may seem simple, it is important to practice thinking about how the components of the confusion matrix relate to one another. In the next section, you will see how these same pieces can be combined in different ways to create a variety of additional performance measures.

Beyond accuracy – other measures of performance

Countless performance measures have been developed and used for specific purposes in disciplines as diverse as medicine, information retrieval, marketing, and signal detection theory, among others. To cover all of them could fill hundreds of pages, which makes a comprehensive description infeasible here. Instead, we'll consider only some of the most useful and most cited measures in machine learning literature.

The *caret* package by Max Kuhn includes functions for computing many such performance measures. This package provides tools for preparing, training, evaluating, and visualizing machine learning models and data; the name “*caret*” is a simplification of “classification and regression training.” Because it is also valuable for tuning models, in addition to its use here, we will also employ the *caret* package extensively in *Chapter 14, Building Better Learners*. Before proceeding, you will need to install the package using the `install.packages("caret")` command.



For more information on *caret*, refer to *Building Predictive Models in R Using the caret Package*, Kuhn, M, *Journal of Statistical Software*, 2008, Vol. 28 or the package's very informative documentation pages at <http://topepo.github.io/caret/index.html>

The caret package adds yet another function for creating a confusion matrix. As shown in the following command, the syntax is similar to `table()`, but with a minor difference. Because caret computes the measures of model performance that reflect the ability to classify the positive class, a `positive` parameter should be specified. In this case, since the SMS classifier is intended to detect spam, we will set `positive = "spam"` as follows:

```
> library(caret)
> confusionMatrix(sms_results$predict_type,
+                  sms_results$actual_type, positive = "spam")
```

This results in the following output:

```
Confusion Matrix and Statistics

              Reference
Prediction    ham  spam
      ham   1203   31
      spam     4  152

Accuracy : 0.9748
95% CI : (0.9652, 0.9824)
No Information Rate : 0.8683
P-Value [Acc > NIR] : < 2.2e-16

Kappa : 0.8825

McNemar's Test P-Value : 1.109e-05

Sensitivity : 0.8306
Specificity : 0.9967
Pos Pred Value : 0.9744
Neg Pred Value : 0.9749
Prevalence : 0.1317
Detection Rate : 0.1094
Detection Prevalence : 0.1122
Balanced Accuracy : 0.9136

'Positive' Class : spam
```

At the top of the output is a confusion matrix much like the one produced by the `table()` function, but transposed. The output also includes a set of performance measures. Some of these, like accuracy, are familiar, while many others are new. Let's look at some of the most important metrics.

The kappa statistic

The **kappa statistic** (labeled Kappa in the previous output) adjusts the accuracy by accounting for the possibility of a correct prediction by chance alone. This is especially important for datasets with a severe class imbalance because a classifier can obtain high accuracy simply by always guessing the most frequent class. The kappa statistic will only reward the classifier if it is correct more often than this simplistic strategy.



There is more than one way to define the kappa statistic. The most common method, described here, uses **Cohen's kappa coefficient**, as described in the paper *A coefficient of agreement for nominal scales*, Cohen, J, *Education and Psychological Measurement*, 1960, Vol. 20, pp. 37-46.

Kappa values typically range from 0 to a maximum of 1, with higher values reflecting stronger agreement between the model's predictions and the true values. It is possible to observe values less than 0 if the predictions are consistently in the wrong direction—that is, the predictions disagree with the actual values or are wrong more often than would be expected by random guessing. This rarely occurs for machine learning models and usually reflects a coding issue, which can be fixed by simply reversing the predictions.

Depending on how a model is to be used, the interpretation of the kappa statistic might vary. One common interpretation is shown as follows:

- Poor agreement = less than 0.2
- Fair agreement = 0.2 to 0.4
- Moderate agreement = 0.4 to 0.6
- Good agreement = 0.6 to 0.8
- Very good agreement = 0.8 to 1.0

It's important to note that these categories are subjective. While "good agreement" may be more than adequate for predicting someone's favorite ice cream flavor, "very good agreement" may not suffice if your goal is to identify birth defects.



For more information on the previous scale, refer to *The measurement of observer agreement for categorical data*, Landis, JR, Koch, GG. *Biometrics*, 1997, Vol. 33, pp. 159-174.

The following is the formula for calculating the kappa statistic. In this formula, $\Pr(a)$ refers to the proportion of actual agreement and $\Pr(e)$ refers to the expected agreement between the classifier and the true values, under the assumption that they were chosen at random:

$$\kappa = \frac{\Pr(a) - \Pr(e)}{1 - \Pr(e)}$$

These proportions are easy to obtain from a confusion matrix once you know where to look. Let's consider the confusion matrix for the SMS classification model created with the `CrossTable()` function, which is repeated here for convenience:

		sms_results\$predict_type		Row Total
sms_results\$actual_type		ham	spam	
ham	ham	1203	4	1207
	16.128	127.580		
	0.997	0.003	0.868	
	0.975	0.026		
	0.865	0.003		
spam	spam	31	152	183
	106.377	841.470		
	0.169	0.831	0.132	
	0.025	0.974		
	0.022	0.109		
Column Total		1234	156	1390
		0.888	0.112	

Remember that the bottom value in each cell indicates the proportion of all instances falling into that cell. Therefore, to calculate the observed agreement $\Pr(a)$, we simply add the proportion of all instances where the predicted type and actual SMS type agree.

Thus, we can calculate $\text{Pr}(a)$ as:

```
> pr_a <- 0.865 + 0.109
> pr_a
[1] 0.974
```

For this classifier, the observed and actual values agree 97.4 percent of the time—you will note that this is the same as the accuracy. The kappa statistic adjusts the accuracy relative to the expected agreement, $\text{Pr}(e)$, which is the probability that chance alone would lead the predicted and actual values to match, under the assumption that both are selected randomly according to the observed proportions.

To find these observed proportions, we can use the probability rules we learned in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. Assuming two events are independent (meaning one does not affect the other), probability rules note that the probability of both occurring is equal to the product of the probabilities of each one occurring. For instance, we know that the probability of both choosing ham is:

$$\text{Pr}(\text{actual_type is ham}) * \text{Pr}(\text{predicted_type is ham})$$

And the probability of both choosing spam is:

$$\text{Pr}(\text{actual_type is spam}) * \text{Pr}(\text{predicted_type is spam})$$

The probability that the predicted or actual type is spam or ham can be obtained from the row or column totals. For instance, $\text{Pr}(\text{actual_type is ham}) = 0.868$ and $\text{Pr}(\text{predicted type is ham}) = 0.888$.

$\text{Pr}(e)$ can be calculated as the sum of the probabilities that the predicted and actual values agree that the message is either spam or ham. Recall that for mutually exclusive events (events that cannot happen simultaneously), the probability of either occurring is equal to the sum of their probabilities. Therefore, to obtain the final $\text{Pr}(e)$, we simply add both products, as follows:

```
> pr_e <- 0.868 * 0.888 + 0.132 * 0.112
> pr_e
[1] 0.785568
```

Since $\text{Pr}(e)$ is 0.786, by chance alone, we would expect the observed and actual values to agree about 78.6 percent of the time.

This means that we now have all the information needed to complete the kappa formula. Plugging the $\text{Pr}(a)$ and $\text{Pr}(e)$ values into the kappa formula, we find:

```
> k <- (pr_a - pr_e) / (1 - pr_e)
> k
```

```
[1] 0.8787494
```

The kappa is about 0.88, which agrees with the previous `confusionMatrix()` output from `caret` (the small difference is due to rounding). Using the suggested interpretation, we note that there is very good agreement between the classifier's predictions and the actual values.

There are a couple of R functions for calculating kappa automatically. The `Kappa()` function (be sure to note the capital "K") in the **Visualizing Categorical Data (VCD)** package uses a confusion matrix of predicted and actual values. After installing the package by typing `install.packages("vcd")`, the following commands can be used to obtain kappa:

```
> library(vcd)
> Kappa(table(sms_results$actual_type, sms_results$predict_type))
```

	value	ASE	z	$\text{Pr}(> z)$
Unweighted	0.8825	0.01949	45.27	0
Weighted	0.8825	0.01949	45.27	0

We're interested in the unweighted kappa. The value of 0.88 matches what we computed manually.



The weighted kappa is used when there are varying degrees of agreement. For example, using a scale of cold, cool, warm, and hot, the value of warm agrees more with hot than it does with the value of cold. In the case of a two-outcome event, such as spam and ham, the weighted and unweighted kappa statistics will be identical.

The `kappa2()` function in the **Interrater Reliability (irr)** package can be used to calculate kappa from vectors of predicted and actual values stored in a data frame. After installing the package using `install.packages("irr")`, the following commands can be used to obtain kappa:

```
> library(irr)
> kappa2(sms_results[1:2])
```

```
Cohen's Kappa for 2 Raters (Weights: unweighted)
```

```
Subjects = 1390
Raters = 2
```

```
Kappa = 0.883
```

```
z = 33
```

```
p-value = 0
```

The `Kappa()` and `kappa2()` functions report the same kappa statistic, so use whichever option you are more comfortable with.



Be careful not to use the built-in `kappa()` function. It is completely unrelated to the kappa statistic reported previously!

The Matthews correlation coefficient

Although accuracy and kappa have been popular measures of performance for many years, a third option has quickly become a de facto standard in the field of machine learning. Like both prior metrics, the **Matthews correlation coefficient (MCC)** is a single statistic intended to reflect the overall performance of a classification model. Additionally, the MCC is like kappa in that it is useful even in the case that the dataset is severely imbalanced—the types of situations in which the traditional accuracy measure can be very misleading.

The MCC has grown in popularity due to its ease of interpretation, as well as a growing body of evidence suggesting that it performs better in a wider variety of circumstances than kappa. Recent empirical research has indicated that the MCC may be the best single metric for describing the real-world performance of a binary classification model. Other studies have identified potential circumstances in which the kappa statistic provides a misleading or incorrect depiction of model performance. In these cases, when the MCC and kappa disagree, the MCC metric tends to provide a more reasonable assessment of the model's true capabilities.

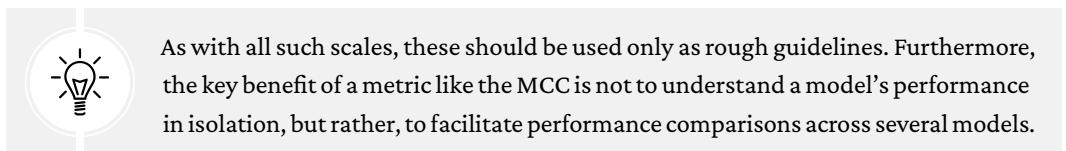


For more information on the relative advantages of the Matthews correlation coefficient versus kappa, see *The Matthews correlation coefficient (MCC) is more informative than Cohen's kappa and brier score in binary classification assessment*, Chicco D, Warrens MJ, Jurman G, IEEE Access, 2021, Vol. 9, pp. 78368-78381. Alternatively, refer to *Why Cohen's Kappa should be avoided as performance measure in classification*, Delgado R, Tibau XA, PLoS One, 2019, Vol. 14(9):e0222916.

Values of the MCC are interpreted on the same scale as Pearson's correlation coefficient, which was introduced in *Chapter 6, Forecasting Numeric Data – Regression Methods*. This ranges from -1 to +1, which indicate perfectly inaccurate and perfectly accurate predictions, respectively. A value of 0 indicates a model that performs no better than random guessing. As most MCC scores fall somewhere in the range of values between 0 and 1, some subjectivity is involved in knowing what is a “good” score. Much like the scale used for Pearson correlations, one potential interpretation is as follows:

- Perfectly incorrect = -1.0
- Strongly incorrect = -0.5 to -1.0
- Moderately incorrect = -0.3 to -0.5
- Weakly incorrect = -0.1 to 0.3
- Randomly correct = -0.1 to 0.1
- Weakly correct = 0.1 to 0.3
- Moderately correct = 0.3 to 0.5
- Strongly correct = 0.5 to 1.0
- Perfectly correct = 1.0

Note that the worst-performing models fall in the middle of the scale. In other words, a model on the negative side of the scale (from perfectly to weakly incorrect) still performs better than a model predicting at random. For instance, even though the accuracy of a strongly incorrect model is poor, the predictions can simply be reversed to obtain the correct result.



The MCC can be computed from the confusion matrix for a binary classifier as shown in the following formula:

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}}$$

Using the confusion matrix for the SMS spam classification model, we obtain the following values:

- TN = 1203
- FP = 4

- FN = 31
- TP = 152

The MCC can then be computed manually in R as follows:

```
> (152 * 1203 - 4 * 31) /  
  sqrt((152 + 4) * (152 + 31) * (1203 + 4) * (1203 + 31))  
  
[1] 0.8861669
```

The `mltools` package by Ben Gorman provides an `mcc()` function which can perform the MCC calculation using vectors of predicted and actual values. After installing the package, the following R code produces the same result as the calculation done by hand:

```
> library(mltools)  
> mcc(sms_results$actual_type, sms_results$predict_type)  
  
[1] 0.8861669
```

Alternatively, for a binary classifier where the positive class is coded as 1 and the negative class is coded as 0, the MCC is identical to the Pearson correlation between the predicted and actual values. We can demonstrate this using the `cor()` function in R, after using `ifelse()` to convert the categorical ("spam" or "ham") values into binary (1 or 0) values as follows:

```
> cor(ifelse(sms_results$actual_type == "spam", 1, 0),  
  ifelse(sms_results$predict_type == "spam", 1, 0))  
  
[1] 0.8861669
```

The fact that such an obvious classification performance metric was hiding in plain sight, as a simple adaptation of Pearson's correlation introduced in the late 1800s, makes it quite remarkable that the MCC has only become popular in recent decades! Biochemist Brian W. Matthews is responsible for popularizing this metric in 1975 for use in two-outcome classification problems and thus receives naming credit for this specific application. However, it seems quite likely that the metric was already used widely, even if it had not garnered much attention until much later. Today, it is used across industry, academic research, and even as a benchmark for machine learning competitions. There may be no single metric that better captures the overall performance of a binary classification model. However, as you will soon see, a more in-depth understanding of model performance can be obtained using combinations of metrics.



Although the MCC is defined here for binary classification, it is unclear whether it is the best metric for multi-class outcomes. For a discussion of this and other alternatives, see *A comparison of MCC and CEN error measures in multi-class prediction*, Jurman G, Riccadonna S, Furlanello C, 2012, PLOS One 7(8): e41882.

Sensitivity and specificity

Finding a useful classifier often involves a balance between predictions that are overly conservative and overly aggressive. For example, an email filter could guarantee to eliminate every spam message by aggressively filtering nearly every ham message. On the other hand, to guarantee that no ham messages will be inadvertently filtered might require us to allow an unacceptable amount of spam to pass through the filter. A pair of performance measures captures this tradeoff: sensitivity and specificity.

The **sensitivity** of a model (also called the **true positive rate**) measures the proportion of positive examples that were correctly classified. Therefore, as shown in the following formula, it is calculated as the number of true positives divided by the total number of positives, both those correctly classified (the true positives) and those incorrectly classified (the false negatives):

$$\text{sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

The **specificity** of a model (also called the **true negative rate**) measures the proportion of negative examples that were correctly classified. As with sensitivity, this is computed as the number of true negatives divided by the total number of negatives—the true negatives plus the false positives.

$$\text{specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Given the confusion matrix for the SMS classifier, we can easily calculate these measures by hand. Assuming that spam is a positive class, we can confirm that the numbers in the `confusionMatrix()` output are correct. For example, the calculation for sensitivity is:

```
> sens <- 152 / (152 + 31)
> sens
```

```
[1] 0.8306011
```

Similarly, for specificity, we can calculate:

```
> spec <- 1203 / (1203 + 4)  
> spec
```

```
[1] 0.996686
```

The `caret` package provides functions for calculating sensitivity and specificity directly from vectors of predicted and actual values. Be careful to specify the `positive` or `negative` parameter appropriately, as shown in the following lines:

```
> library(caret)  
> sensitivity(sms_results$predict_type, sms_results$actual_type,  
    positive = "spam")
```

```
[1] 0.8306011
```

```
> specificity(sms_results$predict_type, sms_results$actual_type,  
    negative = "ham")
```

```
[1] 0.996686
```

Sensitivity and specificity range from 0 to 1, with values close to 1 being more desirable. Of course, it is important to find an appropriate balance between the two—a task that is often quite context-specific.

For example, in this case, the sensitivity of 0.831 implies that 83.1 percent of the spam messages were correctly classified. Similarly, the specificity of 0.997 implies that 99.7 percent of non-spam messages were correctly classified, or alternatively, 0.3 percent of valid messages were rejected as spam. The idea of rejecting 0.3 percent of valid SMS messages may be unacceptable, or it may be a reasonable tradeoff given the reduction in spam.

Sensitivity and specificity provide tools for thinking about such tradeoffs. Typically, changes are made to the model and different models are tested until you find one that meets a desired sensitivity and specificity threshold. Visualizations, such as those discussed later in this chapter, can also assist with understanding the balance between sensitivity and specificity.

Precision and recall

Closely related to sensitivity and specificity are two other performance measures related to compromises made in classification: precision and recall. Used primarily in the context of information retrieval, these statistics are intended to indicate how interesting and relevant a model's results are, or whether the predictions are diluted by meaningless noise.

The **precision** (also known as the **positive predictive value**) is defined as the proportion of positive predictions that are truly positive; in other words, when a model predicts the positive class, how often is it correct? A precise model will only predict the positive class in cases very likely to be positive. It will be very trustworthy.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Consider what would happen if the model was very imprecise. Over time, the results would be less likely to be trusted. In the context of information retrieval, this would be analogous to a search engine like Google returning irrelevant results. Eventually, users would switch to a competitor like Bing. In the case of the SMS spam filter, high precision means that the model is able to carefully target only the spam while avoiding false positives in the ham.

On the other hand, **recall** is a measure of how complete the results are. As shown in the following formula, this is defined as the number of true positives over the total number of positives. You may have already recognized this as the same as sensitivity; however, the interpretation differs slightly.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

A model with high recall captures a large portion of the positive examples, meaning that it has a wide breadth. For example, a search engine with high recall returns a large number of documents pertinent to the search query. Similarly, the SMS spam filter has high recall if the majority of spam messages are correctly identified.

We can calculate precision and recall from the confusion matrix. Again, assuming that spam is a positive class, the precision is:

```
> prec <- 152 / (152 + 4)
> prec
[1] 0.974359
```

And the recall is:

```
> rec <- 152 / (152 + 31)
> rec
[1] 0.8306011
```

The caret package can be used to compute either of these measures from vectors of predicted and actual classes. Precision uses the `posPredValue()` function:

```
> library(caret)
> posPredValue(sms_results$predict_type, sms_results$actual_type,
  positive = "spam")
[1] 0.974359
```

Recall uses the `sensitivity()` function that we used earlier:

```
> sensitivity(sms_results$predict_type, sms_results$actual_type,
  positive = "spam")
[1] 0.8306011
```

Like the tradeoff between sensitivity and specificity, for most real-world problems, it is difficult to build a model with both high precision and high recall. It is easy to be precise if you target only the low-hanging fruit—the easiest-to-classify examples. Similarly, it is easy for a model to have a high recall by casting a very wide net, meaning that the model is overly aggressive at identifying positive cases. In contrast, having both high precision and recall at the same time is very challenging. It is therefore important to test a variety of models in order to find the combination of precision and recall that meets the needs of your project.

The F-measure

A measure of model performance that combines precision and recall into a single number is known as the **F-measure** (also sometimes called the **F₁ score** or the **F-score**). The F-measure combines precision and recall using the **harmonic mean**, a type of average that is used for rates of change. The harmonic mean is used rather than the more common arithmetic mean since both precision and recall are expressed as proportions between 0 and 1, which can be interpreted as rates. The following is the formula for the F-measure:

$$\text{F-measure} = \frac{2 \times \text{precision} \times \text{recall}}{\text{recall} + \text{precision}} = \frac{2 \times \text{TP}}{2 \times \text{TP} + \text{FP} + \text{FN}}$$

To calculate the F-measure, use the precision and recall values computed previously:

```
> f <- (2 * prec * rec) / (prec + rec)
> f
[1] 0.8967552
```

This comes out exactly the same as using the counts from the confusion matrix:

```
> f <- (2 * 152) / (2 * 152 + 4 + 31)  
> f
```

```
[1] 0.8967552
```

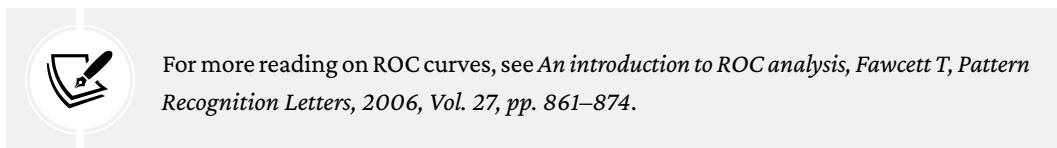
Since the F-measure describes model performance in a single number, it provides a convenient, quantitative metric to compare several models directly. Indeed, the F-measure was once virtually a gold standard measure of a model of performance, but today, it seems to be much less widely used than it was previously. One potential explanation is that it assumes that equal weight should be assigned to precision and recall, an assumption that is not always valid, depending on the real-world costs of false positives and false negatives. Of course, it is possible to calculate F-scores using different weights for precision and recall, but choosing the weights can be tricky at best and arbitrary at worst. This being said, perhaps a more important reason why this metric has fallen out of favor is the adoption of methods that visually depict a model's performance on different subsets of data, such as those described in the next section.

Visualizing performance tradeoffs with ROC curves

Visualizations are helpful for understanding the performance of machine learning algorithms in greater detail. Where statistics such as sensitivity and specificity, or precision and recall, attempt to boil model performance down to a single number, visualizations depict how a learner performs across a wide range of conditions.

Because learning algorithms have different biases, it is possible that two models with similar accuracy could have drastic differences in how they achieve their accuracy. Some models may struggle with certain predictions that others make with ease while breezing through cases that others struggle to get right. Visualizations provide a method for understanding these tradeoffs by comparing learners side by side in a single chart.

The **receiver operating characteristic (ROC)** curve is commonly used to examine the tradeoff between the detection of true positives while avoiding false positives. As you might suspect from the name, ROC curves were developed by engineers in the field of communications. Around the time of World War II, radar and radio operators used ROC curves to measure a receiver's ability to discriminate between true signals and false alarms. The same technique is useful today for visualizing the efficacy of machine learning models.



The characteristics of a typical ROC diagram are depicted in *Figure 10.4*. The ROC curve is drawn using the proportion of true positives on the vertical axis and the proportion of false positives on the horizontal axis. Because these values are equivalent to sensitivity and $(1 - \text{specificity})$, respectively, the diagram is also known as a **sensitivity/specificity plot**.

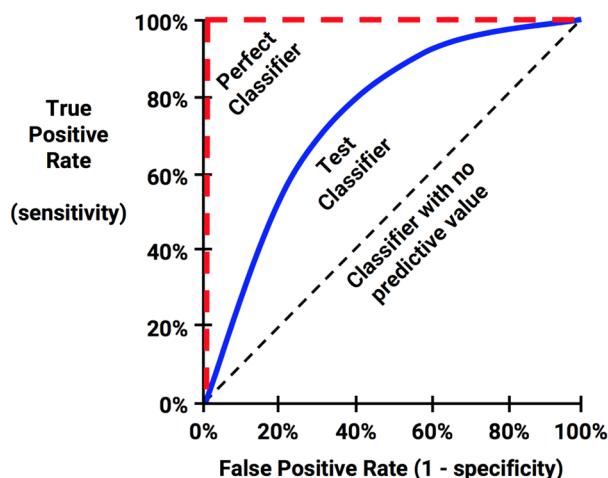


Figure 10.4: The ROC curve depicts classifier shapes relative to perfect and useless classifiers

The points comprising ROC curves indicate the true positive rate at varying false positive thresholds. To illustrate this concept, three hypothetical classifiers are contrasted in the previous plot. First, the *perfect classifier* has a curve that passes through the point at a 100 percent true positive rate and a 0 percent false positive rate. It is able to correctly identify all of the true positives before it incorrectly classifies any negative result. Next, the diagonal line from the bottom-left to the top-right corner of the diagram represents a *classifier with no predictive value*. This type of classifier detects true positives and false positives at exactly the same rate, implying that the classifier cannot discriminate between the two. This is the baseline by which other classifiers may be judged. ROC curves falling close to this line indicate models that are not very useful. Lastly, most real-world classifiers are like the *test classifier*, in that they fall somewhere in the zone between perfect and useless.

The best way to understand how the ROC curve is constructed is to create one by hand. The values in the table depicted in *Figure 10.5* indicate predictions of a hypothetical spam model applied to a test set containing 20 examples, of which six are the positive class (spam) and 14 are the negative class (ham).

Estimated Spam Probability	Actual Message Type
0.95	spam
0.90	spam
0.75	spam
0.70	spam
0.60	ham
0.55	spam
0.51	ham
0.49	spam
0.38	ham
0.35	ham
0.30	ham
0.25	ham
0.21	ham
0.20	ham
0.19	ham
0.19	ham
0.18	ham
0.15	ham
0.11	ham
0.10	ham

Figure 10.5: To construct the ROC curve, the estimated probability values for the positive class are sorted in descending order, then compared to the actual class value

To create the curves, the classifier's predictions are sorted by the model's estimated probability of the positive class, in descending order, with the largest values first, as shown in the table. Then, beginning at the plot's origin, each prediction's impact on the true positive rate and false positive rate results in a curve tracing vertically for each positive example and horizontally for each negative example. This process can be performed by hand on a piece of graph paper, as depicted in *Figure 10.6*:

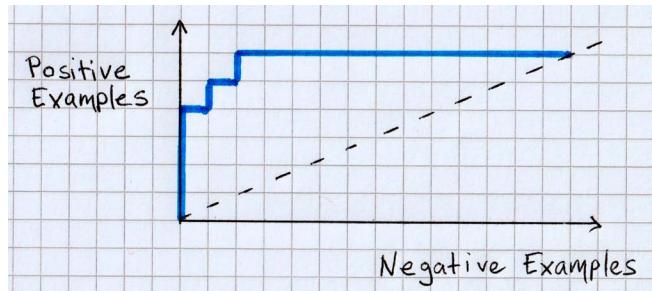


Figure 10.6: The ROC curve can be created by hand on graph paper by plotting the number of positive examples versus the number of negative examples

Note that the ROC curve is not complete at this point, because the axes are skewed due to the presence of more than twice as many negative examples as positive examples in the test set. A simple fix for this is to scale the plot proportionally so that the two axes are equivalent in size, as depicted in *Figure 10.7*:

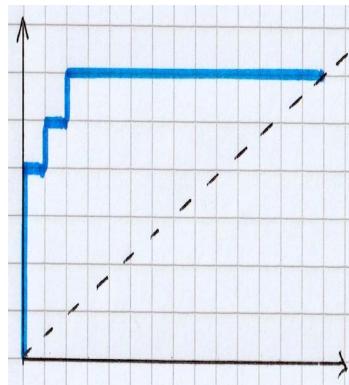


Figure 10.7: Scaling the plot's axes creates a proportionate comparison, regardless of the initial balance of positive and negative examples

If we imagine that both the x and y axes now range from 0 to 1, we can interpret each axis as a percentage. The y axis is the number of positives and originally ranged from 0 to 6; after shrinking it to a scale from 0 to 1, each increment becomes $1/6$. On this scale, we can think of the vertical coordinate of the ROC curve as the number of true positives divided by the total number of positives, which is the true positive rate, or sensitivity. Similarly, the x axis measures the number of negatives; by dividing by the total number of negatives (14 in this example), we obtain the true negative rate, or specificity.

The table in *Figure 10.8* depicts these calculations for all 20 examples in the hypothetical test set:

Example Spam Model						
Predicted Prob. of Spam	Actual Type	True Positives	True Negatives	TP Rate (Sensitivity)	TN Rate (Specificity)	FP Rate (1 - Specificity)
0.95	spam	1	14	16.7%	100.0%	0.0%
0.90	spam	2	14	33.3%	100.0%	0.0%
0.75	spam	3	14	50.0%	100.0%	0.0%
0.70	spam	4	14	66.7%	100.0%	0.0%
0.60	ham	4	13	66.7%	92.9%	7.1%
0.55	spam	5	13	83.3%	92.9%	7.1%
0.51	ham	5	12	83.3%	85.7%	14.3%
0.49	spam	6	12	100.0%	85.7%	14.3%
0.38	ham	6	11	100.0%	78.6%	21.4%
0.35	ham	6	10	100.0%	71.4%	28.6%
0.30	ham	6	9	100.0%	64.3%	35.7%
0.25	ham	6	8	100.0%	57.1%	42.9%
0.21	ham	6	7	100.0%	50.0%	50.0%
0.20	ham	6	6	100.0%	42.9%	57.1%
0.19	ham	6	5	100.0%	35.7%	64.3%
0.19	ham	6	4	100.0%	28.6%	71.4%
0.18	ham	6	3	100.0%	21.4%	78.6%
0.15	ham	6	2	100.0%	14.3%	85.7%
0.11	ham	6	1	100.0%	7.1%	92.9%
0.10	ham	6	0	100.0%	0.0%	100.0%

Figure 10.8: The ROC curve traces what happens to the model's true positive rate versus the false positive rate, for increasingly large sets of examples

An important property of ROC curves is that they are not affected by the class imbalance problem in which one of the two outcomes—typically the positive class—is much rarer than the other. Many performance metrics, such as accuracy, can be misleading for imbalanced data. This is not the case for ROC curves, because both dimensions of the plot are based solely on rates *within* the positive and negative values, and thus the ratio *across* positives and negatives does not affect the result. Because many of the most important machine learning tasks involve severely imbalanced outcomes, ROC curves are a very useful tool for understanding the overall quality of a model.

Comparing ROC curves

If ROC curves are helpful for evaluating a single model, it is unsurprising that they are also useful for comparing across models. Intuitively, we know that curves closer to the top-left of the plot area are better. In practice, the comparison is often more challenging than this, as differences between curves are often subtle rather than obvious, and the interpretation is nuanced and specific to how the model is to be used.

To understand the nuances, let's begin by considering what causes two models to trace different curves on the ROC plot. Beginning at the origin, the curve's length is extended as additional test set examples are predicted to be positive. Because the y axis represents the true positive rate and the x axis represents the false positive rate, a steeper upward trajectory is an implicit ratio, implying that the model is better at identifying the positive examples without making as many mistakes. This is illustrated in *Figure 10.9*, which depicts the start of ROC curves for two imaginary models. For the same number of predictions—indicated by the equal lengths of the vectors emerging from the origin—the first model has a higher true positive rate and a lower false positive rate, which implies it is the better performer of the two:

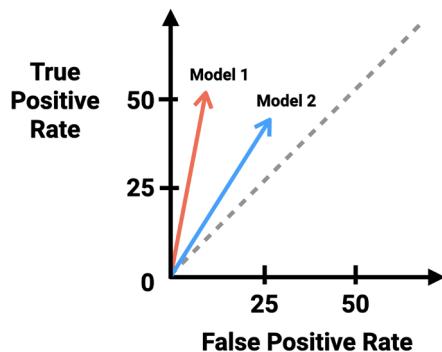


Figure 10.9: For the same number of predictions, model 1 outperforms model 2 because it has a higher true positive rate

Suppose we continue to trace the ROC curves for each of these two models, evaluating the model's predictions on the entire dataset. In this case, perhaps the first model continues to outperform the second at all points on the curve, as shown in *Figure 10.10*.

For all points on the curve, the first model has a higher true positive rate and a lower false positive rate, which means it is the better performer across the entire dataset:

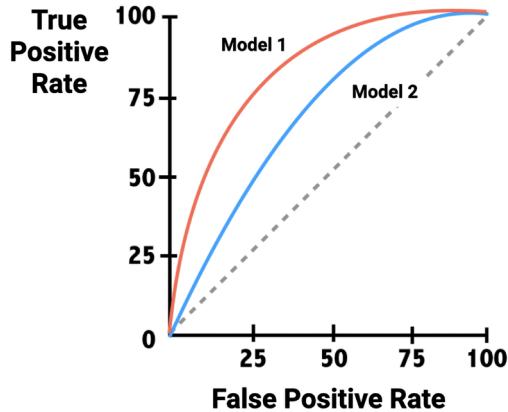


Figure 10.10: Model 1 consistently performs better than model 2, with a higher true positive rate and a lower false positive rate at all points on the curve

Although the second model is clearly inferior in the prior example, choosing the better performer is not always so easy. *Figure 10.11* depicts intersecting ROC curves, which suggests that neither model is the best performer for all applications:

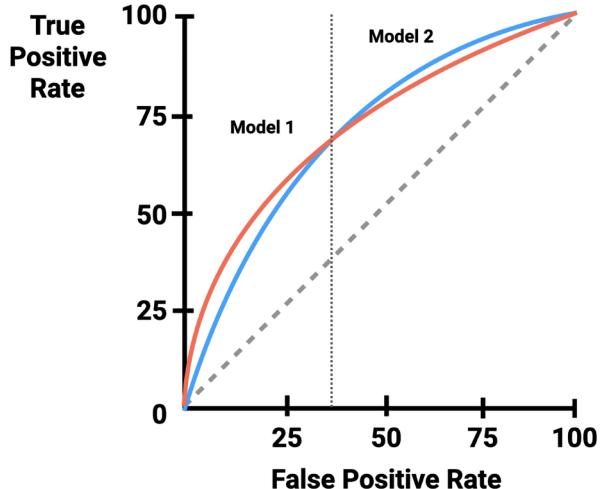


Figure 10.11: Both model 1 and model 2 are the better performers for different subsets of the data

The point of intersection between the two ROC curves splits the plot into two areas: one in which the first model has a higher true positive rate and the other in which the opposite is true. So, how do we know which model is “best” for any given use case?

To answer this question, when comparing two curves, it helps to understand that both models are trying to sort the dataset in order of the highest to the lowest probability that each example is of the positive class. Models that are better able to sort the dataset in this way will have ROC curves closer to the top-left of the plot.

The first model in *Figure 10.11* jumps to an early lead because it was able to sort a larger number of positive examples to the very front of the dataset, but after this initial surge, the second model was able to catch up and outperform the other by slowly and steadily sorting positive examples in front of negative examples over the remainder of the dataset. Although the second model may have better overall performance on the full dataset, we tend to prefer models that perform better early—the ones that take the so-called “low-hanging fruit” in the dataset. The justification for preferring these models is that many real-world models are used only for action on a subset of the data.

For example, consider a model used to identify customers that are most likely to respond to a direct mail advertising campaign. If we could afford to mail all potential customers, a model would be unnecessary. But because we don’t have the budget to send the advertisement to every address, the model is used to estimate the probability that the recipient will purchase the product after viewing the advertisement. A model that is better at sorting the true most likely purchasers to the front of the list will have a greater slope early in the ROC curve and will shrink the marketing budget needed to acquire purchasers. In *Figure 10.11*, the first model would be a better fit for this task.

In contrast to this approach, another consideration is the relative costs of various types of errors; false positives and false negatives often have a different impact in the real world. If we know that a spam filter or a cancer screening needs to target a specific true positive rate, such as 90 percent or 99 percent, we will favor the model that has the lower false positive rate at the desired levels. Although neither model would be very good due to the high false positive rate, *Figure 10.11* suggests that the second model would be slightly preferable for these applications.

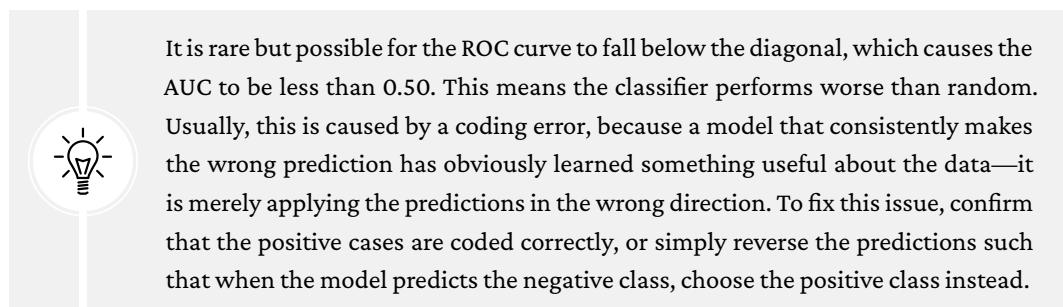
As these examples demonstrate, ROC curves allow model performance comparisons that also consider how the models will be used. This flexibility is appreciated over simpler numeric metrics like accuracy or kappa, but it may also be desirable to quantify the ROC curve in a single metric that can be compared quantitatively, much like these statistics. The next section introduces exactly this type of measure.

The area under the ROC curve

Comparing ROC curves can be somewhat subjective and context-specific, so metrics that reduce performance to a single numeric value are always in demand to simplify and bring objectivity into the comparison. While it may be difficult to say what makes a “good” ROC curve, in general, we know that the closer the ROC curve is to the top-left of the plot, the better it is at identifying positive values. This can be measured using a statistic known as the **area under the ROC curve (AUC)**. The AUC treats the ROC diagram as a two-dimensional square and measures the total area under the ROC curve. The AUC ranges from 0.5 (for a classifier with no predictive value) to 1.0 (for a perfect classifier). A convention for interpreting AUC scores uses a system similar to academic letter grades:

- **A:** Outstanding = 0.9 to 1.0
- **B:** Excellent/Good = 0.8 to 0.9
- **C:** Acceptable/Fair = 0.7 to 0.8
- **D:** Poor = 0.6 to 0.7
- **E:** No Discrimination = 0.5 to 0.6

As with most scales like this, the levels may work better for some tasks than others; the boundaries across categories are naturally somewhat fuzzy.



When the use of AUC started to become widespread, some treated it as a definitive measure of model performance, although unfortunately, this is not true in all cases. Generally speaking, higher AUC values reflect classifiers that are better at sorting a random positive example higher than a random negative example. However, *Figure 10.12* illustrates the important fact that two ROC curves may be shaped very differently, yet have an identical AUC:

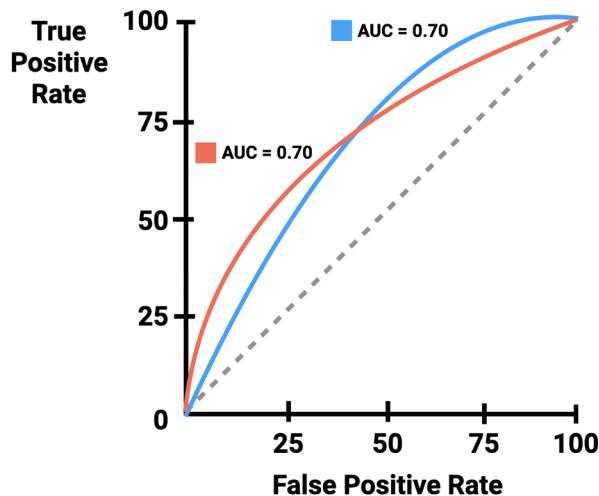


Figure 10.12: ROC curves may have different performances despite having the same AUC

Because the AUC is a simplification of the ROC curve, the AUC alone is insufficient to identify the “best” model for all use cases. The safest practice is to use the AUC in combination with a qualitative examination of the ROC curve, as described earlier in this chapter. If two models have an identical or similar AUC, it is usually preferable to choose the one that performs better early. Furthermore, even in the case that one model has a better overall AUC, a model that has a higher initial true positive rate may be preferred for applications that will use only a subset of the most confident predictions.

Creating ROC curves and computing AUC in R

The pROC package provides an easy-to-use set of functions for creating ROC curves and computing the AUC. The pROC website (at <https://web.expasy.org/pROC/>) includes a list of the full set of features, as well as several examples of visualization capabilities. Before continuing, be sure that you have installed the package using the `install.packages("pROC")` command.



For more information on the pROC package, see *pROC: an open-source package for R and S+ to analyze and compare ROC curves*, Robin, X, Turck, N, Hainard, A, Tiberti, N, Lisacek, F, Sanchez, JC, and Mueller M, *BMC Bioinformatics*, 2011, pp. 12-77.

To create visualizations with pROC, two vectors of data are needed. The first must contain the estimated probability of the positive class and the second must contain the predicted class values.

For the SMS classifier, we'll supply the estimated spam probabilities and the actual class labels to the `roc()` function as follows:

```
> library(pROC)
> sms_roc <- roc(sms_results$prob_spam, sms_results$actual_type)
```

Using the `sms_roc` object, we can visualize the ROC curve with R's `plot()` function. As shown in the following command, many of the standard parameters for adjusting the plot can be used, such as `main` (for adding a title), `col` (for changing the line color), and `lwd` (for adjusting the line width). The `grid` parameter adds faint gridlines to the plot to aid readability, and the `legacy.axes` parameter instructs pROC to label the x axis as $1 - \text{specificity}$, which is a popular convention because it is equivalent to the false positive rate:

```
> plot(sms_roc, main = "ROC curve for SMS spam filter",
      Col = "blue", lwd = 2, grid = TRUE, legacy.axes = TRUE)
```

The result is an ROC curve for the Naive Bayes classifier and a diagonal reference line representing the baseline classifier with no predictive value:

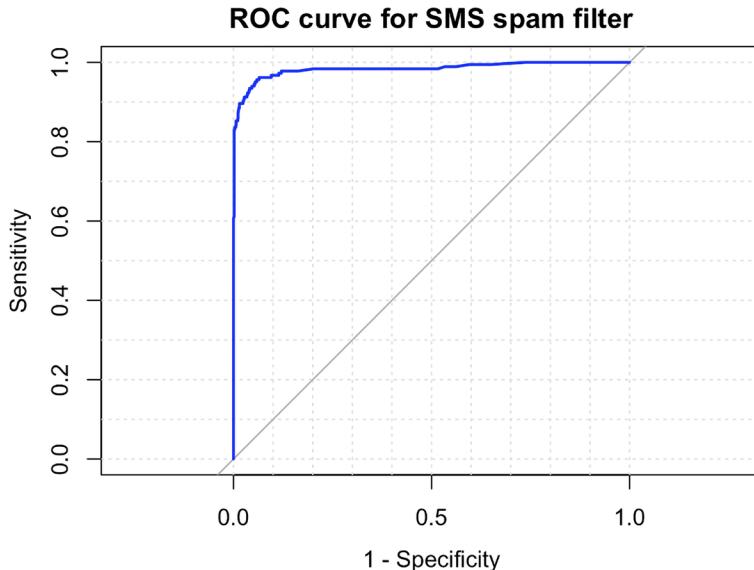


Figure 10.13: The ROC curve for the Naive Bayes SMS classifier

Qualitatively, we can see that this ROC curve appears to occupy the space in the top-left corner of the diagram, which suggests that it is closer to a perfect classifier than the dashed line representing a useless classifier.

To compare this model's performance to other models making predictions on the same dataset, we can add additional ROC curves to the same plot. Suppose that we had also trained a k-NN model on the SMS data using the `knn()` function described in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*. Using this model, the predicted probabilities of spam were computed for each record in the test set and saved to a CSV file, which we can load here. After loading the file, we'll apply the `roc()` function as before to compute the ROC curve, then use the `plot()` function with the parameter `add = TRUE` to add the curve to the previous plot:

```
> sms_results_knn <- read.csv("sms_results_knn.csv")
> sms_roc_knn <- roc(sms_results$actual_type,
  sms_results_knn$p_spam)
> plot(sms_roc_knn, col = "red", lwd = 2, add = TRUE)
```

The resulting visualization has a second curve depicting the performance of the k-NN model making predictions on the same test set as the Naive Bayes model. The curve for k-NN is consistently lower, suggesting that it is a consistently worse model than the Naive Bayes approach:

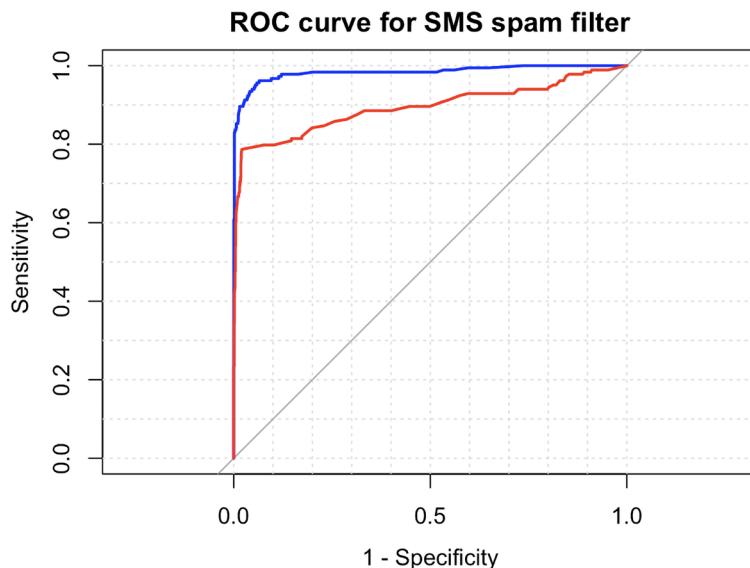


Figure 10.14: ROC curves comparing the performance of Naive Bayes (topmost curve) and k-NN (bottom curve) on the SMS test set

To confirm this quantitatively, we can use the `pROC` package to calculate the AUC. To do so, we simply apply the package’s `auc()` function to the `sms_roc` object for each model, as shown in the following code:

```
> auc(sms_roc)  
Area under the curve: 0.9836  
  
> auc(sms_roc_knn)  
Area under the curve: 0.8942
```

The AUC for the Naive Bayes SMS classifier is 0.98, which is extremely high and substantially better than the k-NN classifier’s AUC of 0.89. But how do we know whether the model is just as likely to perform well on another dataset, or whether the difference is greater than expected by chance alone? In order to answer such questions, we need to better understand how far we can extrapolate a model’s predictions beyond the test data. Such methods are described in the sections that follow.



This was mentioned before, but is worth repeating: the AUC value alone is often insufficient for identifying a “best” model. In this example, the AUC does identify the better model because the ROC curves do not intersect—the Naive Bayes model has a better true positive rate at all points on the ROC curve. When ROC curves *do* intersect, the “best” model will depend on how the model will be used. Additionally, it is possible to combine learners with intersecting ROC curves into even stronger models using the techniques covered in *Chapter 14, Building Better Learners*.

Estimating future performance

Some R machine learning packages present confusion matrices and performance measures during the model-building process. The purpose of these statistics is to provide insight into the model’s **resubstitution error**, which occurs when the target values of training examples are incorrectly predicted, despite the model being trained on this data. This can be used as a rough diagnostic to identify obviously poor performers. A model that cannot perform sufficiently well on the data it was trained on is unlikely to do well on future data.

The opposite is not true. In other words, a model that performs well on the training data cannot be assumed to perform well on future datasets. For example, a model that used rote memorization to perfectly classify every training instance with zero resubstitution error would be unable to generalize its predictions to data it has never seen before. For this reason, the error rate on the training data can be assumed to be optimistic about a model's future performance.

Instead of relying on resubstitution error, a better practice is to evaluate a model's performance on data it has not yet seen. We used such a method in previous chapters when we split the available data into a set for training and a set for testing. In some cases, however, it is not always ideal to create training and test datasets. For instance, in a situation where you have only a small pool of data, you might not want to reduce the sample any further.

Fortunately, as you will soon learn, there are other ways to estimate a model's performance on unseen data. The `caret` package we used to calculate performance measures also offers functions to estimate future performance. If you are following along with the R code examples and haven't already installed the `caret` package, please do so. You will also need to load the package to the R session using the `library(caret)` command.

The holdout method

The procedure of partitioning data into training and test datasets that we used in previous chapters is known as the **holdout method**. As shown in *Figure 10.15*, the **training dataset** is used to generate the model, which is then applied to the **test dataset** to generate predictions for evaluation. Typically, about one-third of the data is held out for testing and two-thirds is used for training, but this proportion can vary depending on the amount of available data or the complexity of the learning task. To ensure that the training and test datasets do not have systematic differences, their examples are randomly divided into two groups.

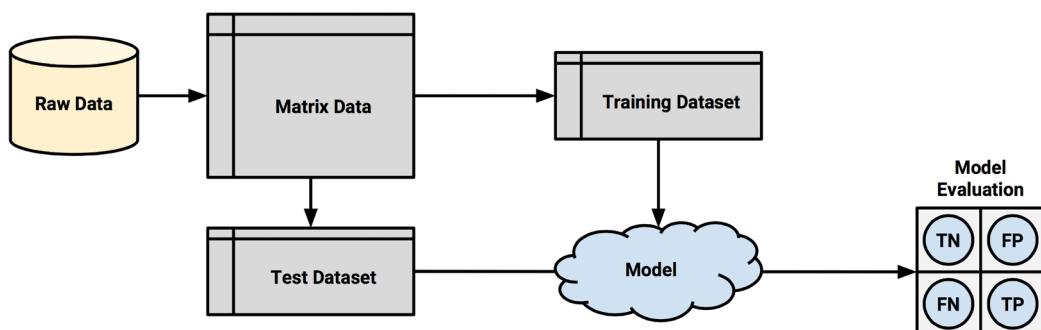


Figure 10.15: The simplest holdout method divides the data into training and test sets

For the holdout method to result in a truly accurate estimate of future performance, at no time should performance on the test dataset be allowed to influence the modeling process. In the words of Stanford professor and renowned machine learning expert Trevor Hastie, “*ideally the test set should be kept in a ‘vault,’ and be brought out only at the end of the data analysis.*” In other words, the test data should remain untouched aside from its one and only purpose, which is to evaluate a single, final model.



For more information, see *The Elements of Statistical Learning (2nd edition)*, Hastie, Tibshirani, and Friedman (2009), p. 222.

It is easy to unknowingly violate this rule and peek into the metaphorical “vault” when choosing one of several models or changing a single model based on the results of repeated testing. For example, suppose we built several models on the training data and selected the one with the highest accuracy on the test data. In this case, because we have used the test dataset to cherry-pick the best result, the test performance is not an unbiased measure of future performance on unseen data.



A keen reader will note that holdout test data was used in previous chapters to both evaluate models and improve model performance. This was done for illustrative purposes but would indeed violate the rule stated previously. Consequently, the model performance statistics shown were not truly unbiased estimates of future performance on unseen data.

To avoid this problem, it is better to divide the original data so that in addition to the training and test datasets, a **validation dataset** is available. The validation dataset can be used for iterating and refining the model or models chosen, leaving the test dataset to be used only once as a final step to report an estimated error rate for future predictions. A typical split between training, test, and validation would be 50 percent, 25 percent, and 25 percent, respectively.

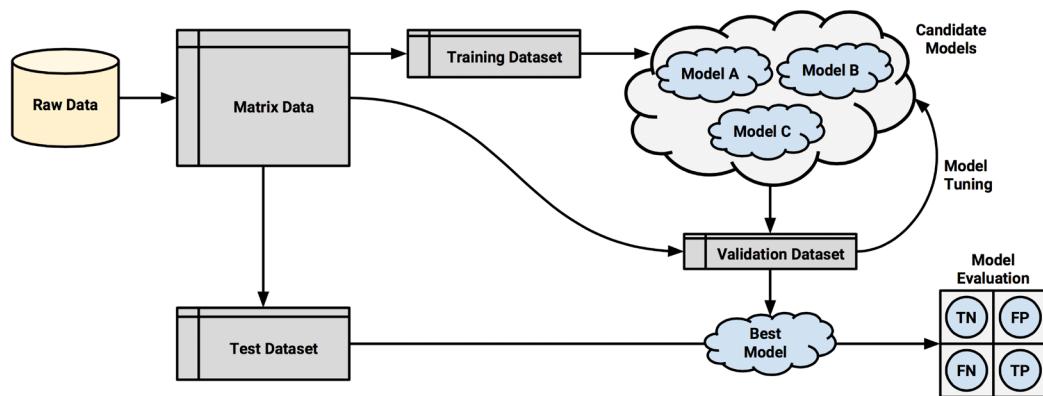


Figure 10.16: A validation dataset can be held out from training to select from multiple candidate models

A simple method for creating holdout samples uses random number generators to assign records to partitions. This technique was first used in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, to create training and test datasets.



If you'd like to follow along with the following examples, download the `credit.csv` dataset from Packt Publishing's website and load it into a data frame using the `credit <- read.csv("credit.csv", stringsAsFactors = TRUE)` command.

Suppose we have a data frame named `credit` with 1,000 rows of data. We can divide this into three partitions as follows. First, we create a vector of randomly ordered row IDs from 1 to 1,000 using the `runif()` function, which, by default, generates a specified number of random values between 0 and 1. The `runif()` function gets its name from the random uniform distribution, which was discussed in *Chapter 2, Managing and Understanding Data*.

The `order()` function then returns a vector indicating the rank order of the 1,000 random numbers. For instance, `order(c(0.5, 0.25, 0.75, 0.1))` returns the sequence 4 2 1 3 because the smallest number (0.1) appears fourth, the second smallest (0.25) appears second, and so on:

```
> random_ids <- order(runif(1000))
```

Next, the random IDs are used to divide the credit data frame into 500, 250, and 250 records comprising the training, validation, and test datasets:

```
> credit_train <- credit[random_ids[1:500], ]  
> credit_validate <- credit[random_ids[501:750], ]  
> credit_test <- credit[random_ids[751:1000], ]
```

One problem with holdout sampling is that each partition may have a larger or smaller proportion of some classes. In cases where one (or more) class is a very small proportion of the dataset, this can lead to it being omitted from the training dataset—a significant problem because the model cannot then learn this class.

To reduce the chance of this occurring, a technique called **stratified random sampling** can be used. Although a random sample should generally contain roughly the same proportion of each class value as the full dataset, stratified random sampling guarantees that the random partitions have nearly the same proportion of each class as the full dataset, even when some classes are small.

The `caret` package provides a `createDataPartition()` function, which creates partitions based on stratified holdout sampling. The steps for creating a stratified sample of training and test data for the `credit` dataset are shown in the following commands. To use the function, a vector of class values must be specified (here, `default` refers to whether a loan went into default), in addition to a parameter, `p`, which specifies the proportion of instances to be included in the partition. The `list = FALSE` parameter prevents the result from being stored as a list object—a capability that is needed for more complex sampling techniques, but is unnecessary here:

```
> in_train <- createDataPartition(credit$default, p = 0.75, list = FALSE)  
> credit_train <- credit[in_train, ]  
> credit_test <- credit[-in_train, ]
```

The `in_train` vector indicates the row numbers included in the training sample. We can use these row numbers to select examples for the `credit_train` data frame. Similarly, by using a negative symbol, we can use the rows not found in the `in_train` vector for the `credit_test` dataset.

Although it distributes the classes evenly, stratified sampling does not guarantee other types of representativeness. Some samples may have too many or too few difficult cases, easy-to-predict cases, or outliers. This is especially true for smaller datasets, which may not have a large enough portion of such cases to divide among training and test sets.

In addition to potentially biased samples, another problem with the holdout method is that substantial portions of data must be reserved for testing and validating the model. Since this data cannot be used to train the model until its performance has been measured, the performance estimates are likely to be overly conservative.



Since models trained on larger datasets generally perform better, a common practice is to retrain the model on the full set of data (that is, training plus test and validation) after a final model has been selected and evaluated.

A technique called **repeated holdout** is sometimes used to mitigate the problems of randomly composed training datasets. The repeated holdout method is a special case of the holdout method that uses the average result from several random holdout samples to evaluate a model's performance. As multiple holdout samples are used, it is less likely that the model is trained or tested on non-representative data. We'll expand on this idea in the next section.

Cross-validation

The repeated holdout is the basis of a technique known as **k-fold cross-validation (k-fold CV)**, which has become the industry standard for estimating model performance. Rather than taking repeated random samples that could potentially use the same record more than once, k-fold CV randomly divides the data into k separate random partitions called **folds**.

Although k can be set to any number, by far the most common convention is to use a 10-fold CV. Why 10 folds? The reason is that empirical evidence suggests that there is little added benefit to using a greater number. For each of the 10 folds (each comprising 10 percent of the total data), a machine learning model is built on the remaining 90 percent of the data. The fold's 10 percent sample is then used for model evaluation. After the process of training and evaluating the model has occurred 10 times (with 10 different training/testing combinations), the average performance across all folds is reported.



An extreme case of k-fold CV is the **leave-one-out method**, which performs k-fold CV using a fold for each one of the data's examples. This ensures that the greatest amount of data is used for training the model. Although this may seem useful, it is so computationally expensive that it is rarely used in practice.

Datasets for CV can be created using the `createFolds()` function in the `caret` package. Like stratified random holdout sampling, this function will attempt to maintain the same class balance in each of the folds as in the original dataset. The following is the command to create 10 folds, using `set.seed(123)` to ensure the results are reproducible:

```
> set.seed(123)
> folds <- createFolds(credit$default, k = 10)
```

The result of the `createFolds()` function is a list of vectors storing the row numbers for each of the `k = 10` requested folds. We can peek at the contents using `str()`:

```
> str(folds)
```

```
List of 10
$ Fold01: int [1:100] 14 23 32 42 51 56 65 66 77 95 ...
$ Fold02: int [1:100] 21 36 52 55 96 115 123 129 162 169 ...
$ Fold03: int [1:100] 3 22 30 34 37 39 43 58 70 85 ...
$ Fold04: int [1:100] 12 15 17 18 19 31 40 45 47 57 ...
$ Fold05: int [1:100] 1 5 7 20 26 35 46 54 106 109 ...
$ Fold06: int [1:100] 6 27 29 48 68 69 72 73 74 75 ...
$ Fold07: int [1:100] 10 38 49 60 61 63 88 94 104 108 ...
$ Fold08: int [1:100] 8 11 24 53 71 76 89 90 91 101 ...
$ Fold09: int [1:100] 2 4 9 13 16 25 28 44 62 64 ...
$ Fold10: int [1:100] 33 41 50 67 81 82 100 105 107 118 ...
```

Here, we see that the first fold is named `Fold01` and stores 100 integers, indicating the 100 rows in the `credit` data frame for the first fold. To create training and test datasets to build and evaluate a model, an additional step is needed. The following commands show how to create data for the first fold. We'll assign the selected 10 percent to the test dataset and use the negative symbol to assign the remaining 90 percent to the training dataset:

```
> credit01_test <- credit[folds$Fold01, ]
> credit01_train <- credit[-folds$Fold01, ]
```

To perform the full 10-fold CV, this step would need to be repeated a total of 10 times, first building a model and then calculating the model's performance each time. In the end, the performance measures would be averaged to obtain the overall performance. Thankfully, we can automate this task by applying several of the techniques we learned earlier.

To demonstrate the process, we'll estimate the kappa statistic for a C5.0 decision tree model of the credit data using 10-fold CV. First, we need to load some R packages: `caret` (to create the folds), `C50` (to build the decision tree), and `irr` (to calculate kappa). The latter two packages were chosen for illustrative purposes; if you desire, you can use a different model or a different performance measure with the same series of steps:

```
> library(caret)
> library(C50)
> library(irr)
```

Next, we'll create a list of 10 folds as we have done previously. As before, the `set.seed()` function is used here to ensure that the results are consistent if the same code is run again:

```
> set.seed(123)
> folds <- createFolds(credit$default, k = 10)
```

Finally, we will apply a series of identical steps to the list of folds using the `lapply()` function. As shown in the following code, because there is no existing function that does exactly what we need, we must define our own function to pass to `lapply()`. Our custom function divides the credit data frame into training and test data, builds a decision tree using the `C5.0()` function on the training data, generates a set of predictions from the test data, and compares the predicted and actual values using the `kappa2()` function:

```
> cv_results <- lapply(folds, function(x) {
  credit_train <- credit[-x, ]
  credit_test <- credit[x, ]
  credit_model <- C5.0(default ~ ., data = credit_train)
  credit_pred <- predict(credit_model, credit_test)
  credit_actual <- credit_test$default
  kappa <- kappa2(data.frame(credit_actual, credit_pred))$value
  return(kappa)
})
```

The resulting kappa statistics are compiled into a list stored in the `cv_results` object, which we can examine using `str()`:

```
> str(cv_results)

List of 10
 $ Fold01: num 0.381
 $ Fold02: num 0.525
```

```
$ Fold03: num 0.247  
$ Fold04: num 0.316  
$ Fold05: num 0.387  
$ Fold06: num 0.368  
$ Fold07: num 0.122  
$ Fold08: num 0.141  
$ Fold09: num 0.0691  
$ Fold10: num 0.381
```

There's just one more step remaining in the 10-fold-CV process: we must calculate the average of these 10 values. Although you will be tempted to type `mean(cv_results)`, because `cv_results` is not a numeric vector, the result would be an error. Instead, use the `unlist()` function, which eliminates the list structure and reduces `cv_results` to a numeric vector. From there, we can calculate the mean kappa as expected:

```
> mean(unlist(cv_results))  
[1] 0.2939567
```

This kappa statistic is relatively low, corresponding to "fair" on the interpretation scale, which suggests that the credit scoring model performs only marginally better than random chance. In *Chapter 14, Building Better Learners*, we'll examine automated methods based on a 10-fold CV, which can assist us with improving the performance of this model.

Because CV provides a performance estimate from multiple test sets, we can also compute the variability in the estimate. For example, the standard deviation of the 10 iterations can be computed as:

```
> sd(unlist(cv_results))  
[1] 0.1448565
```

After finding the average and standard deviation of the performance metric, it is possible to calculate a confidence interval or determine whether two models have a **statistically significant** difference in performance, which means that it is likely that the difference is real and not due to random variation.

Unfortunately, recent research has demonstrated that CV violates assumptions of such statistical tests, particularly the need for the data to be drawn from independent random samples, which is clearly not the case for CV folds, which are linked to one another by definition.



For a discussion of the limitations of performance estimates taken from 10-fold CV, see *Cross-validation: what does it estimate and how well does it do it?*, Bates S, Hastie T, and Tibshirani R, 2022, <https://arxiv.org/abs/2104.00673>.

More sophisticated variants of CV have been developed to improve the robustness of model performance estimates. One such technique is **repeated k-fold CV**, which involves repeatedly applying k-fold CV and averaging the results. A common strategy is to perform 10-fold CV 10 times. Although computationally intensive, this provides an even more robust performance estimate than a standard 10-fold CV, as the performance is averaged over many more trials. However, it too violates statistical assumptions, and thus statistical tests performed on the results may be slightly biased.

Perhaps the current gold standard for estimating model performance is **nested cross-validation**, which literally performs k-fold CV within another k-fold CV process. This technique is described in *Chapter 11, Being Successful with Machine Learning*, and is not only extremely computationally expensive but is also more challenging to implement and interpret. The upside of nested k-fold CV is that it produces truly valid comparisons of model performance compared to standard k-fold CV, which is biased due to its violation of statistical assumptions. On the other hand, the bias caused by this issue seems to be less important for very large datasets, so it may still be reasonable—and remains a common practice—to use the confidence intervals or significance tests derived from the simpler CV approach to help identify the “best” model.

Bootstrap sampling

A slightly less popular, but very important, alternative to k-fold CV is known as **bootstrap sampling**, the **bootstrap**, or **bootstrapping** for short. Generally speaking, these refer to statistical methods that use random samples of data to estimate the properties of a larger set. When this principle is applied to machine learning model performance, it implies the creation of several randomly selected training and test datasets, which are then used to estimate performance statistics. The results from the various random datasets are then averaged to obtain a final estimate of future performance.

So, what makes this procedure different from k-fold CV? Whereas CV divides the data into separate partitions in which each example can appear only once, the bootstrap allows examples to be selected multiple times through a process of **sampling with replacement**. This means that from the original dataset of n examples, the bootstrap procedure will create one or more new training datasets that also contain n examples, some of which are repeated.

The corresponding test datasets are then constructed from the set of examples that were not selected for the respective training datasets.

In a bootstrapped dataset, the probability that any given instance is excluded from the training dataset is 36.8 percent. We can prove this mathematically by recognizing that each example has a $1/n$ chance of being sampled each time one of n rows is added to the training dataset. Therefore, to be in the test set, an example must *not* be selected n times. As the chance of being chosen is $1/n$, the chance of *not* being chosen is therefore $1 - 1/n$, and the probability of going unselected n times is as follows:

$$\left(1 - \frac{1}{n}\right)^n$$

Using this formula, if the dataset to be bootstrapped contains 1,000 rows, the probability of a random record being unselected is:

```
> (1 - (1/1000))^1000
```

```
[1] 0.3676954
```

Similarly, for a dataset with 100,000 rows:

```
> (1 - (1/100000))^100000
```

```
[1] 0.3678776
```

And as n approaches infinity, the formula reduces to $1/e$, as shown here:

```
> 1 / exp(1)
```

```
[1] 0.3678794
```

Given that the probability of being unselected is 36.8 percent, the probability of any instance being selected for the training dataset is $100 - 36.8 = 63.2$ percent. In other words, the training data represents only 63.2 percent of available examples, some of which are repeated. In contrast with 10-fold CV, which uses 90 percent of examples for training, the bootstrap sample is less representative of the full dataset.

Because a model trained on only 63.2 percent of the training data is likely to perform worse than a model trained on a larger training set, the bootstrap's performance estimates may be substantially lower than what will be obtained when the model is later trained on the full dataset.

A special case of bootstrapping, known as the **0.632 bootstrap**, accounts for this by calculating the final performance measure as a function of performance on both the training data (which is overly optimistic) and the test data (which is overly pessimistic). The final error rate is then estimated as:

$$\text{error} = 0.632 \times \text{error}_{\text{test}} + 0.368 \times \text{error}_{\text{train}}$$

One advantage of bootstrap sampling over CV is that it tends to work better with very small datasets. Additionally, bootstrap sampling has applications beyond performance measurement. In particular, in *Chapter 14, Building Better Learners*, you will learn how the principles of bootstrap sampling can be used to improve model performance.

Summary

This chapter presented several of the most common measures and techniques for evaluating the performance of machine learning classification models. Although accuracy provides a simple method for examining how often a model is correct, this can be misleading in the case of rare events because the real-life importance of such events may be inversely proportional to how frequently they appear in the data.

Some measures based on confusion matrices better capture a model's performance as well as the balance between the costs of various types of errors. The kappa statistic and Matthews correlation coefficient are two more sophisticated measures of performance, which work well even for severely unbalanced datasets. Additionally, closely examining the tradeoffs between sensitivity and specificity, or precision and recall, can be a useful tool for thinking about the implications of errors in the real world. Visualizations such as the ROC curve are also helpful to this end.

It is also worth mentioning that, sometimes, the best measure of a model's performance is to consider how well it meets, or doesn't meet, other objectives. For instance, you may need to explain a model's logic in simple language, which would eliminate some models from consideration. Additionally, even if it performs very well, a model that is too slow or difficult to scale to a production environment is completely useless.

Looking ahead to the chapters that follow, an obvious extension of *measuring* performance is finding ways to *improve* performance. As you continue in this book, you will apply many of the principles in this chapter while strengthening your machine learning abilities and adding more advanced skills. CV techniques, ROC curves, bootstrapping, and the *caret* package will reappear regularly in the coming pages, as we build upon our work so far to investigate ways to make smarter models by systematically iterating, refining, and combining learning algorithms.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



11

Being Successful with Machine Learning

An all-too-common problem in the field of machine learning occurs when students, with fresh excitement from learning the methods, struggle to apply what they've learned to real-world projects. Much as the beauty of a forest trail feels sinister in the darkness of night, code and methods that initially seemed straightforward feel daunting in the absence of a step-by-step roadmap. Without such a guide, the learning curve feels so much steeper and pitfalls appear deeper.

It is discouraging to think about the countless students that have been turned away from machine learning, due to the chasm between machine learning in theory and practice. Having worked in the field for over a decade, and having trained, interviewed, hired, and supervised numerous new practitioners, I have seen the challenges of this catch-22 firsthand. It is seemingly a paradox: gaining real-world experience in machine learning seems impossible without first having gained experience in machine learning!

The purpose of this chapter, as well as those that follow, is to serve as a bridge between the simple teaching examples in prior chapters and the unyielding complexity of the real world. In this chapter, you will learn:

- The factors that contribute to the success and failure of machine learning models
- Strategies for designing projects that are likely to perform well
- How to perform data exploration to spot potential issues early
- Why data science and competition are relevant to machine learning

Whether your definition of success involves finding a job in the field, building better machine learning models, or simply deepening your knowledge of the field's tools and techniques, you will find something to learn in the coming pages. You may even find yourself with a newfound desire to join many others in online machine learning competitions, which stretch your skills and put your knowledge to the test.

What makes a successful machine learning practitioner?

To be clear, the challenges of real-world machine learning are not due to the addition of more advanced or complex methods; after all, the first nine chapters of this book covered practical, real-world problems as diverse and challenging as identifying cancer cells, filtering spam messages, and predicting risky bank loans. Instead, the challenges of real-world machine learning have much to do with aspects of the field that are difficult to convey in a scripted setting, like a textbook or lecture. Machine learning is as much art as it is science, and just as it would be challenging to learn to paint, dance, or speak a foreign language without real-world practice, it is equally difficult to apply machine learning methods to new, uncharted domains.

Like pioneers exploring distant lands, you will encounter never-before-seen challenges requiring soft skills, including persistence and creativity. You will encounter large, messy, and complex datasets requiring in-depth exploration and documentation; graphs and visualizations are the field's equivalent to the pioneers' charts and maps. Your analytical and programming skills will be tested, and when you fail, as tends to happen early and often, you will need to iterate and improve upon your mistakes. Creating reproducible experiments using the scientific method will become the breadcrumbs that will prevent you from walking in circles. To become the machine learning equivalent of a rugged individualist involves being both nimble and adaptable, yet also having an insatiable curiosity like a dog with a bone—that is, once it bites down, it doesn't let go.

Although the idea of the rugged individualist is a fantastic metaphor for the solo elements of machine learning, the work is very much also a team sport. Exploring the tundra of Antarctica or climbing the peaks of Mount Everest is a grueling effort, and so are most real-world machine learning projects. It would be unwise or risky to go at such tasks alone—perhaps even dangerous, if the stakes are high. However, even within a team, failure can occur due to poor planning or poor communication. The handoff between the data scientists that develop the models and the data engineers that provide them with the data is particularly treacherous. If a team makes it this far, an even more challenging handoff occurs later when the model must be implemented into business practices and IT systems. For this reason, among many others, the majority of machine learning models are never deployed.

If it seems like real-world machine learning requires superhuman-like skillsets, this may not be far from the truth if a perusal of recent online job postings is any indication. One very specific job posting asks for experience building recommender systems and designing image recognition tools, as well as familiarity with graph representation learning and natural language processing. Another asks for experience building “performant inference pipelines on very large datasets.” Many want experience with deep neural networks, yet some are more general, requiring a “solid understanding of machine learning fundamentals” and the “ability to analyze a wide variety of data both structured and unstructured.” The diverse sets of skills can be somewhat intimidating to early-career practitioners and lead them to wonder where to begin.

Figure 11.1 lists some of the so-called “hard” technical skills commonly used in the field as well as some of the beneficial “soft” traits. By the end of this book, you will have been exposed to most of the tools and skills on the left side of the figure, and by completing the exercises, you will develop the characteristics on the right. Keep in mind that finding a person who has more than a superficial handle on every one of these skills is exceptionally rare. These are the fabled “unicorns” of the field, although even they would likely admit that there is still much to learn. It is always possible to go deeper and learn more about a particular topic. Given limited time and energy, most people must compromise on whether to go broader across many areas or deeper into just a few.

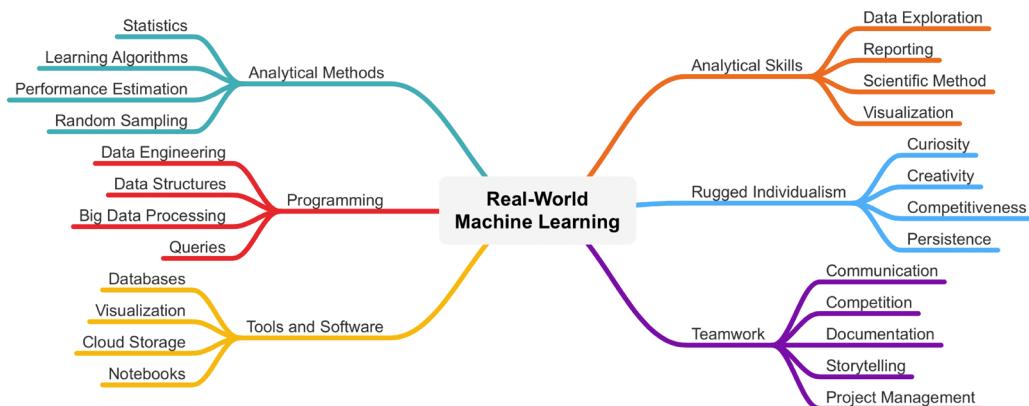


Figure 11.1: Real-world machine learning requires numerous technical skills (left) and soft skills (right)

One of the best ways to hone machine learning skills, and especially soft skills, is through competition. As depicted in *Figure 11.1*, competition improves machine learning results in more ways than one; it is a key component of the individual drive to innovate and improve one’s own performance, yet it also fosters strong teamwork toward meeting a common goal. For these reasons, competition has long been a part of machine learning training.

For instance, academic computer scientists have competed for over 25 years in an exercise called the **Knowledge Discovery and Data Mining (KDD) Cup** (<https://www.kdd.org/kdd-cup>), which chooses winners based on their performance on machine learning tasks that change annually. Similar competitions exist for specialized topics in image, text, and audio data, as well as many others.

In one of the earliest widely known examples of machine learning competitions in the for-profit space, in 2006, the Netflix video streaming service began offering a \$1M prize to make its movie recommendations 10 percent more accurate. The publicity around this event spurred a wave of additional corporate-sponsored challenges, including those listed on Kaggle (<https://www.kaggle.com>), a website that hosts competitions offering cash rewards for advancing the state of the art on tough machine learning tasks across varied domains. Kaggle soon became so popular that it inspired a generation of machine learning practitioners, some of whom used their victories as a springboard for future work in consulting and tech companies. In *Chapter 12, Advanced Data Preparation*, you will learn from the experience of some of these Kaggle champions.

Not everybody enjoys head-to-head competition, but you can still compete against yourself or imagine your company in competition against other businesses. Some practitioners are content with challenging themselves to beat their own “high score” on a model performance statistic. Others are motivated by “survival of the fittest” and the thought that a business market rewards companies that outperform others—some of which eventually go extinct. In any case, the goal of competition is not to boost one’s ego, but rather to motivate innovation and continuous quality improvement, ensuring your skills stay up to date.

The need to continually learn, and to continually apply your learning, may be the most important characteristics of a machine learning devotee. As described in *Chapter 1, Introducing Machine Learning*, the field evolved in an environment in which the volume and complexity of data grew alongside the computing power and statistical methods necessary to make sense of it. This evolution shows no signs of slowing down. Data, tools, and methods will change, but there will always be a need for people to apply them. Therefore, approach each project as an opportunity to learn something new. The iterative and sometimes addictive process of building better models is an apt place to begin this journey.

What makes a successful machine learning model?

Until now, we have taken a largely *quantitative* perspective of what it means to be a successful machine learning model. Supervised learners were initially said to perform well if the accuracy was high.

In *Chapter 10, Evaluating Model Performance*, we expanded this definition to include other, more sophisticated performance measures, such as the Matthews correlation coefficient and the area under the ROC curve, to account for the fact that accuracy is misleading for unbalanced datasets and to consider performance trade-offs for potential use cases.

So far, we have relegated *qualitative* measures of model performance to the realm of unsupervised learning, although there are certainly non-quantifiable considerations in the area of predictive modeling as well. Consider, for example, a credit scoring model that is so computationally expensive that it cannot be implemented in a real-time application, or so algorithmically complex that an explanation for its decisions cannot be provided to the applicants. With this in mind, we may favor a simpler, less accurate model, if the alternative is no model at all. After all, even a simple predictive model is usually better than nothing—the word “usually” being a key qualifier, given what we will examine shortly about models failing dramatically in the real world!

There may be business costs, resource constraints, or human resource factors not easily incorporated into the model itself that impact the success of a modeling project. To illustrate this fact, imagine that you create a customer churn forecasting algorithm that can identify with a high degree of accuracy the most likely customers to stop purchasing a product. Unfortunately, upon deploying the model, you receive complaints from the sales representatives, who have the role of attempting to retain these customers:

- “I already know these people will churn. You’re telling me nothing new.”
- “That customer stopped buying 2 months ago. They already churned.”
- “We actually want these people to churn, as they are low-value customers.”
- “I’ve already spoken with that customer, and there’s nothing we can do.”
- “Your model’s predictions make no sense. I don’t trust it.”
- “What makes you think this customer will churn? They seem happy to me.”
- “We’ll lose money trying to retain that customer.”
- “The predictions don’t seem to be as good as before. What happened?”

These types of comments are typical in real-world machine learning, and represent common barriers to a project’s overall success, even for a model that, by conventional, statistical performance metrics, was deemed accurate or effective. The trouble with these barriers is that they are not easily navigated without deep knowledge of the business in which the model will be used. On the other hand, these types of issues tend to follow similar patterns, which can be foreseen with practice.

The following table categorizes the problems into four groups, along with typical symptoms and potential workarounds:

Pitfall	Symptoms	Potential solutions
Predicting the obvious	<ul style="list-style-type: none"> • A simpler model (or a well-known rule-of-thumb) performs almost as well • The model's performance statistics seem "too good to be true" • The model performs well on training and test sets but makes no impact when deployed • The outcomes seem inevitable; having the predictions provides no actionable way to intervene 	<ul style="list-style-type: none"> • Reformulate the problem to be more challenging to the learning algorithm • Look out for rote memorization, circular logic, or target leakage (having a predictor that is essentially a proxy for the target) • Recode the target variable, or limit access to certain predictors overly correlated with the target, such that the model finds new connections rather than the obvious ones • Examine the mid-range of predicted probabilities, or filter out the most obvious or inevitable predictions

Conducting unfair evaluations	<ul style="list-style-type: none"> The model performs much worse in the real world than during testing Determining the “best” model used a lot of iteration or tuning The correct or incorrect predictions are seemingly predictable; the model may do better or worse than expected on certain segments of data 	<ul style="list-style-type: none"> Ensure an appropriate evaluation dataset is used Use cross-validation correctly and understand its limitations Beware of common forms of internally correlated data and understand how to construct fair test sets in these cases
Being inconsiderate of real-world impacts	<ul style="list-style-type: none"> The results are interesting but not very impactful An unclear business case to implement the model Important subsets of examples are neglected by the model Overreliance on simple, quantitative performance measures Ignoring predicted probabilities and treating all predictions with equal weight 	<ul style="list-style-type: none"> Forecast the impact of the project under various plausible scenarios using simulations and experiments Put filters on the output that reflect real-world constraints Create ROC curves and other cost-aware performance metrics Focus on the high-impact outcomes, not the “low-hanging fruit”

	<ul style="list-style-type: none"> • Stakeholders refuse to act upon the data and rely on intuition instead • A preference to do things the “old” way • Little interest in working through predictions systematically • Stakeholders cherry-picking results they agree/disagree with • Repeatedly asking to justify the predictions 	<ul style="list-style-type: none"> • Identify “champions” that will help promote the project • Include stakeholders in the modeling process (especially data preparation) and iterate on their feedback • Craft an “elevator pitch” and “road show” slide deck, to preemptively address FAQs • Document cases where the model made an impact and tell these stories repeatedly • Output predictions in an actionable form, such as a “stoplight” approach • Use model explainability tools
Lack of trust		

It is likely that many pages could be spent describing a career’s worth of experience with each of these three categories of pitfalls, but unfortunately, this would be no substitute for learning for oneself the hard way. That being said, there are some broad pointers that may help you steer clear of some of the bumps in the road.

Avoiding obvious predictions

When it comes to *predicting the obvious*, it may not be at all obvious how or why this happens in the first place! The short answer to this question is that is often easier than one might think to accidentally construct a model that “cheats” by finding a way to simplify the problem, or “short-circuit” the problem without doing the deeper work necessary to truly understand the problem.

This is especially true for projects that track features and outcomes over time, such as forecasting an event that will happen in the future. These types of projects typically begin with **time series data**, which repeatedly measures the same attributes for examples over time.

We will consider time series data from a data preparation perspective in *Chapter 12, Advanced Data Preparation*, but for now, it suffices to say that data with a time dimension must be carefully coded in order to avoid using values from the future to predict the past. This problem falls under the broader category of **leakage**, or more specifically, **target leakage**, which describes a situation in which the learning algorithm knows something about the target to be predicted, which it would not have available in the real-world deployment environment. When there is a clear out-of-sequence issue, such as using current credit scores to predict past loan defaults, target leakage is quite obvious in hindsight. It still occurs surprisingly often, especially when analysts simply dump all potential predictors into a model without considering what they mean. Other times, target leakage is very, very subtle and more difficult to detect, only to be revealed upon deeper analysis when a stakeholder rejects the results as “too good to be true.”

One of the most subtle forms of leakage occurs when the target variable is defined in such a way that it creates tautological or definitional predictions. The relationship between the target and the predictors need not always be fully deterministic in this case, but merely unduly correlated, or linked together in some unclear way. For example, suppose we use a business definition that defines this month’s churn status using a 3-month rolling average in sales. Using last month and 2 months ago as predictors for the current month’s churn then gets the algorithm two-thirds of the way to the correct answer—customers with low sales in those months are very likely to churn, by definition. This type of mistake is easy to make when using complex survey data, in which the individual survey responses are used as predictors and a score computed from the set of survey responses is used as the target to be predicted. As a general rule, to avoid target leakage, it is best to use a target that is generated by a completely independent process, and is collected at a later time than the data used to make the prediction.



Beware of using the future to predict the past! For a model that will be deployed in the real world, this is almost always a clear sign that target leakage is present.

Leakage can also occur when the target variable is linked in a hidden manner to predictors by a business practice. For instance, imagine a scenario in which a manufacturer attempts to boost customer acquisition rates by building a model to predict which customers are most likely to purchase their brand’s car.

It seems reasonable to create a predictor for whether the person has opened or clicked on marketing emails, but if the marketing emails were sent only to prior customers, then the model is likely to make the common-sense forecast that loyal customers tend to stay loyal customers, and the sales agents are unlikely to be impressed. Due to the strong connection between the target and the predictor, the model essentially ignores the group of people that have never purchased from the brand before, which is the group most likely to impact the company's profits! Excluding this predictor from the model, or building the model only based on first-time car buyers, would focus the algorithm on the most impactful predictions, or the ones that are less inevitable.

Another factor leading to obvious predictions is related to **autocorrelation**, which describes the inertia-like phenomenon in which measurements that are close together in time tend to also be close together in value. Based on this observation, one can conclude that the best predictor of something today is often the value of that same thing yesterday. This is almost universally true: today's energy use, spending, calorie intake, happiness, and virtually anything else one can imagine are all closely linked to the state of these things the day prior. Stated differently, autocorrelation implies that the long-term variation across people or other units of analysis, such as households, businesses, stock values, and so on, tends to be greater than the variation within those same units over small periods of time.

Machine learning algorithms can quickly identify instances of autocorrelation and will happily produce a model that uses yesterday's values to predict today. It will even have high accuracy on the test set, which leads many inexperienced analysts to ignore the underlying issue. Specifically, this is merely an overly complex way to do list-sorting! If a business wanted to predict the customers most likely to spend large amounts tomorrow, they could simply query the database for the top-spending customers as of today and sort the list in a simple spreadsheet application. In fact, this sorting approach has worked quite well for many years under the name **recency, frequency, monetary value (RFM)** analysis, which was introduced in *Chapter 6, Forecasting Numeric Data – Regression Methods*, in contrast to a machine learning based approach. The RFM approach basically says that the customer who purchases more recently and frequently and spends more money is more likely to continue these trends. Unfortunately, this does little to forecast which new customers are most likely to become top spenders in the future.

Forcing the learning algorithm to tackle this more actionable question involves redefining the target variable around the “action.” The target needs to be very specific and indicate the exact circumstances in which the model will make an impact. In the previous example, rather than modeling total spending, it would be more actionable to model the spending increase or decrease over time.

This is known as the **delta**, and forecasting the sales delta will allow sales agents to intervene before the predicted increase or decrease occurs. Alternatively, it is possible to model the intervention's impact itself; for example, rather than predicting the customers most likely to churn, it is better to model the customers most likely to respond positively to the churn intervention. Of course, this requires historic data that records the attributes of previous anti-churn interventions. Often, this type of data is lacking in most businesses.

Conducting fair evaluations

It is not uncommon for machine learning projects that performed well on paper to perform worse in the real world. This is sometimes related to the problem of *conducting unfair evaluations*, and whether it is due to honest oversight or intentional deception, this mistake should not exist. Given the results of case studies in previous chapters, we know not to assume that training performance is an unbiased estimate of future performance. Therefore, we've always constructed a holdout test set to simulate future unseen data and provide this fair estimate. In *Chapter 10, Evaluating Model Performance*, we learned that to compare and choose between multiple candidate models, we should use a validation dataset so that the test set can be “kept in a vault” and remain an unbiased estimate of future performance. The underlying issue is that by cherry-picking a model that performs best on the test set, one essentially overfits to the test set, and the performance is inflated, just as it is when overfitting to the training set. Violating the “vault” rule will, unsurprisingly, lead to unexpectedly poor performance models in the real world.

Perhaps more surprisingly, just as it is possible to overfit to training and testing, it is also possible to overfit to the validation set. This is especially true when numerous models are built iteratively, or the model is “tuned” extensively to identify the optimal parameter values, as will be discussed in *Chapter 14, Building Better Learners*. The problem is that by repeatedly using the same data, some information about the data inevitably “leaks” out to the learning algorithm, and it can eventually become overfitted to the validation set.

It may help to visualize the procedures of model building, model selection, and model evaluation as a sequence of steps, as shown in the following figure. During the training step, the algorithm identifies the optimal fit for the data; in doing so, it optimizes internal values, known as **parameters**, which are the basis of the model abstraction. In some cases, like regression models, neural networks, and support vector machines, the parameters are easily visible to the end user as coefficients, weights, or support vectors. In other cases, such as k-NN, decision trees, and rule learners, the parameters are more conceptual; think of the parameters as the internal choices the algorithm made to fit the data.

In any case, as the model has chosen a single “best” set of parameters to fit the training data, any performance estimate is likely to be optimistic, and it is likely to perform at least slightly more poorly upon generalization to the test set.

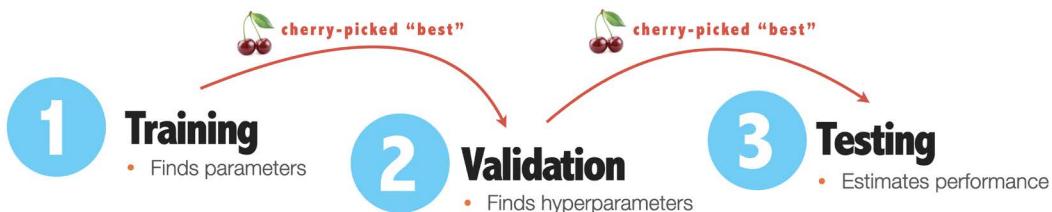


Figure 11.2: Because a “best” model is chosen in training and validation, the performance estimates tend to be optimistic

The validation dataset is used to select between various types of models, test multiple iterations of a single type of model, or both of these simultaneously. This can be understood as the process of identifying the optimal **hyperparameters** for the learner, or any other parameters that are set outside the learner and not estimated by the algorithm itself. Having read the previous chapters, you will already be familiar with several hyperparameters, such as the value of k for the k-NN algorithm, the cost parameter C and the kernel for the SVM algorithm, and the learning rate and the number of hidden nodes for a neural network, among many others. Defined broadly, the notion of a hyperparameter does not only refer to choices that directly influence a specific algorithm but also the overall choice of algorithm itself, as well as how the algorithm can be combined with others, as you will learn later in *Chapter 14, Building Better Learners*. For reasons that will soon become clear, it may be helpful to consider a “hyperparameter” to be any decision that is made outside the learning process and may impact the model’s fit.

Now, suppose you have a validation dataset, and you systematically evaluate numerous approaches on this data. You may test a variety of algorithms, like neural networks versus decision trees and SVMs, and then test various iterations of each of these models using different hyperparameter values. Upon choosing the “best” performer out of all of these dozens or hundreds of possibilities, based upon the validation set performance, this model’s performance is likely to regress when applied to the testing dataset given the potential overfitting to the validation set, just as it did between training and validation. We are left wondering whether we truly selected the true best model and how robust the performance will be.

The 10-fold CV approach, which was introduced in *Chapter 10, Evaluating Model Performance*, seems at first as if it may solve both problems. Indeed, the practice of computing the average and standard deviation of performance across the 10 folds does provide a measure of the robustness of model performance as the underlying training data changes. This provides a sense of how well a model will generalize to future, unseen data. Consequently, a common practice is to run 10-fold CV repeatedly on the same dataset to compare various hyperparameters and choose the winner.

However, as *Figure 11.3* illustrates, 10-fold CV in its standard form (left) does not provide a validation dataset, and thus can only estimate the generalization error that arises from the model's internal search for optimal parameters. For example, suppose we compare 10-fold CV performance statistics to help make a decision about whether a neural network performs better than a decision tree, or we use 10-fold CV to determine which of 25 potential values of C is best for an SVM approach. In both of these cases, notice that we are again cherry-picking the winner, which has the consequence of inflating our performance estimate. Ultimately, if cross-validation is used with extensive hyperparameter tuning, we may be overly optimistic about the model's true future performance.

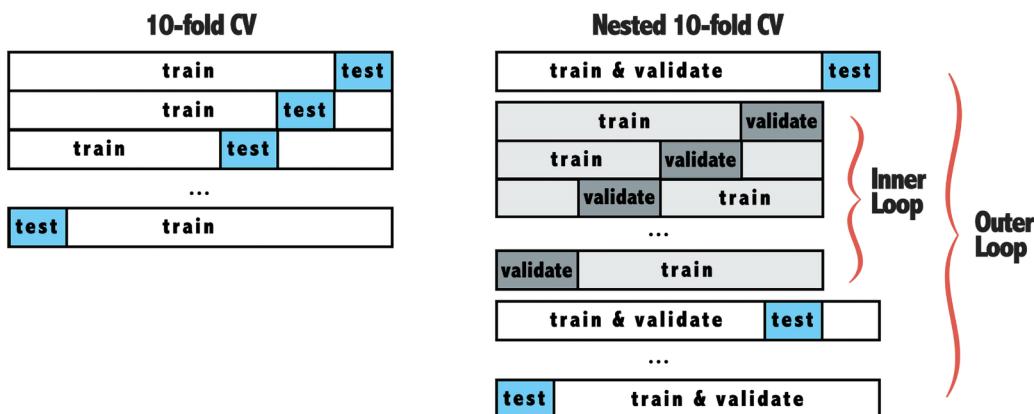


Figure 11.3: Nested 10-fold CV adds an “inner loop” of 10-fold CV for learning optimal hyperparameters on the validation set

It is better to think of cross-validation as not merely estimating the ability of the model to fit the training data (learning the best parameters) but also the entire pipeline of decisions made in the course of choosing the final model (learning the best hyperparameters). Suppose we could write an R function that automates these decisions; it takes several candidate approaches and chooses the best performer among them.

For lack of a more formal term, let's call this an "assessor" function. In this case, we might consider modifying our standard 10-fold CV approach by dividing each of the 10 folds into training and validation sets, rather than training and test sets. Each model would be evaluated by the assessor for its performance on the validation dataset, and the winning model would be chosen by the assessor as the one with the best average performance across the 10 folds. At this point, we are still left trying to figure out how well the performance will generalize to new, unseen datasets in the future.

As depicted in *Figure 11.3*, the purpose of **nested cross-validation** (typically, nested 10-fold CV) is to use cross-validation as an inner loop in which parameters are learned on each of the inner folds, and the best hyperparameters are determined by an assessor function at each of the folds of an outer loop (also typically 10-fold CV). Within each of the inner loop folds, any number of models can be evaluated by the assessor function on the validation dataset. Perhaps we are evaluating three different types of models, such as decision trees, k-NN, and SVMs, or we may be evaluating 25 different learning rates for a single neural network. In the end, whether we are assessing 3 or 25 models, only the single best one is nominated by the assessor function to be sent for evaluation in the outer loop, where the 10 best inner loop models' performance values are averaged on the corresponding fold's test set. This means that nested cross-validation does not measure the performance of a single model but the entire process of selecting models and learning the best parameters and hyperparameters.

As noted in *Chapter 10, Evaluating Model Performance*, given the complexity of nested cross-validation—both in implementation and interpretation—standard 10-fold CV is often good enough for most real-world applications of machine learning. On one hand, as the amount of information leak occurring in the validation process is relatively minor, the larger the dataset, the smaller the impact will be on the analysis. On the other hand, if the performance difference between two models is small, it is possible that the magnitude of the overfitting is enough to cause the wrong choice to be made. This is especially true as the amount of tuning, iteration, and hyperparameterization is increased.

Overall, whether nested cross-validation is necessary or overkill for a fair evaluation depends much on how the results will be used. For an industry benchmark or an academic publication, the more complex nested technique may be justified. For lower-stakes work, it may be wiser to choose the simpler standard 10-fold cross-validation, and use the time saved to consider how the model will be deployed. As will become clear in the next section, it is not uncommon for a model that performs well in theory to fail in the real world. Thus, simpler approaches that can get to failure faster will allow you more time to course-correct.

Considering real-world impacts

The adrenaline rush of creating a machine learning project that, by all objective measures, appears like it will perform well at its intended task often leads to another common pitfall: *being inconsiderate of real-world impacts*. Real-world machine learning projects are generally not exercises performed for fun; they are typically costly, time-consuming endeavors. The stakeholders that commission a machine learning project generally expect a **return on investment (ROI)**, not just for the time and money needed to produce the model but also for the resources that will be required to act upon the result. A model that doesn't work at all is a one-time sunk cost, but a model that provides poor recommendations or squanders or misdirects the company's future time and resources is one that throws good money after bad. Escalating commitment to a good idea that simply didn't work as intended is the root of the **sunk cost fallacy**, in which one believes that a failing project can be saved with more investment. A machine learning project used this way is worse than nothing at all.

In addition to wasting the resources of the implementation team, it is also possible that a project can cause unexpected harm. This is true even if the machine learning algorithm is doing what is rational in light of its training. To provide one humorous personal example of this possibility, see the following image, which shows dozens of mail pieces I received from the same credit card company over the period of just a few months. Making matters worse, this isn't even all of them, as I didn't start collecting them until I realized what was happening! There were even some periods when I received a letter every day.

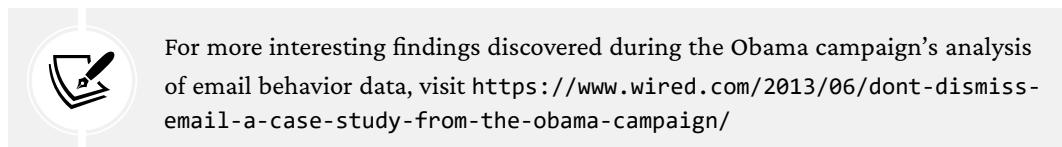
Knowing that banks are generally not inclined to waste money on postage without good reason, my suspicion is that a customer acquisition machine learning model determined that I would be a valuable customer to acquire, even if it meant spending a lot to do so.



Figure 11.4: A solution that seems reasonable to an algorithm may have negative real-world impacts, such as “spamming” end users with solicitations

Although I should have been flattered that an algorithm considered me valuable enough to mail relentlessly, it may have ultimately tarnished my impression of the credit card company. While I cannot be certain that this was not caused by some sort of glitch, while it was happening, I couldn't help but recall what I had heard during a presentation from Rayid Ghani, the Chief Data Scientist of the 2012 Barack Obama presidential campaign. Specifically, the campaign had run thousands of experiments on their email solicitations and, in doing so, discovered that the more emails they sent, the more money they made. This occurred with effectively no limit.

At no point did the number of people unsubscribing from their mailing list outweigh the additional donations they received by being constantly atop of email inboxes. Perhaps a finding like this explains my flood of paper mail from the credit card company; it certainly explains the huge increase in email marketing that consumers now see in their inboxes. Only time will tell what the long-term consequences of this approach will be.



For more interesting findings discovered during the Obama campaign's analysis of email behavior data, visit <https://www.wired.com/2013/06/dont-dismiss-email-a-case-study-from-the-obama-campaign/>

Rather than leaving it to chance, the single best way to be considerate of real-world impacts is to design the machine learning project as a close approximation of the ultimate deployment scenario. Simulations, experiments, and small **proof-of-concept (POC)** trial runs are especially helpful tools to this end. Before the project is even considered for deployment, the machine learning practitioner should be able to answer typical stakeholder questions such as:

- How many dollars, lives, widgets, and so on, can be saved using this model?
- How many “misses” will happen for every successful prediction?
- Is the ROI dependent on high-risk, high-reward events, or does it accumulate smaller, slow-and-steady wins?
- Is the model’s performance markedly better on certain types of examples, or does it perform evenly well across the full set?
- Does the model systematically favor or ignore categories of interest, such as protected age, race, or ethnic groups, geographic regions, or customer segments?
- What are the potential unintended consequences? Can the algorithm cause harm, or be exploited by bad actors to do so?

Cleverly designed projects that include a simulated deployment and an estimated ROI calculation will help provide data to answer these questions. In these projects, it is important to not simply calculate accuracy but also take a step further, and compute a number that describes how this accuracy translates into a real-world impact once the model has been deployed.

Making a fair comparison requires enough business knowledge to construct or identify an appropriate **control group**—the existing status quo or situation that serves as a benchmark or baseline frame of reference.

For a cancer identification model, for example, one might estimate the number of lives saved using the model’s predictions, and then compare this to a baseline comprising the lives saved using traditional human diagnosis. In cases where the intervention of a machine learning model has no obvious frame of reference, the baseline may be a scenario using no model at all, one in which the outcome is essentially decided at random, or one decided using a “dumb” model that always picks the majority class or predicts the mean value. You may also use a simple model like the OneR rule learning algorithm discussed in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, to simulate a simple rule-of-thumb heuristic.

Naturally, the real world is extremely complex and constantly changing, and thus it is important to not only estimate the impact of a model but also examine how robust its impact is under various constraints. Some of these constraints may be ethical, such as the need to ensure that it performs evenly well across subgroups of interest. In a business environment, these important subgroups may consist of categories like age, race, ethnicity, gender, and economic status; in medical contexts, these may be based additionally on health characteristics like body mass index and whether the subject smokes. It is up to the analyst to determine what real-world contexts are most important to evaluate for performance. Once these have been decided, the analyst can make predictions on each of the subgroups and compare the model’s performance across the groups, checking for bias reflected in systematically over-performing or underperforming groups.

Aside from variation across subgroups, a model’s impact may also vary in the real world due to changes over time in the data used for training or evaluation. In practice, 10-fold CV and even the more sophisticated nested cross-validation variant oversimplify many aspects of how machine learning models are built and deployed in the real world. They do not reflect many potential external factors, beyond mere variations in the training data, which may influence a model’s future performance.

To help understand these other factors, *Figure 11.5* provides a simplified view of how models are generally built and evaluated in most real-world environments. It imagines a data stream composed of the entities for which the model is to make its predictions; you might imagine this as a single-file line of potential cancer patients, customers, loan applicants, and so on. To forecast their future outcomes, we generally take a snapshot of this data stream today, at a single point in historical time, or record observations for a period of time, and then divide this data into separate sets for training, validation, and testing—possibly using 10-fold CV or similar methods. The performance estimates from models built and evaluated on this snapshot are assumed to be reasonable estimates of future performance, but we cannot be certain until the future eventually happens.

Of course, in an ideal scenario, we would simply build our model on today's data (or past data) and then wait some time for the "future" to occur and conduct the evaluation, but this is very rare in practice. Consider yourself fortunate if your business had the foresight to gather sufficient historical data or has the patience to wait while it is gathered; in many cases, business moves too quickly and resources are too scarce for this to be true.

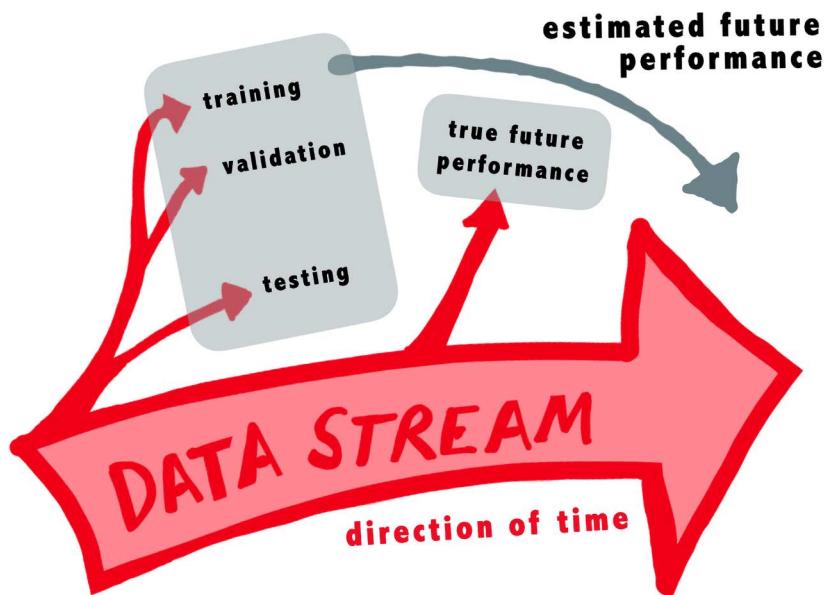


Figure 11.5: The data used to train and evaluate a model is often from a snapshot in time much earlier than the time of deployment

Even when a model can be evaluated on truly never-before-seen future data, the relentlessness of time and the data stream often cause problems after deployment that are difficult to foresee ahead of time. Specifically, as the real world is constantly changing, deployed machine learning projects tend to be subject to **model decay**, a term that describes the common phenomenon in which their performance deteriorates over time after implementation.

Sometimes, this is due to systematic changes in the input data over time, known as **data drift**, which can happen due to cyclical patterns like purchasing behavior or disease spread, which vary by season, as well as changes in the meaning or magnitude of the data itself. Data drift occurs after changing the scale on which something was measured, as in switching from a scale from 1 to 5 to 1 to 10, as well as inflating the values overall, which occurs with currency inflation. It can also occur subtly even without changes in the attribute values themselves if, for instance, a survey question's wording changed.

Perhaps the survey used a value of three to indicate “neither agree nor disagree” but was later translated into another language as “no opinion.” This drift in meaning over time may contribute to degraded performance over time, but it can be somewhat mitigated with strict maintenance of codes and definitions—easier said than done!

Another contributor to model decay is **model drift**, which occurs when the relationship between the target and the predictors changes over time, even if the meaning of the underlying data remains constant. This generally reflects a change external to the model, or an external force that would have been difficult to foresee. For example, there may be broad changes to the economy or customer behavior and preferences, evolutionary changes in how a disease behaves, or other such factors that fundamentally disrupt the patterns the learning algorithm discovered during training. Fortunately, model drift can be remedied via more frequent training—the model simply learns the new patterns—but this leads to confusion and further complexity, as it is not clear how frequently or infrequently one should retrain the model.

Although one might think that frequent or nearly-real-time training is always best, this substantially increases the complexity of deployment and can lead to increased variability in the model’s predictions over time, and thus can contribute to a lack of trust in the model’s output—a problem described in the next section. Perhaps the best approach is to experiment and identify a schedule to refresh models that works best for the specific use case, such as annually, seasonally, or upon suspicion of data drift. Then, closely monitor the results and refine as needed. As usual, there is no such thing as a free lunch!

Building trust in the model

The final category of pitfall contributing to failed data science projects has little to do with the technical details of model implementation or the performance of the model itself; instead, it stems from a fundamental *lack of trust* in the project from key stakeholders. This pitfall is especially burnout-inducing because it occurs so late in the workflow. It is disheartening to invest countless hours in a project only to see it fail to gain traction with the stakeholders who asked for it, or the end users that would benefit most from its implementation. As frustrating as this sounds, it is even more troubling to hear the raw statistics about the problem’s magnitude; a quick web search reveals a variety of estimates—none of them good—of the proportion of machine learning projects that make it into production. Some firms estimate the number of failed machine learning projects at over 60 percent, while other estimates are as high as a staggering 85 percent.

How can it be possible that only around 15 percent of projects are ever launched successfully? Even using the more optimistic estimate, less than half will be implemented! Can this possibly be true? Unfortunately, if this seems to be an impossibly low number, you are unlikely to have worked in the field of machine learning for very long, or are one of the lucky few to work for a company that has found a solution to this epidemic of failed projects. Instead, most practitioners follow a similar pattern in which their efforts continually fall flat in an organization, and thus they look for another workplace where they can make a bigger impact. As most organizations suffer from the same issue, the cycle of discontent inevitably begins again soon.

The reasons for this failure to launch are myriad, and it is tempting to place the blame solely on the stakeholders who often have unrealistic expectations of the upside of machine learning or the costs and resources needed to follow a project through to completion. Perhaps they bought into the hype surrounding artificial intelligence and expected it to be plug-and-play with minimal investment. Under-resourced information technology teams are, after all, not a recent phenomenon. This being said, there is much that machine learning practitioners can do to proactively build the stakeholders' trust in the project and make it much more likely to take root. A key part of building trust in modeling projects is recognizing that machine learning is both an art and a science. With experience comes the understanding that while soft skills are not necessarily fundamental to performing the work, they are virtually essential to its ultimate success.

There is a lot that machine learning practitioners can learn from the study of magicians or illusionists, particularly with respect to showmanship. Of course, that is not to say in any way that the work itself should be phony—you don't want to be a snake oil salesman—rather, consider the fact that to the end user, machine learning and artificial intelligence are a black box that might as well be magic. If it works as intended, it undoubtedly stirs a magical feeling. Being heavily invested in building the tool, the practitioner may not even realize this.

As noted by David Copperfield, the most commercially successful magician in world history, “*magicians lose the opportunity to experience a sense of wonder.*” Savvy practitioners will take advantage of the magical allure of machine learning and identify early-adopter “champions” who will promote the work and help it take root in the organization. These stakeholders will know more about business operations, and including such stakeholders in the process of building the model, particularly during the phases of gathering data and translating the model into action, will provide an end user perspective and help ensure that the project will eventually be useful rather than merely interesting.

Most successful magicians take their show on the road to audiences all over the world. In machine learning, this is beneficial not only to sell the project to ever-growing audiences of stakeholders but also to gather feedback on what will work in practice and what will not. During this road-show period, it is wise to craft an elevator pitch, or a short two- or three-sentence description of the project that could be given in a brief elevator ride. Honing this pitch by repeatedly practicing it in one-on-one settings with potential end users will not only improve your ability to deliver the pitch but also assist with identifying additional champions of the project and gathering success stories. The successes can later be peppered into future conversations to build even greater buzz around the project. Of course, be sure to use the appropriate level of detail for the audience. Paraphrasing famous sleight-of-hand magician Jerry Andrus, it's our job to dazzle and "fool" the audience, but not to make them feel like fools.

Questions, criticisms, and even outright negativity can be helpful; these can lead to improvements in the project or can be added to an FAQ document or a slide deck, which can be presented to larger audiences. During the first few large audience presentations, incorporating success stories from audience members that are known to be in attendance, or even planting a champion in the audience to ask predefined questions, can contribute to trust in the project, much like a magician often plants an associate in the audience to "volunteer" for the illusion. Over time, you will begin to recognize similar questions and criticisms and preemptively address them or, even better, provide just enough information to allow the audience members to discover the answer for themselves. The trick of leaving some of the project faults in plain sight, and revealing them to the audience seemingly by accident, can be especially effective on known detractors. This is especially powerful because, according to Teller, of the long-running Las Vegas performing duo Penn and Teller, "*when a magician lets you notice something on your own, his lie becomes impenetrable.*"

Sometimes, rather than glitz and showmanship, the path to building trust is to use statistical tools and methods to provide a more intuitive means of acting upon the model's predictions. For example, suppose a model has been built to predict whether someone will develop lung cancer. Since lung cancer is relatively rare in the overall population (roughly 1 in 16 Americans will develop it in their lifetime), many models will tend toward low predicted probabilities of cancer, even among those most likely to develop cancer. For a person who is eight times more likely than the average person to develop lung cancer, it is still only a coin flip overall on whether they will develop it in their lifetime, and thus a reasonable prediction for this person may still be "no cancer." Given this person's high relative risk, early intervention may be impactful, but a prediction that simply states "no cancer" is of no use to distinguish this person from the others with much lower risk; likewise, a predicted probability of 49.999 percent is of little use without knowledge that the baseline cancer probability is 6.25 percent.

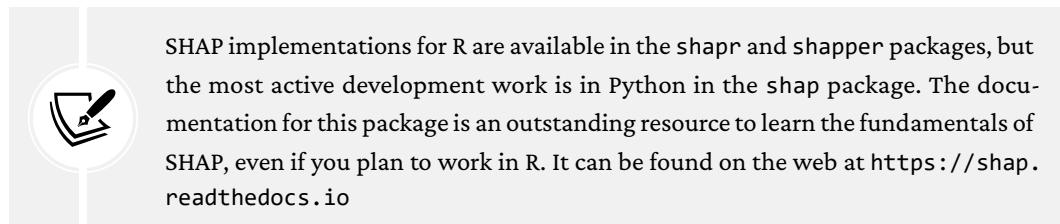
As many machine learning projects focus on rare events, translating the raw predicted probabilities into categories of relative risk can help drive trust and action. For the lung cancer model described previously, this may mean creating a red, yellow, and green “stoplight” system in which red indicates the highest possible risk, yellow indicates a moderate risk, and green indicates a low risk of developing the disease. These red, yellow, and green labels can then be presented directly in a report or dashboard and understood intuitively by the agents that need to act upon this information.

For models intended for other use cases, other formats may be more appropriate to drive action. One business might use a primary school A, B, C, D, or E letter grade system for loan applicants, another might transform raw predicted churn probabilities into a percentile-based system that ranks customers on their likelihood to churn, and some may even warrant more complex presentations that include not only a prediction but also a confidence rating. Part of the benefit of the road show is that, hopefully, while dazzling audiences with the predictive power of the model, you have also gained a sense of what output format will drive them to use it.

Even if end user adoption is high, or perhaps *especially* if end user adoption is high, there will be predictions that are either counterintuitive or appear to be nonsensical. Occasionally, end users are merely curious about why a specific prediction was made. These raise questions that can only be answered by a dive under the hood of the model itself, which is virtually impossible for black-box models like neural networks and challenging even for simpler approaches like regression and decision trees, especially when these models have been applied to large and complex datasets. These differences get at the notion of **model interpretability**, which is the ability for a human to understand how a model works. If a model is simple and transparent, it will be readily understood, and stakeholders will tend to trust it.

Closely related to interpretability, which describes generally *how* a model makes predictions, it is also important to understand *why* a specific prediction was made. The growing field of **model explainability** involves the development of methods that can be used to probe models, to develop a simplified or intuitive understanding of the factors the prediction was based on. Model explainability may be currently one of the fastest-evolving subsections of machine learning and artificial intelligence due to the rapid adoption of deep learning models, which are notoriously impossible to interpret, yet are used in fields like finance and medicine, which have strict requirements for model explainability. Model explainability tools allow powerful but uninterpretable models to be used even for applications where it is crucial to justify the decisions.

Model explainability is a rapidly advancing field of study, and new methods and best practices are being discovered regularly. One promising technique, called **Shapley Additive Explanations (SHAP)**, uses principles from game theory to allocate credit for a prediction to individual features that are most responsible for the predicted value. This is a more challenging task than it may seem at first because, for complex models, a given feature might not have a simple, linear impact on the outcome. Instead, the feature's impact may also depend on the value of the other features, which themselves may have different impacts depending on how they are combined. Because it is computationally expensive to compute all possible permutations, most SHAP implementations use heuristics that simplify this computation, and the average impact of each feature is measured across all possibilities.



SHAP implementations for R are available in the `shapr` and `shapper` packages, but the most active development work is in Python in the `shap` package. The documentation for this package is an outstanding resource to learn the fundamentals of SHAP, even if you plan to work in R. It can be found on the web at <https://shap.readthedocs.io>.

Because machine learning, at its core, is about turning data into action, explainability tools help build trust in a model, which leads to increased adoption and greater impact. For instance, a model that identifies a hospital patient to have a high risk of mortality will cause more harm than good unless we know what factors are causing the risk to be elevated. Without an explanation, a patient will be in fear for no reason. On the other hand, knowing that the risk is due to preventable factors will lead to interventions that save lives. Making these connections between the model and the real world is up to the practitioner. Thus, much in the same way that explainability techniques can lead to more effective models, your own practices and storytelling skills can contribute to the project's success, as you will see in the sections that follow.

Putting the “science” in data science

In the time since the first edition of *Machine Learning with R* was published, a new phrase has become somewhat ubiquitous within the field of machine learning. That buzzword, of course, is **data science**—a term that has been defined by many but is generally agreed to describe a field of work or study encapsulating aspects of statistics, data preparation and visualization, subject-matter expertise, as well as machine learning.

It is debatable whether data science is synonymous with what used to be called data mining, but it is safe to assume that there is a lot of overlap between the two. A reasonable outsider might observe that data science is simply a more formalized version of data mining. The methods and techniques in data mining were often learned informally on the job or passed between practitioners at industry events. This is in stark contrast to the field of data science, which offers countless opportunities to earn formal credentials and experience via online training courses and in-person degree programs.

As depicted in *Figure 11.6*, a search of Google Trends data suggests that the term truly began to grow in popularity just as the first edition of this book was published. While I would love to take credit for popularizing the phrase, unfortunately, I cannot do so, as it barely appeared in the first edition at all! Of only two appearances of the phrase in the text, the most notable appearance was literally on the book's final page, where I wrote briefly about the "burgeoning" data science community. If only I had known how true this would be! At least I was in good company; even the Wikipedia page for data science was just in its infancy in 2012 when the earliest pages of this book's first edition were being written.

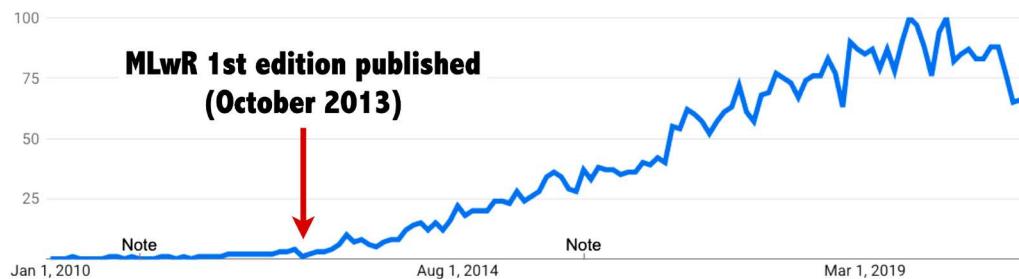


Figure 11.6: A Google Trends search shows the rapid rise of “data science” in the past decade

Since then, my role as a machine learning practitioner in the workforce, like the roles of many others around the world at the time, was transformed from the title of data analyst in to data scientist. Yet, despite the rapid change in title and perception, it seems that the work itself changed very little. Applied machine learning was essentially just data mining, historically, and today's data scientists are expected to use statistics and machine learning, as well as a strong hacker or tinkerer's work ethic, to find useful insights in data—just like the data miners of yore. What made this new field of data science different from what we previously did?

Numerous blogs and news publications attempted to answer this question on the road to understanding the hype and why data science suddenly became one of the “hottest new career fields of the 21st century,” as it was so often called.

Early in this trend, a common theme was to use a Venn diagram to illustrate the requisite skills. As shown in the following Bing image search, this depiction of data science was and continues to be pervasive, and the Venn diagram visualization has practically reached meme-like status. There are subtle variations, but most share the same overall structure: data science is found at the intersection of computer science, statistics, and domain expertise.

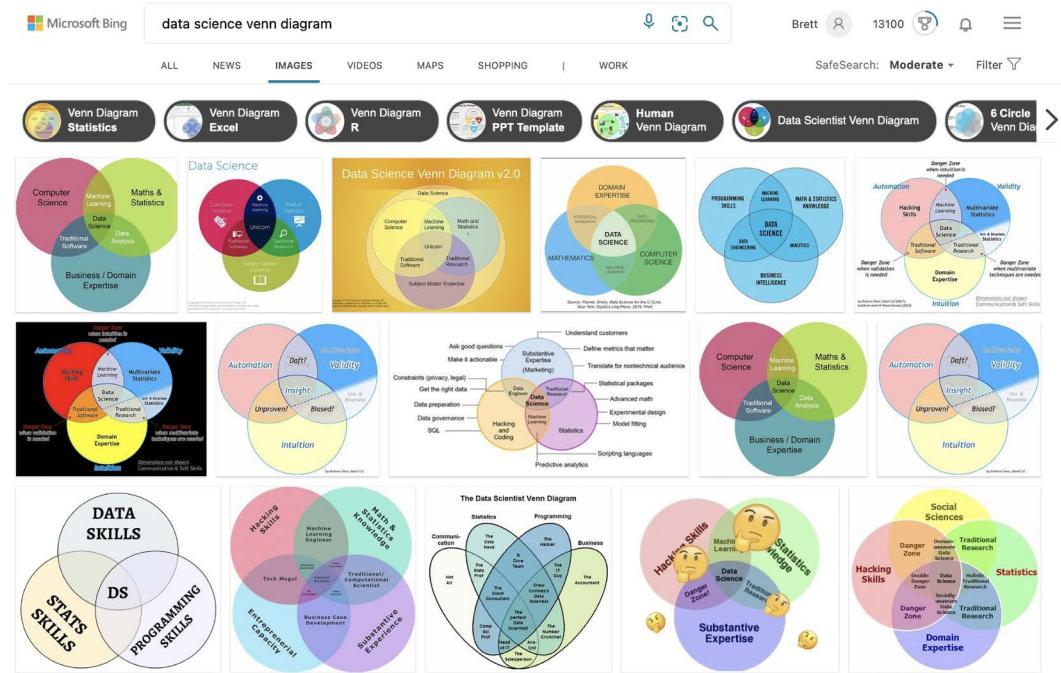


Figure 11.7: The data science Venn diagram has reached meme-like status; most place data science at the intersection of programming, statistics, and domain expertise

The problem with this type of reductionist conception is that while it captures the broad strokes of data science, it misses the soul and key distinction: a scientific mindset. One can have the requisite skills of statistics, programming, and domain knowledge, but if the work is treated without scientific rigor, then it is no different than the data mining of years prior. To be clear, this is not to say that data mining as it was performed before was useless even then, but it is certainly doubtful that it was deemed scientific by anybody who knew how the work was performed in the trenches.

So, how do we put the “science” into data science? Answering this question involves realizing why the science matters now when it may not have previously. In particular, the operationalization of data science occurred as organizations of all sizes and across many domains began rapidly staffing and resourcing business intelligence teams to find insights in the so-called “treasure trove” of big data.

These growing layers of complexity necessitated more sophisticated tools and processes to coordinate efforts and link findings across parts of an organization. When one or two people were doing data mining in the dark recesses of a business, it was not essential to be very scientific, but as the teams and tools grew, a more methodical approach was warranted to avoid having the effort devolve into chaos.

Knowing that data science at its core is a team sport, it is also important to incorporate elements of the scientific method into solo machine learning projects. Small machine learning projects can quickly grow in complexity, even for relatively simple tasks, via the use of trial and error and iteration, which are essential to the scientific method. *Figure 11.8* is intended to illustrate some of the dead ends and tangents that one explores during a rigorous machine learning project. Hypotheses are generated and examined during data exploration, only some of which prove to be fruitful. These insights inform feature engineering, which itself may have multiple false starts. Several models are tested; some fail, while others can be used to springboard more sophisticated models. Ultimately, the most promising models are evaluated, and then tuned for better performance in concert with additional feature engineering before deployment. From start to finish, the entire sequence can take days, weeks, or even months and years for complex, real-world projects. Recognizing the fits and starts of this process as natural steps in the scientific method helps communicate to stakeholders that progress isn't always a straight line forward, proportional to time invested.

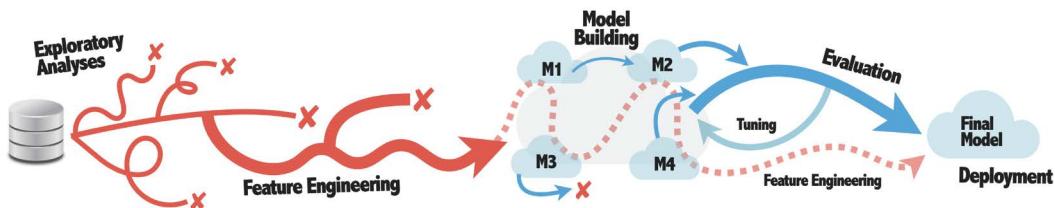


Figure 11.8: Machine learning projects rarely proceed in a straight line from start to finish

Likewise, it is important for you, the data science practitioner, to recognize that your work will not proceed in a linear fashion. Unlike machine learning tutorials in books and on the web, the messy complexity of real-world projects requires more careful attention to avoid getting lost in code or reinventing the wheel. No longer is it sufficient to create a single R code file, which is executed line by line by hand. Instead, we assemble our code and output in a single, well-organized place, in a form that we hope will also serve as an artifact of our investigation for future readers or our future, forgetful selves. Thankfully, R and RStudio make this work seamless.

Using R Notebooks and R Markdown

Upon completion of a large machine learning project, after letting out a sigh of relief, you may find yourself looking back and wondering where the time went. Dwelling on this question for too long may lead to insecurity, as inevitably, you may start to ask whether you might have avoided some of the more obvious mistakes, or perhaps made different design choices. “If only!” you may find yourself saying repeatedly. How is it possible that a project seemingly so simple in hindsight consumed so much time and effort?

This question stems from a newfound perspective atop the summit of a difficult data analysis project, with a clear view of the outcome. Recall the meandering pathway of a typical machine learning project depicted in *Figure 11.8* in the previous section. At the project’s completion, we tend to forget the numerous time-consuming dead ends and false starts and simplify the journey as a straight line from start to finish, rather than recalling the convoluted pathway it actually took.

Early in a data science career, people tend to assume that there is a way to avoid these detours and jump straight to the conclusion without “wasting” so much time chasing pointless leads. There isn’t. This work is not in vain but is an essential part of the machine learning process. The work is not wasted at all; as you become smarter about the data, the machine likewise will become smarter.

Later in a data science career, you may recognize that these initial exploratory ventures are a necessary step in all projects. However, you may have a suspicion that the work is not as impactful as it could be. While data exploration may inform a single analysis, it doesn’t seem to make a lasting impression, and the same mistakes are often repeated. Part of this may relate to the fact that it is much easier to recall what worked than what didn’t work. Successes stick in one’s mind while failures are forgotten and, in many cases, literally deleted from the R code file. In this way, the exploratory work doesn’t seem to accumulate in the same way that other experiences do. Failures, hypotheses ruled out, and paths not taken are, therefore, not easily remembered and not easily transferred to others to build historical knowledge about a project’s roots. This is to the detriment of your future self, or others who may take over your code upon your retirement. Rather than deleting everything except the final, clean solution, it would be better to have a way to present the full investigation—dead ends, mistakes, and all.

The RStudio development environment provides a solution to this problem in the form of **R notebooks**, which are a special type of R code file that combines both R code and explanatory free-form text. These notebooks can easily be compiled into HTML, PDF, or Microsoft Word formats, or even slideshows and books with a bit more effort. The resulting output document embeds code within a report’s text, or text within code, depending on your perspective.

This provides an artifact that can be used to document the entire machine learning process from start to finish, yet it doesn't feel tedious due to the fact that the code can still be run interactively line by line or block by block during development. By spending a little extra time to add explanatory or contextual documentation to the R code file, the result is a report that can be shared with others or reviewed by your future self to refresh your memory.

R notebooks are simply plain text files, much like a standard R code file, but saved with the `.Rmd` file extension. These notebooks allow code to be executed interactively within the notebook, and the output will be displayed inline with the surrounding text. In Rstudio, a new file can be created by using the **File** menu, selecting **New File**, and choosing the **R Notebook** option. This will create a new R notebook using the default template, as shown in the following image:

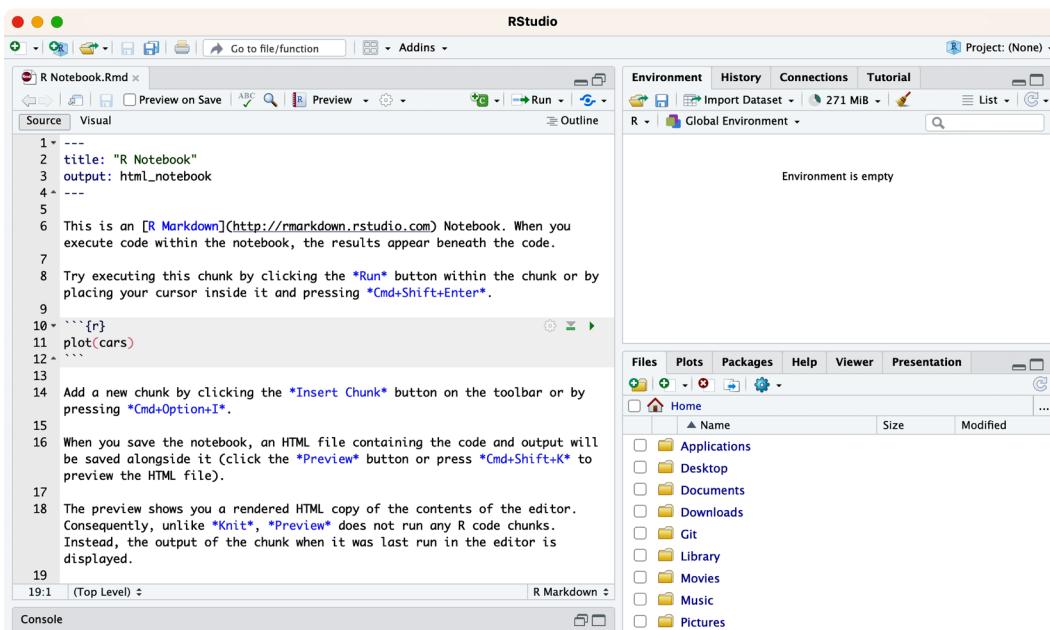
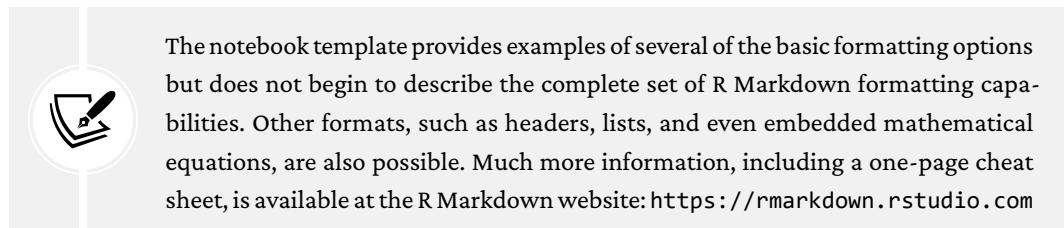


Figure 11.9: An R notebook file open in RStudio allows code and output to be integrated within a report

The top of the file between the `---` dashes includes metadata about the notebook, such as the title and the intended output format. The “gear” icon to the right of the **Preview** button in Rstudio provides settings to switch between the default HTML notebook format and PDF or Microsoft Word document if you do not want to edit this setting manually. These settings govern the output format when the R notebook is compiled upon completion of the project.

Directly below the header metadata, we find a key distinction between an R notebook and traditional R code files. In particular, this section is not R code but rather **R Markdown**, which is a simple specification for formatting reports within plain text files. Because R and RStudio were not designed to be word processors, the styles are not controlled via a graphical user interface but rather by simple formatting codes, such as ***italics*** and ****bold****, which are translated into *italics* and **bold**, respectively, in the final output file.



Because the R notebook format defaults to R Markdown, any R code must be embedded into the file using special indicators, denoting where the code begins and ends. The indicators are three backtick characters followed by the code language, surrounded by curly brackets. For example, a section of R code would begin using the ````{r}` statement. The end of this section is denoted by three backtick characters as in a ````` statement. Alternatively, these sections can be added to a notebook using the graphical user interface **Insert** button, just above the editor window and to the right of the **Preview** and “gear” buttons. The **Insert** button provides a drop-down selection of the programming languages available to use in the notebook, but keep in mind that these other languages may not be able to take advantage of the objects in the R environment—at least not without some additional steps.

Executing a code block, either by clicking the **Run** (green triangle) button within the chunk or by pressing your environment’s key combination, displays the command’s output inline with the R Markdown text. Options to manage the output format for each code block can be found using the “gear” icon at the top-right of the block. Here, one can govern whether the code or results are hidden from the final document, and whether or not the code should be executed at all. These features may be useful to suppress extraneous output from the report or prevent long-running code from being run unnecessarily.

Clicking the **Preview** button at the top of the notebook file generates a preview version of the final output report, using whatever R code output has been run interactively. For HTML notebooks, this file will open in a simple viewer, as shown in the screenshot that follows, or it can be opened in a web browser.

Because the file only uses output as it is generated in real time, the preview file is regenerated automatically by RStudio every time the notebook is saved. Leaving it open in the viewer window will allow you to see approximately how the final report will look at the end of the project.

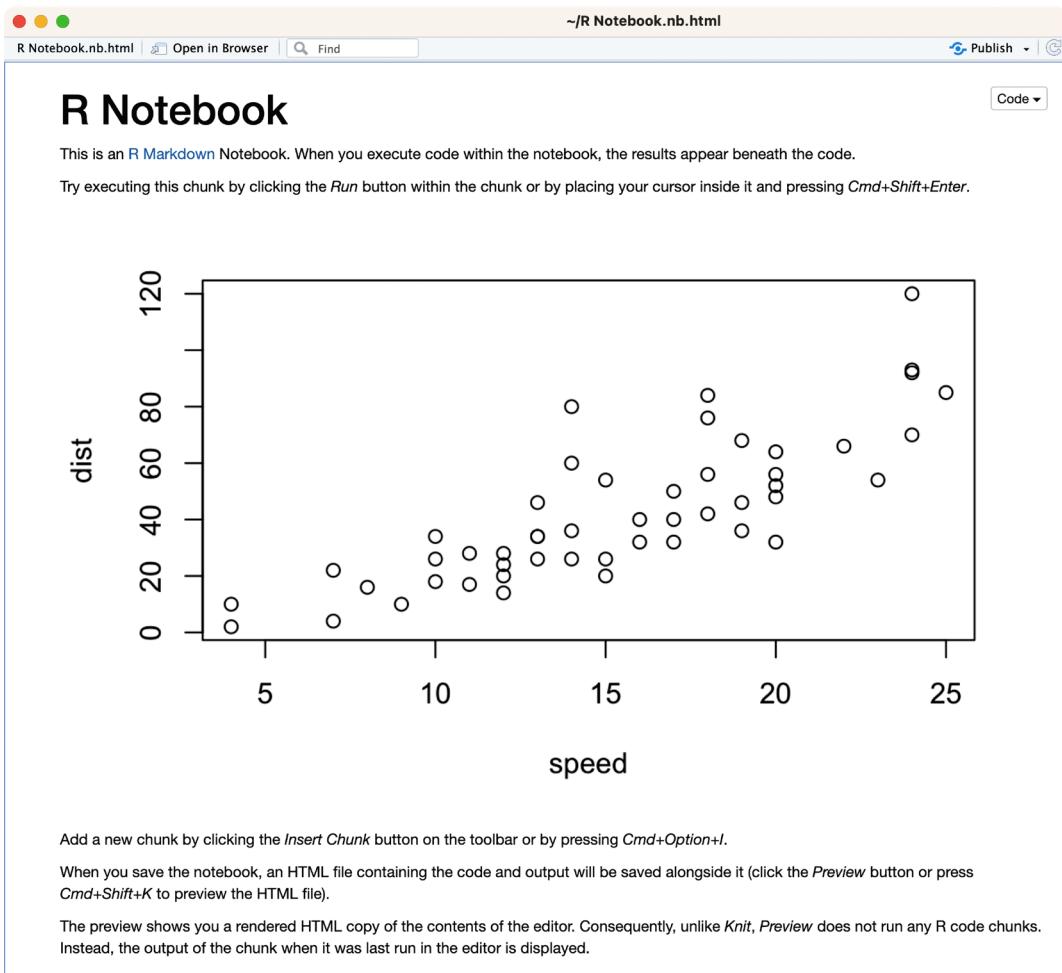


Figure 11.10: The preview file for an HTML notebook embeds the output within the text documentation

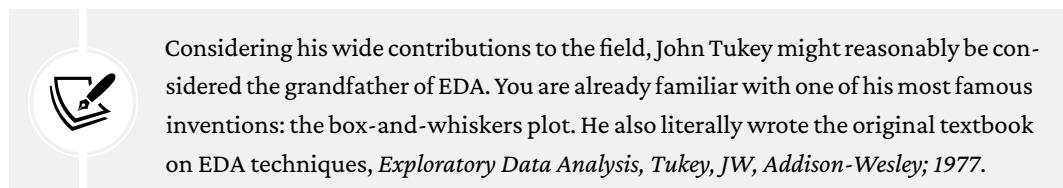
The drop-down menu button to the right of the **Preview** button provides a means to compile or “knit” the document to its final output format. This runs the complete set of R code blocks from start to finish and uses the `knitr` package to bring together the code and text into a single report. Knitting to HTML is usually straightforward, but knitting to PDF or Microsoft Word may require installing additional packages.

The resulting files are inclusive of code, text, and images and without dependences, so they can be easily shared via email. This is true even of the final HTML notebook format, which is saved with the file extension `.nb.html` and offers some simple interactivity when viewed in a web browser. This format also embeds the original `.Rmd` R Markdown file so that a recipient can open the file and recreate the analysis if needed.

Performing advanced data exploration

Falling squarely on the “art” side of data science, data exploration is a topic rarely given much coverage in academic textbooks. Tutorials may provide lip service to the practice, showing learners how to create graphs and visualizations, but rarely explaining how these are useful or why they may be necessary. Even this book is guilty of this; although the first few chapters performed simple data exploration, these exploratory analyses very rarely expanded beyond the five-number summary statistics described in *Chapter 2, Managing and Understanding Data*. Based on the limited coverage of this topic, one might gain the impression that it is not very important in practice, but this couldn’t be further from the truth; in fact, data exploration is a key component of real-world data science, and it is especially important for large, complex, and unfamiliar datasets.

Even though we have already performed simple exploratory analyses, we have not yet formally defined what it means to do so. The pioneering mathematician and statistician John W. Tukey, whose 1977 book on the subject brought the term into widespread awareness, noted that **exploratory data analysis (EDA)** involves allowing a dataset to suggest hypotheses and reveal useful insights rather than simply answer predetermined questions. It is often aided by graphs and charts, which in Tukey’s view, force us “*to notice what we never expected to see.*” One might imagine Tukey’s perspective as the notion that presenting the data in clear yet surprising ways, and listening carefully to what these analyses tell us, is not a ritual performed to merely understand the data itself but to better understand how we might ask questions about the data. In short, a rigorous precursory exploratory analysis is likely to lead to a more accurate main analysis.



As the goal of machine learning is not merely to answer predetermined questions, the form of exploratory data analysis that should be performed in concert with a machine learning project is very much in line with Tukey's line of thinking. Advanced data exploration, conducted well, allows data to suggest insights that can be exploited to improve the performance of the machine learning task. Given the goal of improving machine learning models, it is best when data exploration is performed systematically and iteratively, but this is no easy task without prior experience. Without direction, one can explore countless dead ends. To counter this, the next few sections provide some ideas on how and where to begin this journey.

Constructing a data exploration roadmap

If we are to follow Tukey's conception of data exploration, we are to believe that data exploration is less like an interrogation and more like a conversation, or perhaps even a one-sided listening session in which the data shares its nuggets of wisdom. Unfortunately, when this impression is combined with the superficial manner in which exploratory data analysis is depicted in many contexts, many new data explorers are left in a state of so-called "analysis paralysis" and unable to determine where to begin. It is as if the data scientist has been led into a dark room, told to conduct a séance, and wait for the data's illuminating response. It's no wonder this is a bit intimidating!

No exploratory analysis is exactly like another, and each data scientist should develop the confidence to perform the work in their own way. However, while building your own experience and your own data exploration roadmap, you may find it helpful to learn by example. With that in mind, this section provides advice that may be of assistance in guiding exploratory analyses in general. It is not able to cover the exhaustive set of approaches, nor is it intended to imply a single best approach for data exploration. Again, the best approach is a systematic, iterative, and perhaps even intimate conversation with the dataset, and just as there is no textbook to completely prepare you for a human conversation, there is no single formula to converse with data.

That being said, although every verbal conversation may be unique, they tend to begin similarly with a greeting and an exchange of names and pleasantries. Likewise, your data exploration roadmap may also begin with you simply becoming familiar with the data. Obtain a data dictionary, or create one in a text file or spreadsheet, which describes each of the features available for use. You may also record additional metadata such as the number of rows, the data source, when and where it was collected, and whether there are any known problems with the data. Such details may prompt questions during the analysis, or they may help enlighten when unexpected results are encountered.

You may find it fruitful to print a paper copy of the data dictionary and work methodically row by row, exploring each feature one at a time. Although this work can certainly be performed in an electronic document, with large datasets having hundreds of predictors or more, the task somehow feels less daunting when it is performed with pen, paper, and highlighter pens—not to mention the satisfying feeling of making check marks and notes and crossing items off lists! The following figure illustrates the result of one such real-world data exploration process; rows indicate the available features, which have been annotated with stars, highlighting, and notes, indicating the perceived importance of each potential predictor, as well as any potential concerns found during the exploration.

LoanStatNew	Description
acc_now_delinq	The number of accounts on which the borrower is now delinquent.
acc_open_past_24mths	Number of trades opened in past 24 months.
addr_state	The state provided by the borrower in the loan application
all_util	Balance to credit limit on all trades
annual_inc	The self-reported annual income provided by the borrower during registration. <i>correlated</i>
annual_inc_joint	The combined self-reported annual income provided by the co-borrowers during registration
application_type	Indicates whether the loan is an individual application or a joint application with two co-borrowers
avg_cur_bal	Average current balance of all accounts
bc_open_to_buy	Total open to buy on revolving bankcards.
bc_util	Rate of total current balance to high credit/credit limit for all bankcard accounts.
chargeoff_within_12_mths	Number of charge-offs within 12 months
collection_recovery_fee	post charge off collection fee
collections_12_mths_ex_med	Number of collections in 12 months excluding medical collections
delinq_2yrs	The number of 30+ days past-due incidences of delinquency in the borrower's credit file for the past 2 years
delinq_amnt	The past-due amount owed for the accounts on which the borrower is now delinquent.
desc	Loan description provided by the borrower <i>text data</i>
dti	A ratio calculated using the borrower's total monthly debt payments on the total debt obligations, excluding mortgage and the requested LC loan
dti_joint	A ratio calculated using the co-borrowers' total monthly payments on the total debt obligations, excluding mortgages and the requested LC loan
earliest_cr_line	The month the borrower's earliest reported credit line was opened <i>is this really month (01 month + year?)</i>
emp_length	Employment length in years. Possible values are between 0 and 10 where 0 means less than one year and 10 means ten or more years.
emp_title	The job title supplied by the Borrower when applying for the loan.*
fico_range_high	The upper boundary range the borrower's FICO at loan origination belongs to. <i>what to do with these?</i>
fico_range_low	The lower boundary range the borrower's FICO at loan origination belongs to.

Figure 11.11: When performing data exploration, it can be helpful to print a data dictionary (or list of available attributes) and write notes directly on the paper by hand

Working systematically down the list of features, you may first scout for any potential pitfalls. An attribute that at first appears to be incredibly useful may ultimately prove to be useless, due to newly discovered flaws or issues. For each variable, you may consider whether any of the following potential issues are present:

- Missing or unexpected values
- Outliers or extreme or unusual values
- Numeric features with high skew
- Numeric features with multiple modes
- Numeric features with very high or very low variance
- Categorical features with very many levels (known as high **cardinality**)

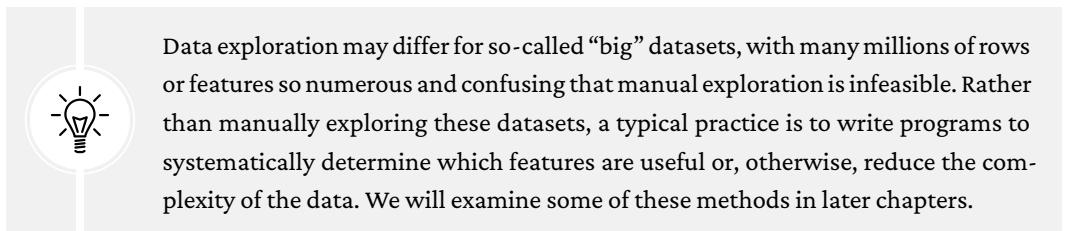
- Categorical features with levels that have very few observations (known as “sparse” data)
- Features strongly or weakly associated with the target or each other

Keep in mind that even if these potential issues are encountered, they do not always indicate a problem. Many of them can be worked around, and in fact, we will explore (or have already explored) solutions for all of them in this book. For now, by focusing only on the exploration and not on the workarounds, it is likely that you are already familiar with analysis methods that may help to identify cases of each of the listed issues.

Simple one-way tables or visualizations like histograms can provide a look into individual features and identify problematic values, but more complex visuals may be necessary to investigate the data more deeply. It is important not to merely perform univariate analyses, which consider features in isolation, but also consider each feature’s relationship to the others and the target. This will require bivariate analysis such as cross tables or visualizations, like stacked bar charts, heat maps, and scatterplots. Given that the number of potential bivariate analyses is large for large numbers of predictors, R’s sophisticated visualization capabilities, described later in this chapter, make data exploration less tedious than it otherwise might be.

The power of data exploration is not simply to probe the data for anything of negative value but also to identify aspects of positive value. By working one by one down the list, you should ask whether each potential feature could provide any useful information about the outcome. Conversely, you might ask whether the feature is completely useless, or whether it might provide even a tiny bit of assistance toward the model’s goals. This is where human intelligence and subject-matter expertise are helpful for having an insightful conversation with the data.

As truly useless data is extremely rare, you may turn this into a kind of game in which you act as a detective, trying to discover the hidden information encoded in the supposedly “useless” attribute. As the saying goes, “One man’s trash is another man’s treasure.” Data scientists that are very good at turning trash into treasure will have a strong edge over the competition, as they will develop models that use more and better data.



Data exploration may differ for so-called “big” datasets, with many millions of rows or features so numerous and confusing that manual exploration is infeasible. Rather than manually exploring these datasets, a typical practice is to write programs to systematically determine which features are useful or, otherwise, reduce the complexity of the data. We will examine some of these methods in later chapters.

Encountering outliers: a real-world pitfall

Just as how the process of data exploration, which once seemed quaint, became much more intricate in light of real-world complexity, many of the seemingly simple concepts of data exploration are actually much more nuanced in reality than they may have at first appeared. We will experience this many times firsthand when working through more complex real-world examples throughout the remaining chapters of this book; however, the nature of outliers may be the epitome of this phenomenon.

Thus far, we've taken our definition of outliers for granted; in *Chapter 2, Managing and Understanding Data*, we simply said that an outlier is "atypically high or low relative to the majority of data." We observed such outliers quite easily on a box-and-whiskers plot, denoted by circles that were 1.5 times above or below the **interquartile range (IQR)** beyond the median. In fact, these are not merely outliers but, specifically, **Tukey outliers**, named after—if you haven't already guessed—John W. Tukey, our previously noted forebearer of exploratory data analysis. This outlier definition is by no means wrong, but it may be slightly narrow. It is likely safe to assume that Tukey himself would agree that his own definition is but one of many ways to conceive of an "outlier."

Let's consider a slightly broadened definition of the term and define an **outlier** as a value that is unusual compared to others in a dataset; it is not necessarily high or low, but simply "unusual." Although this may seem only slightly different, technically speaking, from the prior definition, the word "unusual" has been precisely chosen to convey a very specific meaning. In particular, the word "unusual" does not imply a particular way to fix the data, whereas terms like "high" and "low" suggest that a data point is wrong in a specific way. You generally cannot easily correct "unusual" to "usual" without first having a firm grasp of what "usual" means. Unusual things are simply odd or curious; we should investigate them further.

With this mindset, study the following hypothetical dataset comprising images of road signs taken from a simple Bing image search. Which of these are outliers? Most stop signs are red, so it would seem that the yellow (middle) and blue (bottom left) stop signs, as well as the "stop ahead" signs, are clearly outliers, but there are some other oddities too. There's a stop sign with a hand, some with additional text, and many with slight variations to the sign's font and border. Moreover, what about stop signs on a plain white background versus a natural landscape? Or, perhaps if you are from another country, literally all of these would be unusual, and therefore, all would be considered outliers. If you are from Hawaii, where the picture of the blue stop sign was apparently taken, then even a blue stop sign may be completely within the ordinary!



Figure 11.12: Which of the images in this hypothetical stop sign dataset are outliers?

The takeaway from this exercise, of course, is that an outlier is almost always a matter of perspective, and thus, detecting and fixing outliers becomes much more complicated. On one hand, it does become a bit easier to discern if an outlier is obviously the product of a data error, as in a “mistake” that was made when recording a value. For example, suppose a data entry error recorded someone’s wealth as 1 trillion dollars rather than 1 billion. The extremeness of this value even relative to other wealthy people makes the value easy to detect, and the fact that it is obviously wrong makes for an easy fix: simply input the correct value. On the other hand, outliers that are “real,” such as Elon Musk who, at the time of writing, is worth nearly \$200 billion, are much less straightforward to handle. This distinction between “real” and “mistake” outliers is intended to illustrate the idea of whether or not an outlier is explainable. It is often but surely not always best to try to model the explainable outliers; this makes the model more robust. On the other hand, modeling the “mistake” outliers, which are essentially random variations, will usually just add noise and make for a weaker model.

The most important question to consider when encountering outliers during data exploration is whether including the outlier in the training data will ultimately improve or detract from the learning algorithm's ability to perform the desired task. This speaks to the **generalizability** of the model, or its ability to perform well on data that it has not seen before. While doing a thorough job of data exploration, keep in mind the deployment scenario and whether the model will need to be robust against similar outliers in the future. For example, if the prior stop sign images were being used to train an autonomous vehicle driving algorithm, then one might remove outliers that are not expected to be encountered on public roadways. Yet, a real-world self-driving vehicle would be expected to encounter signs defaced with graffiti, concealed by darkness, or obscured by plants and weather conditions, so one might also argue that this dataset has *too few* outliers!

As has been and will continue to be the theme for real-world machine learning, there is no single one-size-fits-all approach to handling this problem. Deleting outliers is likely the most common strategy, and is often taught in introductory statistics courses, but it is perhaps one of the worst. It is certainly easy, but this ease comes with a dark side: deleting outliers may discard very important details about the learning task. The practice precludes the data scientist from engaging in a deeper conversation with the dataset about whether the information is useful or useless.

Other approaches require more effort but may be more likely to improve the model's generalizability. In the case of events that present as outliers due to their rarity, it may be possible to collect more data on these rare events. Alternatively, it may be possible to group outliers into a single, more frequent category through binning or bucketing rare values, or capping values at a maximum level. Ideally, these groups will be based on an intuitive sense of how the learning algorithm will use the data, but in the absence of subject-matter expertise, it is often sufficient to group them into a top decile or create groups of values that have a similar impact on the target variable.

Returning to the question of what it means to be an outlier, we have already observed that context is key. Something that appears unusual in one context, such as a blue stop sign, may be ordinary in another context. Likewise, something that is completely ordinary in one context may be highly irregular in another. In short, not only can a reasonable value falsely appear to be an outlier but actual outliers can also be hidden in plain sight. Truly grasping this fact is central to rigorous data exploration. For example, consider a dataset with a typical population distribution. We'd expect to see a fair number of elderly women and a fair number of pregnant women, but observing a pregnant elderly woman would be highly irregular! Exploratory data analysis, performed well, helps identify these types of anomalies and ultimately leads to better-performing models.

Example – using ggplot2 for visual data exploration

As noted previously, data exploration is at its best when aided by graphs and charts, which according to John Tukey—himself a pioneer of innovative data visualization techniques—help us “to notice what we never expected to see.” We’ve explored a variety of datasets in previous chapters, yet until now, we have only used R’s built-in graphing capabilities to create simple visualizations, like boxplots, histograms, and scatterplots.

For a deeper, more thorough job of data exploration, we’ll need to build more complex visuals, and although we could do so using base R, a better option is available. That option comes in the form of the `ggplot2` package, which provides a “grammar of graphics” that describes how the elements of a plot relate to each other and the visualization itself. The package has been widely used for over a decade and is highly popular. It can create professional, publication-ready images, and its output can be seen in many academic journals and on many common websites. Even if you didn’t know it at the time, you are likely to have seen its output before.



Entire books have been dedicated to the `ggplot2` package and the “grammar of graphics.” This section covers only the essentials necessary to get started using the package. For many free resources on this topic, visit the website at <https://ggplot2.tidyverse.org>, where you can even download a single-page cheat sheet with the most used commands.

It would fill an entire book to demonstrate the capabilities of the `ggplot2` package, but the fundamentals can be illustrated with several basic recipes. To this end, we’ll use it to explore a dataset over 100 years in the making. The dataset describes the passengers of the Titanic ship, which sunk in the year 1912. The machine learning application is used to predict which of the 1,309 passengers were tragically killed in the disaster, and although a predictive model is of little use today, the dataset is well suited to practicing data exploration, due to having many hidden patterns, which visualizations can help reveal.



The Titanic dataset is a widely popular teaching dataset and is available from numerous online sources. The original file and documentation are available via the Vanderbilt University Department of Biostatistics, located on the web at <https://hbiostat.org/data/>. This book uses a variant of the Titanic dataset that was created to introduce learners to the Kaggle competition format. To read more or join the competition, visit <https://www.kaggle.com/c/titanic>.

We'll begin by loading the Titanic model training dataset and examining its features:

```
> titanic_train <- read.csv("titanic_train.csv")
> str(titanic_train)

'data.frame': 891 obs. of 12 variables:
 $ PassengerId: int 1 2 3 4 5 6 7 8 9 10 ...
 $ Survived    : int 0 1 1 1 0 0 0 0 1 1 ...
 $ Pclass      : int 3 1 3 1 3 3 1 3 3 2 ...
 $ Name        : chr "Braund, Mr. Owen Harris" ...
 $ Sex         : chr "male" "female" "female" "female" ...
 $ Age         : num 22 38 26 35 35 NA 54 2 27 14 ...
 $ SibSp       : int 1 1 0 1 0 0 0 3 0 1 ...
 $ Parch       : int 0 0 0 0 0 0 0 1 2 0 ...
 $ Ticket      : chr "A/5 21171" "PC 17599" "STON/O2. 3101282" ...
 $ Fare        : num 7.25 71.28 7.92 53.1 8.05 ...
 $ Cabin       : chr "" "C85" "" "C123" ...
 $ Embarked    : chr "S" "C" "S" "S" ...
```

The output shows that the dataset includes 12 features for 891 of the Titanic's 1,309 passengers; the remaining 418 passengers can be found in the `titanic_test.csv` file, representing a roughly 70/30 split for training and testing. The binary target feature `Survived` indicates whether the passenger survived the shipwreck, with 1 indicating survival and 0 indicating the less fortunate outcome. Note that in the test set, `Survived` is left blank to simulate unseen future data for prediction.

In the spirit of building a data exploration roadmap, we can start thinking about each feature's potential value for prediction. The `Pclass` column indicates the passenger class, as in first-, second-, or third-class ticket status. This, as well as the `Sex` and `Age` attributes, seem like potentially useful predictors of survival. We'll use the `ggplot2` package to explore these potential relationships in more depth. If you haven't already installed this package, do so using the `install.packages("ggplot2")` before proceeding.

Every `ggplot2` visualization is composed of layers, which place graphics upon a blank canvas. Executing the `ggplot()` function alone creates an empty gray plot area with no data points:

```
> library(ggplot2)
> p <- ggplot(data = titanic_train)
> p
```

To create something more interesting than a blank gray coordinate system, we'll need to add additional layers to the plot object stored in the `p` object. Additional layers are specified by a `geom` function, which determines the type of layer to be added. Each of the many `geom` functions requires a `mapping` parameter, which invokes the package's aesthetic function, `aes()`, to link the dataset's features to their visual depiction. This series of steps can be somewhat confusing, so the best way to learn is by example.

Let's begin by creating a simple boxplot of the `Age` feature. You'll recall that in *Chapter 2, Managing and Understanding Data*, we used R's built-in `boxplot()` feature to construct such visualizations as follows:

```
> boxplot(titanic_train$Age)
```

To accomplish the same in the `ggplot2` environment, we simply add `geom_boxplot()` to the blank coordinate system, with the `aes()` aesthetic mapping function indicating that we would like the `Age` feature to be mapped to the `y` coordinate as follows:

```
> p + geom_boxplot(mapping = aes(y = Age))
```

The resulting figures are largely similar, with only a few stylistic differences in how the data is presented. Even the use of Tukey outliers is the same across both plots:

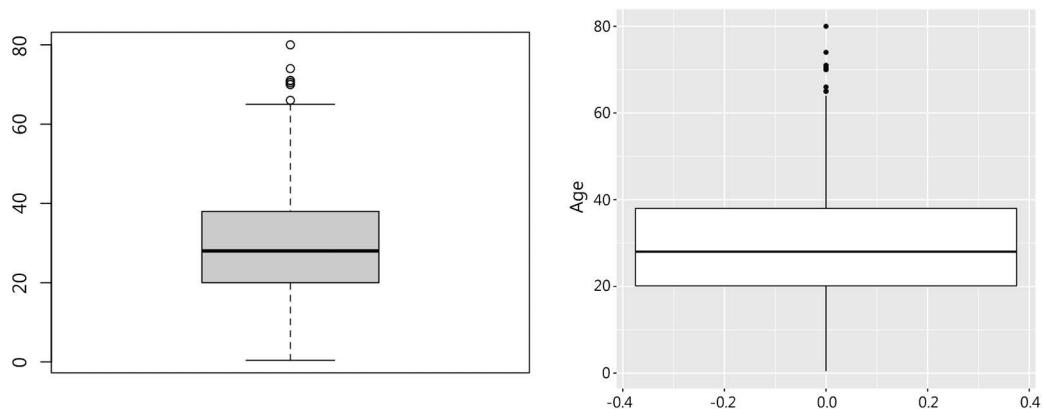


Figure 11.13: R's built-in `boxplot` function (left) compared to the `ggplot2` version of the same (right). Both depict the distribution of Titanic passenger ages.

Although it may seem pointless to use the more complicated `ggplot()` visualization when the simpler function will suffice, the strength of the framework is its ability to visualize bivariate relationships with only small changes to the code. For example, suppose we'd like to examine how age is related to survival status. We can do so using a simple modification of our previous code. Note that the `Age` has been mapped to the `x` dimension in order to create a horizontal boxplot rather than the vertical boxplot used previously. Supplying a factor-converted `Survived` as the `y` dimension creates a boxplot for each of the two levels of the factor. Using this plot, it appears that survivors tended to be a bit younger than non-survivors:

```
> p + geom_boxplot(aes(x = Age, y = as.factor(Survived)))
```

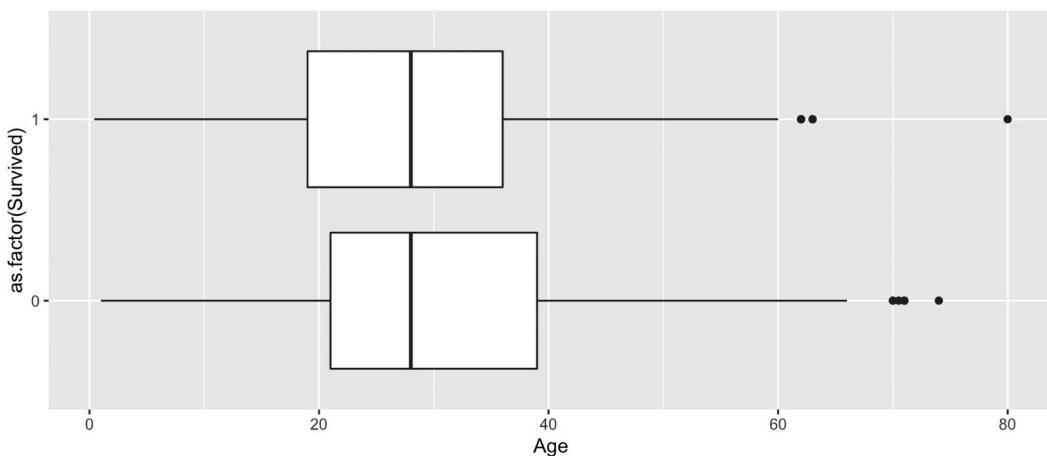


Figure 11.14: Side-by-side boxplots help compare the age distribution of Titanic's survivors and non-survivors

Sometimes a slightly different visualization can better tell a story. With this in mind, recall that in *Chapter 2, Managing and Understanding Data*, we also used R's `hist()` function to examine the distribution of numeric features. We'll begin by replicating this in `ggplot` to compare the two side by side. The built-in function is quite simple:

```
> hist(titanic_train$Age)
```

The ggplot version uses the `geom_histogram()`:

```
> p + geom_histogram(aes(x = Age))
```

The resulting figures are largely the same, aside from stylistic differences and the defaults regarding the number of bins:

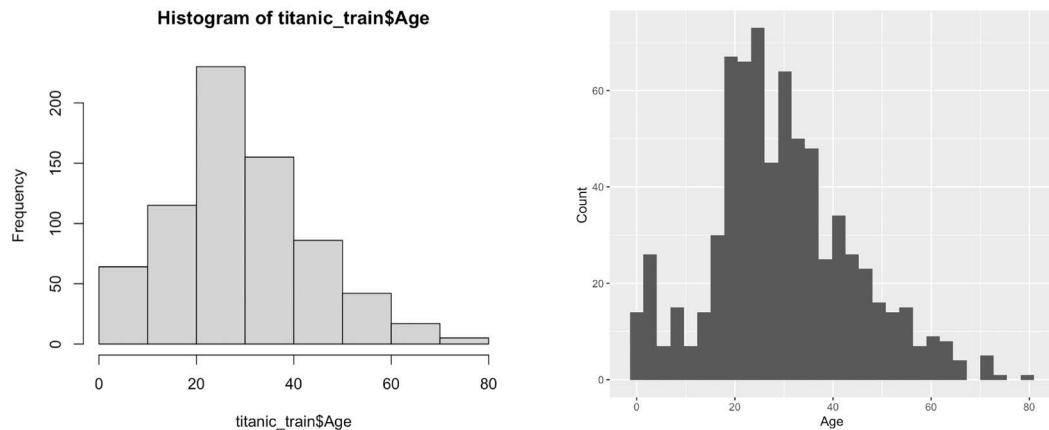


Figure 11.15: R's built-in histogram (left) compared to the ggplot2 version of the same (right) examining the distribution of Titanic passenger ages

Again, where the `ggplot2` framework shines is the ability to make a few small tweaks and reveal interesting relationships in the data. Here, let's examine three variants of the same comparison of age and survival.

First, we can construct overlapping histograms by adding a `fill` parameter to the `aes()` function. This colorizes the bars according to the levels of the factor provided. We'll also use the `ggtitle()` function to add an informative title to the figure:

```
> p + geom_histogram(aes(x = Age, fill = as.factor(Survived))) +
  ggtitle("Distribution of Age by Titanic Survival Status")
```

Second, rather than having overlapping histograms, we can create a grid of side-by-side plots using the `facet_grid()` function. This function takes `rows` and `cols` parameters to define the cells in the grid. In our case, to create side-by-side plots, we need to define the columns for survivors and non-survivors using the `Survived` variable. This must be wrapped by the `vars()` function to denote that it's a feature from the accompanying dataset:

```
> p + geom_histogram(aes(x = Age)) +
  facet_grid(cols = vars(Survived)) +
  ggtitle("Distribution of Age by Titanic Survival Status")
```

Third, rather than using the histogram `geom`, we can use `geom_density()` to create a density plot. This type of visualization is like a histogram but uses a smoothed curve instead of individual bars to depict the proportion of records at each value of the `x` dimension. We'll set the color of the line based on the levels of `Survived` and fill the area beneath the curve with the same color. Because the areas overlap, the `alpha` parameter allows us to control the level of transparency so that both may be seen at the same time. Note that this is a parameter of the `geom` function and not the `aes()` function. The complete command is as follows:

```
> p + geom_density(aes(x = Age,
  color = as.factor(Survived),
  fill = as.factor(Survived)),
  alpha = 0.25) +
  ggtitle("Density of Age by Titanic Survival Status")
```

The resulting three figures visualize the same data in different ways:

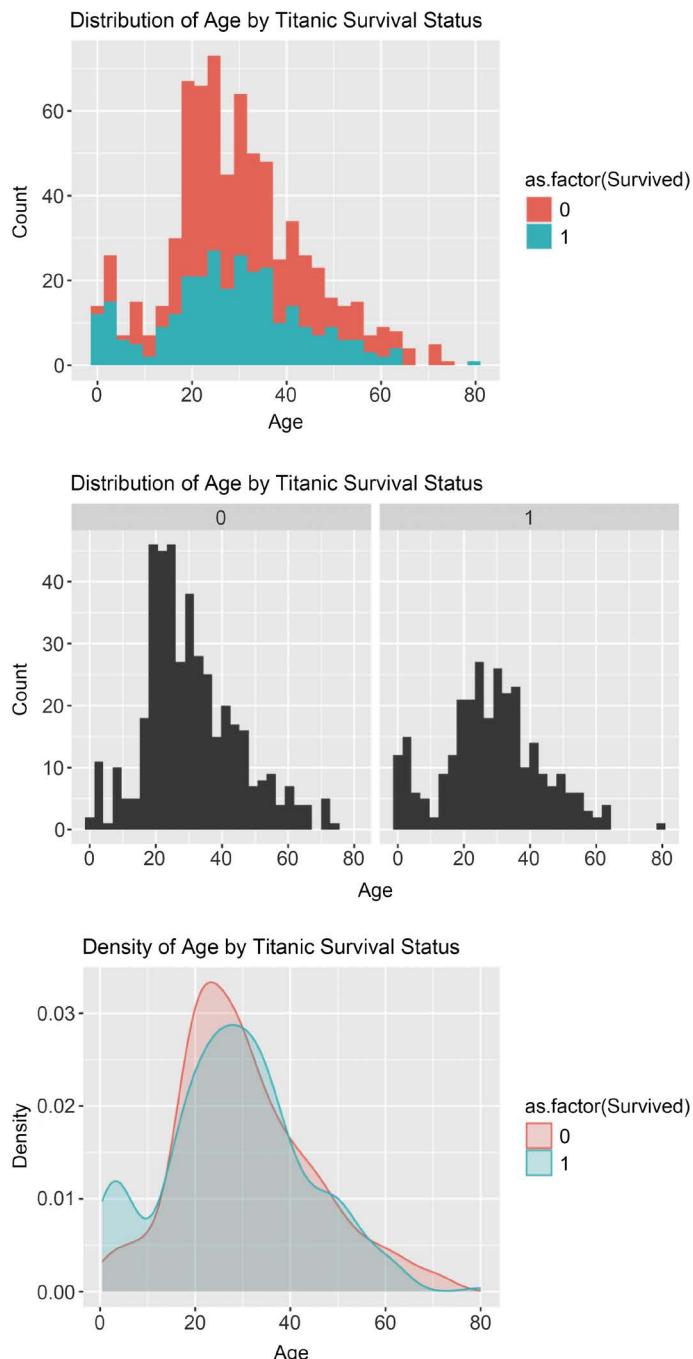


Figure 11.16: Small changes in the `ggplot()` function call can create vastly different outputs

These three visualizations demonstrate the fact that different visualizations of the same data can help tell different stories. For example, the top figure with overlapping histograms seems to highlight the fact that a relatively small proportion of people survived. In contrast, the bottom figure clearly depicts the spike in survival for travelers below 10 years of age; this provides evidence of a “women and children first” policy for the lifeboats—at least with respect to children.

Let’s examine a few more plots to see if we can uncover more details of the Titanic’s evacuation policies. We’ll begin by confirming the assumed differences in survival by gender. For this, we’ll create a simple bar chart using the `geom_bar()` layer. By default, this simply counts the number of occurrences of the supplied dimension. The following command creates a bar chart, illustrating the fact that there were nearly twice as many males as females on board the Titanic:

```
> p + geom_bar(aes(x = Sex)) +
  ggtitle("Titanic Passenger Counts by Gender")
```

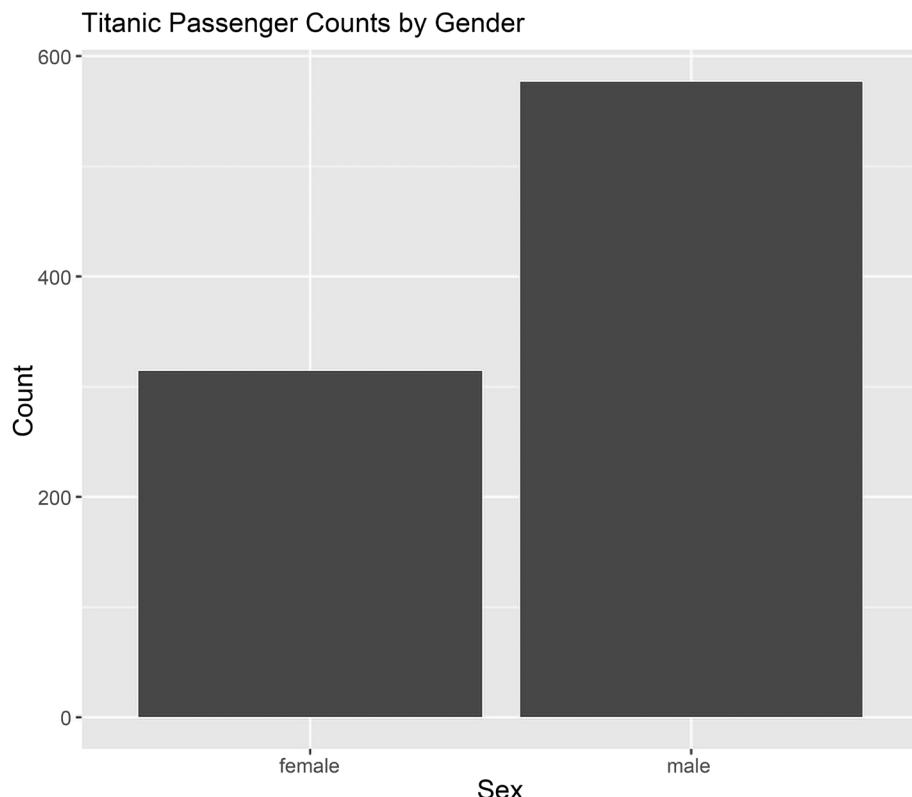


Figure 11.17: A simple bar chart of a single feature helps to put count data into perspective

A more interesting visualization would be to compare the rate of survival by gender. To do this, we must not only supply the `Survived` outcome as the `y` parameter to the `aes()` function but also tell the `geom_bar()` function to compute a summary statistic of the data—in particular, using the `mean` function—using the `stat` and `fun` parameters as shown:

```
> p + geom_bar(aes(x = Sex, y = Survived),  
+                 stat = "summary", fun = "mean") +  
+                 ggtitle("Titanic Survival Rate by Gender")
```

The resulting figures confirm the assumption of a “women and children first” lifeboat policy. Although there were almost twice as many men on board, women were three times more likely to survive:

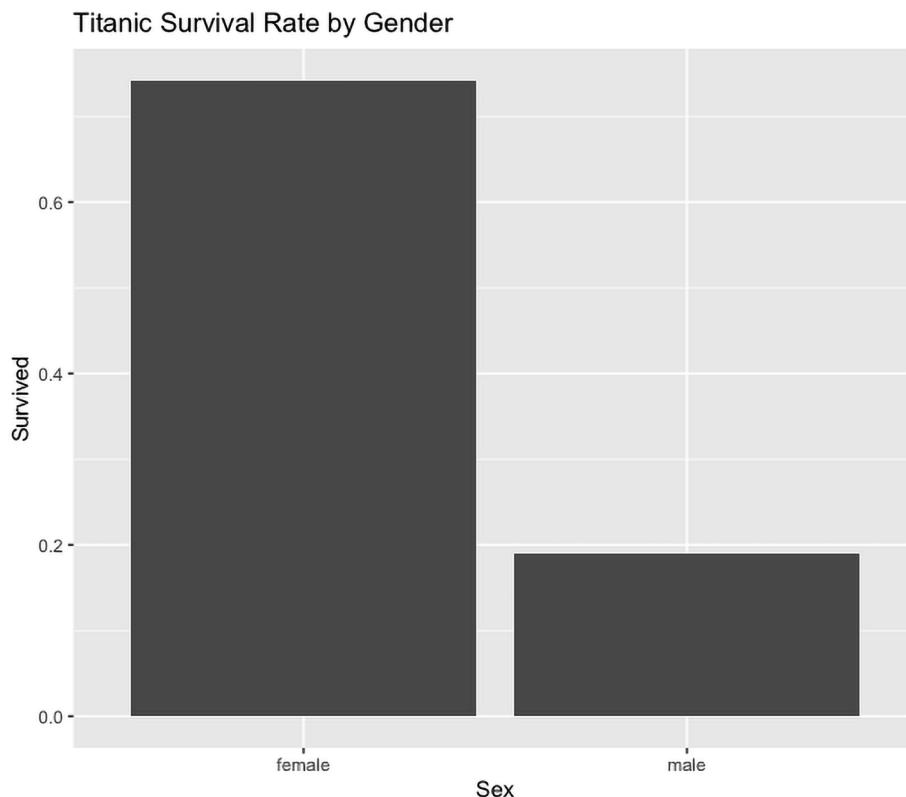


Figure 11.18: More complex bar charts can illustrate disparities in survival rates by gender

To once again demonstrate ggplot's ability to create a large variety of visualizations and tell different stories about the data with relatively small changes to the code, we'll examine the passenger class (`Pclass`) feature in a few different ways. First, we'll create a simple bar chart that depicts the survival rate using the `stat` and `fun` parameters, just as we did for survival rate by gender:

```
> p + geom_bar(aes(x = Pclass, y = Survived),  
               stat = "summary", fun = "mean") +  
  ggtitle("Titanic Survival Rate by Passenger Class")
```

The resulting figure depicts a substantial decline in survival likelihood for second- and third-class passengers:

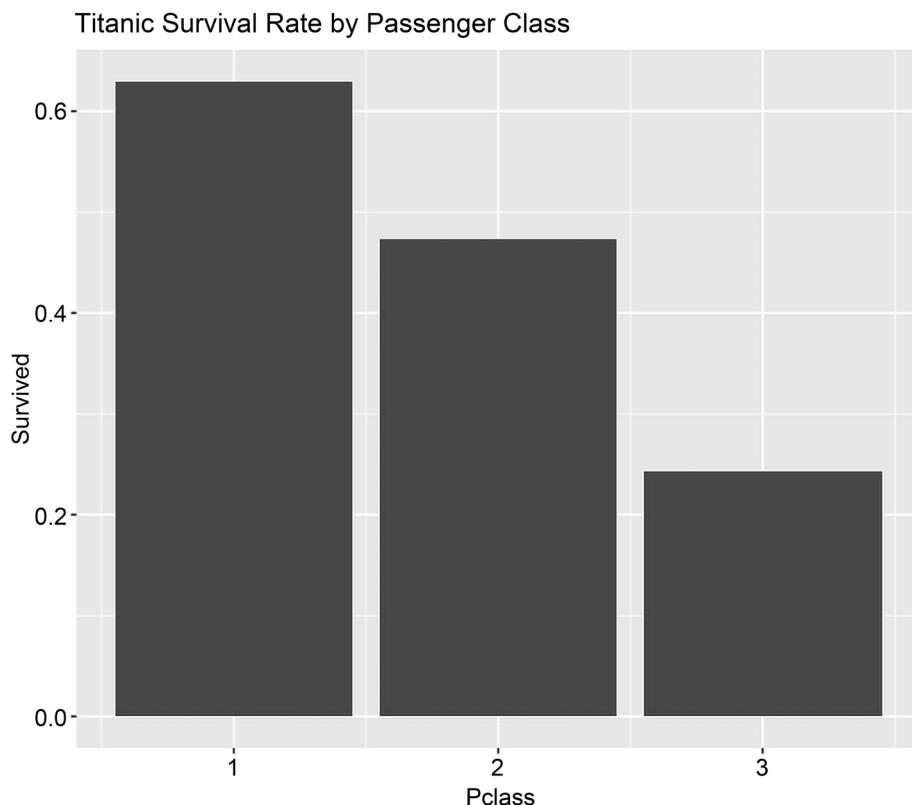


Figure 11.19: A bar chart shows a clear disparity in survival outcomes for Titanic's lower passenger classes

Color can be an effective tool to communicate additional dimensions. Using the `fill` parameter, we'll create a simple bar chart of passenger counts with bars that are filled with color according to survival status, which is converted to a factor:

```
> p + geom_bar(aes(x = Pclass,
                     fill = factor(Survived,
                                   labels = c("No", "Yes")))) +
  labs(fill = "Survived") +
  ylab("Number of Passengers") +
  ggtitle("Titanic Survival Counts by Passenger Class")
```

The result highlights the fact that the overwhelming number of deceased came from the third-class section of the ship:

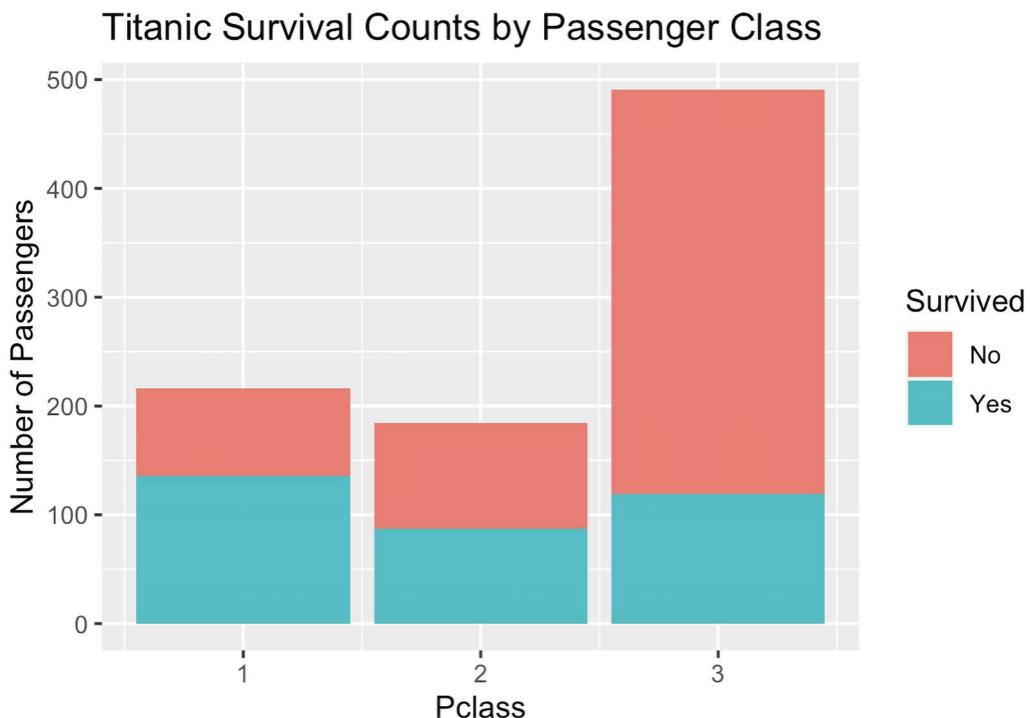


Figure 11.20: A bar chart emphasizing the number of third-class passengers that died on the *Titanic*

Next, we'll modify this plot using a position parameter that informs `ggplot()` how to arrange the colorized bars. In this case, we'll set `position = "fill"`, which creates a stacked bar chart that fills the vertical space—essentially giving each color in the stack a relative proportion out of 100 percent:

```
> p + geom_bar(aes(x = Pclass,
                    fill = factor(Survived,
                                  labels = c("No", "Yes"))),
                    position = "fill") +
  labs(fill = "Survived") +
  ylab("Proportion of Passengers") +
  ggtitle("Titanic Survival by Passenger Class")
```

The resulting figure emphasizes the decreased odds of survival for the lower classes:

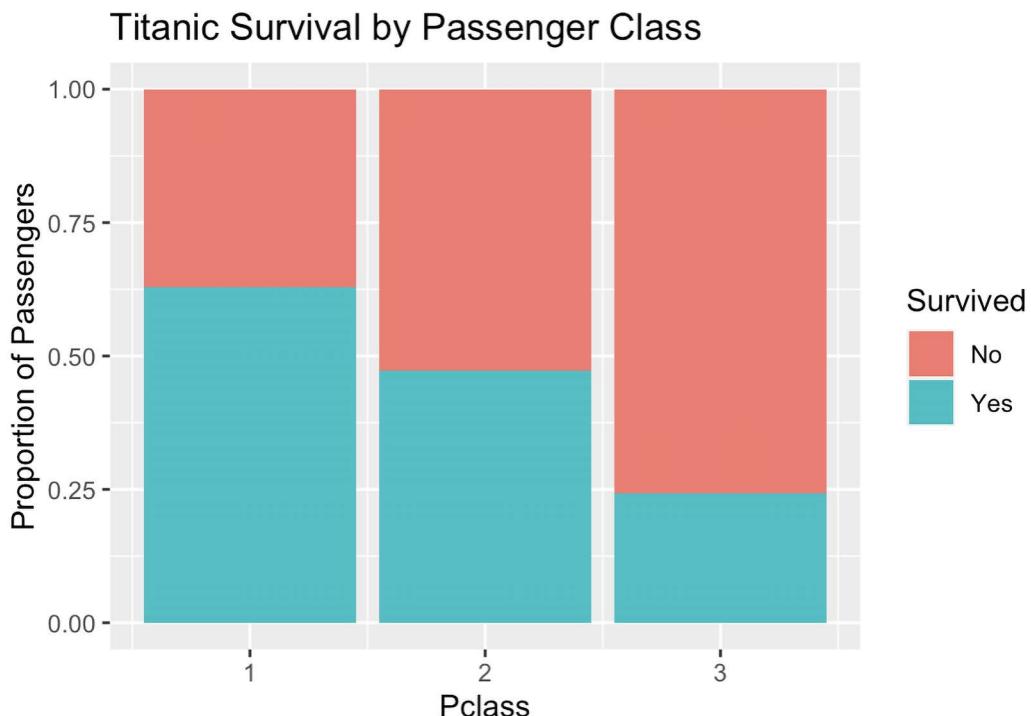


Figure 11.21: A bar chart contrasting the survival rates by passenger class

Lastly, we'll attempt to visualize the relationship between three dimensions: passenger class, gender, and survival. The Pclass and Survived features define the x and y dimensions, leaving Sex to define the bar colors via the fill parameter. Setting the position = "dodge" tells ggplot() to place the colored bars side-by-side rather than stacked, while the stat and fun parameters compute the survival rate. The full command is as follows:

```
> p + geom_bar(aes(x = Pclass, y = Survived, fill = Sex),  
               position = "dodge", stat = "summary", fun = "mean") +  
  ylab("Survival Proportion") +  
  ggtitle("Titanic Survival Rate by Class and Sex")
```

This figure reveals the fact that nearly all first- and second-class female passengers survived, whereas men of all classes were more likely to perish:

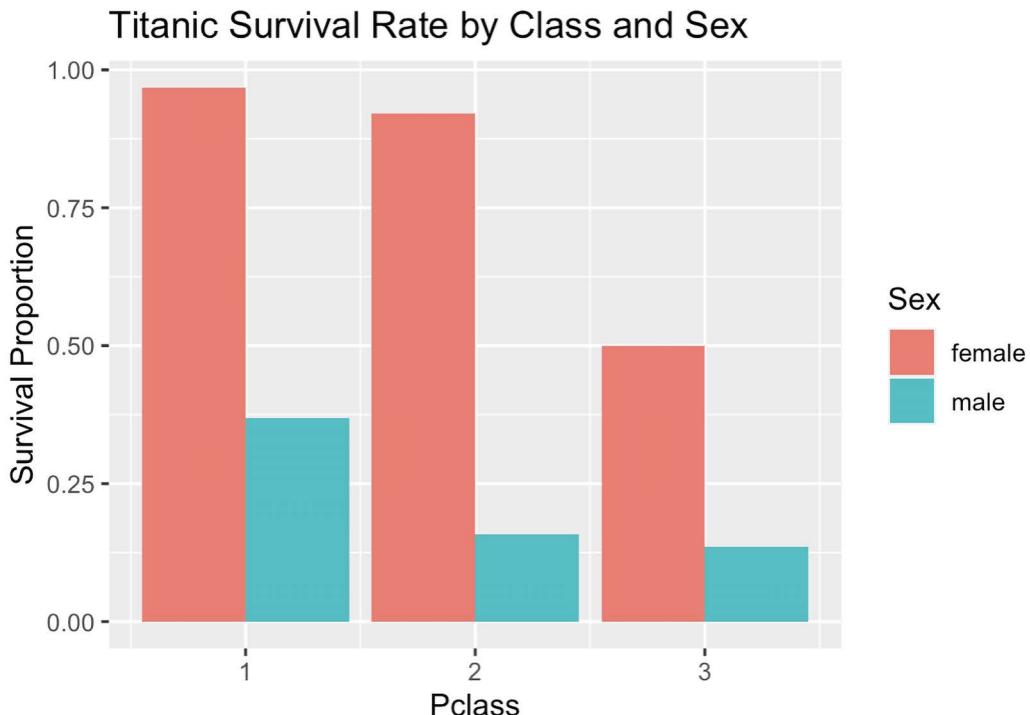
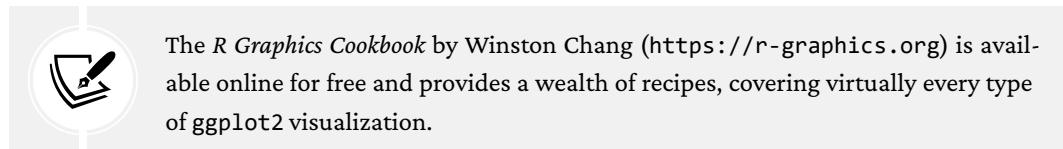


Figure 11.22: A bar chart illustrating the low survival rate for males, regardless of passenger class

Examining more facets of the Titanic data is an exercise best left to the reader. After all, data exploration may be best thought of as a personal conversation between the data and the data scientist. Similarly, as mentioned before, it is beyond the scope of this book to cover every aspect of the `ggplot2` package. Still, this section should have demonstrated ways in which data visualization can help identify connections between features, which is useful for developing a rich understanding of the data. Diving deeper into the capabilities of the `ggplot()` function, perhaps by exploring a dataset of personal interest to you, will do much to improve your model building and storytelling skills—both of which are important elements of being successful with machine learning.



The *R Graphics Cookbook* by Winston Chang (<https://r-graphics.org>) is available online for free and provides a wealth of recipes, covering virtually every type of `ggplot2` visualization.

Summary

In this chapter, you learned the fundamentals of what it means to be a successful machine learning practitioner and the skills necessary to build successful machine learning models. These require not only a broad set of requisite knowledge and experience but also a thorough understanding of the learning algorithms, the training dataset, the real-world deployment scenario, and the myriad ways that the work can go wrong—either by accident or by design.

The data science buzzword suggests a relationship between the data, the machine, and the people who guide the learning process. This is a team effort, and the growing emphasis on data science as a distinct outgrowth from the field of data mining that came before it, with numerous degree programs and online certifications, reflects its operationalization as a field of study concerned with not just statistics, data, and computer algorithms but also the technologic and bureaucratic infrastructure that enables applied machine learning to be successful.

Applied machine learning and data science ask their practitioners to be compelling explorers and storytellers. The audacious use of data must be carefully balanced with what truly can be learned from the data, and what may reasonably be done with what is learned. This is certainly both an art and a science, and therefore, few can master the field in its entirety. Instead, striving to constantly improve, iterate, and compete will lead to an improvement of self that inevitably leads to models that better perform at their intended real-world applications. This contributes to the so-called “virtuous cycle” of artificial intelligence, in which a flywheel-like effect rapidly increases the productivity of organizations employing data science methods.

Just as this chapter revisited familiar topics and revealed the newfound complexity in the real-world practice of machine learning, the next chapter revisits data preparation to consider solutions to common problems found in large and messy datasets. We'll work our way back into the trenches by learning about a completely new way to program in R, which is not only more capable of handling such challenges but also, once the initial learning curve has been passed, perhaps even more fun and intuitive to use.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



12

Advanced Data Preparation

The truism that 80 percent of the time invested in real-world machine learning projects is spent on data preparation is so widely cited that it is mostly accepted without question. Earlier chapters of this book helped perpetuate the cliché by stating it as a matter of fact without qualification, and although it is certainly a common experience and perception, it is also an oversimplification, as tends to be the case when generalizing from a statistic. In reality, there is no single, uniform experience for data preparation. Yet, it is indeed true that data prep work almost always involves more effort than anticipated.

Rare is the case in which you will be provided a single CSV formatted text file, which can be easily read into R and processed with just a few lines of R code, as was the case in previous chapters. Instead, necessary data elements are often distributed across databases, which must then be gathered, filtered, reformatted, and combined before the features can be used with machine learning. This can require significant effort even before considering the time expended gaining access to the data from stakeholders, as well as exploring and understanding the data.

This chapter is intended to prepare you (pun intended!) for the larger and more complex datasets that you'll be preparing in the real world. You will learn:

- Why data preparation is crucial to building better models
- Tips and tricks for transforming data into more useful predictors
- Specialized R packages for efficiently preparing data

Different teams and different projects require their data scientists to invest different amounts of time preparing data for the machine learning process, and thus, the 80 percent statistic may overstate or understate the effort needed for any given project or from any single contributor.

Still, whether it is you or someone else performing this work, you will soon discover the undeniable fact that advanced data preparation is a necessary step in the process of building strong machine learning projects.

Performing feature engineering

Time, effort, and imagination are central to the process of **feature engineering**, which involves applying subject-matter expertise to create new features for prediction. In simple terms, it might be described as the art of making data more useful. In more complex terms, it involves a combination of domain expertise and data transformations. One needs to know not just what data will be useful to gather for the machine learning project, but also how to merge, code, and clean the data to conform to the algorithm's expectations.

Feature engineering is closely interrelated with data exploration, as described in *Chapter 11, Being Successful with Machine Learning*. Both involve interrogating data through the generation and testing of hypotheses. Exploring and brainstorming are likely to lead to insights about which features will be useful for prediction, and the act of engineering the features may lead to new questions to explore.

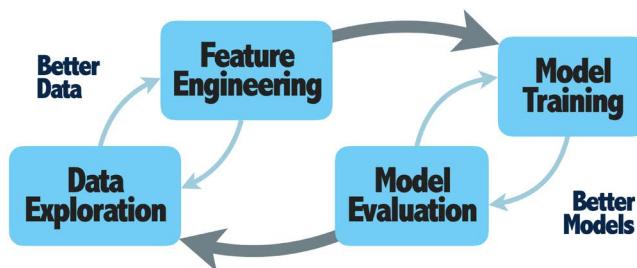


Figure 12.1: Feature engineering is part of a cycle that helps the model and data work together

Feature engineering is part of a cycle within a cycle in which effort is invested to help the model and data work better together. A round of data exploration and feature engineering leads to improvements to the data, which leads to iterations of training better models, which then informs another round of potential improvements to the data. These potential improvements are not only the bare minimum cleaning and preparation tasks needed to address simple data issues and allow the algorithm to run in R, but also the steps that lead an algorithm to learn more effectively. These may include:

- Performing complex data transformations that help the algorithm to learn faster or to learn a simpler representation of the data

- Creating features that are easier to interpret or better represent the underlying theoretical concepts
- Utilizing unstructured data or merging additional features onto the main source

All three of these require both intense thought and creativity, and are improvisational and domain-specific rather than formulaic. This being said, the computer and the practitioner can share this work using complementary strengths. What the computer lacks in creativity and ability to improvise, it may be able to address with computational horsepower, brute force, and unwavering persistence.

The role of human and machine

Feature engineering can be viewed as a collaboration between the human and the machine during the learning process stage of abstraction. Recall that in *Chapter 1, Introducing Machine Learning*, the abstraction step was defined as the translation of stored data into broader concepts and representations. In other words, during abstraction, connections are made between elements of raw data, which will represent important concepts for the learning objective. These relationships are generally defined by a model, which links the learned concepts to an outcome of interest. During feature engineering, the human gently guides or nudges the abstraction process in a specific direction, with the goal of producing a better-performing model.

Imagine it this way: recall an instance in your past where you attempted to learn a difficult concept—possibly even while reading this very textbook! Reading and later re-reading the text proves to be of no help to understanding the concept, and frustrated, you contact a friend or colleague for help. Perhaps this friend explains the concept in a different way, using analogies or examples that help connect the concept to your prior experience, and in doing so, it leads you to a moment of enlightenment: “Eureka!” All is suddenly clear, and you wonder how you couldn’t understand the concept in the first place. Such is the power of abstractions, which can be transferred from one learner to another to aid the learning process. The process of feature engineering allows the human to transfer their intuitive knowledge or subject-matter expertise to the machine through intentionally and purposefully designed input data.

Given the fact that abstraction is the cornerstone of the learning process, it can be argued that machine learning is fundamentally feature engineering. The renowned computer scientist and artificial intelligence pioneer Andrew Ng said, “*Coming up with features is difficult, time-consuming, and requires expert knowledge. Applied machine learning is basically feature engineering.*” Pedro Domingos, a professor of computer science and author of the machine learning book *The Master Algorithm*, said that “*some machine learning projects succeed and some fail. What makes the difference? Easily the most important factor is the features used.*”



Andrew Ng's quote appears in a lecture titled *Machine Learning and AI via Brain simulations*, which is available online via web search. In addition to Pedro Domingos' book *The Master Algorithm* (2015), see also his excellent paper "A few useful things to know about machine learning" in *Communications of the ACM* (2012). <https://doi.org/10.1145/2347736.2347755>.

Feature engineering performed well can turn weaker learners into much stronger learners. Many machine learning and artificial intelligence problems can be solved with simple linear regression methods, assuming that the data has been sufficiently cleaned. Even very complex machine learning methods can be replicated in standard linear regression given sufficient feature engineering. Linear regression can be adapted to model nonlinear patterns, using splines and quadratic terms, and can approach the performance of even the most complex neural networks, given a sufficient number of new features that have been engineered as carefully designed interactions or transformations of the original input data.

The idea that simple learning algorithms can be adapted to more complex problems is not limited to regression. For example, decision trees can work around their axis-parallel decision boundaries by rotations of the input data, while hyperplane-based support vector machines can model complex nonlinear patterns with a well-chosen kernel trick. A method as simple as k-NN could be used to mimic regression or perhaps even more complex methods, given enough effort and sufficient understanding of the input data and learning problem, but herein lies the catch. Why invest large amounts of time performing feature engineering to employ a simple method when a more complex algorithm will perform just as well or better, while also performing the feature engineering for us automatically?

Indeed, it is probably best to match the complexity of the data's underlying patterns with a learning algorithm capable of handling them readily. Performing feature engineering by hand when a computer can do it automatically is not only wasted effort but also prone to mistakes and missing important patterns. Algorithms like decision trees and neural networks with a sufficiently large number of hidden nodes—and especially, deep learning neural networks—are particularly capable of doing their own form of feature engineering, which is likely to be more rigorous and thorough than what can be done by hand. Unfortunately, this does not mean we can blindly apply these same methods to every task—after all, there is no free lunch in machine learning!

Applying the same algorithm to every problem suggests that there is a one-size-fits-all approach to feature engineering, when we know that it is as much an art as it is a science. Consequently, if all practitioners apply the same method to all tasks, they will have no way of knowing whether better performance is possible. Perhaps a slightly different feature engineering approach could have resulted in a model that more accurately predicted churn or cancer, and would have led to greater profits or more lives saved. This is clearly a problem in the real world, where even a small performance boost can mean a substantial edge over the competition.

In a high-stakes competition environment, such as the machine learning competitions on Kaggle, each team has access to the same learning algorithms and is readily capable of rapidly applying each of them to identify which one performs best. It is no surprise, then, that a theme emerges while reading interviews with Kaggle champions: they often invest significant effort into feature engineering. Xavier Conort, who was the top-rated data scientist on Kaggle in 2012–2013, said in an interview that:



"The algorithms we used are very standard for Kagglers... We spent most of our efforts on feature engineering... We were also very careful to discard features likely to expose us to the risk of overfitting."

Because feature engineering is one of the few proprietary aspects of machine learning, it is one of the few points of distinction across teams. In other words, teams that perform feature engineering well tend to outperform the competition.



To read the full interview with Xavier Conort, which was originally posted on the Kaggle “No Free Hunch” blog, visit <https://web.archive.org/web/20190609154949/http://blog.kaggle.com/2013/04/10/qa-with-xavier-conort/>. Interviews with other Kaggle champions are available at <https://medium.com/kaggle-blog/tagged/kaggle-competition>.

Based on Conort’s statement, it would be easy to assume that the need to invest in feature engineering necessitates greater investment in human intelligence and the application of subject-matter expertise, but this is not always true. Jeremy Achin, a member of a top-performing “DataRobot” team on Kaggle, remarked on the surprisingly limited utility of human expertise. Commenting on his team’s time spent on feature engineering, he said in an interview that:



"The most surprising thing was that almost all attempts to use subject matter knowledge or insights drawn from data visualization led to drastically worse results. We actually arranged a 2-hour whiteboard lecture from a very talented biochemist and came up with some ideas based on what we learned, but none of them worked out."

Jeremy Achin, along with Xavier Conort and several other high-profile Kaggle Grand Masters, bootstrapped their Kaggle competition successes into an artificial intelligence company called DataRobot, which is now worth billions of dollars. Their software performs machine learning automatically, suggesting that a key lesson learned from their Kaggle work was that computers can perform many steps in the machine learning process just as well as humans, if not better.

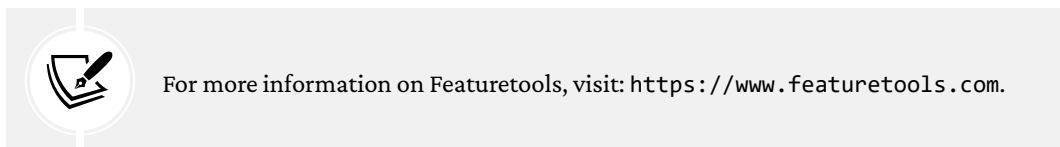


To read the full interview with Jeremy Achin, which was originally posted on the Kaggle "No Free Hunch" blog, visit <https://web.archive.org/web/20190914030000/http://blog.kaggle.com/2012/11/04/team-datarobot-merck-2nd-place-interview/>. The DataRobot company is found on the web at <https://www.datarobot.com>.

Of course, there is a balance between building models piece by piece using subject-matter expertise and throwing everything at the machine to see what sticks. Although feature engineering today is largely still a manual process, the future of the field seems to be headed toward the scatter-shot "see what sticks" approach, as **automated feature engineering** is a rapidly growing area of research. The foundation of automated feature engineering tools is the idea that a computer can make up for its lack of creativity and domain knowledge by testing many more combinations of features than a human would ever have time to attempt. Automated feature engineering exchanges narrow-but-guided human thought for broad-and-systematic computer thought, with the potential upside of finding a more optimal solution and potential downsides including loss of interpretability and greater likelihood of overfitting.

Before getting too excited about the potential for automation, it is worth noting that while such tools may allow a human to outsource certain parts of *thinking* about feature engineering, effort must still be invested in the *coding* part of the task. That is to say, time that was once spent hand-coding features one by one is instead spent coding functions that systematically find or construct useful features.

There are promising algorithms in development, such as the Python-based `Featuretools` package (and corresponding R package `featuretoolsR`, which interacts with the Python code), that may help automate the feature-building process, but the use of such tools is not yet widespread. Additionally, such methods must be fed by data and computing time, both of which may be limiting factors in many machine learning projects.



For more information on Featuretools, visit: <https://www.featuretools.com>.

The impact of big data and deep learning

Whether feature engineering is performed by a human or by automated machine methods, a point is inevitably reached at which additional invested effort leads to little or no boost to the learning algorithm's performance. The application of more sophisticated learning algorithms may also improve the model's performance somewhat, but this is also subject to diminishing returns, as there exists only a finite number of potential methods to apply and their performance differences tend to be relatively minor. Consequently, if additional performance gains are truly necessary, we are left with one remaining option: increasing the size of the training dataset with additional features or examples. Moreover, because adding additional columns would require revising data generated by past business processes, in many cases, collecting more rows is the easier option of the two.

In practice, there is a relatively low ceiling on the performance gains achievable via the inclusion of more rows of data. Most algorithms described in this book plateau quickly and will perform little better on a dataset of 1 million rows than on a dataset containing a few thousand. You may have already observed this firsthand if you've applied machine learning methods to real-world projects in your own areas of interest. Once a dataset is big enough—often just a few thousand rows for many real-world applications—additional examples merely cause additional problems, such as extended computation time and running out of memory. If more data causes more problems, then the natural follow-up question is, why is there so much hype around the so-called “big data” era?

To answer this question, we must first begin by making a philosophical distinction between datasets of various sizes. To be clear, “big data” does not merely imply a large number of rows or a large amount of storage consumed in a database or filesystem. In fact, it comprises both of these and more, as size is just one of four elements that may indicate the presence of big data.

These are the so-called four V's of big data:

- **Volume:** The literal size of the data, whether it be more rows, more columns, or more storage
- **Velocity:** The speed at which data accumulates, which impacts not only the volume but also the complexity of data processing
- **Variety:** The differences in types or definitions of data across different systems, particularly the addition of unstructured sources such as text, images, and audio data
- **Veracity:** The trustworthiness of the input data and the ability to match data across sources

Reading this list from top to bottom, the elements become less intuitively obvious, yet are more challenging to handle when encountered. The first two elements, volume and velocity, are the basis of what might be dubbed the **medium data** space. While this is not to say that there aren't challenges working with high-volume, high-velocity data, these challenges can often be solved by scaling up what we are already doing. For example, it may be possible to use faster computers with more memory or apply a more computationally efficient algorithm. The presence of a greater variety and reduced veracity of data requires a completely different approach for use in machine learning projects, especially at high-velocity and high-volume scales. The following table lists some of the distinctions between the small, medium, and big data spaces:

	Small Data	Medium Data	Big Data
Storage Format	Spreadsheets	Databases	+ Data Lakes & NoSQL Databases
Analysis Methods	Traditional Statistics	+ Machine Learning	+ Deep Learning & Artificial Intelligence
Computing Skills	Excel, Visualizations	+ SQL, R, Python	+ Hadoop/Spark, TensorFlow, H2O, etc.
Computing Hardware	Laptop	Laptop/Server	Cloud

Figure 12.2: Most machine learning projects are on the scale of “medium data” while additional skills and tools are required to make use of “big data”

Moving from small to medium and then from medium to big data requires exponential investment. As datasets increase in size and complexity, the required infrastructure becomes much more complex, adding increasingly specialized databases, computing hardware, and analysis tools, some of which will be covered in *Chapter 15, Making Use of Big Data*. These tools are rapidly changing, which necessitates constant training and re-skilling. With the increased scale of data, time becomes a more significant constraint; not only are the projects more complex with many more moving pieces, requiring more cycles of iteration and refinement, but the work simply takes longer to complete—literally! A machine learning algorithm that runs in minutes on a medium-sized dataset may take hours or days on a much larger dataset, even with the benefit of cloud computing power.

Given the high stakes of big data, there is often an order of magnitude difference in how such projects are staffed and resourced—it is simply considered part of “the cost of doing business.” There may be dozens of data scientists, with matching numbers of IT professionals supporting the required infrastructure and data processing pipeline. Typical big data solutions require numerous tools and technologies to work together. This creates an opportunity for **data architects** to plan and structure the various computing resources and to monitor their security, performance, and cloud hosting costs. Similarly, data scientists are often matched by an equal or greater number of **data engineers**, who are responsible for piping data between sources and doing the most complex programming work. Their efforts in processing large datasets allow data scientists to focus on analysis and machine learning model building.

From the perspective of those working on the largest and most challenging machine learning projects today, many everyday projects, including nearly all the examples covered in this book, fall squarely into what has been called a **small data regime**. In this paradigm, datasets can grow to be “large” in terms of the number of rows or in sheer storage volume, but they will never truly be “big data.” Computer science and machine learning expert Andrew Ng has noted that in the realm of small data, the role of the human is still impactful; the human can greatly impact a project’s performance via hand-engineering features or by the selection of the most performant learning algorithm. However, as a dataset grows beyond “large” and into “huge” sizes and into the **big data regime**, a different class of algorithms breaks through the performance plateau to surpass the small gains of manual tweaks.

Figure 12.3, which is adapted from Ng's work, illustrates this phenomenon:

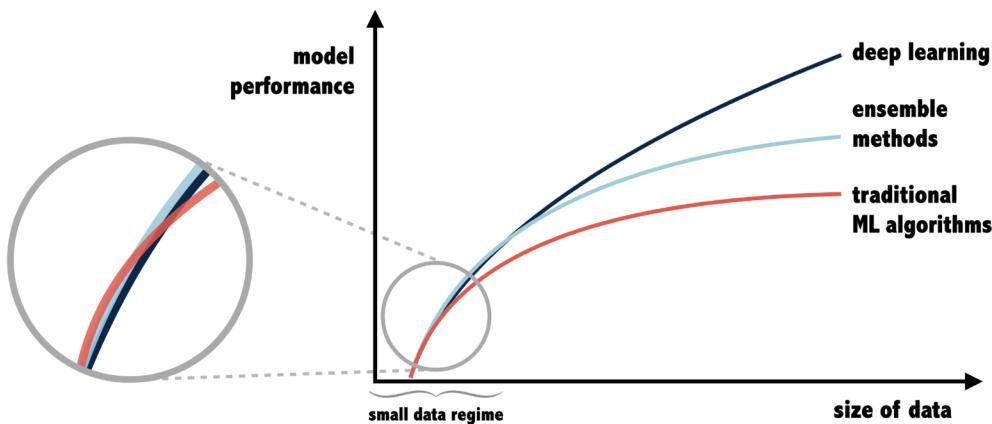


Figure 12.3: In the small data regime, traditional machine learning algorithms are competitive with and may even perform better than more complex methods, which perform much better as the size of data increases

Within the confines of the small data regime, no single algorithm or class of algorithms performs predictably better than the others. Here, clever feature engineering including subject-matter expertise and hand-coded features may allow simpler algorithms to outperform much more sophisticated approaches or deep learning neural networks.

As the size of data increases to the medium data regime, ensemble approaches (described in *Chapter 14, Building Better Learners*) tend to perform better than even a carefully handcrafted model that uses traditional machine learning algorithms. For the largest datasets found in the big data regime, only deep learning neural networks (introduced in *Chapter 7, Black-Box Methods – Neural Networks and Support Vector Machines*, and to be covered in more detail in *Chapter 15, Making Use of Big Data*) appear to be capable of the utmost performance, as their capability to learn from additional data practically never plateaus. Does this imply that the “no free lunch” theorem is incorrect and there truly is one learning algorithm to rule them all?



The visualization of the performance of different learning algorithms under the small and big data regimes can be found described by Andrew Ng in his own words. To find these, simply search YouTube for “Nuts and Bolts of Applying Deep Learning” (appears 3 minutes into the video) or “Artificial Intelligence is the New Electricity” (appears 20 minutes into the video).

To understand why certain algorithms perform better than others under the big data regime and why the “no free lunch” principle still applies, we must first consider the relationship between the size and complexity of the data, the capability of a model to learn a complex pattern, and the risk of overfitting. Let’s begin by considering a case in which the size and complexity of the data is held constant, but we increase the complexity of the learning algorithm to more closely model what is observed in the training data. For example, we may grow a decision tree to an overly large size, increase the number of predictors in a regression model, or add hidden nodes in a neural network. This relationship is closely linked to the idea of the bias-variance trade-off; by increasing the model complexity, we allow the model to conform more closely to the training data and, therefore, reduce its inherent bias and increase its variance.

Figure 12.4 illustrates the typical pattern that occurs as models increase in complexity. Initially, when the model is underfitted to the training dataset, increases in model complexity lead to reductions in model error and increases in model performance. However, there is a point at which increases in model complexity contribute to overfitting the training dataset. Beyond this point, although the model’s error rate on the training dataset continues to be reduced, the test set error rate increases, as the model’s ability to generalize beyond training is dramatically hindered. Again, this assumes a limit to the dataset’s ability to support the model’s increasing complexity.

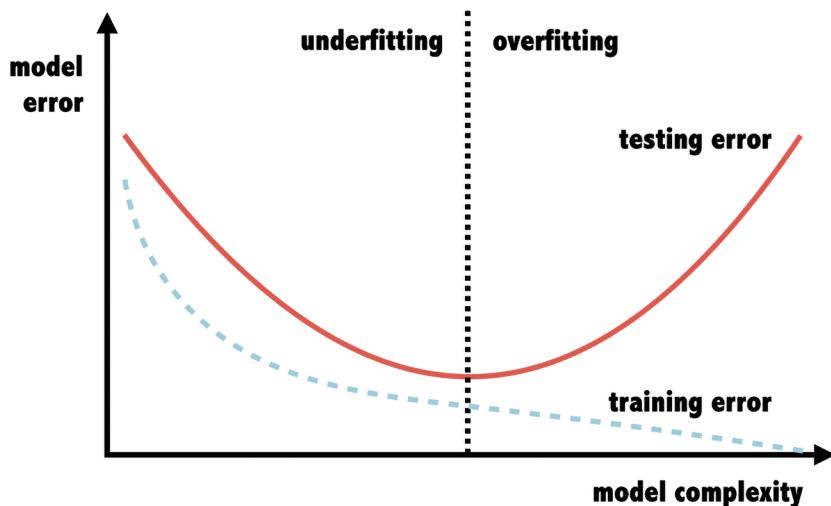


Figure 12.4: For many training datasets, increasing the complexity of the learning algorithm runs the risk of overfitting and increased test set error

If we can increase the size and scope of the training dataset, the big data regime may unlock a second tier of machine learning performance, but only if the learning algorithm is likewise capable of increasing its complexity to make use of the additional data. Many traditional algorithms, such as those covered so far in this book, are incapable of making such an evolutionary leap—at least not without some extra help.

The missing link between the traditional machine learning algorithms and those capable of making this leap has to do with the number of parameters that the algorithms attempt to learn about the data. Recall that in *Chapter 11, Being Successful with Machine Learning*, parameters were described as the learner’s internal values that represent its abstraction of the data. Traditionally, for a variety of reasons, including the bias-variance trade-off depicted above, as well as the belief that simpler, more parsimonious models should be favored over more complex ones, models with fewer parameters have been favored. It was assumed that increasing the number of parameters too high would allow the dataset to simply memorize the training data, leading to severe overfitting.

Interestingly, this is true, but only to a point, as *Figure 12.5* depicts. As model complexity—that is, the number of parameters—increases, the test set error follows the same U-shaped pattern as before. However, a new pattern emerges once complexity and parameterization have reached the **interpolation threshold**, or the point at which there are enough parameters to memorize and accurately classify virtually all the training set examples. At this threshold, generalization error is at its maximum, as the model has been greatly overfitted to the training data. However, as model complexity increases even further, test set error once again begins to drop. With sufficient additional complexity, a heavily overfitted model may even surpass the performance of a well-tuned traditional model, at least according to our existing notion of “overfitted.”

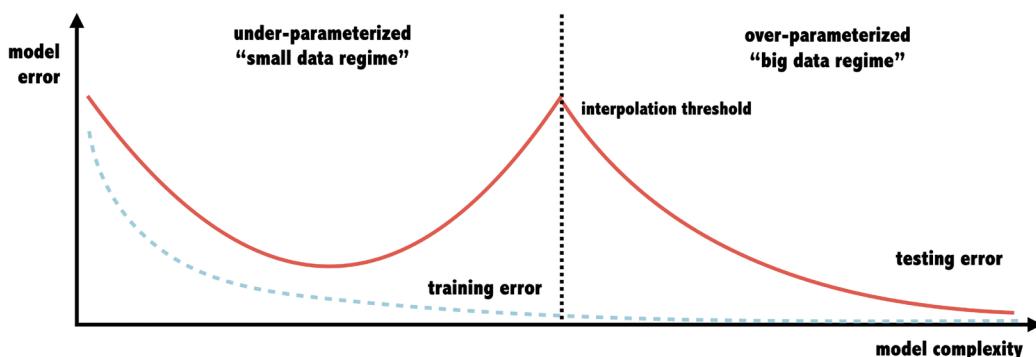


Figure 12.5: Some algorithms are able to make use of big data to generalize well even after they seemingly overfit the training data



For more information on the apparent contradiction of the “double descent” curve depicted here, see this groundbreaking paper: *Reconciling modern machine-learning practice and the classical bias–variance trade-off*, Belkin M, Hsu D, Ma S, and Mandal S, 2019, *Proceedings of the National Academy of Sciences*, Vol. 116(32), pp. 15,849–15,854.

The mechanism that explains this unexpected result has to do with an interesting and perhaps even magical transformation that occurs in models capable of additional parameterization beyond the interpolation threshold. Once a learner has sufficient parameters to interpolate (to sufficiently conform to) the training data, additional parameters lead to a state of **overparameterization**, in which the additional complexity enables higher levels of thinking and abstraction. In essence, an overparameterized learner is capable of learning higher-order concepts; in practice, this means it is capable of learning how to engineer features or learning how to learn. A significant jump in model complexity beyond the interpolation threshold is likely to lead to a significant leap in the way the algorithm approaches the problem, but of course, not every algorithm is capable of this leap.

Deep neural networks, which can add additional complexity endlessly and trivially via the addition of hidden nodes arranged in layers, are the ideal candidate for consuming big data. As you will learn in *Chapter 15, Making Use of Big Data*, a cleverly designed neural network can engineer its own features out of unstructured data such as images, text, or audio. Similarly, its designation as a universal function approximator implies that it can identify the best functional form to model any pattern it identifies in the data. Thus, we must once again revisit the early question of how exactly this doesn’t violate the principle of “no free lunch.” It would appear that for datasets of sufficient size, deep learning neural networks are the single best approach.

Putting a couple of practical issues aside—notably, the fact that most real-world projects reside in the small data regime and the fact that deep neural networks are computationally expensive and difficult to train—a key reason that deep learning doesn’t violate the “no free lunch” principle is based on the fact that once the neural network becomes large and substantially overparameterized, and assuming it has access to a sufficiently large and complex training dataset, it ceases to be a single learning *algorithm* and instead becomes a generalized learning *process*. If this seems like a distinction without a difference, perhaps a metaphor will help: rather than providing us with a free lunch, the process of deep learning provides an opportunity to teach the algorithm how to make its own lunch. Given the limited availability of truly big data and the limited applicability of deep learning to most business tasks, to produce the strongest models, it is still necessary for the machine learning practitioner to assist in the feature engineering process.

Feature engineering in practice

Depending on the project or circumstances, the practice of feature engineering may look very different. Some large, technology-focused companies employ one or more data engineers per data scientist, which allows machine learning practitioners to focus less on data preparation and more on model building and iteration. Certain projects may rely on very small or very massive quantities of data, which may preclude or necessitate the use of deep learning methods or automated feature engineering techniques. Even projects requiring little initial feature engineering effort may suffer from the so-called “last mile problem,” which describes the tendency for costs and complexity to be disproportionately high for the small distances to be traveled for the “last mile” of distribution. Relating this concept to feature engineering implies that even if most of the work is taken care of by other teams or automation, a surprising amount of effort may still be required for the final steps of preparing the data for the model.

It is likely that the bulk of real-world machine learning projects today require a substantial amount of feature engineering. Most companies have yet to achieve the level of analytics maturity at the organizational level needed to allow data scientists to focus solely on model building. Many companies and projects will never achieve this level due to their small size or limited scope. For many small-to-mid-sized companies and small-to-mid-sized projects, data scientists must take the lead on all aspects of the project from start to finish. Consequently, it is necessary for data scientists to understand the role of the feature engineer and prepare to perform this role if needed.

As stated previously, feature engineering is more art than science and requires as much imagination as it does programming skills. In a nutshell, the three main goals of feature engineering might be described as:

- Supplementing what data is already available with additional external sources of information
- Transforming the data to conform to the machine learning algorithm’s requirements and to assist the model with its learning
- Eliminating the noise while minimizing the loss of useful information—conversely, maximizing the use of available information

An overall mantra to keep in mind when practicing feature engineering is “be clever.” One should strive to be a clever, frugal data miner and try to think about the subtle insights that you might find in every single feature, working systematically, and avoiding letting any data go to waste. Applying this rule serves as a reminder of the requisite creativity and helps to inspire the competitive spirit needed to build the strongest-performing learners.

Although each project will require you to apply these skills in a unique way, experience will reveal certain patterns that emerge in many types of projects. The sections that follow, which provide seven “hints” for the art of feature engineering, are not intended to be exhaustive but, rather, provide a spark of inspiration on how to think creatively about making data more useful.



There has been an unfortunate dearth of feature engineering books on the market, until recently, when a number have been published. Two of the earliest books on this subject are Packt Publishing’s *Feature Engineering Made Easy* (Ozdemir & Susara, 2018) and O’Reilly’s *Feature Engineering for Machine Learning* (Zheng & Casari, 2018). The book *Feature Engineering and Selection* (Kuhn & Johnson, 2019) is also a standout and even has a free version, available on the web at: <http://www.feat.engineering>.

Hint 1: Brainstorm new features

The choice of topic for a new machine learning project is typically motivated by an unfulfilled need. It may be motivated by a desire for more profit, to save lives, or even simple curiosity, but in any case, the topic is almost surely not selected at random. Instead, it relates to an issue at the core of the company, or a topic held dear by the curious, both of which suggest a fundamental interest in the work. The company or individual pursuing the project is likely to already know a great deal about the subject and the important factors that contribute to the outcome of interest. With this domain experience and subject-matter expertise, the company, team, or individual that commissioned the project is likely to hold proprietary insights about the task that they alone can bring.

To capitalize on these insights, at the beginning of a machine learning project, just prior to feature engineering, it can be helpful to conduct a brainstorming session in which stakeholders are gathered and ideas are generated about the potential factors that are associated with the outcome of interest. During this process, it is important to avoid limiting yourself to what is readily available in existing datasets. Instead, consider the process of cause-and-effect at a more abstract level, imagining the various metaphorical “levers” that can be pulled in order to impact the outcome in a positive or negative direction. Be as thorough as possible and exhaust all ideas during this session. If you could have literally anything you wanted in the model, what would be most useful?

The culmination of a brainstorming session may be a **mind map**, which is a method of diagramming ideas around a central topic. Placing the outcome of interest at the center of the mind map, the various potential predictors radiate out from the central theme, as shown in the following example of a mind mapping session designing a model to predict heart disease mortality.

A mind map diagram may use a hierarchy to link associated concepts or group factors that are related in a similar data source:

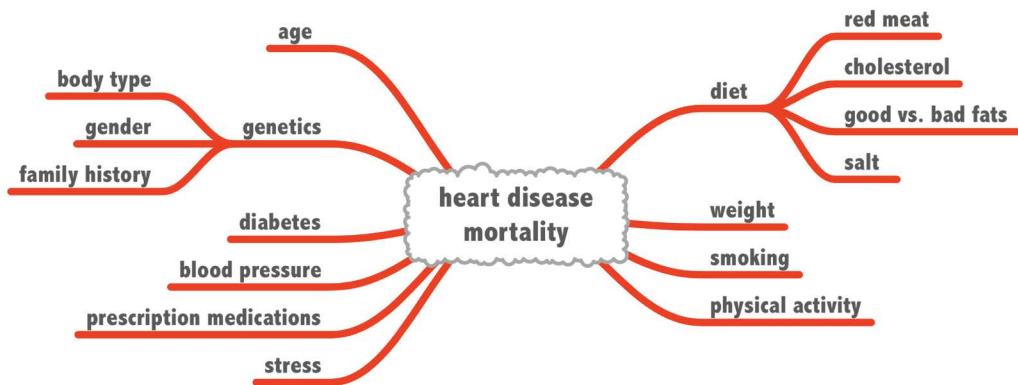


Figure 12.6: Mind maps can be useful to help imagine the factors that contribute to an outcome

While constructing the mind map, you may determine that some of the desired features are unavailable in the existing data sources. Perhaps the brainstorming group can help identify alternative sources of these data elements or find someone willing to help gather them. Alternatively, it may be possible to develop a **proxy measure** that effectively measures the same concept using a different method. For example, it may be impossible or practically infeasible to directly measure someone's diet, but it may be possible to use their social media activity as a proxy by counting the number of fast-food restaurants they follow. This is not perfect, but it is something, and is certainly better than nothing.

A mind mapping session can also help reveal potential interactions between features in which two or more factors have a disproportionate impact on the outcome; a joint effect may be greater (or lesser) than the sum of its parts. In the heart disease example, one might hypothesize that the combined effect of stress and obesity is substantially more likely to cause heart disease than the sum of their separate effects. Algorithms such as decision trees and neural networks can find these interaction effects automatically, but many others cannot, and in either case, it may benefit the learning process or result in a simpler, more interpretable model if these combinations are coded explicitly in the data.

Hint 2: Find insights hidden in text

One of the richest sources of hidden data and, therefore, one of the most fruitful areas for feature engineering is text data. Machine learning algorithms are generally not very good at realizing the full value of text data because they lack the external knowledge of semantic meaning that a human has gained over a lifetime of language use.

Of course, given a tremendous amount of text data, a computer may be able to learn the same thing, but this is not feasible for many projects, and would greatly add to a project's complexity. Furthermore, text data cannot be used as-is, as it suffers from the curse of dimensionality; each block of text is unique and, therefore, serves as a form of fingerprint linking the text to an outcome. If used in the learning process, the algorithm will severely overfit or ignore the text data altogether.



The curse of dimensionality applies to unstructured “big” data more generally as image and audio data are likewise difficult to use directly in machine learning models.

Chapter 15, Making Use of Big Data, covers some methods that allow these types of data sources to be used with traditional machine learning approaches.

The humans in charge of constructing features for the learning algorithm can add insight to the text data by coding reduced-dimensionality features derived from the interpretation of the text. In selecting a small number of categories, the implicit meaning is made explicit. For example, in a customer churn analysis, suppose a company has access to the public Twitter timeline for its customers. Each customer’s tweets are unique, but a human may be able to code them into three categories of positive, negative, and neutral. This is a simple form of **sentiment analysis**, which analyzes the emotion of language. Computer software, including some R packages, may be able to help automate this process using models or rules designed to understand simple semantics. In addition to sentiment analysis, it may be possible to categorize text data by topic; in the churn example, perhaps customers tweeting about customer service are more likely to switch to another company than customers tweeting about price.



There are many R packages that can perform sentiment analysis, some of which require subscriptions to paid services. To get started quickly and easily, check out the aptly named `SentimentAnalysis` and `RSentiment` packages, as well as the `Syuzhet` package. All of these can classify sentences as positive or negative with just a couple of lines of R code. For a deeper dive into text mining and sentiment analysis, see the book *Text Mining with R: A Tidy Approach*, 2017, Silge J and Robinson D, which is available on the web at <https://www.tidytextmining.com>. Additionally, see *Text Mining in Practice with R*, 2017, Kwartler T.

Beyond coding the overt meaning of text, one of the subtle arts of feature engineering involves finding the covert insights hidden in the text data. In particular, there may be useful information encoded in the text that is not related to the direct interpretation of the text, but it appears in the text coincidentally or accidentally, like a “tell” in the game of poker—a micro-expression that reveals the player’s secret intentions.

Hidden text data may help reveal aspects of a person's identity, such as age, gender, career level, location, wealth, or socioeconomic status. Some examples include:

- Names and salutations such as Mr. and Mrs., or Jr. and Sr., traditional and modern names, male and female names, or names associated with wealth
- Job titles and categories such as CEO, president, assistant, senior, or director
- Geographic and spatial codes such as postal codes, building floor numbers, foreign and domestic regions, first-class tickets, PO boxes, and similar
- Linguistic markers such as slang or other expressions that may reveal pertinent aspects of identities

To begin searching for these types of hidden insights, keep the outcome of interest in mind while systematically reviewing the text data. Read as many of the texts as possible while thinking about any way in which the text might reveal a subtle clue that could impact the outcome. When a pattern emerges, construct a feature based on the insight. For instance, if the text data commonly includes job titles, create rules to classify the jobs into career levels such as entry-level, mid-career, and executive. These career levels could then be used to predict outcomes such as loan default or churn likelihood.

Hint 3: Transform numeric ranges

Certain learning algorithms are more capable than others of learning from numeric data. Among algorithms that can utilize numeric data at all, some are better at learning the important cut points in the range of numeric values or are better at handling severely skewed data. Even a method like decision trees, which is certainly apt at using numeric features, has a tendency to overfit on numeric data and, thus, may benefit from a transformation that reduces the numeric range into a smaller number of potential cut points. Other methods like regression and neural networks may benefit from nonlinear transformations of numeric data, such as log scaling, normalization, and step functions.

Many of these methods have been covered and applied in prior chapters. For example, in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*, we considered the technique of discretization (also known as “binning” or “bucketing”) as a means of transforming numeric data into categorical data so that it could be used by the naive Bayes algorithm. This technique is also sometimes useful for learners that can handle numeric data natively, as it can help clarify a decision boundary.

The following figure illustrates this process for a hypothetical model predicting heart disease, using a numeric age predictor. On the left, we see that as the numeric age increases, the darker the color becomes, indicating a greater prevalence of heart disease with increasing age. Despite this seemingly clear trend, a decision tree model may struggle to identify an appropriate cut point and it may do so arbitrarily, or it may choose numerous small cut points; both of these are likely to be overfitted to the training data. Instead of leaving this choice to the model, it may be better to use *a priori* knowledge to create predefined groups for “young” and “old” patients. Although this loses some of the nuances of the true underlying gradient, it may help the model generalize better to future data by trading the decision tree’s “high variance” approach for a “high bias” approach of theory-driven discretization.



Figure 12.7: Discretization and other numeric transformations can help learners identify patterns more easily

In general, for datasets containing numeric features, it may be worth exploring each feature systematically, while also considering the learning algorithm’s approach to numeric data, to determine whether a transformation is necessary. Apply any domain or subject-matter expertise you may have to inform the creation of bins, buckets, step points, or nonlinear transformations in the final version of the feature. Even though many algorithms are capable of handling numeric data without recoding or transformation, additional human intelligence may help guide the model to a better overall fit.

Hint 4: Observe neighbors’ behavior

One of the lesser-known methods of surfacing hidden insights during feature engineering is to apply the common knowledge that “birds of a feather flock together.” We applied this principle to prediction in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, but it is also a useful mindset for identifying useful predictors. The idea hinges on the fact that there may be explicit or implicit groupings across the dataset’s rows, and there may be insights found by examining how one example relates to the others in its neighborhood or grouping.

An example of an explicit grouping found often in real-world data is households. Many datasets include not only rows based on individuals but also a household identifier, which allows you to link rows into household groups and, thus, create new features based on the groups’ compositions.

For instance, knowing that someone is in a household may provide an indication of marital status and the number of children or dependents, even if these features were not included in the original individual-level dataset. Simply counting or aggregating some of the group's features can result in highly useful predictors.

From here, it is also possible to share information among records in the groups. For instance, knowing one spouse's income is helpful, but knowing both provides a better indication of the total available income. Measures of variance within a group can also be enlightening. There may be aspects of households that provide a bonus effect if the partners match or disagree on certain attributes; for example, if both partners report satisfaction with a particular telephone company, they may be especially loyal compared to households where only one member is satisfied.

These principles also apply to less obvious but still explicit groupings, like postal codes or geographic regions. By collecting the rows falling into the group, one can count, sum, average, take the maximum or minimum value, or examine the diversity within the group to construct new and potentially useful predictors. Groups with more or less agreement or diversity may be more or less robust to certain outcomes.

There may be value in identifying implicit groupings as well—that is, a grouping not directly coded in the dataset. Clustering methods, such as those described in *Chapter 9, Finding Groups of Data – Clustering with k-means*, are one potential method of finding these types of groupings, and the resulting clusters can be used directly as a predictor in the model. For example, in a churn project, using clusters as features for the model may reveal that some clusters are more likely to churn than others. This may imply that churn is related to the cluster's underlying demographics, or that churn is somewhat contagious among cluster members.

In other words, if birds of a feather flock together, it makes sense to borrow leading indicators from the experiences of similar neighbors—they may have a similar reaction to some external factor or may directly influence one another. Implicit groups that exhibit rare or unique traits may be interesting in themselves; perhaps some are the bellwether or “canary in the coal mine”—trend-setters that respond to change earlier than other groups. Observing their behavior and coding these groups explicitly into the model may improve the model’s predictive ability.



If you do use information from neighbors (or from related rows, as described in the next section), beware of the problem of data leakage, which was described in *Chapter 11, Being Successful with Machine Learning*. Be sure to only engineer features using information that will be available at the time of prediction when the model is deployed. For example, it would be unwise to use both spouses' data for a credit scoring model if only one household member completes the loan application and the other spouse's data is added after the loan is approved.

Hint 5: Utilize related rows

The practice of utilizing “follow the leader” behavior as hinted in the previous section can be especially powerful given the related rows of time series datasets, where the same attribute is measured repeatedly at different points in time. Data that contains repeated measures offers many such additional opportunities to construct useful predictors. Whereas the previous section considered grouping related data *across* the unit of analysis, the current section considers the value of grouping related observations *within* the unit of analysis. Essentially, by observing the same units of analysis repeatedly, we can examine their prior trends and make better predictions of the future.

Revisiting the hypothetical churn example, suppose we have access to the past 24 months of data from subscribers to an online video streaming service. The unit of observation is the customer-month (one row per customer per month), while our unit of analysis is the customer. Our goal is to predict which customers are most likely to churn so that we might intervene. To construct a dataset for machine learning, we must collect the units of observation and aggregate them into one row per customer. Here is where feature engineering is especially needed. In the process of “rolling up” the historical data into a single row for analysis, we can construct features that examine trends and loyalty, asking questions such as:

- Is the customer’s average monthly activity greater than or less than their peers?
- What is the customer’s monthly activity over time? Is it up, down, or stable?
- How frequent is their activity? Are they loyal? Is their loyalty stable across months?
- How consistent is the customer’s behavior? Does the behavior vary a lot from month to month?

If you are familiar with basic calculus, it may help to reflect on the concept of the first and second derivative, as both can be useful features in a time series model. The first derivative here refers to the velocity of the behavior—that is, the behavior count over a unit of time. For example, we may compute the number of dollars spent per month on the streaming service, or the number of television shows and movies streamed per month. These are useful predictors alone, but they can be made even more useful in the context of the second derivative, which is the acceleration (or deceleration) of the behavior. The acceleration is the change in velocity over time, such as the change in monthly spending or the change in the shows streamed per month. High-velocity customers with high spending and usage might be less likely to churn, but a rapid deceleration (that is, a large reduction in usage or spending) from these same customers might indicate an impending churn.

In addition to velocity and acceleration, measures of consistency, reliability, and variability can be constructed to further enhance predictive ability. A very consistent behavior that suddenly changes may be more concerning than a wildly varying behavior that changes similarly. Calculating the proportion of recent months with a purchase, or with spending or behavior meeting a given threshold, provides a simple loyalty metric, but more sophisticated measures using variance are also possible.

Hint 6: Decompose time series

The repeated measures time series data described in the previous section, with multiple related rows per unit of analysis, is said to be in the **long format**. This contrasts with the type of data required for most R-based machine learning methods. Unless a learning algorithm is designed to understand the related rows of repeated measures data, it will require time series data to be specified in the **wide format**, which transposes the repeated rows of data into repeated columns. For example, if a weight measurement is recorded monthly for 3 months for 1,000 patients, the long-format dataset will have $3 * 1,000 = 3,000$ rows and 3 columns (patient identifier, month, and weight). As depicted in *Figure 12.8*, the same dataset in wide format would contain only 1,000 rows but 4 columns: 1 column for the patient identifier, and 3 columns for the monthly weight readings:

Long Format

patient_id	month	weight
1	1	78.6 kg
1	2	77.9 kg
1	3	78.3 kg
2	1	56.7 kg
2	2	55.9 kg
2	3	55.3 kg
:	:	:
1000	1	65.8 kg
1000	2	64.9 kg
1000	3	64.1 kg

**Wide Format**

patient_id	weight_m1	weight_m2	weight_m3
1	78.6 kg	77.9 kg	78.3 kg
2	56.7 kg	55.9 kg	55.3 kg
:	:	:	:
1000	65.8 kg	64.9 kg	64.1 kg

Figure 12.8: Most machine learning models require long-format time series data to be transformed into a wide format

To construct a wide format dataset, one must first determine how much history will be useful for prediction. The more history that is needed, the more columns that will need to be added to the wide dataset. For example, if we wanted to forecast a customer’s energy usage 1 month into the future, we may decide to use their prior 12 months of energy use as predictors so that a full year of seasonality would be covered. Therefore, to build a model forecasting energy usage in June 2023, we might create 12 predictor columns measuring energy use in May 2023, April 2023, March 2023, and so on, for each of the 12 months prior to June 2023. A 13th column would be the target or dependent variable, recording the actual energy usage in June 2023. Note that a model trained upon this dataset would learn to predict energy use in June 2023 based on data in the months from June 2022 to May 2023, but it would not be able to predict other future months because the target and predictors are linked to specific months.

Instead, a better approach is to construct **lagged variables**, which are computed relative to the target month. The lagged variables are essentially measures that are delayed in time to be carried forward to a later, more recent row in the dataset. A model using lagged variables can be retrained on a rolling, monthly basis as additional months of data become available over time. Rather than having column names like `energy_june2023` and `energy_may2023`, the resulting dataset will have names that indicate the relative nature of the measurements, such as `energy_lag0`, `energy_lag1`, and `energy_lag2`, which indicate the energy use in the current month, the prior month, and 2 months ago. This model will always be applied to the most recent data to predict the forthcoming time period.

Figure 12.9 visualizes this approach. Each month, a model is trained on the past 13 months of data; the most recent month is used for the target or dependent variable (denoted as DV) while the earlier 12 months are used as lagged predictors. The model can then be used to predict the future month, which has not yet been observed. Each successive month following the first shifts the rolling window 1 month forward, such that data older than 13 months is unused in the model. A model trained using data constructed in this way does not learn the relationship between specific calendar months, as was the case with the non-lagged variables; rather, it learns how prior behavior relates to future behavior, regardless of the calendar month.

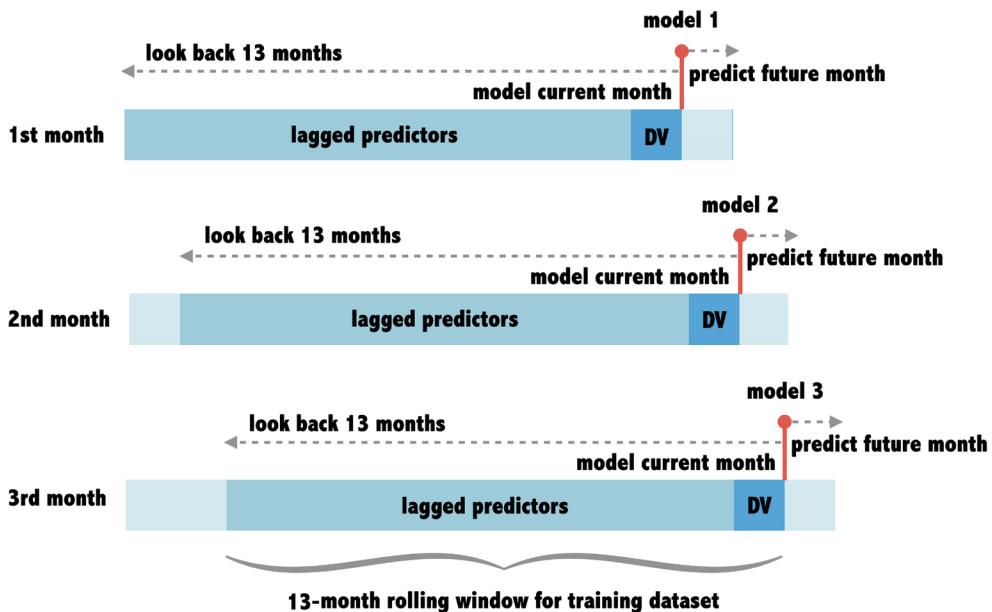


Figure 12.9: Constructing lagged predictors is one method to model time series data

A problem with this approach, however, is that this method has disregarded calendar time, yet certain calendar months may have an important impact on the target variable. For example, energy use may be higher in winter and summer than in spring and fall, and thus, it would be beneficial for the model to know not only the relationship between past and future behavior but also to gain a sense of seasonal effects, or other patterns broader than the local patterns, within the rows related to the unit of analysis.

One might imagine that the value of the target to be predicted is composed of three sources of variation, which we would like to decompose into features for the model:

1. There is the local or internal variation, which is based on the attributes unique to the unit of analysis. In the example of forecasting energy demand, the local variation may be related to the size and construction of the household, the residents' energy needs, where the house is located, and so on.
2. There may be broader global trends, such as fuel prices or weather patterns, that affect the energy use of most households.
3. There may be seasonal effects, independent of the local and global effects, that explain changes in the target. This is not limited to the annual weather patterns mentioned before, but any cyclical or predictable pattern can be considered a seasonal effect.

Some specific examples relevant to the energy forecasting project may include higher or lower demand on:

- Different days of the week, particularly weekdays versus weekends
- Religious or government holidays
- Traditional school or business vacation periods
- Mass gatherings such as sporting events, concerts, and elections

If local, global, and seasonal features can be incorporated into the training dataset as predictors, the model can learn their effect on the outcome. The challenge thereafter is twofold: subject-matter knowledge or data exploration is required to identify the important seasonal factors, and there must be ample training data for the target to be observed in each of the included seasons. The latter implies that the training data should be composed of more than a single month cross-section of time; lacking this, the learning algorithm will obviously be unable to discover the relationship between the seasons and the target!

Though it would seem to follow that we should revert to the original long-format data, this is actually not the case. In fact, the wide data with lagged variables from each month can be stacked in a single unified dataset with multiple rows per unit of analysis. Each row indicates an individual at a particular moment in time, with a target variable measuring the outcome at that moment, and a wide set of columns that have been constructed as lagged variables for periods of time prior to the target. Additional columns can also be added to further widen the matrix and decompose the various components of time variance, such as indicators for seasons, days of the week, and holidays; these columns will indicate whether the given row falls within one of these periods of interest.

The figure that follows depicts a hypothetical dataset using this approach. Each household (denoted by the `household_id` column) can appear repeatedly with different values of the target (`energy_use`) and predictors (`season`, `holiday_month`, `energy_lag1`, and so on). Note that the lag variables are missing (as indicated by the `NA` values) for the first few rows of the dataset, which means that these rows cannot be used for training or prediction. The remaining rows, however, can be used with any machine learning method capable of numeric prediction, and the trained model will readily forecast next month's energy use given the row of data for the current month.

<code>household_id</code>	<code>billing_period</code>	<code>energy_use</code>	<code>season</code>	<code>holiday_month</code>	<code>energy_lag1</code>	<code>energy_lag2</code>	<code>energy_lag3</code>
123	December 2022	965	Winter	1	NA	NA	NA
123	January 2023	1034	Winter	0	965	NA	NA
123	February 2023	933	Winter	0	1034	965	NA
123	March 2023	710	Spring	0	933	1034	965
123	April 2023	653	Spring	0	710	933	1034
123	May 2023	545	Spring	0	653	710	933
123	June 2023	748	Summer	0	545	653	710
456	December 2022	899	Winter	1	NA	NA	NA
456	January 2023	932	Winter	0	899	NA	NA
456	February 2023	917	Winter	0	932	899	NA
...

Figure 12.10: Datasets including historical data may include both seasonal effects and lagged predictors

Before rushing into modeling time series data, it is crucial to understand an important caveat about the data preparation methods described here: because the rows from repeated observations from the same unit of analysis are related to one another, including them in the training data violates the assumption of independent observations for methods like regression. While models built upon such data may still be useful, other methods for formal time series modeling may be more appropriate, and it is best to consider the methods described here as a workaround to perform forecasting with the machine learning methods previously covered. Linear mixed models and recurrent neural networks are two potential approaches that can handle this type of data natively, although both methods are outside the scope of this book.

The `lme4` package is used to build mixed models in R, but it would be unwise to jump in without understanding the statistical underpinnings of these types of models; they are a significant step up in complexity over traditional regression modeling. The book *Linear Mixed-Effects Models Using R* (Gałecki & Burzykowski, 2013) provides the theoretical background needed to build this type of model. To build recurrent neural networks, R may not be the right tool for the job, as specialized tools exist for this purpose. However, the `rnn` package can build simple RNN models for time series forecasting.



Hint 7: Append external data

Unlike the teaching examples in this book, when a machine learning project begins in the real world, a dataset cannot simply be downloaded from the internet with prebuilt features and examples describing the topic of interest. It is unfortunate how many deeply interesting projects are killed before they begin for this simple reason. Businesses hoping to predict customer churn realize they have no historical data from which a model can be built; students hoping to optimize food distribution in poverty-stricken areas are limited by the scarce amounts of data from these areas; and countless projects that might increase profits or change the world for the better are stunted before they start. What begins as excitement around a machine learning project soon fizzles out due to the lack of data.

Rather than ending with discouragement, it is better to channel this energy into an effort to create the necessary data from scratch. This may mean dialing colleagues on the telephone or firing off a series of email messages to connect with those that can grant access to databases containing relevant pieces of data. It may also require rolling up your sleeves and getting your hands dirty. After all, we live in the so-called big data era where data is not only plentiful but easily recorded, with assistance from electronic sensors and automated data entry tools.

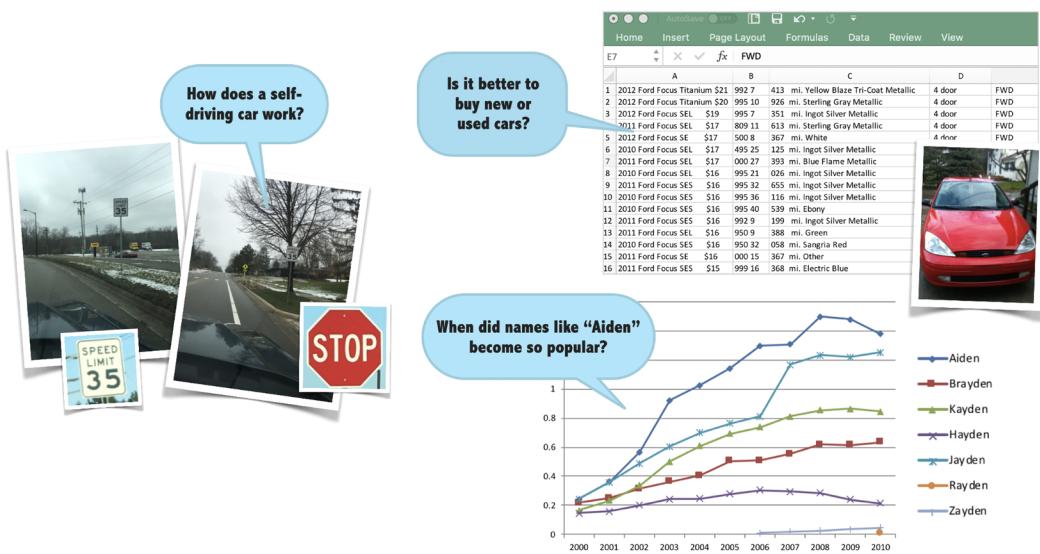


Figure 12.11: Little effort is often sufficient to generate datasets useful for machine learning

In the worst case, an investment of time, effort, and imagination can build useful datasets from nothing. Typically, this is easier than one might think. The previous figure illustrates several cases in which I created datasets to satisfy my own curiosity.

Fascinated by autonomous vehicles, I drove around my neighborhood and took pictures of road signs to build a stop sign classification algorithm. To predict used car prices, I copied and pasted hundreds of listings from used car websites. And, to understand exactly when and why names rhyming with “Aiden” became so popular in the United States, I gathered dozens of years of data from the Social Security baby name database. None of these projects required more than a few hours of effort, but enrolling friends, colleagues, or internet forums as a form of crowdsourcing the effort or even paying for data entry assistance could have parallelized the task and helped my database grow larger or faster. Paid services like Amazon Mechanical Turk (<https://www.mturk.com>) provide an affordable means of distributing large and tedious data entry or collection tasks.

To further enrich existing datasets, there is often the potential to append additional features from external sources. This is especially true when the main dataset of interest includes geographic identifiers such as postal codes, as many publicly available databases measure attributes for these regions. Of course, a postal code-level dataset will not reveal a specific individual’s exact characteristics; however, it may provide insight into whether the average person in the area is wealthier, healthier, younger, or more likely to have kids, among numerous other factors that may help improve the quality of a predictive model. These types of data can be readily found on many governmental agency websites and downloaded at no charge; simply merge them onto the main dataset for additional possible predictors.

Lastly, many social media companies and data aggregator services like Facebook, Zillow, and LinkedIn provide free access to limited portions of their data. Zillow, for example, provides home value estimates for postal code regions. In some cases, these companies or other vendors may sell access to these datasets, which can be a powerful means of augmenting a predictive model. In addition to the financial cost of such acquisitions, they often pose a significant challenge in terms of **record linkage**, which involves matching entities across datasets that share no common unique identifier. Solving this problem involves building a **crosswalk** table, which maps each row in one source to the corresponding row in the other source. For instance, the crosswalk may link a person identified by a customer identification number in the main dataset to a unique website URL in an external social media dataset. Although there are R packages such as RecordLinkage that can help perform such matching across sources, these rely on heuristics that may not perform as well as human intelligence and require significant computational expense, particularly for large databases. In general, it is safe to assume that record linkage is often costly from human resource and computational expense perspectives.



When considering whether to acquire external data, be sure to research the source's terms of use, as well as your region's laws and organizational rules around using such sources. Some jurisdictions are stricter than others, and many rules are becoming stricter over time, so it is important to keep up to date on the legality and liability associated with outside data.

Given the work involved in advanced data preparation, R itself has evolved to keep up with the new demands. Historically, R was notorious for struggling with very large and complex datasets, but over time, new packages have been developed to address these shortcomings and make it easier to perform the types of operations described so far in this chapter. In the remainder of this chapter, you will learn about these packages, which modernize the R syntax for real-world data challenges.

Exploring R's tidyverse

A new approach has rapidly taken shape as the dominant paradigm for working with data in R. Championed by Hadley Wickham—the mind behind many of the packages that drove much of R's initial surge in popularity—this new wave is now backed by a much larger team at Posit (formerly known as RStudio). The company's user-friendly RStudio Desktop application integrates nicely into this new ecosystem, known as the **tidyverse**, because it provides a universe of packages devoted to tidy data. The entire suite of tidyverse packages can be installed with the `install.packages("tidyverse")` command.

A growing number of resources are available online to learn more about the tidyverse, starting with its homepage at <https://www.tidyverse.org>. Here, you can learn about the various packages included in the set, a few of which will be described in this chapter. Additionally, the book *R for Data Science* by Hadley Wickham and Garrett Grolemund is available freely online at <https://r4ds.hadley.nz> and illustrates how the tidyverse's self-proclaimed “opinionated” approach simplifies data science projects.



I am often asked the question of how R compares to Python for data science and machine learning. RStudio and the tidyverse are perhaps R's greatest asset and point of distinction. There is arguably no easier way to begin a data science journey. Once you've learned the “tidy” way of doing data analysis, you are likely to wish the tidyverse functionality existed everywhere!

Making tidy table structures with tibbles

Whereas the data frame is the center of the base R universe, the data structure at the heart of the tidyverse is found in the `tibble` package (<https://tibble.tidyverse.org>), the name of which is a pun on the word “table” as well as a nod to the infamous “tribble” in *Star Trek* lore. A **tibble** acts almost exactly like a data frame but includes additional modern functionality for convenience and simplicity. Tibbles can be used almost everywhere a data frame can be used. Detailed information about tibbles can be found by typing the command `vignette("tibble")` in R.

Most of the time, using tibbles will be transparent and seamless, as tibbles can pass as a data frame in most R packages. However, in the rare case where you need to convert a tibble to a data frame, use the `as.data.frame()` function. To go in the other direction and convert a data frame in to a tibble, use the `as_tibble()` function. Here, we’ll create a tibble from the Titanic dataset first introduced in the previous chapter:

```
> library(tibble) # not necessary if tidyverse is already loaded
> titanic_csv <- read.csv("titanic_train.csv")
> titanic_tbl <- as_tibble(titanic_csv)
```

Typing the name of this object demonstrates the tibble’s cleaner and more informative output than a standard data frame:

```
> titanic_tbl
# A tibble: 891 x 12
  PassengerId Survived Pclass Name          Sex   Age SibSp Parch Ticket      Fare Cabin Embarked
  <int>     <int> <int> <chr>        <dbl> <dbl> <int> <int> <chr>    <dbl> <chr> <chr>
1       1         0     3 Braund, Mr. Owen Harris   male   22     1     0 A/5 21171 7.25   ""      S
2       2         1     1 Cumings, Mrs. John Bradley (F... female  38     1     0 PC 17599 71.3   "C85"   C
3       3         1     3 Heikkinen, Miss. Laina  female  26     0     0 STON/O2. 3... 7.92   ""      S
4       4         1     1 Futrelle, Mrs. Jacques Heath ... female  35     1     0 113803 53.1   "C12... S
5       5         0     3 Allen, Mr. William Henry male   35     0     0 373450 8.05   ""      S
6       6         0     3 Moran, Mr. James    male  NA     0     0 330877 8.46   ""      Q
7       7         0     1 McCarthy, Mr. Timothy J male   54     0     0 17463 51.9   "E46"   S
8       8         0     3 Palsson, Master. Gosta Leonard male   2      3     1 349909 21.1   ""      S
9       9         1     3 Johnson, Mrs. Oscar W (Elisab... female  27     0     2 347742 11.1   ""      S
10      10        1     2 Nassar, Mrs. Nicholas (Adele ... female  14     1     0 237736 30.1   ""      C
# ... with 881 more rows
```

Figure 12.12: Displaying a tibble object results in more informative output than a standard data frame

It is important to note the distinctions between tibbles and data frames, as the tidyverse will automatically create a tibble object for many of its operations. Overall, you are likely to find that tibbles are faster and easier to work with than data frames. They generally make smarter assumptions about the data, which means you will spend less time redoing R’s work—like recoding strings as factors or vice versa.

Indeed, one simple distinction between tibbles and data frames is that a tibble never assumes `stringsAsFactors = TRUE`, which was the default behavior in base R until relatively recently with the release of R version 4.0. As described in previous chapters, R's `stringsAsFactors` setting sometimes led to confusion or programming bugs when character columns were automatically converted in to factors by default. Another distinction between tibbles and data frames is that, as long as the name is surrounded by the backtick (`) character, a tibble can use non-standard column names like `my var` that violate base R's object naming rules. Other benefits of tibbles are unlocked by complementary tidyverse packages, as described in the sections that follow.

Reading rectangular files faster with `readr` and `readxl`

Nearly every chapter so far has used the `read.csv()` function to load data into R data frames. Although we could convert these data frames in to tibbles, there is a faster and more direct path to get data into the tibble format. The tidyverse includes the `readr` package (<https://readr.tidyverse.org>) for loading tabular data. This is described in the data import chapter in *R for Data Science* at <https://r4ds.hadley.nz/data-import.html>, but the basic functionality is simple.

The `readr` package provides a `read_csv()` function that loads data from CSV files much like base R's `read.csv()` function. A key difference, aside from the subtle difference in their function names, is that the tidyverse's version is much speedier—and not merely because it automatically converts the data into a tibble. It is about 10x faster at reading data according to the package authors. It is also smarter about the format of the columns to be loaded. For example, it has the capability to handle numbers with currency characters, parse date columns, and is better at handling international data.

To create a tibble from a CSV file, simply use the `read_csv()` function as follows:

```
> library(readr) # not necessary if tidyverse is already loaded  
> titanic_train <- read_csv("titanic_train.csv")
```

This will use the default parsing settings, which attempt to infer the correct data type (that is, character or numeric) for each column. The column specification will be displayed in the R output upon completion of the file read. The inferred data types may be overridden by providing the correct column specifications via a `col()` function call passed to the `read_csv()` function. For more information on the syntax, view the documentation using the `vignette("readr")` command.

The `readxl` package (<https://readxl.tidyverse.org>) provides a method to read data directly from the Microsoft Excel spreadsheet format. To create a tibble from an XLSX file, simply use the `read_excel()` function as follows:

```
> library(readxl)
> titanic_train <- read_excel("titanic_train.xlsx")
```

Alternatively, as first introduced in *Chapter 2, Managing and Understanding Data*, the RStudio desktop application can write the data import code for you. In the upper-right of the interface, under the **Environment** tab, there is an **Import Dataset** button. This menu reveals a list of data import options, including plaintext formats like CSV files (using base R or the `readr` package), as well as Excel and the SPSS, SAS, and Stata formats created by other statistical computing software tools. Using the **From Text (readr)** option reveals the following graphical interface, allowing the import process to be easily customized:

The screenshot shows the 'Import Text Data' dialog box. At the top, it says 'File/URL:' with a text input field containing the path `~/Documents/Machine Learning with R/Chapter 12/titanic_train.csv`. Below this is a 'Data Preview:' section showing the first 10 rows of the titanic_train.csv dataset. The columns are: PassengerId (double), Survived (double), Pclass (double), Name (character), Sex (character), Age (double), and SibSp (double). The preview data includes entries for passengers 1 through 10, showing details like name, sex, age, and survival status. Below the preview is a note 'Previewing first 50 entries.' On the left, 'Import Options:' include 'Name:' set to `titanic_train`, 'Skip:' set to 0, and checkboxes for 'First Row as Names', 'Trim Spaces', 'Open Data Viewer', 'Delimiter:' (set to Comma), 'Escape:' (set to None), 'Quotes:' (set to Default), 'Comment:' (set to Default), 'Locale:' (button to Configure...), and 'NA:' (set to Default). On the right, a 'Code Preview:' window shows the R code generated for the import: `library(readr)`, `titanic_train <- read_csv("titanic_train.csv")`, and `View(titanic_train)`. At the bottom are 'Import' and 'Cancel' buttons.

Figure 12.13: RStudio's Import Dataset feature automatically writes R code to easily import a variety of data formats

The interface displays a preview of the data that updates as the import parameters are customized. The default column data types can be customized by clicking on the drop-down menu in the column header, and the code preview in the lower-right will update accordingly.

Clicking the **Import** button will immediately execute the code, but a better practice is to copy and paste the code into your R source code file so that the import process can be easily run again in the future.

Preparing and piping data with `dplyr`

The `dplyr` package (<https://dplyr.tidyverse.org>) provides the infrastructure for the tidyverse, as it includes the basic functionality that allows data to be transformed and manipulated. It also provides a straightforward way to begin working with larger datasets in R. Though there are other packages that have greater raw speed or are capable of handling even more massive datasets, `dplyr` is still quite capable and a good first step to take if you run into speed or memory limitations with base R.

When used with tibble objects, `dplyr` unlocks some impressive functionality:

- Because `dplyr` focuses on data frames rather than vectors, new operators are introduced that allow common data transformations to be performed with much less code while remaining highly readable.
- The package makes reasonable assumptions about data frames, which optimizes your effort as well as memory use. If possible, it avoids making copies of data by pointing to the original value instead.
- Key portions of the code are written in C++, which, according to the authors, yields a 20x to 1,000x performance increase over base R for many operations.
- R data frames are limited by available memory. With `dplyr`, tibbles can be linked transparently to disk-based databases exceeding what can be stored in memory.

The `dplyr` grammar of working with data becomes second nature after the initial learning curve has been passed. There are five key verbs in the grammar, which perform many of the most common transformations to data tables. Beginning with a tibble, one may choose to:

- `filter()` rows of data by values of the columns
- `select()` columns of data by name
- `arrange()` rows of data by sorting the values
- `mutate()` columns into new columns by transforming the values
- `summarize()` rows of data by aggregating values into a summary

These five dplyr verbs are brought together in sequences using a **pipe operator**, which is natively supported in R as of version 4.1 or later. Represented by the `|>` symbols, which vaguely resembles an arrowhead pointing to the right, the pipe operator “pipes” data by moving it from one function to another. The use of pipes allows you to create powerful chains of functions to sequentially process datasets.



In versions of R prior to 4.1.0 update, the pipe operator was denoted by the `%>%` character sequence and required the `magrittr` package. The differences between the old and new pipe functionality are relatively minor, but as a native operator the new pipe may have a small speed advantage. For a shortcut to typing the pipe operator, the RStudio Desktop IDE, the key combination *ctrl + shift + m* will insert the character sequence. Note that for this shortcut to produce the updated pipe, you may need to change the setting to “**Use the native pipe operator, |>**” in the RStudio “**Global Options**” menu under the “**Code**” heading.

After loading the package with the `library(dplyr)` command, data transformations begin with a tibble being piped into one of the package’s verbs. For example, one might `filter()` rows of the Titanic dataset to limit rows to women:

```
> titanic_train |> filter(Sex == "female")
```

Similarly, one might `select()` only the name, sex, and age columns:

```
> titanic_train |> select(Name, Sex, Age)
```

Where dplyr starts to shine is through its ability to chain together verbs in a sequence with pipes. For example, we can combine the prior two verbs, sort alphabetically using the verb `arrange()`, and save the output to a tibble as follows:

```
> titanic_women <- titanic_train |>
  filter(Sex == "female") |>
  select(Name, Sex, Age) |>
  arrange(Name)
```

Although this may not seem like a revelation just yet, when combined with the `mutate()` verb, we can perform complex data transformations with simpler, more readable code than in the base R language. We will see several examples of `mutate()` later on, but for now, the important thing to remember is that it is used to create new columns in the tibble. For example, we might create a binary `elderly` feature that indicates whether a passenger is at least 65 years old.

This uses the `dplyr` package's `if_else()` function to assign a value of `1` if the passenger is elderly, and `0` otherwise:

```
> titanic_train |>  
  mutate(elderly = if_else(Age >= 65, 1, 0))
```

By separating the statements by commas, multiple columns can be created within a single `mutate()` statement. This is demonstrated here to create an additional `child` feature that indicates whether the passenger is less than 18 years old:

```
> titanic_train |>  
  mutate(  
    elderly = if_else(Age >= 65, 1, 0),  
    child = if_else(Age < 18, 1, 0)  
)
```

The remaining `dplyr` verb, `summarize()`, allows us to create aggregated or summarized metrics by grouping rows in the tibble. For example, suppose we would like to compute the survival rate by age or sex. We'll begin with sex, as it is the easier of the two cases. We simply pipe the data into the `group_by(Sex)` function to create the male and female groups, then follow this with a `summarize()` statement to create a `survival_rate` feature that computes the average survival by group:

```
> titanic_train |>  
  group_by(Sex) |>  
  summarize(survival_rate = mean(Survived))
```

```
# A tibble: 2 × 2  
  Sex     survival_rate  
  <chr>      <dbl>  
 1 female     0.742  
 2 male       0.189
```

As shown in the output, females were substantially more likely to survive than males.

To compute survival by age, things are slightly more complicated due to the missing age values. We'll need to filter out these rows and use the `group_by()` function to compare children (less than 18 years old) to adults as follows:

```
> titanic_train |>  
  filter(!is.na(Age)) |>
```

```
mutate(child = if_else(Age < 18, 1, 0)) |>
  group_by(child) |>
  summarize(survival_rate = mean(Survived))
```

```
# A tibble: 2 x 2
  child survival_rate
  <dbl>        <dbl>
1     0         0.381
2     1         0.540
```

The results suggest that children were about 40% more likely to survive than adults. When combined with the comparison between males and females, this provides strong evidence of the hypothesized “women and children first” policy for evacuation of the sinking ship.



Because summary statistics by group can be computed using other methods in base R (including the `ave()` and `aggregate()` functions described in previous chapters), it is important to note that the `summarize()` command is also capable of much more than this. In particular, one might use it for the feature engineering hints described earlier in this chapter, such as observing neighbors’ behavior, utilizing related rows, and decomposing time series. All three of these cases involve `group_by()` options like households, zip codes, or units of time. Using `dplyr` to perform the aggregation for these data preparation operations is much easier than attempting to do so in base R.

To put together what we’ve learned so far and provide one more example using pipes, let’s build a decision tree model of the Titanic dataset. We’ll `filter()` missing age values, use `mutate()` to create a new `AgeGroup` feature, and `select()` only the columns of interest for the decision tree model. The resulting dataset is piped to the `rpart()` decision tree algorithm, which illustrates the ability to pipe data to functions outside of the tidyverse:

```
> library(rpart)
> m_titanic <- titanic_train |>
  filter(!is.na(Age)) |>
  mutate(AgeGroup = if_else(Age < 18, "Child", "Adult")) |>
  select(Survived, Pclass, Sex, AgeGroup) |>
  rpart(formula = Survived ~ ., data = _)
```

Note that the series of steps reads almost like plain-language pseudocode. It is also worth noting the arguments within the `rpart()` function call. The `formula = Survived ~ .` argument uses R's formula interface to model survival as a function of all predictors; the dot here represents the other features in the dataset not explicitly listed. The `data = _` argument uses the underscore (`_`) as a placeholder to represent the data being fed to `rpart()` by the pipe. The underscore can be used in this way to indicate the function parameter to which the data should be piped.

This is usually unnecessary for `dplyr`'s built-in functions, because they look for the piped data as the first parameter by default, but functions outside the `tidyverse` may require the pipe to target a specific function parameter in this way.



It is important to note that the underscore placeholder character is new as of R version 4.2 and will not work in prior versions! In older code that uses the `magrittr` package, the dot character (`.`) was used as the placeholder.

For fun, we can visualize the resulting decision tree, which shows that women and children are more likely to survive than adults, men, and those in third passenger class:

```
> library(rpart.plot)
> rpart.plot(m_titanic)
```

This produces the following decision tree diagram:

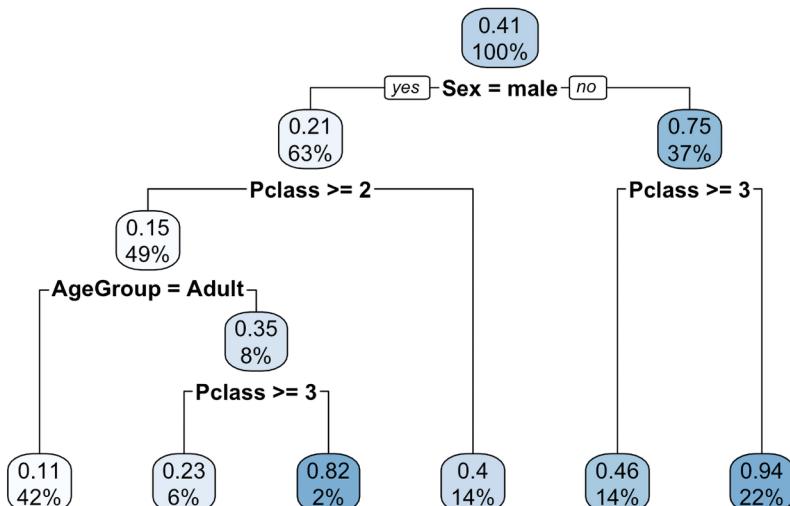


Figure 12.14: A decision tree predicting Titanic survival, which was built using a series of pipes

These are just a few small examples of how sequences of dplyr commands can make complex data manipulation tasks simpler. This is on top of the fact that, due to dplyr’s more efficient code, the steps often execute more quickly than the equivalent commands in base R! Providing a complete dplyr tutorial is beyond the scope of this book, but there are many learning resources available online, including the *R for Data Science* chapter at <https://r4ds.hadley.nz/transform.html>.

Transforming text with stringr

The `stringr` package (<https://stringr.tidyverse.org>) adds functions to analyze and transform character strings. Base R, of course, can do this too, but the functions are inconsistent in how they work on vectors and are relatively slow; `stringr` implements these functions in a form more attuned to the tidyverse workflow. The free resource *R for Data Science* has a tutorial that introduces the package’s complete set of capabilities, at <https://r4ds.hadley.nz/strings.html>, but here, we’ll examine some of the aspects most relevant to feature engineering. If you’d like to follow along, be sure to load the `Titanic` dataset and install and load the `stringr` package before proceeding.

Earlier in this chapter, the second tip for feature engineering was to “find insights hidden in text.” The `stringr` package can assist with this effort by providing functions to slice strings and detect patterns within text. All `stringr` functions begin with the prefix `str_`, and a few relevant examples are as follows:

- `str_detect()` determines whether a search term is found in a string
- `str_sub()` slices a string by position and returns a substring
- `str_extract()` searches for a string and returns the matching pattern
- `str_replace()` replaces characters in a string with something else

Although these functions seem quite similar, they are used for quite different purposes. To demonstrate these purposes, we’ll begin by examining the `Cabin` feature to determine whether certain rooms on the *Titanic* are linked to greater survival. We cannot use this feature as-is, because each cabin code is unique.

However, because the codes are in forms like A10, B101, or E67, perhaps the alphabetical prefix indicates a position on the ship, and perhaps passengers in some of these locations may have been more able to escape the disaster. We'll use the `str_sub()` function to take a 1-character substring beginning and ending at position 1, and save this to a `CabinCode` feature as follows:

```
> titanic_train <- titanic_train |>  
  mutate(CabinCode = str_sub(Cabin, start = 1, end = 1))
```

To confirm that the cabin code is meaningful, we can use the `table()` function to see a clear relationship between it and the passenger class. The `useNA` parameter is set to "ifany" to display the NA values caused by missing cabin codes for some passengers:

```
> table(titanic_train$Pclass, titanic_train$CabinCode,  
  useNA = "ifany")
```

	A	B	C	D	E	F	G	T	<NA>
1	15	47	59	29	25	0	0	1	40
2	0	0	0	4	4	8	0	0	168
3	0	0	0	0	3	5	4	0	479

The NA values appear to be more common in the lower ticket classes, so it seems plausible that cheaper fares may have not received a cabin code.

We can also plot the survival probability by cabin code by piping the file into a `ggplot()` function:

```
> library(ggplot2)  
> titanic_train |> ggplot() +  
  geom_bar(aes(x = CabinCode, y = Survived),  
    stat = "summary", fun = "mean") +  
  ggtitle("Titanic Survival Rate by Cabin Code")
```

The resulting figure shows that even within the first-class cabin types (codes A, B, and C) there are differences in survival rate; additionally, the passengers without a cabin code are the least likely to survive:

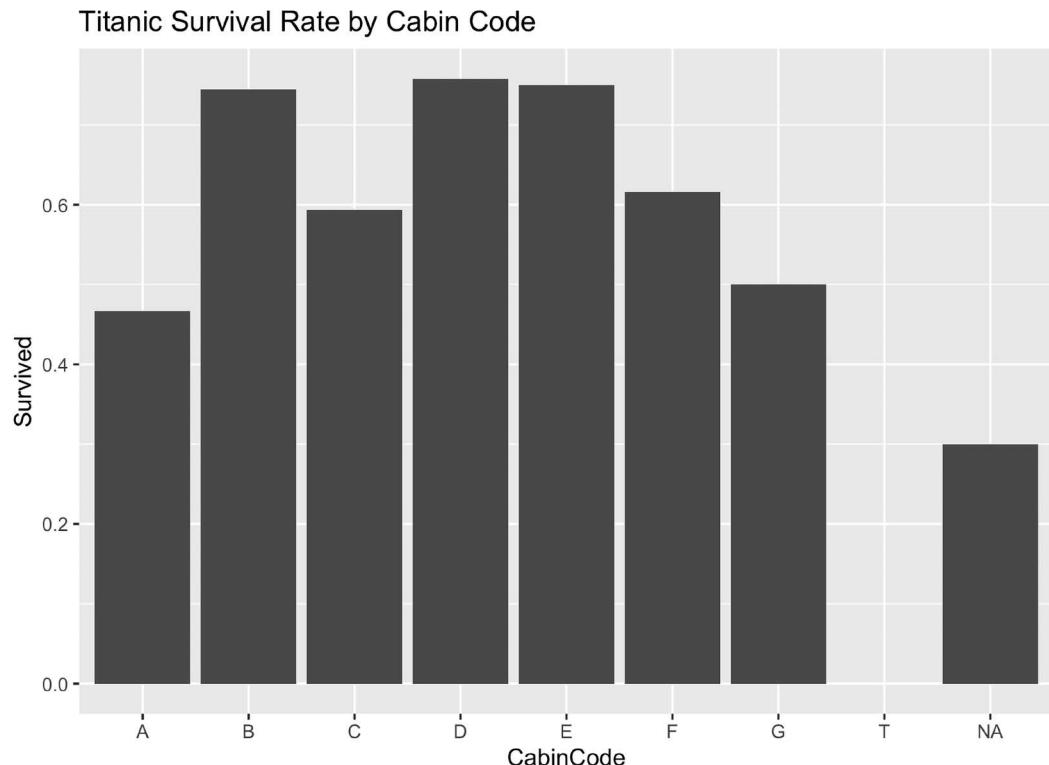


Figure 12.15: The cabin code feature seems related to survival, even within first-class cabins (A, B, and C)

Without processing the Cabin text data first, a learning algorithm would be unable to use the feature as the codes are unique to each cabin. Yet by applying a simple text transformation, we've decoded the cabin codes into something that can be used to improve the model's survival predictions.

With this success in mind, let's examine another potential source of hidden data: the Name column. One might assume that this is unusable in a model, because the name is a unique identifier per row and training a model on this data will inevitably lead to overfitting. Although this is true, there is useful information hiding within the names. Looking at the first few rows reveals some potentially useful text strings:

```
> head(titanic_train$name)

[1] "Braund, Mr. Owen Harris"
[2] "Cumings, Mrs. John Bradley (Florence Briggs Thayer)"
[3] "Heikkinen, Miss. Laina"
[4] "Futrelle, Mrs. Jacques Heath (Lily May Peel)"
[5] "Allen, Mr. William Henry"
[6] "Moran, Mr. James"
```

For one, the salutation (Mr., Mrs., and Miss.) might be helpful for prediction. The problem is that these titles are located at different positions within the name strings, so we cannot simply use the `str_sub()` function to extract them. The correct tool for this job is `str_extract()`, which is used to match and extract shorter patterns from longer strings. The trick with working with this function is knowing how to express a text pattern rather than typing each potential salutation separately.

The shorthand used to express a text search pattern is called a **regular expression**, or **regex** for short. Knowing how to create regular expressions is an incredibly useful skill, as they are used for the advanced find-and-replace features in many text editors, in addition to being useful for feature engineering in R. We'll create a simple regex to extract the salutations from the name strings.

The first step in using regular expressions is to identify the common elements across all the desired target strings. In the case of the Titanic names, it looks like each salutation is preceded by a comma followed by a blank space, then has a series of letters before ending with a period. This can be coded as the following regex string:

```
", [A-z]+\\."
```

This seems to be nonsense but can be understood as a sequence that attempts to match a pattern character by character. The matching process begins with a comma and a blank space, as expected. Next, the square brackets tell the search function to look for any of the characters inside the brackets. For instance, `[AB]` would search for A or B, and `[ABC]` would search for A, B, or C. In our usage, the dash is used to search for any characters within the range between A and z. Note that capitalization is important—that is, `[A-Z]` is different from `[A-z]`. The former will search 26 characters comprising the uppercase alphabet while the latter will search 52 characters, including uppercase and lowercase. Keep in mind that `[A-z]` only matches a single character.

To have the expression match more characters, we follow the brackets with a + symbol to tell the algorithm to continue matching characters until it reaches something not inside the brackets. Then, it checks to see whether the remaining part of the regex matches.

The remaining piece is the \\. sequence, which is three characters that represent the single period character at the end of our search pattern. Because the dot is a special term that represents an arbitrary character, we must escape the dot by prefixing it with a slash. Unfortunately, the slash is also a special character in R, so we must escape it as well by prefixing it with yet another slash.



Regular expressions can be tricky to learn but are well worth the effort. You can find a deep dive into understanding how they work at <https://www.regular-expressions.info>. Alternatively, there are many text editors and web applications that demonstrate matching in real time. These can be hugely helpful to understand how to develop the regex search patterns and diagnose errors. One of the best such tools is found at <https://regexr.com>.

We can put this expression to work on the Titanic name data by combining it in a `mutate()` function with `str_extract()`, as follows:

```
> titanic_train <- titanic_train |>  
  mutate>Title = str_extract(Name, ", [A-z]+\\.".")
```

Looking at the first few examples, it looks like these need to be cleaned up a bit:

```
> head(titanic_train>Title)  
[1] ", Mr."    ", Mrs."   ", Miss." ", Mrs."   ", Mr."    ", Mr."
```

Let's use the `str_replace()` function to eliminate the punctuation and blank spaces in these titles. We begin by constructing a regex to match the punctuation and empty space. One way to do this is to match the comma, blank space, and period using the "[, \\.]" search string. Used with `str_replace()` as shown here, any comma, blank space, and period characters in `Title` will be replaced by the empty (null) string:

```
> titanic_train <- titanic_train |>  
  mutate>Title = str_replace_all>Title, "[, \\.]", "")
```

Note that the `str_replace_all()` variant of the replace function was used due to the fact that multiple characters needed replacement; the basic `str_replace()` would have only replaced the first instance of a matching character. Many of `stringr`'s functions have "all" variants for this use case. Let's see the result of our effort:

```
> table(titanic_train>Title)
```

Capt	Col	Don	Dr	Jonkheer	Lady
1	2	1	7	1	1
Major	Master	Miss	Mlle	Mme	Mr
2	40	182	2	1	517
Mrs	Ms	Rev	Sir		
125	1	6	1		

Given the small counts for some of these titles and salutations, it may make sense to group them together. To this end, we can use dplyr's `recode()` function to change the categories. We'll keep several of the high-count levels the same, while grouping the rest into variants of `Miss` and a catch-all bucket, using the `.missing` and `.default` values to assign the `Other` label to NA values and anything else not already coded:

```
> titanic_train <- titanic_train |>
  mutate>TitleGroup = recode(Title,
    "Mr" = "Mr", "Mrs" = "Mrs", "Master" = "Master",
    "Miss" = "Miss",
    "Ms" = "Miss", "Mlle" = "Miss", "Mme" = "Miss",
    .missing = "Other",
    .default = "Other"
  )
)
```

Checking our work, we see that our cleanup worked as planned:

```
> table(titanic_train$TitleGroup)
```

Master	Miss	Mr	Mrs	Other
40	186	517	125	23

We can also see that the title is meaningful by examining a plot of survival rates by title:

```
> titanic_train |> ggplot() +
  geom_bar(aes(x = TitleGroup, y = Survived),
            stat = "summary", fun = "mean") +
  ggtitle("Titanic Survival Rate by Salutation")
```

This produces the following bar chart:

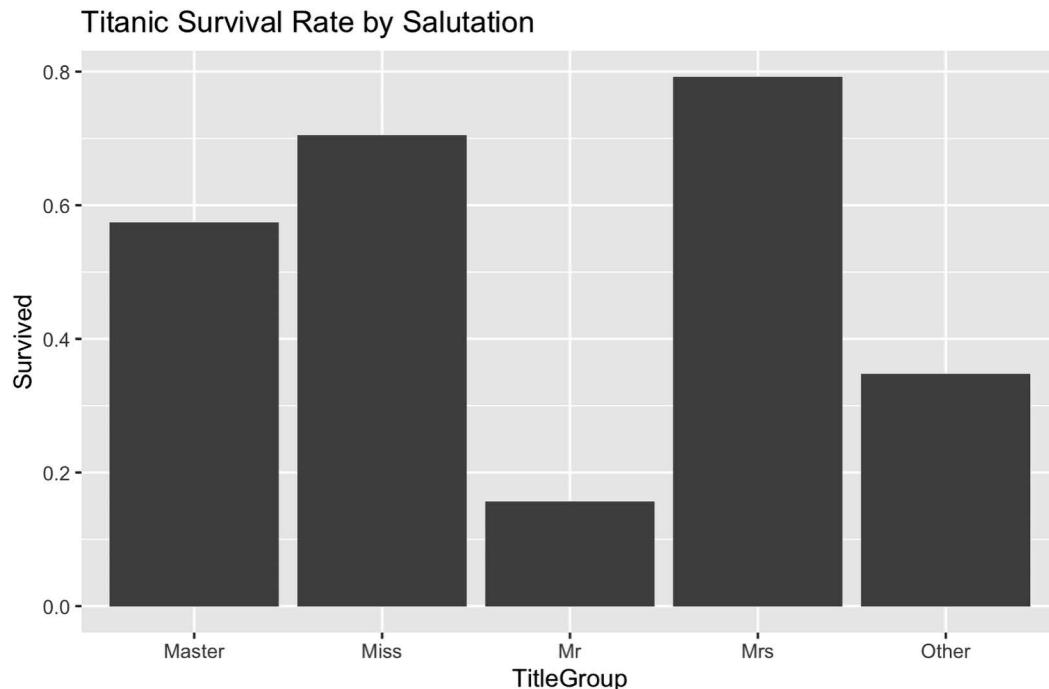


Figure 12.16: The constructed salutation captures the impact of both age and gender on survival likelihood

The creation of `CabinCode` and `TitleGroup` features exemplifies the feature engineering technique of finding hidden information in text data. These new features are likely to provide additional information beyond the base features in the Titanic dataset, which learning algorithms can use to improve performance. A bit of creativity combined with `stringr` and knowledge of regular expressions may provide the edge needed to surpass the competition.

Cleaning dates with lubridate

The `lubridate` package (<https://lubridate.tidyverse.org>) is an important tool for working with date and time data. It may not be needed for every analysis, but when it is needed, it can save a lot of grief. With dates and times, seemingly simple tasks can quickly turn into adventures, due to unforeseen subtleties like leap years and time zones—just ask anyone who has worked on birthday calculations, billing cycles, or similar date-sensitive tasks.

As with the other tidyverse packages, the *R for Data Science* resource has an in-depth lubridate tutorial at <https://r4ds.hadley.nz/datetime.html>, but we'll briefly cover three of its most important feature engineering strengths here:

- Ensuring date and time data is loaded into R correctly while accounting for regional differences in how dates and times are expressed
- Accurately calculating differences between dates and times while accounting for time zones and leap years
- Accounting for differences in how increments in time are understood in the real world, such as the fact that people become “1 year older” on their birthday

Reading dates into R is challenge number one, because dates are presented in many different formats. For example, the publication date of the first edition of *Machine Learning with R* can be expressed as:

- October 25, 2013 (a common longhand format in the United States)
- 10/25/13 (a common shorthand format in the United States)
- 25 October 2013 (a common longhand format in Europe)
- 25.10.13 (a common shorthand format in Europe)
- 2013-10-25 (the international standard)

Given these diverse formats, lubridate is incapable of determining the correct format without help because months, days, and years can all fall in the range from 1 to 12. Instead, we provide it the correct date constructor—either `mdy()`, `dmy()`, or `ymd()`, depending on the order of the month (`m`), day (`d`), and year (`y`) components of the input data. Given the order of the date components, the functions will automatically parse longhand and shorthand variants, and will handle leading zeros and two- or four-digit years. To demonstrate this, the dates expressed previously can be handled with the appropriate lubridate function, as follows:

```
> mdy(c("October 25, 2013", "10/25/2013"))
```

```
[1] "2013-10-25" "2013-10-25"
```

```
> dmy(c("25 October 2013", "25.10.13"))
```

```
[1] "2013-10-25" "2013-10-25"
```

```
> ymd("2013-10-25")
```

```
[1] "2013-10-25"
```

Notice that in each case, the resulting Date object is exactly the same. Let's create a similar object for each of the three previous editions of this book:

```
> MLwR_1stEd <- mdy("October 25, 2013")
> MLwR_2ndEd <- mdy("July 31, 2015")
> MLwR_3rdEd <- mdy("April 15, 2019")
```

We can do simple math to compute the difference between two dates:

```
> MLwR_2ndEd - MLwR_1stEd
```

Time difference of 644 days

```
> MLwR_3rdEd - MLwR_2ndEd
```

Time difference of 1354 days

Notice that by default, the difference between the two dates is returned as days. What if we hope to have an answer in years? Unfortunately, because these differences are a special lubridate `difftime` object, we cannot simply divide these numbers by 365 days to perform the obvious calculation. One option is to convert them into a `duration`, which is one of the ways lubridate computes date differences, and in particular, tracks the passage of physical time—imagine it acting much like a stopwatch. The `as.duration()` function performs the needed conversion:

```
> as.duration(MLwR_2ndEd - MLwR_1stEd)
```

[1] "55641600s (~1.76 years)"

```
> as.duration(MLwR_3rdEd - MLwR_2ndEd)
```

[1] "116985600s (~3.71 years)"

We can see here that the gap between the 2nd and 3rd editions of *Machine Learning with R* was almost twice as long as the difference between the 1st and 2nd editions. We can also see that the duration seems to default to seconds while also providing the approximate number of years. To obtain only years, we can divide the duration by the duration of 1 year, which lubridate provides as a `dyears()` function:

```
> dyears()
```

[1] "31557600s (~1 years)"

```
> as.duration(MLwR_2ndEd - MLwR_1stEd) / dyears()
```

```
[1] 1.763176
```

```
> as.duration(MLwR_3rdEd - MLwR_2ndEd) / dyears()
```

```
[1] 3.70705
```

You may find it more convenient or easier to remember the `time_length()` function, which can perform the same calculation:

```
> time_length(MLwR_2ndEd - MLwR_1stEd, unit = "years")
```

```
[1] 1.763176
```

```
> time_length(MLwR_3rdEd - MLwR_2ndEd, unit = "years")
```

```
[1] 3.70705
```

The `unit` argument can be set to units like days, months, and years, depending on the desired result. Notice, however, that these durations are exact to the second like a stopwatch, which is not always how people think about dates.

In particular, for birthdays and anniversaries, people tend to think in terms of calendar time—that is, the number of times the calendar has reached a particular milestone. In lubridate, this approach is called an **interval**, which implies a timeline-or calendar-based view of date differences, rather than the stopwatch-based approach of the duration methods discussed previously.

Let's imagine we'd like to compute the age of the United States, which was born, so to speak, on July 4, 1776. This means that on July 3, 2023, the country will be 246 birthdays old, and on July 5, 2023, it will be 247. Using durations, we don't get quite the right answers:

```
> USA_DOB <- mdy("July 4, 1776") # USA's Date of Birth
```

```
> time_length(mdy("July 3 2023") - USA_DOB, unit = "years")
```

```
[1] 246.9897
```

```
> time_length(mdy("July 5 2023") - USA_DOB, unit = "years")
```

```
[1] 246.9952
```

The problem has to do with the fact that durations deviate from calendar time due to calendar irregularities such as leap years and time changes. By explicitly converting the date difference into an interval with the `interval()` function, and then dividing by the `years()` function, we get closer to the right answer:

```
> interval(USA_DOB, mdy("July 3 2023")) / years()
```

```
[1] 246.9973  
> interval(USA_DOB, mdy("July 5 2023")) / years()  
[1] 247
```

Before going any further, be sure to notice the fact that the `interval()` uses `start`, `end` syntax, in contrast to the date difference, which used `end - start`. Also note that the `years()` function returns a lubridate **period**, which is yet another way to understand differences between dates and times. Periods are always relative to their position on a calendar, which means that a 1-hour period can be a 2-hour duration during a time change, and a 1-year period can include 365 or 366 1-day periods, depending on the calendar year—these are the types of challenging subtleties when working with dates that were mentioned in this section’s opening paragraph!

To create our final age calculation, we’ll use the `%--%` interval construction operator as shorthand, and use the integer division operator `%/%` to return only the integer component of the age. These return the expected age values:

```
> USA_DOB %--% mdy("July 3 2023") %/% years()  
[1] 246  
> USA_DOB %--% mdy("July 5 2023") %/% years()  
[1] 247
```

Generalizing this work, we can create a function to compute the calendar-based age for a given date of birth as of today:

```
> age <- function(birthdate) {  
  birthdate %--% today() %/% years()  
}
```

To prove that it works, we’ll check the ages of a few famous tech billionaires:

```
> age(mdy("February 24, 1955")) # Jeff Bezos  
[1] 59  
> age(mdy("June 28, 1971")) # Elon Musk  
[1] 51  
> age(mdy("Oct 28, 1955")) # Bill Gates
```

[1] 67

If you are following along in R, be aware that your results may vary depending on when you run the code—we’re all, unfortunately, still getting older by the day!

Summary

This chapter demonstrated the importance of data preparation. Because the tools and algorithms used to build machine learning models are the same across projects, data preparation is a key that unlocks the highest levels of model performance. This allows some aspects of human intelligence and creativity to have a large impact on the machine’s learning process, although clever practitioners use their strengths in concert with the machine’s by developing automated data engineering pipelines that take advantage of the computer’s ability to tirelessly search for useful insights in the data. These pipelines are especially important in the so-called “big data regime,” where data-hungry approaches like deep learning must be fed large amounts of data to avoid overfitting.

In traditional small and medium data regimes, feature engineering by hand still reigns supreme. Using intuition and subject matter expertise, one can guide the model to the most useful signal in the training dataset. As this is more art than science, tips and tricks are learned on the job, or passed along second-hand from one data scientist to another. This chapter provided seven hints to help guide you on the journey, but the only way to truly become skilled at feature engineering is through practice.

Tools like the tidyverse suite of R packages make it much less laborious than in years past to gain the necessary experience to perform feature engineering tasks. This chapter demonstrated how the tidyverse packages can be used to turn data into more useful predictors, and how information hidden in text data can be extracted to turn what seem like useless features into important predictors. The tidyverse packages are much more capable of handling large and ever-growing datasets than the base R functions, and they make R a pleasure to use even as datasets grow in size and complexity.

The skills developed in this chapter will provide a foundation for the work to come. In the next chapter, you will add new tidyverse packages to your toolkit and see even more examples of how it integrates into the machine learning workflow. You will continue to see the importance of data preparation skills as you explore data issues that begin as relatively minor challenges but quickly grow into massive problems if taken to an extreme.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



13

Challenging Data – Too Much, Too Little, Too Complex

Challenging data takes many forms throughout the course of a machine learning project, and the journey of each new project represents an adventure requiring a pioneer spirit. Beginning with uncharted data that must be explored, the data must then be wrangled before it can be used with the learning algorithm. Even then, there may still be wild aspects of the data that need to be tamed for the project to be successful. Extraneous information must be culled, small-but-important details must be cultivated, and tangled webs of complexity must be cleared from the learner’s path.

Conventional wisdom in the big data era suggests that data is treasure, but as the saying goes, one can have “too much of a good thing.” Most machine learning algorithms will happily indulge in as much data as they are fed, which leads to a new set of problems akin to overeating. An abundance of data can overwhelm the learner with unnecessary information, obscure important patterns, and shift the learner’s attention from the details that matter to those that are obvious. Thus, it may be better to avoid the “more is always better” mindset and instead find a balance between quantity and quality.

The purpose of this chapter is to consider techniques that can be used to adapt to a dataset’s signal-to-noise ratio. You will learn:

- How to handle datasets with an overwhelming number of features
- Methods for making use of feature values that are missing or appear very infrequently
- Approaches for modeling rare target outcomes

You will discover that some learning algorithms are better equipped at performing these techniques independently, while others will require you to intervene more extensively in the process. In either case, due to the prevalence of these types of data issues and their status as some of the most challenging problems in machine learning, it is important to understand the ways that they can be remedied.

The challenge of high-dimension data

If someone says that they are struggling to handle the size of a dataset, it is easy to assume that they are talking about having too many rows or that the data uses too much memory or storage space. Indeed, these are common issues that cause problems for new machine learning practitioners. In this scenario, the solutions tend to be technical rather than methodological; one generally chooses a more efficient algorithm or uses hardware or a cloud computing platform capable of consuming large datasets. In the worst case, one can take a random sampling and simply discard some of the excessive rows.

The challenge of having too much data can also apply to a dataset's columns, making the dataset overly wide rather than overly long. It may require some creative thinking to imagine why this happens, or why it is a problem, because it is rarely encountered in the tidy confines of teaching examples. Even in real-world practice, it may be quite some time before someone encounters this problem, as useful predictors can be scarce, and datasets are often scrounged piece by piece. For such projects, having too many predictors would be a good problem to have!

However, consider a situation in which a data-driven organization, acutely aware of the competitive advantage of big data, has amassed a war chest of information from a variety of sources. Perhaps they collected some of the data directly through the ordinary course of business, purchased supplemental data from vendors, and gathered some via additional sensors or indirect, passive interactions via the internet. All these sources are merged into a single table that provides a rich but highly complex and varied set of features. The resulting table was not carefully constructed piece by piece, but rather through a mishmash of data elements, some of which will be more useful than others. Today, this type of data treasure trove is found predominantly in very large or very data-savvy organizations, but it is likely that an increasing number will have access to similar datasets in the future. Datasets are growing increasingly wide over time, even before considering inherently feature-rich sources such as text, audio, image, or genetic data.

The challenge of these types of high-dimension datasets, in short, has much to do with the fact that more data points have been collected than are truly needed to represent the underlying pattern.

The additional data points add noise or subtle variations across examples and may distract a learning algorithm from the important trends. This describes the **curse of dimensionality** in which learners fail as the number of features increases. If we imagine each additional feature as a new dimension of the example—and here, the word “dimension” is used both in a literal and a metaphorical sense—then as the dimensions increase, the richness of our understanding of any given example increases, but so does the example’s relative uniqueness. In a sufficiently high-dimension space, every example is unique, as it is comprised of its own distinct combination of feature values.

Consider an analogy. A fingerprint uniquely identifies individual people but it is not necessary to store all of a fingerprint’s details to make an accurate match. In fact, out of the limitless details found in each fingerprint, a forensic investigator may use only 12 to 20 distinct points to confirm a match; even computerized fingerprint scanners use only 60 to 80 points. Any additional detail is likely to be superfluous and would detract from the match quality, and taken to an extreme, might cause failed matches—even if the fingerprints are from the same person! For example, including too much detail might lead to false negatives as the learning algorithm is distracted by the print’s orientation or image quality, but too little detail may lead to false positives as the algorithm has too few features to distinguish among similar candidates. Clearly, it is important to find a balance between too much and too little detail. This is, in essence, the goal of **dimensionality reduction**, which seeks to remedy the curse of dimensionality by identifying the important details.



Figure 13.1: Dimensionality reduction helps to ignore noise and emphasize the key details that will be helpful to learn the underlying pattern

In contrast to the problem of very long datasets, the solutions needed to learn from wide datasets are completely different and are as much conceptual as they are practical. One cannot simply randomly discard columns as is possible with rows because some columns are more useful than others. Instead, a systematic approach is taken, often cooperating with the learning algorithm itself to find the balance between too much and too little detail. As you will learn in the coming sections, some of these methods are integrated into the learning process while others will require a more hands-on approach.

Applying feature selection

In the context of supervised machine learning, the goal of feature selection is to alleviate the curse of dimensionality by choosing only the most important predictors. Feature selection may also be beneficial even in the case of unsupervised learning due to its ability to simplify datasets by eliminating redundant or useless information. In addition to feature selection's primary goal of assisting a learning algorithm's attempts to separate the signal from the noise, additional benefits of the practice include:

- Shrinking the size of the dataset and decreasing storage requirements
- Reducing the time or computational expense for model training
- Enabling data scientists to focus on fewer features for data exploration and visualization

Rather than attempting to find the single most optimal complete set of predictors, which can be very computationally expensive, feature selection tends to focus on identifying useful individual features or subsets of features. To do so, feature selection typically relies on heuristics that reduce the number of subsets that are searched. This reduces the computing cost but may lead to missing the best possible solution.

To search for subsets of useful features is to assume that some predictors are useless, or at least less useful than others. Yet, despite the validity of this premise, it is not always clear what makes some features useful and others not. Of course, there may be obviously irrelevant features that provide no predictive value, but there may also be useful features that are redundant and therefore unnecessary for the learning algorithm. The trick is recognizing that something that appears redundant in one context may actually be useful in a different context.

The following figure illustrates the ability of useful features to disguise themselves as seemingly useless and redundant predictors. The scatterplot depicts a relationship between two hypothetical features, each having values in the approximate range of -50 to 50, and being used to predict a binary outcome, triangles versus circles.

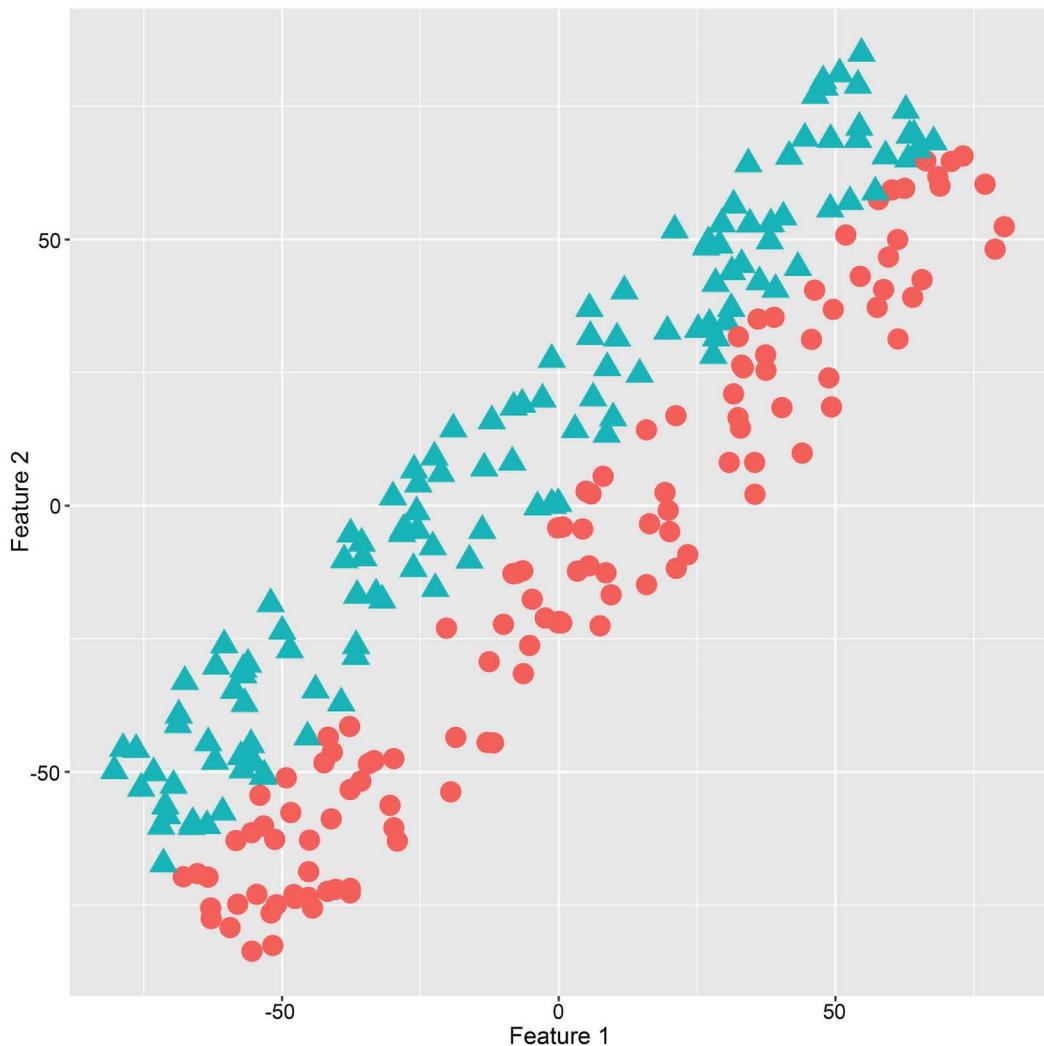
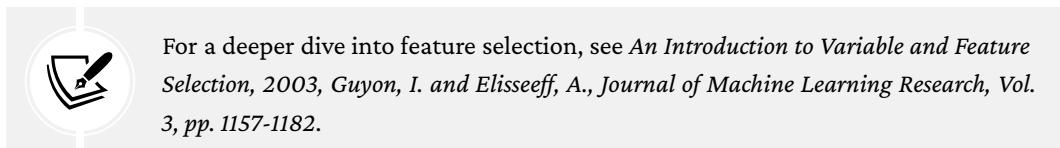


Figure 13.2: Features 1 and 2 are seemingly useless and redundant but have predictive value when used together

Knowing the value of feature 1 or 2 alone provides virtually no value toward predicting the outcome of the target, as the circles and triangles are almost completely evenly split for any value of either feature. In quantitative terms, this is demonstrated by a very weak correlation between the features and the outcome. A simple feature selection algorithm that examines only the relationship between one feature and the outcome may thus determine that neither feature is useful for prediction. Additionally, because the correlation between the two features is about 0.90, a more sophisticated feature selection algorithm that simultaneously considers the pair may inadvertently exclude one of the two due to the seeming redundancy.

Despite the seemingly useless and redundant nature of the two features, the scatterplot clearly depicts their predictive ability when used together: if feature 2 is greater than feature 1, then predict triangle; otherwise, predict circle. A useful feature selection method ought to be able to recognize these types of patterns; otherwise, it risks excluding important predictors from the learning algorithm. However, the feature selection technique also needs to consider computational efficiency, as examining every potential combination of features is infeasible except for the smallest of datasets.

The need to balance the search for useful, non-redundant features with the possibility that features may only be useful in combination with others is part of the reason there is no one-size-fits-all approach to feature selection. Depending on the use case and the chosen learning algorithm, different techniques can be applied that perform a less rigorous or more thorough search of the features.



For a deeper dive into feature selection, see *An Introduction to Variable and Feature Selection, 2003, Guyon, I. and Elisseeff, A., Journal of Machine Learning Research, Vol. 3, pp. 1157-1182.*

Filter methods

Perhaps the most accessible form of feature selection is the category of **filter methods**, which use a relatively simple scoring function to measure each feature's importance. The resulting scores can then be used to rank the features and limit the number used in the predictive model. Due to the simplicity of this approach, filter methods are often used as a first step in an iterative process of data exploration, feature engineering, and model building. One might initially apply a crude filter to identify the most interesting candidate features for in-depth exploration and visualization, then apply more vigorous feature selection methods later if further reduction is desired.

A single defining characteristic of filter methods is the use of a proxy measure of feature importance. The measure is a proxy because it is a substitute for what we truly care about—the predictive ability of the feature—but we cannot know this without first building the predictive model. Instead, we choose a much simpler metric, which we hope reflects the utility of the feature when it is later added to the model. For instance, in a numeric prediction model, one might compute bivariate correlations between each feature and the target and select only the features that are substantially correlated with the target. For a binary or categorical target, a comparable approach might involve constructing a single-variable classifier, examining contingency tables for strong bivariate relationships between the features and the target, or using a metric like information gain, which was described in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*. A benefit of these types of simple feature selection metrics is that they are unlikely to contribute to overfitting because the proxy measures use a different approach and make different assumptions about the data than the learning algorithm.

The greatest benefit of filter methods may be the fact that they are scalable even for datasets with very large numbers of features. This efficiency stems from the fact the filtering method only computes one importance score for each feature and then sorts the predictors by these scores from most to least important. Thus, as the number of features increases, the computational expense grows relatively slowly and in direct proportion to the number of predictors. Note that the product of this approach is a rank-ordered list of features rather than a single best set of features; therefore, subjective judgment is required to determine the optimal cutoff between important and not important features.

Although filter methods are computationally efficient, they lack the ability to consider groups of features, which means that important predictors may be excluded if they are only useful in combination with others. Additionally, the fact that filter methods are unlikely to contribute to overfitting comes with the potential downside that they also may not result in the set of features that are best suited to work with the desired learning algorithm. The feature selection method described in the next section sacrifices computational efficiency to address each of these concerns.

Wrapper methods and embedded methods

In contrast to filter methods, which use a proxy measure of variable importance, **wrapper methods** use the machine learning algorithm itself to identify the importance of variables or subsets of variables. Wrapper methods are based on the simple idea that as more important features are provided to the algorithm, its ability to perform the learning task should improve. In other words, its error rate should be reduced as important predictors are included or the correct combinations are included.

Thus, by iteratively building models composed of different combinations of features and examining how the model’s performance changes, it is possible to identify the important predictors and sets of predictors. By systematically testing all possible combinations of features, it is even possible to identify the overall best set of predictors.

However, as one might expect, the process of testing all possible combinations of features is extremely computationally inefficient. For a dataset with p predictors, there are 2^p potential sets of predictors that must be tested, which causes the computational expense of this technique to grow relatively quickly as additional features are added. For example, a dataset with only 10 predictors would require $2^{10} = 1,024$ different models to be evaluated, while a dataset adding just five more predictors would require $2^{15} = 32,768$ models, which is over 30 times the computational cost! Clearly, this approach is not viable except for the smallest of datasets and the simplest of machine learning algorithms. One solution to this problem might be to first reduce the number of features using a filter method, but not only does this risk missing important combinations of features but it would also require such a reduction in dimensionality that it may negate many of the benefits of wrapper methods.

Rather than letting its inefficiency prevent us from capitalizing on its upsides, we can instead use heuristics to avoid searching every combination of features. In particular, the “greedy” approach described in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, which helped grow trees efficiently, can also be used here. You may recall that the idea of a greedy algorithm is to use data on a first-come, first-served basis, with the most predictive features used first. Although this technique is not guaranteed to find the optimal solution, it drastically reduces the number of combinations that must be tested.

There are two basic approaches for adapting wrapper methods for greedy feature selection. Both involve probing the learning algorithm by changing one variable at a time. The technique of **forward selection** begins by feeding each feature to the model one by one, to determine which of them results in the best one-predictor model. The next iteration of forward selection keeps the first best predictor in the model and tests the remaining features to identify which makes the best two-predictor model. As might be expected, this process can continue selecting the best three-predictor model, four-predictor model, and so on, until all features have been selected. However, as the point of feature selection is specifically not to select the entire set of features, the process of forward selection stops early, when adding additional features no longer improves the model’s performance beyond a specific threshold.

The similar technique of **backward elimination** works the same way, but in reverse. Beginning with a model containing all features, the model iterates repeatedly, eliminating the least-predictive feature each time, until stopping when eliminating a feature decreases the model’s performance more than a desired threshold.

Learning algorithms known as **embedded methods** have a form of built-in wrappers much like forward selection. These methods select the best features automatically during the model training process. You are already familiar with one such method, decision trees, which uses greedy forward selection to determine the best feature subset. Most machine learning techniques do not have embedded feature selection; the dimensions must be reduced beforehand. The next section demonstrates how these methods can be applied in R via a variant of the machine learning algorithm introduced in *Chapter 6, Forecasting Numeric Data – Regression Methods*.

Example – Using stepwise regression for feature selection

One widely known implementation of wrapper methods is **stepwise regression**, which uses forward or backward selection to identify a set of features for a regression model. To demonstrate this technique, we’ll revisit the Titanic passenger dataset used in the previous two chapters and build a logistic regression model that predicts whether each passenger survived the ill-fated voyage. To begin, we’ll use the tidyverse to read the data and apply some simple data preparation steps. The following sequence of commands creates a missing value indicator for Age, imputes the average age for the missing Age values, imputes X for the missing Cabin and Embarked values, and converts Sex to a factor:

```
> library(tidyverse)
> titanic_train <- read_csv("titanic_train.csv") |>
  mutate(
    Age_MVI = if_else(is.na(Age), 1, 0),
    Age = if_else(is.na(Age), mean(Age, na.rm = TRUE), Age),
    Cabin = if_else(is.na(Cabin), "X", Cabin),
    Embarked = factor(if_else(is.na(Embarked), "X", Embarked)),
    Sex = factor(Sex)
  )
```

The stepwise process needs to know the starting and ending conditions for feature selection, or the minimum and maximum set of variables that can be included. In our case, we’ll define the simplest possible model as one containing no variables at all—a model with only a constant intercept term.

To define this model in R, we'll use the `glm()` function to model survival as a function of a constant intercept using the `Survived ~ 1` formula. Setting the `family` parameter to `binomial` defines a logistic regression model:

```
> simple_model <- glm(Survived ~ 1, family = binomial,
                        data = titanic_train)
```

The full model still uses logistic regression, but includes many more predictors:

```
> full_model <- glm(Survived ~ Age + Age_MVI + Embarked +
                      Sex + Pclass + SibSp + Fare,
                      family = binomial, data = titanic_train)
```

Forward selection will begin with the simple model and determine which of the features in the full model are worth including in the final model. The `step()` function in the base R `stats` package provides this functionality; however, because other packages also have `step()` functions, specifying `stats::step()` ensures the correct one is used. The first function argument provides the starting model, the `scope` parameter requires the `formula()` of the full model, and the `direction` is set to forward stepwise regression:

```
> sw_forward <- stats::step(simple_model,
                            scope = formula(full_model),
                            direction = "forward")
```

This command generates a set of outputs for each iteration of the stepwise process, but only the first and last iterations are included here for brevity.



If you are selecting from a large number of variables, set `trace = 0` in the `step()` function to turn off the output for each iteration.

At the start of the stepwise process, it begins with the simple model using the `Survived ~ 1` formula, which models survival using only a constant intercept term. The first block of output thus displays the model quality at the start and after evaluating seven other candidate models each with a single additional predictor added. The row labeled `<none>` refers to the model's quality at the start of this iteration and how it ranks compared to the seven other candidates:

```
Start: AIC=1188.66
Survived ~ 1
```

	Df	Deviance	AIC
+ Sex	1	917.8	921.8
+ Pclass	1	1084.4	1088.4
+ Fare	1	1117.6	1121.6
+ Embarked	3	1157.0	1165.0
+ Age_MVI	1	1178.9	1182.9
+ Age	1	1182.3	1186.3
<none>		1186.7	1188.7
+ SibSp	1	1185.5	1189.5

The quality measure used, AIC, is a measure of a model's relative quality compared to other models. In particular, it refers to the **Akaike information criterion**. While a formal definition of AIC is outside the scope of this chapter, the measure is intended to balance model complexity and model fit. Lower AIC values are better. Therefore, the model that includes `Sex` is the best out of the six other candidate models as well as the original model. In the final iteration, the base model uses `Sex`, `Pclass`, `Age`, and `SibSp`, and no additional features reduce the AIC further—the `<none>` row is ranked above the candidate models adding `Embarked`, `Fare`, and `Age_MVI` features:

```
Step:  AIC=800.84
Survived ~ Sex + Pclass + Age + SibSp

Df Deviance    AIC
<none>      790.84 800.84
+ Embarked   3    785.27 801.27
+ Fare        1    789.65 801.65
+ Age_MVI    1    790.59 802.59
```

At this point, the forward selection process stops. We can obtain the formula for the final model:

```
> formula(sw_forward)

Survived ~ Sex + Pclass + Age + SibSp
```

We can also obtain the final model's estimated regression coefficients:

```
> sw_forward$coefficients

(Intercept)      Sexmale      Pclass       Age      SibSp
  5.19197585 -2.73980616 -1.17239094 -0.03979317 -0.35778841
```

Backward elimination is even simpler to execute. By providing a model with the complete set of features to test and setting `direction = "backward"`, the model will iterate and systematically eliminate any features that will result in a better AIC. For example, the first step begins with a full set of predictors, but eliminating the `Fare`, `Age_MVI`, or `Embarked` features results in a lower AIC:

```
> sw_backward <- stats::step(full_model, direction = "backward")
```

```
Start: AIC=803.49
Survived ~ Age + Age_MVI + Embarked + Sex + Pclass + SibSp +
Fare
```

	Df	Deviance	AIC
- Fare	1	783.88	801.88
- Age_MVI	1	784.81	802.81
- Embarked	3	789.42	803.42
<none>		783.49	803.49
- SibSp	1	796.34	814.34
- Age	1	810.97	828.97
- Pclass	1	844.74	862.74
- Sex	1	1016.36	1034.36

At each iteration, the worst feature is eliminated, but by the final step, eliminating any of the remaining features leads to a higher AIC, and therefore leads to a lower-quality model than the baseline. Thus, the process stops here:

```
Step: AIC=800.84
Survived ~ Age + Sex + Pclass + SibSp
```

	Df	Deviance	AIC
<none>		790.84	800.84
- SibSp	1	805.33	813.33
- Age	1	819.32	827.32
- Pclass	1	901.80	909.80
- Sex	1	1044.10	1052.10

In this case, forward selection and backward elimination resulted in the same set of predictors, but this is not necessarily always the case. Differences may arise if certain features work better in groups or if they are interrelated in some other way.

As noted previously, one of the downsides of the heuristics used by wrapper methods is that they are not guaranteed to find the single most optimal set of predictors; however, this shortcoming is exactly what makes the feature selection process computationally feasible.

Example – Using Boruta for feature selection

For a more robust yet much more computationally intensive feature selection method, the `Boruta` package implements a wrapper around the random forest algorithm, which will be introduced in *Chapter 14, Building Better Learners*. For now, it suffices to know that random forests are a variant of decision trees, which provide a measure of variable importance. By systematically testing random subsets of variables repeatedly, it is possible to determine whether a feature is significantly more or less important than others using statistical hypothesis testing techniques.



Because of its heavy reliance on the random forest technique, it is no surprise that the technique shares a name with Boruta, a mythological Slavic creature thought to dwell in swamps and forests. To read more about Boruta's implementation details, see *Feature Selection with the Boruta Package*, Kursa, M. B. and Rudnicki, W. R., 2010, *Journal of Statistical Software*, Vol. 36, Iss. 11.

The Boruta technique employs a clever trick using so-called “shadow features” to determine whether a variable is important. These shadow features are copies of the dataset’s original features, but with the values shuffled randomly so that any association between the feature and the target outcome is broken. Thus, these shadow features are, by definition, nonsense and unimportant, and should provide zero predictive benefits to the model except by random chance. They serve as a baseline by which the other features are judged.

After running the original features and shadow features through the random forest modeling process, the importance of each original feature is compared to the most important shadow feature. Features that are significantly better than the shadow feature are deemed important; those significantly worse are deemed unimportant and permanently removed. The algorithm iterates repeatedly until all features are deemed important or unimportant, or the process hits a predetermined limit of iterations.

To see this in action, let’s apply the Boruta algorithm to the same Titanic training dataset constructed in the previous section. Just to prove that the algorithm can detect truly useless features, for demonstration purposes we can add one to the dataset. First, we’ll set the random seed to the arbitrary number 12345 to ensure your results match those shown here. Then, we’ll assign each of the 891 training examples a random value between 1 and 100.

Because the numbers are completely random, this feature should almost certainly be found useless, except in the case of dumb luck:

```
> set.seed(12345)
> titanic_train$rand_vals <- runif(n = 891, min = 1, max = 100)
```

Next, we'll load the `Boruta` package and apply it to the Titanic dataset. The syntax is similar to training a machine learning model; here, we specify the model using the formula interface to list the target and predictors:

```
> library(Boruta)
> titanic_boruta <- Boruta(Survived ~ PassengerId + Age +
+                               Sex + Pclass + SibSp + random_vals,
+                               data = titanic_train, doTrace = 1)
```

The `doTrace` parameter is set to 1 to request verbose output, which produces a status update as the algorithm reaches key points in the iteration process. Here, we see the output after 10 iterations, which shows that the `rand_vals` feature has unsurprisingly been rejected as unimportant, while four features were confirmed as important and one feature remains undetermined:

```
After 10 iterations, +0.51 secs:
confirmed 4 attributes: Age, Pclass, Sex, SibSp;
rejected 1 attribute: rand_vals;
still have 1 attribute left.
```

Once the algorithm has completed, type the name of the object to see the results:

```
> titanic_boruta

Boruta performed 99 iterations in 4.555043 secs.
4 attributes confirmed important: Age, Pclass, Sex, SibSp;
1 attributes confirmed unimportant: rand_vals;
1 tentative attributes left: PassengerId;
```

The `Boruta()` function is set to a limit of 100 runs by default, which it hit after iterating 99 times in about 4.5 seconds. Before stopping, four features were found to be important and one was found to be unimportant. The `PassengerId` feature, which is listed as tentative, was unable to be confirmed as important or unimportant. Setting the `maxRuns` parameter to a higher value than 100 can help come to a conclusion—in this case, setting `maxRuns = 500` will confirm `PassengerId` to be unimportant after 486 iterations.

It is also possible to plot the importance of the features relative to one another:

```
> plot(titanic_boruta)
```

The resulting visualization is shown in *Figure 13.3*. For each of the six features, as well as the max, mean (average), and min performing shadow features, a boxplot shows the distribution of importance metrics for that feature. Using these results, we can confirm that the PassengerId is slightly less important than the max shadow feature, and rand_vals is even less important than that:

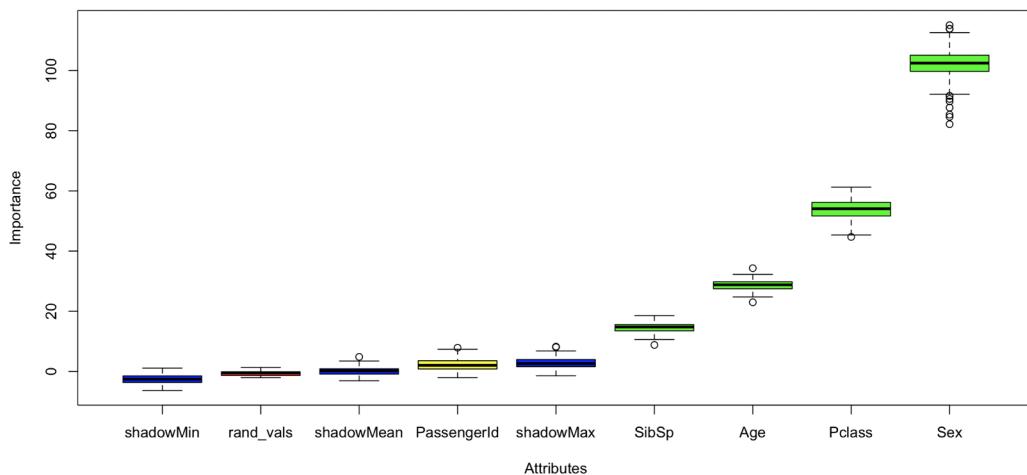


Figure 13.3: Plotting the Boruta output shows the relative importance of features compared to each other and the shadow features

Based on the exploration of the Titanic dataset we performed in *Chapter 11, Being Successful with Machine Learning*, the high importance of the Sex and Pclass features is unsurprising. Likewise, we would not expect the PassengerId to be important, unless the IDs were somehow linked to Titanic survival rather than being assigned at random. This being said, even though the results of this feature selection process did not reveal new insights, the technique would be much more helpful for datasets that are not as easy to explore by hand, or where the real-world meaning of the features is unknown. Of course, this is just one approach for dealing with a large number of features of undetermined importance; the next section describes an alternative that may perform better, especially if many of the features are correlated.



The Boruta technique can be very computationally intensive, and on real-world datasets, it will generally take minutes or even hours to complete rather than seconds as with the Titanic data. The authors of the package estimate that on a modern computer, it needs roughly one hour per million feature-example combinations. For example, a dataset with 10,000 rows and 50 features will take roughly half an hour to complete. Increasing the size of this dataset to 100,000 rows would require about five hours of processing time!

Performing feature extraction

Feature selection is not the only approach available to reduce the dimensionality of a highly dimensional dataset. Another possibility is to synthesize a smaller number of composite predictors. This is the goal of **feature extraction**, a dimensionality reduction technique that creates new features rather than selecting a subset of existing features. The extracted features are constructed such that they reduce the amount of redundant information while keeping as much useful information as possible. Of course, finding the ideal balance between too much and too little information is a challenge in itself.

Understanding principal component analysis

To begin to understand feature extraction, start by imagining a dataset with a very large number of features. For instance, to predict applicants likely to default on a loan, a dataset may include hundreds of applicant attributes. Obviously, some of the features are going to be predictive of the target outcome, but it is likely that many of the features are predictive of each other as well. For example, a person's age, education level, income, zip code, and occupation are all predictive of their likelihood to pay back a loan, but they are also predictive of each other to varying degrees. Their interrelatedness suggests that there is a degree of overlap or joint dependency among them, which is reflected in their covariance and correlation.

It may be the case that the reason these five attributes of loan applicants are related is that they are components of a smaller number of attributes that are the true, underlying drivers of loan payment behavior. In particular, we might believe that loan payment likelihood is based on an applicant's responsibility and affluence, but because these concepts are difficult to measure directly, we instead use multiple readily available proxy measures. The following figure illustrates how each of the five features might capture aspects of the two hidden dimensions of interest. Note that none of the features fully captures either component dimension, but rather, each component dimension is a composite of several features.

For instance, a person's level of responsibility might be captured by their age and education level, while their affluence might be reflected in their income, occupation, and zip code.

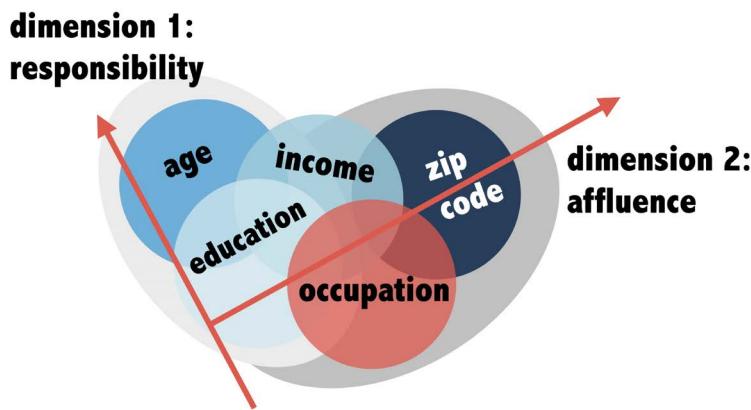


Figure 13.4: Five hypothetical attributes of loan applicants might be more simply expressed in two dimensions created from composites of the covariance of each attribute

The goal of **principal component analysis (PCA)** is to extract a smaller number of underlying dimensions from a larger number of features by expressing the covariance of multiple correlated attributes as a single vector. Put simply, the covariance refers to the extent to which attributes vary in concert. When one goes up or down, the other tends to go up or down. The resulting vectors are known as **principal components** and are constructed as weighted combinations of the original attributes. When applied to a dataset with many correlated features, a much smaller number of principal components may be capable of expressing much of the total variance of the higher-dimension dataset. Although this seems like a lot of technical jargon, and the math required to implement PCA is beyond the scope of the book, we will work toward a conceptual understanding of the process.



Principal component analysis is closely related to another technique, called **factor analysis**, which is a more formal approach for exploring the relationships between observed and unobserved (latent) factors, such as those depicted in the figures here. In practice, both can be applied similarly, but PCA is simpler and avoids building a formal model; it merely reduces the number of dimensions while retaining maximal variation. For a deeper dive into the many subtle distinctions, see the following Stack Exchange thread: <https://stats.stackexchange.com/questions/1576/what-are-the-differences-between-factor-analysis-and-principal-component-analysis/>.

Revisiting *Figure 13.4*, each circle is intended to represent the relationships among each of the five features. Circles with greater overlap represent correlated features that may measure a similar underlying concept. Keep in mind that this is a highly simplified representation that does not depict the individual data points that would be used to compute the correlations among the features. In reality, these individual data points would represent individual loan applicants and would be positioned in a five-dimensional space with coordinates determined by each applicant's five feature values. Of course, this is difficult to depict in the two dimensions of this book's pages, so the circles in this simplified representation should be understood as a cloud-like mass of people with high values of the attribute. In this case, if two features are highly correlated, such as income and education, the two clouds will overlap, because people with high values of one attribute will tend to have high values of the other. *Figure 13.5* depicts this relationship:

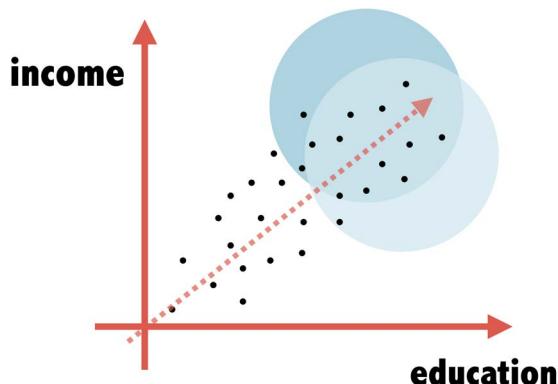


Figure 13.5: When two features are highly correlated, points with high values of one tend to have high values of the other

When examining *Figure 13.5*, note that the diagonal arrow that represents the relationship between income and education reflects the covariance between the two features. Knowing whether a point is closer to the start or end of the arrow would provide a good estimate of both income and education. Highly covariant features are thus likely to express similar underlying attributes and therefore may be redundant. In this way, the information expressed by two dimensions, income and education, could be expressed more simply in a single dimension, which would be the principal component of these two features.

Applying this relationship to a diagram in three dimensions, we might imagine this principal component as the z dimension in the following figure:

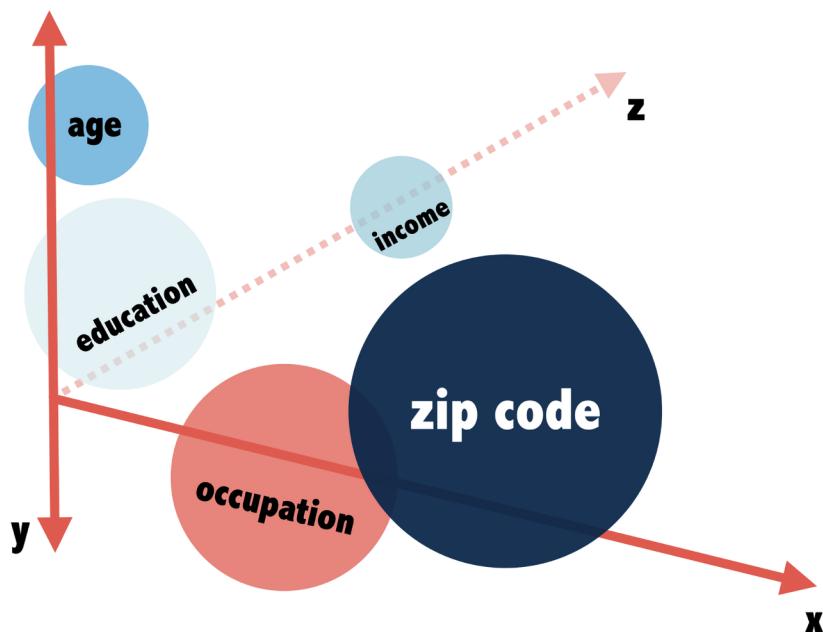


Figure 13.6: Five attributes with varying degrees of covariance in three dimensions

As with the two-dimensional case, the positioning of the circles is intended to represent the covariance among the features; the circle sizes are meant to represent depth, with larger or smaller circles closer to the front or back of the space. In the three dimensions here, age and education are close on one dimension, occupation and zip code are close on another, and income varies in a third dimension.

If we hoped to capture most of the variance while reducing the number of dimensions from three to two, we might project this three-dimensional plot onto a two-dimensional plot as follows:

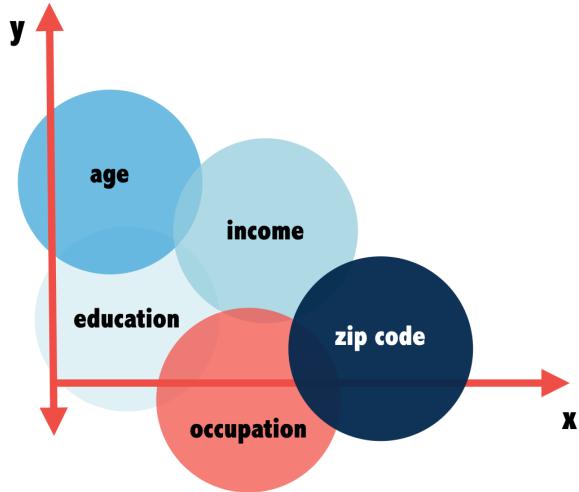


Figure 13.7: Principal component analysis reduces many dimensions into a smaller number of key components

With these two dimensions, we have constructed the two principal components of the dataset, and in doing so, we have reduced the dimensionality of the dataset from five dimensions with real-world meaning to two dimensions, x and y , with no inherent real-world connection. Instead, the two resulting dimensions now reflect linear combinations of the underlying data points; they are useful summaries of the underlying data, but are not easily interpretable.

We could reduce the dimensionality even further by projecting the dataset onto a line to create a single principal component, as illustrated in *Figure 13.8*:

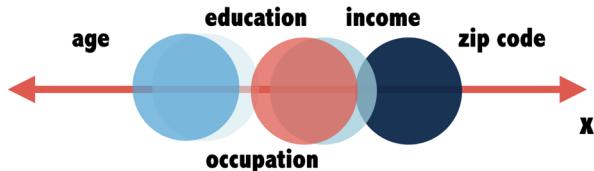


Figure 13.8: The first principal component captures the dimension with the greatest variance

In this example, the PCA approach extracted one hybrid feature from the dataset's five original dimensions. Age and education are treated as somewhat redundant, as are occupation and income.

Additionally, age and education have an opposite impact on the new feature than zip code—they are pulling the value of x in opposite directions. If this one-dimensional representation loses too much of the information stored within the original five features, the earlier approaches with two or three components could be used instead. As with many techniques in machine learning, there is a balance between over- and underfitting the data. We'll see this reflected in a real-world example shortly.

Before applying PCA, it's important to know that principal components are identified by a deterministic algorithm, which means that the solution is consistent every time the process is completed on a given dataset. Each component vector is also always orthogonal, or perpendicular, to all previous component vectors. The first principal component captures the dimension of highest variance, the next captures the next most, and so on, until a principal component has been constructed for each in the original dataset, or the algorithm stops early when the desired number of components has been reached.

Example – Using PCA to reduce highly dimensional social media data

As mentioned previously, PCA is a feature extraction technique that reduces the dimensionality of a dataset by synthesizing a smaller set of features from the complete set. We'll apply this technique to the social media data first described in *Chapter 9, Finding Groups of Data – Clustering with k-means*. You may recall that this dataset includes counts of 36 different words that appeared on the social media pages of 30,000 teens in the United States. The words reflect various interests and activities such as sports, music, religion, and shopping, and although 36 is not an unreasonable number for most machine learning algorithms to handle, if we had more—perhaps hundreds of features—some algorithms might begin to struggle with the curse of dimensionality.

We'll use the tidyverse suite of functions to read and prepare the data. First, we'll load the package, and use its `read_csv()` function to read the social media data as a tibble:

```
> library(tidyverse)
> sns_data <- read_csv("snsdata.csv")
```

Next, we will `select()` only the columns corresponding to the features recording the number of times 36 words were used in each social media profile. The notation here selects from the column named `basketball` through the column named `drugs` and saves the result in a new tibble called `sns_terms`:

```
> sns_terms <- sns_data |> select(basketball:drugs)
```

The PCA technique will only work with a matrix of numeric data. However, because each of the resulting 36 columns is a count, no more data preparation is needed. If the dataset included categorical features, it would be necessary to convert these to numeric before proceeding.

Base R includes a built-in PCA function called `prcomp()`, which becomes slow to run as datasets get larger. We'll use a drop-in substitute from the `irlba` package by Bryan W. Lewis, which can be stopped early to return only a subset of the full set of potential principal components. This truncated approach, plus the use of a generally more efficient algorithm, makes the `irlba_prcomp()` function much more speedy than `prcomp()` on larger datasets, while keeping the syntax and compatibility virtually identical to the base function, in case you are following along with older online tutorials.



The `irlba` package gets its strange-seeming name from the technique it uses: the “implicitly restarted Lanczos bidiagonalization algorithm” developed by Jim Baglama and Lothar Reichel. For more information on this approach, see the package vignette using the `vignette("irlba")` command.

Before beginning, we'll set the random seed to an arbitrary value of 2023 to ensure your results match the book. Then, after loading the required package, we'll pipe the `sns_terms` dataset into the PCA function. The three parameters allow us to limit the result to the first 10 principal components while also standardizing the data by centering each feature around zero and scaling them to have a variance of one. This is usually desirable for much the same reason it is in the k-Nearest Neighbors approach: it prevents features with larger variance from dominating the principal components. The results are saved as an object named `sns_pca`:

```
> set.seed(2023)
> library(irlba)
> sns_pca <- sns_terms |>
  prcomp_irlba(n = 10, center = TRUE, scale = TRUE)
```



Although PCA is a deterministic algorithm, the sign—positive or negative—is arbitrary and can vary from run to run, hence the need to set the random seed beforehand to guarantee reproducibility. This Stack Exchange thread has more information on this phenomenon: <https://stats.stackexchange.com/questions/88880/>

Recall that each component in the PCA captures a decreasing amount of the dataset's variance and that we requested 10 of the possible 36 components. A **scree plot**, named after the “scree” landslide patterns that form at the bottom of cliffs, helps visualize the amount of variance captured by each component and may thus help to determine the optimal number of components to use. R’s built-in `screeplot()` function can be applied to our result to create such a plot. The four parameters supply our PCA result, indicate that we want to plot all 10 components, use a line graph rather than a bar plot, and apply the plot title:

```
> screeplot(sns_pca, ncp = 10, type = "lines",
  main = "Scree Plot of SNS Data Principal Components")
```

The resulting plot appears as follows:

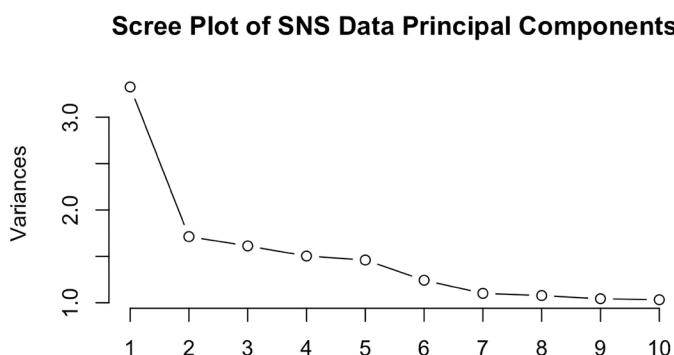


Figure 13.9: The scree plot depicting the variance of the first 10 principal components of the social media dataset

The scree plot shows that there is a substantial drop in the variance captured between the first and second components. The second through fifth components capture approximately the same amount of variance, and then there are additional substantial drops between the fifth and sixth components and between the sixth and seventh components. The seventh through tenth components capture approximately the same amount of variance. Based on this result, we might decide to use one, five, or six principal components as our reduced-dimensionality dataset. We can see this numerically by applying the `summary()` function to our PCA results object:

```
> summary(sns_pca)
```

Importance of components:					
	PC1	PC2	PC3	PC4	PC5
Standard deviation	1.82375	1.30885	1.27008	1.22642	1.20854
Proportion of Variance	0.09239	0.04759	0.04481	0.04178	0.04057
Cumulative Proportion	0.09239	0.13998	0.18478	0.22657	0.26714
	PC6	PC7	PC8	PC9	PC10
Standard deviation	1.11506	1.04948	1.03828	1.02163	1.01638
Proportion of Variance	0.03454	0.03059	0.02995	0.02899	0.02869
Cumulative Proportion	0.30167	0.33227	0.36221	0.39121	0.41990

The output shows the standard deviation, the proportion of the total variance, and the cumulative proportion of variance for each of the 10 components (labeled PC1 to PC10). Because standard deviation is the square root of variance, squaring the standard deviations produces the variance values depicted in the scree plot; for example, $1.82375^2 = 3.326064$, which is the value shown for the first component in the scree plot. A component's proportion of variance is its variance out of the total for all components—not only the 10 shown here, but also the remaining 26 that we could have created. Therefore, the cumulative proportion of variance maxes out at 41.99% rather than the 100% that would be explained by all 36 components.

Using PCA as a dimensionality reduction technique requires the user to determine how many components to keep. In this case, if we choose five components, we will capture 26.7% of the variance, or one-fourth of the total information in the original data. Whether or not this is sufficient depends on how much of the remaining 73.3% of variance is signal or noise—something that we can only determine by attempting to build a useful learning algorithm. One thing that makes this process easier is that regardless of how many components we ultimately decide upon, our PCA process is complete; we can simply use as few or as many of the 10 components as desired. For instance, there is no need to re-run the algorithm to obtain the best three versus the best seven components; finding the first seven components will naturally already include the best three and the results will be identical. In a real-world application of PCA, it may be wise to test several different cut points.

For simplicity here, we'll reduce the original 36-dimension dataset to five principal components. By default, the `irlba_prcmp()` function automatically saves a version of the original dataset that has been transformed into the lower-dimension space. This is found in the resulting `sns_pca` list object with the name `x`, which we can examine with the `str()` command:

```
> str(sns_pca$x)

num [1:30000, 1:10] 1.448 -3.492 0.646 1.041 -4.322 ...
- attr(*, "dimnames")=List of 2
..$ : NULL
..$ : chr [1:10] "PC1" "PC2" "PC3" "PC4" ...
```

The transformed dataset is a numeric matrix with 30,000 rows like the original dataset but 10 rather than 36 columns with names from PC1 to PC10. We can see this more clearly by using the `head()` command output to see the first few rows:

```
> head(sns_pca$x)

      PC1      PC2      PC3      PC4      PC5
[1,] -1.4477620  0.07976310 0.3357330 -0.3636082  0.03833596
[2,]  3.4922144  0.36554520 0.7966735 -0.1871626  0.57126163
[3,] -0.6459385 -0.67798166 0.8000251  0.6243070  0.25122261
[4,] -1.0405145  0.08118501 0.4099638 -0.2555128 -0.02620989
[5,]  4.3216304 -1.01754361 3.4112730 -1.9209916 -0.43409869
[6,]  0.2131225 -0.65882053 1.6215828  0.9372545  1.47217369

      PC6      PC7      PC8      PC9      PC10
[1,] -0.01559079  0.007278589 -0.004582346  0.19226144  0.08086065
[2,]  3.02758235 -0.306304037 -1.142422251  0.72992534  0.11203923
[3,] -0.40751994  0.454614417  0.704544996 -0.43734980 -0.07735574
[4,]  0.27837411  0.462898314 -0.175251793 -0.08843005  0.26784326
[5,] -1.11734548 -2.122420077 -2.287638056  2.19992650 -0.26536161
[6,]  0.04614790 -0.654207687  0.285263646  0.69439745 -0.89649127
```

Recall that in the original dataset, each of the 36 columns indicated the number of times a particular word appeared in the social media profile text. If we standardized the data to have a mean of zero, as we did in *Chapter 9, Finding Groups of Data – Clustering with k-means*, and has been done here for the principal components, then positive and negative values indicate profiles with higher or lower-than-average values, respectively. The trick is that each of the 36 original columns had an obvious interpretation, whereas the PCA results are without apparent meaning.

We can attempt to understand the components by visualizing the **PCA loadings**, or the weights that transform the original data into each of the principal components. Large loadings are more important to a particular component. These loadings are found in the `sns_pca` list object with the name `rotation`.

This is a numeric matrix with 36 rows corresponding to each of the original columns in the dataset and 10 columns that provide the loadings for the principal components. To construct our visualization, we will need to pivot this data such that it has one row per social media term per principal component; that is, we will have $36 * 10 = 360$ rows in the longer version of the dataset.

The following command uses two steps to create the required long dataset. The first step creates a tibble including a `SNS_Term` column with one row for each of the 36 terms as well as the `sns_pca$rotation` matrix, which is converted into a tibble using `as_tibble()`. The combined tibble, with 11 columns and 36 rows, is piped into the `pivot_longer()` function, which pivots the table from wide to long format. The three parameters tell the function to pivot the 10 columns from PC1 to PC10, with the former column names now becoming the rows for a column named `PC` and the former column values now becoming row values of a column named `Contribution`. The full command creates a tibble with 3 columns and 360 rows:

```
> sns_pca_long <- tibble(SNS_Term = colnames(sns_terms),
                           as_tibble(sns_pca$rotation)) |>
  pivot_longer(PC1:PC10, names_to = "PC", values_to = "Contribution")
```

The `ggplot()` function can now be used to plot the most important contributing terms for a given principal component. For example, to look at the third principal component, we'll `filter()` the rows to limit to only PC3, select the top 15 largest contribution values—considering both positive and negative values using the `abs()` absolute value function—and mutate the `SNS_Term` to reorder by the contribution amount. Ultimately, this is piped into `ggplot()` with a number of adjustments to the formatting:

```
> sns_pca_long |>
  filter(PC == "PC3") |>
  top_n(15, abs(Contribution)) |>
  mutate(SNS_Term = reorder(SNS_Term, Contribution)) |>
  ggplot(aes(SNS_Term, Contribution, fill = SNS_Term)) +
  geom_col(show.legend = FALSE, alpha = 0.8) +
  theme(axis.text.x = element_text(angle = 90, hjust = 1,
                                    vjust = 0.5), axis.ticks.x = element_blank()) +
  labs(x = "Social Media Term",
       y = "Relative Importance to Principal Component",
       title = "Top 15 Contributors to PC3")
```

The result is shown in the plot that follows. Because the terms with positive and negative impacts seem to be split across subjects related to sex, drugs, and rock and roll, one might argue that this principle component has identified a stereotypical dimension of teen identity:

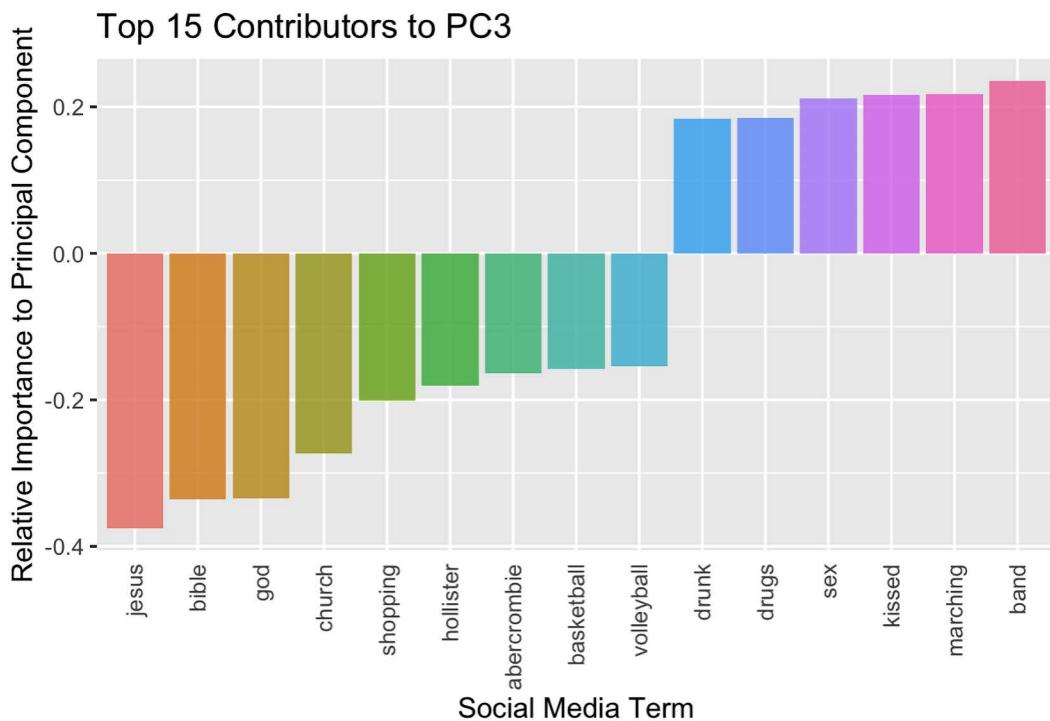


Figure 13.10: The top 15 terms contributing to PC3

By repeating the above ggplot code for the four other principal components among the first five, we observe similar distinctions, as shown in the figure that follows:

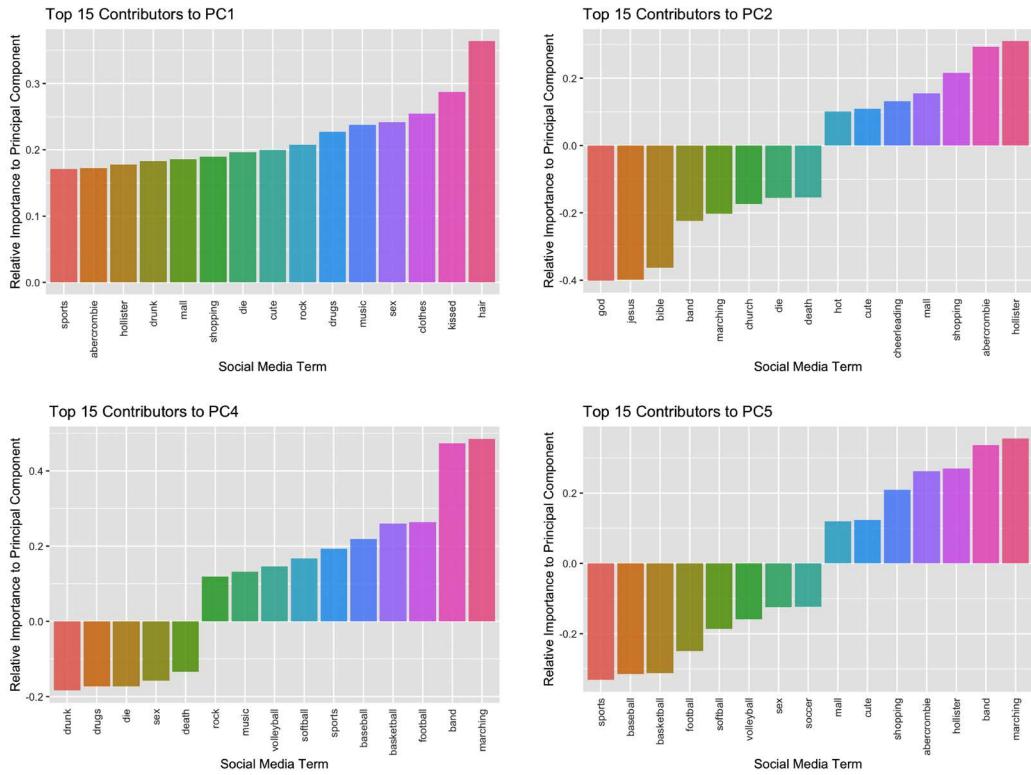


Figure 13.11: The top 15 terms contributing to the other four principal components

PC1 is particularly interesting, as every term has a positive impact; this may be distinguishing people who have anything versus nothing at all on their social media profiles. PC2 seems to favor shopping-related terms, while PC4 seems to be a combination of music and sports, without sex and drugs. Lastly, it seems that PC5 may be distinguishing between sports and non-sports-related terms. Examining the charts in this way will help to understand each component's impact on the predictive model.



The previous visualization method was adapted from an outstanding tutorial from Julia Silge, author of *Text Mining with R: A Tidy Approach* (2017). For a deeper dive into PCA, see <https://juliasilge.com/blog/stack-overflow-pca/>.

An understanding of principal component analysis is of little value if the technique is not useful for building machine learning models. In the previous example, we reduced the dimensionality of a social media dataset from 36 to 10 or fewer components. By merging these components back into the original dataset, we can use them to make predictions about a profile's gender or number of friends. We'll begin by using the `cbind()` function to combine the first four columns of the original data frame with the transformed profile data from the PCA result:

```
> sns_data_pca <- cbind(sns_data[1:4], sns_pca$x)
```

Next, we'll build a linear regression model predicting the number of social media friends as a function of the first five principal components. This modeling approach was introduced in *Chapter 6, Forecasting Numeric Data – Regression Methods*. The resulting output is as follows:

```
> m <- lm(friends ~ PC1 + PC2 + PC3 + PC4 + PC5, data = sns_data_pca)
> m
```

```
Call:
lm(formula = friends ~ PC1 + PC2 + PC3 + PC4 + PC5, data = sns_data_pca)
```

Coefficients:

(Intercept)	PC1	PC2	PC3	PC4
30.1795	1.9857	0.9748	-2.5230	1.1160
	PC5			
	0.8780			

Because the value of the intercept is approximately 30.18, the average person in this dataset has about 30 friends. People with higher values of PC1, PC2, PC4, and PC5 are expected to have more friends, while higher values of PC3 are associated with fewer friends, assuming all else is equal. For example, for each unit increase in PC2, we would anticipate about one additional friend on average. Given our understanding of the components, these findings make sense; the positive values of PC2, PC3, and PC5 were associated with more social activities. In contrast, PC3 was about sex, drugs, and rock and roll, which may be somewhat antisocial.

Although this is a very simple example, PCA can be used in the same way with much larger datasets. In addition to mitigating the curse of dimensionality, it also has the benefit of reducing complexity. For instance, a dataset with a very large number of predictors may be too computationally expensive for k-Nearest Neighbors or an artificial neural network to run as is, but by selecting a smaller number of principal components, such techniques may be within reach.

Feel free to experiment with PCA and contrast this type of feature extraction with other feature selection methods like filters and wrappers; you may find that you have better luck with one approach or the other. Even if you choose not to use dimensionality reduction, there are other problems with highly dimensional data that you will discover in the next section.

Making use of sparse data

As datasets increase in dimension, some attributes are likely to be **sparse**, which means most observations do not share values of the attribute. This is a natural consequence of the curse of dimensionality in which this ever-increasing detail turns observations into outliers identified by their unique combination of attributes. It is very uncommon for sparse data to have any specific value, or perhaps even any value at all—as was the case in the sparse matrices for text data found in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*, and the sparse matrices for shopping cart data in *Chapter 8, Finding Patterns – Market Basket Analysis Using Association Rules*.

This is not the same as missing data, where typically a relatively small portion of values are unknown. In sparse data, most values are known, but the number of interesting, meaningful values is dwarfed by an overwhelming number of values that add little value to the learning task. With missing data, machine learning algorithms struggle to learn something from nothing; with sparse data, machine learning algorithms struggle to find the needle in the haystack.

Identifying sparse data

Sparse data can appear in several interrelated forms. Perhaps the most encountered form is categorical, in which a single feature has a very large number of levels or categories, with some having extremely small counts relative to the others. Features like this are said to have high **cardinality** and will lead to sparse data problems when fed to a learning algorithm. An example of this is zip codes; in the United States, there are over 40,000 postal codes, with some having more than 100,000 residents and others having less than 100. Consequently, if a sparse zip code feature is included in a modeling project, the learning algorithm is likely to struggle to find the balance between ignoring and overemphasizing areas with few residents.

Categorical features with many levels are often expressed as a series of binary features with one feature per level. We have used such features many times when we manually constructed binary dummy variables, and many learning algorithms do the same automatically for categorical data. This can lead to a situation in which binary features are sparse as the 1 values are overwhelmed by the 0 values.

For example, in a zip code dataset for the U.S. population, a tiny fraction of the 330 million residents will fall into each of the 40,000 postal codes, thus making each binary zip code feature highly sparse and difficult for a learning algorithm to use.

Many forms of so-called “big” data are inherently highly dimensional and sparse. Sparseness is closely related to the curse of dimensionality. Just like how the ever-expanding universe creates greater voids of empty space between objects, one might argue that every dataset becomes sparse as more dimensions are added. Text data is usually sparse because each word can be treated as a dimension and there are countless words that can appear, each having a low probability of appearing in a specific document. Other big data forms, like DNA data, transactional market basket data, and image data, also often exhibit the problem of sparseness. Unless the dataset’s density is increased, many learning algorithms will struggle to make use of the rich, big dataset.

Even a simple numeric range of data can be sparse. This occurs when the distribution of numeric values is wide, which leads to some ranges of the distribution having a very low density. Income is one example of this, because the values generally become increasingly sparse for higher incomes. This is closely related to the problem of outliers, but here we clearly hope to model the outlying values. Sparse numeric data can also be found in cases when the numbers have been stored in an overly specific degree of precision. For example, if age values are stored with decimals rather than integers, such as 24.9167 and 36.4167 rather than simply 24 and 36, this creates an implied void between numbers that some learning algorithms may struggle to ignore. For instance, a decision tree might distinguish between people that are 24.92 and 24.90 years old—probably more likely to be related to overfitting than a meaningful distinction in the real world.

Reducing the sparseness of a dataset manually can assist a learning algorithm with identifying the important signals and ignoring the noise. The approach used depends on the type and degree of sparse data as well as the modeling algorithm used. Some algorithms are better than others at handling certain types of sparse data. For example, naive Bayes performs relatively well with sparse categorical data, regression methods do relatively well with sparse numeric data, and decision trees tend to struggle with sparse data in general due to their preference for features with a larger number of categories. More sophisticated methods like deep neural networks and boosting can help, but in general, it is better if the dataset can be made denser prior to the learning process.

Example – Remapping sparse categorical data

As we have seen in prior chapters, when adding a categorical feature to a dataset, it is usually transformed into a set of binary variables equal to the number of levels of the original feature using dummy or one-hot encoding.

For example, if there are 40,000 zip codes in the United States, the machine learning algorithm would have 40,000 binary predictors for this feature. This is called a **one-of-n mapping** because only one of the 40,000 features would have a value of 1 while the remainder would have values of 0—a case of extreme growth in dimensionality and sparseness.

To increase the density of a one-of-n mapping, an **m-of-n mapping** may be used instead, which reduces the n binary variables to a smaller set of m variables. For example, with zip codes, instead of creating a 40,000-level feature with one level per zip code, one might choose to map into 100 levels by using the first two digits of the zip code from 00 to 99. Similarly, if it would create too much sparseness to include a binary feature for each of the 200 countries in the world, it might be possible to map countries to a smaller set of continents, like Europe, North America, and Asia, instead.

When creating an m-of-n mapping, it is best if the groupings represent a shared underlying characteristic, but it is possible to use other approaches as well. Domain knowledge can be helpful for creating a remapping that reflects the shared characteristics of the more granular units. In the absence of domain expertise, the following methods may be appropriate:

- Leave the larger categories as is and group only the categories with small numbers of observations. For example, zip codes for dense urban areas could be included directly, but sparse rural zip codes could be grouped into larger geographic areas.
- Examine the impact of the categories on the target variable by creating a two-way cross table or computing the average outcome by level and group levels that have a similar impact on the response variable. For example, if certain zip codes are more likely to default on a loan, create a new category composed of these zip codes.
- As a more sophisticated variant of the previous method, it may also be possible to build a simple machine learning model that predicts the target using the highly dimensional feature and then group levels of the feature that have a similar relationship with the target or with other predictors. Simple methods like regression and decision trees would be ideal for this approach.

Once a remapping strategy has been chosen, helpful functions for recoding categorical variables can be found in the `forcats` package (<https://forcats.tidyverse.org>), which is part of the base set of packages comprising the tidyverse. The package includes options for automatically recoding categorical variables with sparse levels, or manually recoding if a more guided approach is desired. Detailed information about the package is available in the *R for Data Science* chapter at <https://r4ds.hadley.nz/factors.html>.

We'll examine a couple of approaches for remapping using the Titanic dataset and the passenger titles created in *Chapter 12, Advanced Data Preparation*. Because the `forcats` package is included in the base tidyverse, it can be loaded with the entire suite or on its own using the `library(forcats)` command. We'll begin by loading the tidyverse, reading the Titanic dataset, and then examining the levels of the title feature:

```
> library(tidyverse)
> titanic_train <- read_csv("titanic_train.csv") |>
  mutate>Title = str_extract(Name, "[A-z]+\\.") |>
  mutate>Title = str_replace_all>Title, "[, \\.]", "") |>
> table(titanic_train$title, useNA = "ifany")
```

Capt	Col	Don	Dr	Jonkheer	Lady	Major
1	2	1	7	1	1	2
Master	Miss	Mlle	Mme	Mr	Mrs	Ms
40	182	2	1	517	125	1
Rev	Sir	<NA>				
6	1	1				

In the previous chapter, we used base R's `recode()` function to combine the variants of *Miss*, such as *Ms*, *Mlle*, and *Mme*, into a single group. The `forcats` package includes an `fct_collapse()` function, which is more convenient to use for categorical features with a large number of levels. We'll use it here to create an m-of-n mapping that creates groups based on knowledge of the titles' real-world meanings. Note that several of the new categories are one-to-one mappings of the previous categories, but by including a vector of labels, we can map several of the old levels to a single new level, as follows:

```
> titanic_train <- titanic_train |>
  mutate>TitleGroup = fct_collapse>Title,
  Mr = "Mr",
  Mrs = "Mrs",
  Master = "Master",
  Miss = c("Miss", "Mlle", "Mme", "Ms"),
  Noble = c("Don", "Sir", "Jonkheer", "Lady"),
  Military = c("Capt", "Col", "Major"),
  Doctor = "Dr",
  Clergy = "Rev",
```

```

    other_level = "Other")
) |>
mutate>TitleGroup = fct_na_value_to_level>TitleGroup,
           level = "Unknown"))

```

Examining the new categorization, we see that the 17 original categories have been reduced to 9:

```
> table(titanic_train>TitleGroup)
```

	Military	Noble	Doctor	Master	Miss	Mr	Mrs
	5	4	7	40	186	517	125
	Clergy	Unknown					
	6	1					

If we had a much larger set of levels, or in the absence of knowledge of how categories should be grouped, we can leave large categories as is and group the levels with few examples. The `forcats` package includes a simple function for examining the levels of our feature. Although this can also be done with base R functions, the `fct_count()` function provides a sorted list of the feature levels and their proportions of the overall total:

```
> fct_count(titanic_train>Title, sort = TRUE, prop = TRUE)
```

	f	n	p
1	Mr	517	0.580
2	Miss	182	0.204
3	Mrs	125	0.140
4	Master	40	0.0449
5	Dr	7	0.00786
6	Rev	6	0.00673
7	Col	2	0.00224
8	Major	2	0.00224
9	Mlle	2	0.00224
10	Capt	1	0.00112
11	Don	1	0.00112
12	Jonkheer	1	0.00112
13	Lady	1	0.00112
14	Mme	1	0.00112

15 Ms	1 0.00112
16 Sir	1 0.00112
17 NA	1 0.00112

This output can inform groupings based on a minimum number or minimum proportion of observations. The `forcats` package has a set of `fct_lump()` functions to help with this process of “lumping” factor levels into an “other” group. For example, we might take the top three levels and treat everything else as other:

```
> table(fct_lump_n(titanic_train$title, n = 3))
```

Miss	Mr	Mrs	Other
182	517	125	66

Alternatively, we can lump together all levels with less than one percent of the observations:

```
> table(fct_lump_prop(titanic_train$title, prop = 0.01))
```

Master	Miss	Mr	Mrs	Other
40	182	517	125	26

Lastly, we might choose to lump together all levels with less than five observations:

```
> table(fct_lump_min(titanic_train$title, min = 5))
```

Dr	Master	Miss	Mr	Mrs	Rev	Other
7	40	182	517	125	6	13

The choice of which of these three functions to use, as well as the appropriate parameter value, will depend on the dataset used and the desired number of levels for the m-of-n mapping.

Example – Binning sparse numeric data

While many machine learning methods handle numeric data without trouble, some approaches like decision trees are more likely to struggle with numeric data, especially when it exhibits some of the characteristics of sparseness. A common solution to this problem is called **discretization**, which converts a range of numbers into a smaller number of discrete categories called bins. We encountered this method previously in *Chapter 4, Probabilistic Learning – Classification Using Naïve Bayes*, as we discretized the numeric data to work with the naive Bayes algorithm. Here, we will apply a similar approach, using modern tidyverse methods, to reduce the dimensionality of the number range to help address the tendency of some methods to over- or underfit to sparse numeric data.

As is the case with many machine learning approaches, ideally one would apply subject-matter expertise to determine the cut points for discretizing a numeric range. For example, on a range of age values, perhaps meaningful break points could occur between well-established childhood, adulthood, and elderly age groups, to reflect the impact of these age values in the real world. Similarly, bins may be created for salary levels such as lower, middle, and upper class.

In the absence of real-world knowledge of important categories, it is often advisable to use cut points that reflect natural percentiles of data or intuitive increments of values. This may mean dividing a range of numbers using strategies such as:

- Creating groups based on tertiles, quartiles, quintiles, deciles, or percentiles that contain equal proportions of examples (33%, 25%, 20%, 10%, or 1%).
- Using familiar cut points for the underlying range of values, such as grouping time values by hours, half hours, or quarter hours; grouping 0-100 scale values by fives, tens, or twenty-fives; or bucketing large numeric ranges like income by large multiples of 10 or 25.
- Applying the notion of log scaling to skewed data, so that the bins are proportionally wider for the skewed portion of the data where the values are sparser; for example, income might be bucketed into groups of 0-10,000 followed by 10,000-100,000, then 100,000-1,000,000, and 1,000,000 or more.

To illustrate these approaches, we'll apply discretization techniques to the fare values in the Titanic dataset used previously. The `head()` and `summary()` functions illustrate that the values are highly granular and highly sparse on the high end due to their severe right skew:

```
> head(titanic_train$Fare)
[1] 7.2500 71.2833 7.9250 53.1000 8.0500 8.4583

> summary(titanic_train$Fare)
  Min.  1st Qu.   Median     Mean  3rd Qu.    Max.
  0.00     7.91    14.45   32.20   31.00  512.33
```

Suppose we are most interested in the difference between first class and other passengers and that we assume that the top 25 percent of fares reflect first-class tickets. We could easily create a binary feature using the tidyverse `if_else()` function as follows. If the fare has a value of at least £31, which is the value for the third quartile, then we'll assume it is a first-class fare and assign a value of 1 to the binary-coded `fare_firstclass` feature; if not, it receives a 0 value. The `missing` parameter tells the function to assign the value 0 if the fare was missing, under the assumption that first-class fares are very unlikely to be unknown:

```
> titanic_train <- titanic_train |> mutate(  
  fare_firstclass = if_else(Fare >= 31, 1, 0, missing = 0)  
)
```

This reduces a feature with nearly 250 distinct values into a new feature with only two:

```
> table(titanic_train$fare_firstclass)  
  
 0   1  
666 225
```

Although this was a very simple example, it's a first step toward more complex binning strategies. The `if_else()` function, although simple here, would be unwieldy to use for creating a new feature with more than two levels. This would require nesting `if_else()` functions within each other, which quickly becomes difficult to maintain. Instead, a tidyverse function called `case_when()` allows the construction of a more complex series of checks to determine the result.

In the code that follows, the fare data is binned into three levels corresponding roughly to first-, second-, and third-class fare levels. The `case_when()` statement is evaluated as a series of ordered if-else statements. The first statement checks whether the fare is at least 31 and assigns these examples the first-class category. The second can be read as an else-if statement; that is, if the first statement is not true—the “else”—we check “if” the fare is at least 15 and assign the second-class level if true. The final statement is the ultimate “else” as `TRUE` always evaluates to true, and thus all records not categorized by the first and second lines are assigned the third-class level:

```
> titanic_train <- titanic_train |>  
  mutate(  
    fare_class = case_when(  
      Fare >= 31 ~ "1st Class",  
      Fare >= 15 ~ "2nd Class",  
      TRUE ~ "3rd Class"  
    )  
)
```

The resulting feature has three levels as expected:

```
> table(titanic_train$fare_class)  
  
1st Class 2nd Class 3rd Class  
225        209       457
```

In the case that we have zero understanding of the real-world meaning of the fares, such as the knowledge of first, second, and third-class fares, we might instead apply the discretization heuristics described previously, which use natural percentiles or intuitive cut points of values instead of meaningful groups.

The `cut()` function is included in base R and provides a simple method for creating a factor from a numeric vector. The `breaks` parameter specifies the cut points for the numeric range, shown as follows for a three-level factor that matches the previous discretization. The `right = FALSE` parameter indicates that the levels should not include the rightmost, or highest, value and the `Inf` break point indicates that the final category can span the range of values from 31 to infinity. The resulting categories are identical to the prior result, but use different labels:

```
> table(cut(titanic_train$Fare, breaks = c(0, 15, 31, Inf),
             right = FALSE))

[0,15) [15,31) [31,Inf)
 457     209     225
```

By default, `cut()` sets labels for factors that indicate the range of values falling into each level. Square brackets indicate that the bracketed number is included in the level, while parentheses indicate a number that is not included. A `labels` parameter can be assigned a vector of factor labels for the result, if desired.

The `cut()` function becomes more interesting when combined with a sequence of values generated by the `seq()` function. Here, we create levels for the 11 ranges of values from 0 to 550 in increments of 50:

```
> table(cut(titanic_train$Fare, right = FALSE,
             breaks = seq(from = 0, to = 550, by = 50)))

[0,50)  [50,100) [100,150) [150,200) [200,250) [250,300)
 730      108       24        9       11        6

[300,350) [350,400) [400,450) [450,500) [500,550)
   0         0         0         0         3
```

Using evenly wide intervals here reduces the dimensionality but doesn't solve the problem of sparseness. The first two levels contain most of the examples, but the remainder have very few, or even zero in some cases.

As an alternative to having equally sized intervals, we can construct bins with an equal number of examples. We have used the `quantile()` function in previous chapters to identify the cut points for quintiles and percentiles, but we would still need to use these values with a `cut()` function to create the factor levels. The following code creates five bins for the quintiles, but could be adapted for quartiles, deciles, or percentiles:

```
> table(cut(titanic_train$Fare, right = FALSE,
             breaks = quantile(titanic_train$Fare,
                                probs = seq(0, 1, 0.20))))
```

[0,7.85)	[7.85,10.5)	[10.5,21.7)	[21.7,39.7)	[39.7,512)
166	173	196	174	179

Note that the bins do not contain exactly the same number of examples due to the presence of ties.

The tidyverse also includes a function for creating quantile-based groups, which may be easier to use in some cases. This `ntile()` function divides the data into `n` groups of equal size. For example, it can create five groups as follows:

```
> table(ntile(titanic_train$Fare, n = 5))
```

1	2	3	4	5
179	178	178	178	178

Because the function assigns the groups numeric labels, it is important to convert the resulting vector to a factor. This can be done directly in a `mutate()` statement:

```
> titanic_train <- titanic_train |>
  mutate(fare_level = factor(ntile(Fare, n = 11)))
```

The resulting feature has 11 equally proportioned levels:

```
> table(titanic_train$fare_level)
```

1	2	3	4	5	6	7	8	9	10	11
81	81	81	81	81	81	81	81	81	81	81

Although the level still has numeric labels, because the feature has been coded as a factor, it will still be treated as categorical by most R functions. Of course, it is still important to find the right balance between too few and too many levels.

Handling missing data

The teaching datasets used for examples in previous chapters rarely had the problem of missing data, where a value that should be present is instead absent. The R language uses the special value `NA` to indicate these missing values, which cannot be handled natively by most machine learning functions. In *Chapter 9, Finding Groups of Data – Clustering with k-means*, we were able to replace missing values with a guess of the true value based on other information available in the dataset in a process called imputation. Specifically, the missing age values of high school students were imputed with the average age of students that had the same graduation year. This provided a reasonable estimate of the unknown age value.

Missing data is a much greater problem in real-world machine learning projects than would be expected given its rarity so far. This is not only because real-world projects are messier and more complex than simple textbook examples. Additionally, as datasets increase in size—as they include more rows or more columns—a relatively small proportion of missingness will cause more problems, as it becomes more likely that any given row or any given column contains at least one missing value. For example, even if the rate of missingness is only one percent, in a dataset with 100 columns, we would expect the average row to have one missing value. In this case, simply excluding all rows with missing values would drastically reduce the size of the dataset to the point of nothingness!

In fields like economics, biostatistics, and the social sciences, the gold standard approach to missing data is **multiple imputation**, which uses statistical modeling or machine learning techniques to impute all the missing feature values given the non-missing feature values. Because this tends to decrease the variability of the data, and thus inflates the certainty of predictions, modern multiple imputation software tends to add random variation to the imputed values to avoid biasing the inferences made from the dataset. R has many packages for performing multiple imputation, such as:

- `mice`: Multivariate Imputation by Chained Equations
- `Amelia`: A Program for Missing Data (named after the famous pilot Amelia Earhart, who went missing in 1937 during an attempt to become the first female pilot to fly around the globe)
- `Simputation`: Simple Imputation, which attempts to simplify missing data handling via the use of tidyverse-compatible functions

- `missForest`: Nonparametric Missing Value Imputation using Random Forest, a package that uses state-of-the-art machine learning methods to impute any type of data, even types with complex, nonlinear relationships among the features

Despite the wealth of multiple imputation software tools, in comparison to projects in traditional statistics and the social sciences, machine learning projects apply simpler methods for handling missing data. This is because the goals and considerations differ. Machine learning projects tend to focus on methods that work on very large datasets and facilitate prediction on a future, unseen test set, even if certain statistical assumptions are violated. On the other hand, the more formal methods of the social sciences focus on strategies that tend to be more computationally intensive but lead to unbiased estimates for inference and hypothesis testing. Keep this distinction in mind while reading the sections that follow, which cover common practical techniques for handling missing data, but are generally not advisable for formal scientific analysis.

Understanding types of missing data

Not all missing data is created equally, and some types are more problematic than others. For this reason, when preparing data with missing values, it is useful to consider the underlying reasons why a particular value is missing. Try to picture yourself inside the process that generated the dataset and ask yourself why certain values were left blank. Is there a logical reason it is missing? Or, was it left blank purely by mistake or chance alone? Answering these questions helps inform the solution for replacing the missing values in a responsible manner. The answers to these questions also distinguish three different types of missing data, from least problematic to most severe:

1. **Data missing completely at random (MCAR)** is independent of the other features and its own value; in other words, it would not be possible to predict whether any particular value is missing. The missingness may be caused due to a random data entry error, or some other process that randomly skips the value. Missing completely at random can be imagined as a completely unpredictable process that takes the final matrix of data and randomly selects cells to delete.
2. **Data missing at random (MAR)** may depend on other features but not on the underlying value, which means that certain, predictable rows are more likely than others to contain missing values. For example, households in certain geographic regions may be less willing to report their household income, but assuming they do disclose such information, they do so honestly. Essentially, MAR implies that the missing values are randomly selected after controlling for the underlying factor or factors causing the missingness.

3. **Data missing not at random (MNAR)** is missing due to a reason related to the missing value itself. This data is in essence censored from the dataset for some reason impossible to discern from the other features in the dataset. For instance, poorer individuals may feel less comfortable sharing their income, so they simply leave it blank. Another example might be a temperature sensor that reports a missing value for extremely high or low temperatures. It is probable that most real-world missing values are MNAR, as there is usually some unmeasured, hidden mechanism causing the missingness. Very little is truly random in the real world.

Imputation methods work well for the first two types of missing data. Although one might be led to believe that MCAR data is the most challenging to impute due to its independence and unpredictability, it is actually the ideal type of missing data to handle. Even though the missingness is completely random, the values that have been randomly hidden may be predictable given the other available features. Stated differently, the *missingness* itself is unpredictable, but the underlying missing *values* may be quite predictable. Similarly, MAR data is also readily predictable by the given features.

Unfortunately, NMAR data, which is perhaps the most common type of missing data, is the least capable of being predicted. Because the missing values were censored by an unknowable process, any model built on this data will have an incomplete picture of the relationship between the missing and non-missing data, and the results are likely to be biased toward the non-missing data. For example, suppose we are trying to build a model of loan default, and poorer people are more likely to leave the income field blank on the loan application. If we impute the missing incomes, the imputed values will tend to be higher than the true values, as our imputation was based only on the available data, which is missing more low values than high values. If lower-income households are more likely to default, a model that uses the biased imputed income values to predict loan outcomes will underestimate the probability of default for households that left income blank.

Due to the possibility of such bias, strictly speaking, we should only impute MCAR and MAR data. Yet, imputation may be the lesser of two imperfect options, since excluding rows with missing data from the training dataset will also bias the model if the data is not missing completely at random. Thus, despite violating statistical assumptions, machine learning practitioners often impute missing values rather than removing missing data from the dataset. The following sections demonstrate a few common strategies employed toward this end.

Performing missing value imputation

Because NA values cannot be handled directly by many R functions nor most machine learning algorithms, they must be replaced with something else, and ideally, in a way that improves the model's performance. In machine learning, this type of missing value imputation is a barrier to prediction, which means that simpler approaches that work reasonably well are favored over more complex approaches—even if the complex approaches may be more methodologically and theoretically sound.

Missing character-type data may provide the form of missingness with the simplest possible solution, as it is possible to merely treat the missing values like any other value by recoding the NA values to a literal character string like 'Missing', 'Unknown', or another label of your choosing. The string itself is arbitrary; it just needs to be consistent for each missing value within the column. For example, the Titanic dataset includes two categorical features with missing data: Cabin and Embarked. We can easily impute 'X' in place of the missing Cabin values and 'Unknown' in place of missing Embarked values as follows:

```
> titanic_train <- titanic_train |>  
  mutate(  
    Cabin = if_else(is.na(Cabin), "X", Cabin),  
    Embarked = if_else(is.na(Embarked), "Unknown", Embarked)  
)
```

Although this method has eliminated the NA values by replacing them with valid character strings, it seems as if more sophisticated approaches ought to be possible. After all, couldn't we use machine learning to predict missing values using the remaining columns in the dataset? Indeed, this is possible, as we will learn shortly. However, using this type of advanced approach may be overkill if not actually detrimental to the model's predictive performance.

The reasons that one might perform missing value imputation vary across disciplines. In traditional statistics and the social sciences, models are often used for inference and hypothesis testing rather than for predicting and forecasting. When used for inference, it is very important that the relationships among features within the dataset are preserved as carefully as possible, as statisticians seek to carefully estimate and understand each feature's individual connection to the outcome of interest. Imputing arbitrary values into the missing slots may distort these relationships—particularly in the case that the values are not missing completely at random.

Rather, more sophisticated approaches use the other available information to impute a reasonable guess as to the true value, keeping as many rows of data as possible, while also ensuring that the data's important internal relationships across features are preserved. Ultimately, this increases the **statistical power** of the analysis, which relates to the capabilities of detecting patterns and testing hypotheses.

In contrast, machine learning practitioners are often less concerned with the internal relationships among a dataset's features, and more focused on the features' relationship with an external target outcome. From this perspective, there is no strong reason to apply sophisticated imputation strategies. Such methods do not contribute new information that can be used to better predict the target because they merely reinforce internal patterns. On the other hand, by the assumption that the data is not missing at random, it may be less helpful to focus on the specific value to impute in a missing slot, and instead focus effort on trying to use the missingness itself as a predictor of the target.

Simple imputation with missing value indicators

The practice described in the previous section, in which missing categorical values were replaced with an arbitrary string like 'Missing' or 'Unknown', is one form of **simple imputation**, in which NA values are simply replaced by a constant value. For numeric features, we can use an equivalent approach. For each feature with a missing value, choose a value to impute in place of the NA values. This can be a summary statistic such as mean, median, or mode, or it may be an arbitrary number—the specific value generally doesn't matter.



Although the exact value generally doesn't matter, the most common approach may be **mean imputation**, perhaps due to the common practice of doing this in the field of traditional statistics. An alternative approach is to use a value on the same order of magnitude but outside the range of actual values found in the data. For example, for missing age values in the range of 0 to 100, you may choose to impute the value -1 or 999. Keep in mind that regardless of the value chosen, any summary statistics computed on the feature will be distorted by the imputed values.

In addition to imputing a value in place of the NA, it is especially important to create a **missing value indicator (MVI)**, which is a binary variable that indicates whether the feature value was imputed. We'll do this for the missing Titanic passenger age values using the following code:

```
> titanic_train <- titanic_train |>  
  mutate(
```

```
Age_MVI = if_else(is.na(Age), 1, 0),
Age = if_else(is.na(Age), mean(Age, na.rm = TRUE), Age)
)
```

Both the feature that has been imputed and the MVI should be included as predictors in the machine learning model. The fact that a value was missing is often an important predictor of the target, and surprisingly often, one of the strongest predictors of the target. This may not actually be unexpected under the simple belief that very little in the real world happens at random; if a value is missing, there is probably an explanation for it. For example, in the Titanic dataset, perhaps the missing age implies something about the passenger's social status or family background. Similarly, someone who refuses to report their income or occupation on a loan application may be hiding the fact that they make a very small amount of money—which may be a strong predictor of a loan default. This finding that missing values are interesting predictors is also true for larger amounts of missing data; more missingness may lead to even more interesting predictors.

Missing value patterns

Expanding upon the belief that a missing value may be a highly useful predictor, each additional missing value may contribute to our ability to forecast a specific outcome. In the case of loan application data, a person that has missing data on a single feature may have been intentionally hiding their answer, or they may have just accidentally skipped this question on the loan application form. If a person has missing data on multiple features, the latter excuse no longer applies, or perhaps it implies that they rushed through the application or are generally more irresponsible. We might assume that people with more missing data are more likely to default, but of course, we won't know until we train the model; it could just as easily be the case that people with more missing data are in fact less likely to default, perhaps because some of the loan application questions don't apply to their situation.

Suppose there actually is a pattern to be found among records with large amounts of missingness, but it is not solely based on the number of missing values, but instead, the specific features that are missing. For example, someone who is afraid to report their income because it is too low may skip related questions on a loan application, whereas a small business owner may skip a different section of questions on employment history because they do not apply to someone who is self-employed. Both cases may have roughly equal numbers of missing values, but their patterns may differ substantially.

A **missing value pattern (MVP)** can be constructed to capture such behaviors and use them as features for machine learning. A missing value pattern is essentially a character string composed of a series of MVIs, with each character in the string representing a feature with missing values. *Figure 13.12* illustrates how the process works for a simplified loan application dataset. For each of the eight features, we construct a missing value indicator that indicates whether the corresponding cell was missing:

married	age	gender	country	emp_len	income	zipcode	creditscore
NA	NA	NA	USA	10	\$54,000	90210	670
Yes	48	M	USA	6	\$99,000	46656	780
Yes	37	F	USA	5	NA	NA	NA
No	NA	M	NA	NA	NA	32817	NA

x1_mvi	x2_mvi	x3_mvi	x4_mvi	x5_mvi	x6_mvi	x7_mvi	x8_mvi
1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1
0	1	0	1	1	1	0	1

Figure 13.12: Constructing a missing value pattern begins with creating missing value indicators for each feature with missing data

These binary MVIs are then concatenated into a single string. For example, the first row would be represented by the string '11100000', which indicates that the first three features for this loan applicant were missing. The second applicant, who had no missing data, would be represented by '00000000', while the second and third would be represented by '00000111' and '01011101', respectively. The resulting `mvp` R character vector would be converted into a factor to allow a learning algorithm to use it for prediction. Each level of the factor represents a specific pattern of missingness; loan applicants that follow the same pattern may be likely to have similar outcomes.

Although missing value patterns can be extremely powerful predictors, they are not without some challenges. First and foremost, in a dataset containing k features, there are 2^k potential values of a missing value pattern. A dataset with just 10 features may have as many as 1,024 levels of the MVP predictor, while a dataset with 25 features would have over 33 million potential levels. A relatively small dataset with 50 features would have almost an uncountable number of potential MVP levels, which would make the predictor useless for modeling.

Despite this potential issue, the hope with the MVP approach is that the potentially huge number of levels avoids the curse of dimensionality due to patterns of missingness that are far from uniform or random. In other words, the MVP approach depends strongly on data that is not missing at random; we hope there is a strong underlying pattern driving the missing values, which the missing value patterns will reflect. Overall, the less heterogeneity that is present in the missingness, the more often that certain patterns of missingness appear frequently in the data. Unfortunately, even if one feature is missing completely at random, it can reduce the utility of the MVP approach because even if rows are similar on nearly all of the binary missing value indicators, if a single one differs, it will be treated as a completely different missing value pattern. To address this issue, an alternative is to use the MVI dataset with an unsupervised clustering algorithm like the k-means algorithm covered in *Chapter 9, Finding Groups of Data – Clustering with k-means*, to create clusters of people with similar patterns of missingness.

The problem of imbalanced data

One of the most challenging data issues is **imbalanced data**, which occurs when one or more class levels are much more common than the others. Many, if not most, machine learning algorithms struggle mightily to learn heavily imbalanced datasets, and although there isn't a specific threshold that determines when a dataset is too off-balance, the problems caused by the lack of balance become increasingly serious as the problem becomes more severe.

In the early stages of class imbalance, small problems are found. For instance, simple performance measures like accuracy begin to lose relevance and more sophisticated performance measures like those described in *Chapter 10, Evaluating Model Performance*, are needed. As the imbalance widens, bigger problems occur. For example, with extremely imbalanced datasets, some machine learning algorithms might struggle to predict the minority group at all. With this in mind, it might be wise to begin worrying about imbalanced data when the split is worse than 80% versus 20%, worry more when it is worse than 90% versus 10%, and assume the worst when the split is more severe than 99% to 1%.



A class imbalance can also occur if the real-world misclassification cost of one or more class levels is significantly higher or lower than the others.

Imbalanced data is a prevalent yet important challenge because many of the real-world outcomes we care to predict are significant mainly because they are both rare and costly. This includes prediction tasks to identify outcomes such as:

- Severe illnesses or diseases
- Extreme weather and natural disasters
- Fraudulent activity
- Loan defaults
- Hardware or mechanical failure
- Wealth or so-called “whale” customers

As you will soon learn, there is unfortunately no single best way to handle imbalanced classification problems like these and even the more advanced techniques are not without downsides. Perhaps the most important approach is to be aware of the problem of unbalanced data while recognizing that all solutions are imperfect.

Simple strategies for rebalancing data

If a dataset has a severe imbalance, with some class levels having too many or too few examples, a simple solution to this problem is to subtract examples from the majority classes or add examples of the minority classes. The former strategy is called **undersampling**, which in the simplest case involves discarding records at random from the majority classes. The latter approach is called **oversampling**. Ideally, one would simply collect more rows of data, but this is usually not possible. Instead, examples of the minority classes are duplicated at random until the desired class balance is achieved.

Under- and oversampling each have significant drawbacks but can be effective in certain circumstances. The main danger of undersampling is the risk of dropping examples that express small but important patterns in the data. Therefore, undersampling works best if a dataset is large enough to reduce the risk that removing a substantial portion of the majority classes will completely exclude key training examples. Moreover, it always leads to a feeling of defeat to voluntarily surrender information in the era of big data.

Oversampling avoids this disappointment by generating additional minority class examples but risks overfitting to unimportant patterns or noise in the minority cases. Both under- and oversampling have been included in more advanced **focused sampling** approaches that avoid simple random sampling in favor of favoring records that maximize the decision boundaries between the groups.

Such techniques are rarely used in practice due to their computational inefficiency and limited real-world efficacy.



For a more in-depth review of strategies for handling imbalanced data, refer to *Data Mining for Imbalanced Datasets: An Overview*, Chawla, N., 2010, in *Data Mining and Knowledge Discovery Handbook*, 2nd Edition, Maimon, O. and Rokach, L.

To illustrate resampling techniques, we'll return to the teenage social media dataset used previously in this chapter, and begin by loading and preparing it with several tidyverse commands. First, using `fct_` functions from the `forcats` package, the gender feature is recoded as a factor with `Male` and `Female` labels and the `NA` values are recoded to `Unknown`. Then, outlier ages below 13 years or greater than 20 years are replaced by `NA` values. Next, using `group_by()` in combination with `mutate()` allows us to impute the missing ages with the median age by graduation year. Lastly, we `ungroup()` the data and reorder the columns with `select()` such that our features of interest appear first in the dataset. The full command is as follows:

```
> snsdata <- read_csv("snsdata.csv") |>
  mutate(
    gender = fct_recode(gender, Female = "F", Male = "M"),
    gender = fct_na_value_to_level(gender, level = "Unknown"),
    age = ifelse(age < 13 | age > 20, NA, age)
  ) |>
  group_by(gradyear) |>
  mutate(age_imp = if_else(is.na(age),
    median(age, na.rm = TRUE), age)) |>
  ungroup() |>
  select(gender, friends, gradyear, age_imp, basketball:drugs)
```

In this dataset, males and people of unknown gender are underrepresented, which we can confirm with the `fct_count()` function:

```
> fct_count(snsdata$gender, prop = TRUE)
```

```
# A tibble: 3 × 3
  f          n      p
  <fct>    <int>  <dbl>
1 Female    22054  0.735
2 Male      5222   0.174
3 Unknown   2724   0.0908
```

One approach would be to undersample the female and male groups such that all three have the same number of records. The `caret` package, which was first introduced in *Chapter 10, Evaluating Model Performance*, includes a `downSample()` function that can perform this technique. The `y` parameter is the categorical feature with the levels to be balanced, the `x` parameter specifies the remaining columns to include in the resampled data frame, and the `yname` parameter is the name of the target column:

```
> library(caret)
> sns_undersample <- downSample(x = snsdata[2:40],
                                 y = snsdata$gender,
                                 yname = "gender")
```

The resulting dataset includes 2,724 examples of each of the three class levels:

```
> fct_count(sns_undersample$gender, prop = TRUE)

# A tibble: 3 × 3
  f          n      p
  <fct>    <int> <dbl>
1 Female    2724  0.333
2 Male      2724  0.333
3 Unknown   2724  0.333
```

The `caret` package's `upSample()` function performs oversampling, such that all three levels have the same number of examples as the majority class:

```
> library(caret)
> sns_oversample <- upSample(x = snsdata[2:40],
                               y = snsdata$gender,
                               yname = "gender")
```

The resulting dataset includes 22,054 examples of each of the three gender categories:

```
> fct_count(sns_oversample$gender, prop = TRUE)

# A tibble: 3 × 3
  f          n      p
  <fct>    <int> <dbl>
1 Female    22054  0.333
2 Male      22054  0.333
3 Unknown   22054  0.333
```

Whether the oversampling or undersampling approach works better depends on the dataset as well as the machine learning algorithm used. It may be wise to build models trained on datasets created by each of these resampling techniques and see which one performs better in testing. However, it is very important to keep in mind that the performance measures should be computed on an unbalanced test set; the evaluation should reflect the original class imbalance, as this is how the model will need to perform during real-world deployment.

Generating a synthetic balanced dataset with SMOTE

In addition to undersampling and oversampling, a third rebalancing approach, called **synthetic generation**, creates brand-new examples of the minority class with the goal of reducing oversampling's tendency to overfit the minority class examples. Today, there are many synthetic generation rebalancing methods, but one of the first to gain widespread prominence was the **SMOTE** algorithm introduced by Chawla et al. in 2002, with a name that refers to its use of a synthetic minority oversampling technique. Put simply, the algorithm uses a set of heuristics to construct new records that are similar to but not exactly the same as those previously observed. To construct similar records, SMOTE uses the notion of similarity described in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, and in fact, uses aspects of the k-NN approach directly.



For more information on the SMOTE algorithm, see *SMOTE: Synthetic Minority Over-Sampling Technique, 2002, Chawla, N., Bowyer, K., Hall, L., and Kegelmeyer, W., Journal of Artificial Intelligence Research, Vol. 16, pp. 321-357.*

To understand how SMOTE works, suppose we wanted to oversample a minority class such that the resulting dataset has twice as many examples of this class. In the case of standard oversampling, we would simply duplicate each minority record so that it appears twice. In *synthetic* oversampling techniques like SMOTE, rather than duplicating each record, we will create a new synthetic record. If more or less oversampling is desired, we simply generate more or less than one new synthetic record per original record.

A question remains: how exactly are the synthetic records constructed? This is where the k-Nearest Neighbors technique comes in. The algorithm finds the k nearest neighbors of each of the original observations of the minority class. By convention, k is often set to five, but it can be set larger or smaller if desired. For each synthetic record to be created, the algorithm randomly selects one of the original observations' k nearest neighbors. For example, to double the minority class, it would randomly select one nearest neighbor out of five for each of the original observations; to triple the original data, two out of the five nearest neighbors would be selected for each observation, and so on.

Because randomly selecting the nearest neighbors merely copies the original data, one more step is needed to generate synthetic observations. In this step, the algorithm identifies the vector between each of the original observations and its randomly chosen nearest neighbors. A random number between 0 and 1 is chosen to reflect the proportion of distance along this line to place the synthetic data point. This point's feature values will be somewhere between 100 percent identical to the original observation's feature values and 100 percent identical to the neighbor's feature values—or anywhere in between. This is depicted in the following figure, which illustrates how synthetic observations can be randomly placed on the lines connecting the four original data points to their neighbors. Adding the six synthetic observations creates a much better balance of the circle and square classes and thus strengthens the decision boundary and potentially makes the pattern easier for a learning algorithm to discover.

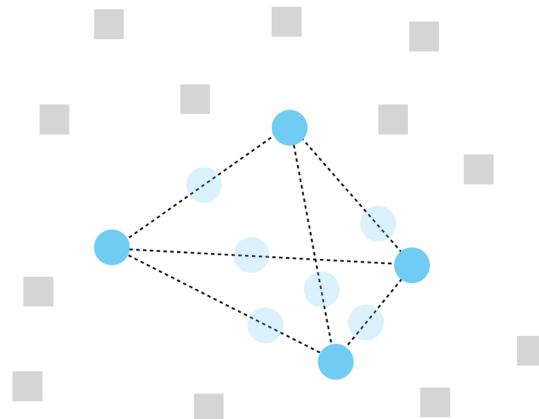


Figure 13.13: SMOTE can create six synthetic observations from four original minority observations, which reinforces the decision boundary between the two classes (circles and squares)

Of course, the SMOTE algorithm's reliance on nearest neighbors and the use of distance functions means that the same data preparation caveats apply as would with k-NN. First, the dataset needs to be completely numeric. Second, although it is not strictly necessary, it may be a good idea to transform the numeric feature values to fall on the same scale so that large ranges do not dominate the selection of nearest neighbors. We'll see this in practice in the section that follows.

Example – Applying the SMOTE algorithm in R

There are several R packages that include implementations of the SMOTE algorithm. The DMwR package has a `SMOTE()` function that is the subject of many tutorials, but at present, is unavailable for recent versions of R.

The `smotefamily` package includes a variety of SMOTE functions and is well documented, but has not been updated in several years. Thus, we will use the `smote()` function in the `themis` package (<https://themis.tidymodels.org>), which is named after Themis, the Greek goddess of justice that is often depicted holding balance scales. This package is both easy to use and well incorporated into the tidyverse.

To illustrate the basic syntax of the `smote()` function, we'll begin by piping in the `snsdata` dataset and using `gender` as the feature to balance:

```
> library(themis)
> sns_balanced <- snsdata |> smote("gender")
```

Checking the result, we use the `table()` function on the dataset, which grew from 30,000 rows to 66,162 rows but is now balanced across the three gender categories:

```
> table(sns_balanced$gender)
```

Female	Male	Unknown
22054	22054	22054

Although this created a gender balance, because the SMOTE algorithm relies on nearest neighbors that are determined by distance calculations, it may be better to normalize the data prior to generating the synthetic data. For instance, because the `friends` feature ranges from 0 to 830 while the `football` feature ranges only from 0 to 15, it is likely that the nearest neighbors will gravitate toward those with similar friend counts rather than similar interests. Applying min-max normalization can help alleviate these concerns by rescaling all features to have a range between 0 and 1.

We've previously written our own normalization function, which we will implement again here:

```
> normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
```

To return the data back to its original scale, we'll also need an `unnormalize()` function. As defined here, the function takes two parameters: the first is a vector, `norm_values`, which stores the values that have been normalized; the second is a string with the name of the column that has been normalized. We need this column name so that we can obtain the minimum and maximum values for this column from the original, unnormalized data in the `snsdata` dataset. The resulting `unnormalized_vals` vector uses these min and max values to reverse the normalization, and then the values are rounded to integers as they were in the original data, except for the `age_imp` feature, which was originally a decimal.

The full unnormalize() function is as follows:

```
> unnormalize <- function(norm_vals, col_name) {
  old_vals <- snsdata[col_name]
  unnormalized_vals <- norm_vals *
    (max(old_vals) - min(old_vals)) + min(old_vals)
  rounded_vals <- if(col_name != "age_imp")
    { round(unnormalized_vals) }
  else {unnormalized_vals}
  return (rounded_vals)
}
```

With a sequence of pipes, we can apply normalization before using the smote() function, followed by unnormalization afterward. This uses the dplyr across() function to normalize and unnormalize the columns where the data type is numeric. In the case of the unnormalize() function, the syntax is slightly more complex due to the use of a lambda, denoted by the tilde (~) character, which defines a function to be used across the columns where the data type is numeric. The normalize() function did not require the use of a lambda because it uses only one parameter, whereas unnormalize() uses two. The .x refers to the vector of data in the column and is passed as the first parameter, while the cur_column() function is used to pass the name of the current column as the second parameter. The complete sequence of commands is as follows:

```
> snsdata_balanced <- snsdata |>
  mutate(across(where(is.numeric), normalize)) |>
  smote("gender") |>
  mutate(across(where(is.numeric), ~unnormalize(.x, cur_column()))))
```

As before, comparing the gender balance before and after SMOTE, we see that the categories are now equivalent:

```
> table(snsdata$gender)
```

Female	Male	Unknown
22054	5222	2724

```
> table(snsdata_balanced$gender)
```

Female	Male	Unknown
22054	22054	22054

Note that we now have over four times as many males and over eight times as many records with unknown gender—or, that roughly three synthetic male records and seven synthetic unknown gender records have been added for each original record with male or unknown gender, respectively. The number of female examples has stayed the same. This balanced dataset can now be used with machine learning algorithms, keeping in mind that the model will be based mostly on synthetic cases rather than “real” examples of the minority classes. Whether or not this results in improved performance may vary from project to project, for reasons that are discussed in the following section.

Considering whether balanced is always better

Although it is undeniable that severely imbalanced datasets cause challenges for learning algorithms, the best approach for handling the imbalance is quite unclear. Some even argue that the best approach is to do nothing at all! The issue is whether artificially balancing the dataset can improve the *overall* performance of a learning algorithm, or if it is just trading a reduction in specificity for an improvement in sensitivity. Because a learning algorithm that has been trained on an artificially balanced dataset will someday be deployed on the original, imbalanced dataset, it seems that the practice of balancing is simply adjusting the learner’s sense of the cost of one type of error versus the other. It is therefore counterintuitive to understand how throwing away data could result in a smarter model—that is, one that is better able to *truly* distinguish among the outcomes.

Among those skeptical of artificially balancing the training data, prolific biostatistician Frank Harrell has written much on this subject. In a thoughtful blog post, he wrote that:



"Users of machine classifiers know that a highly imbalanced sample with regard to a binary outcome variable Y results in a strange classifier... For this reason the odd practice of subsampling the controls is used in an attempt to balance the frequencies and get some variation that will lead to sensible looking classifiers (users of regression models would never exclude good data to get an answer). Then they have to, in some ill-defined way, construct the classifier to make up for biasing the sample."

Clearly, Harrell does not think balancing the sample is generally a wise approach!



For more of Harrell's writings on this subject, see <http://www.fharrell.com/post/classification/> as well as <http://www.fharrell.com/post/class-damage/>.

Nina Zumel, author of *Practical Data Science with R*, performed experiments to determine whether artificially balancing the dataset improved classification performance. After conducting an experiment, she concluded that:



"Classification tends to be easier when the classes are nearly balanced... But I have always been skeptical of the claim that artificially balancing the classes always helps, when the model is to be run on a population with the native class prevalences... balancing the classes, or enrichment in general, is of limited value if your goal is to apply class labels... [it] is not a good idea for logistic regression models."

Much like Frank Harrell, Nina Zumel is also suspicious of the need to artificially balance datasets for classification models. Yet, both perspectives are in opposition to a body of empirical and anecdotal evidence suggesting that artificially balancing a dataset, in fact, does improve the performance of a model.



For a full description of Zumel's experiment on imbalanced data classification, see <https://win-vector.com/2015/02/27/does-balancing-classes-improve-classifier-performance/>.

What explains this contradictory result? It may have something to do with the choice of tool. Statistical learning algorithms, such as regression, may be well **calibrated**, meaning that they do a good job estimating the true underlying probabilities of an outcome—even for rare outcomes. Many machine learning algorithms, such as decision trees and naive Bayes, are decidedly not well calibrated, and thus may need a bit of help via artificial balancing in order to produce reasonable probabilities.

Whether or not a balancing strategy is employed, it is important to use a model evaluation approach that reflects the natural imbalance that the model will be expected to perform on during deployment. This means favoring cost-aware measures like kappa, sensitivity and specificity, or precision and recall, as well as an examination of the **receiver operating characteristic (ROC)** curve, as discussed in *Chapter 10, Evaluating Model Performance*.

While it is a good idea to be skeptical of artificially balancing the dataset, it may also be worth a shot for the most challenging data problems.

Summary

This chapter was intended to expose you to several new types of challenging data that, although infrequently found in simple teaching examples, are regularly encountered in practice. Despite popular adages that tell us that “one can’t have too much of a good thing” or that “more is always better,” this is not always the case for machine learning algorithms, which may be distracted by irrelevant data or have trouble finding the needle in the haystack if overwhelmed by less important details. One of the seeming paradoxes of the so-called big data era is the fact that more data is simultaneously what makes machine learning possible and what makes it challenging; indeed, too much data can even lead to a so-called “curse of dimensionality.”

As disappointing as it is to throw away some of the treasure of big data, this is sometimes necessary to help the learning algorithm perform as desired. Perhaps it is better to think of this as data curation in which the most relevant details are brought to the forefront. Dimensionality reduction techniques like feature selection and feature extraction are important for algorithms that don’t have built-in selection methods, but also provide benefits such as improved computational efficiency, which can be a key bottleneck for large datasets. Sparse data also requires a helping hand to bring the important details to the attention of the learning algorithm, much like the problems of outliers and missing data.

Missing data, which has only been a minor problem in the book so far, presents a significant challenge in many real-world datasets. Machine learning practitioners often choose the simplest approach to solving the problem—that is, the least work necessary to get the model to perform reasonably well—yet machine learning-based approaches like multiple imputation are being used to create complete datasets in the fields of traditional statistics, biostatistics, and the social sciences.

The problem of imbalanced data may be the most difficult form of challenging data to address. Many of the most important machine learning applications involve predictions on imbalanced datasets, but there are no easy solutions, only compromises. Techniques like over- and undersampling are simple but have significant downsides, while more complex techniques like SMOTE are promising but may introduce new problems of their own, and the community is divided on the best approach. The most important lesson, regardless, is to ensure that the evaluation strategy reflects the conditions the model will encounter during deployment.

For example, even if the model is trained on an artificially balanced dataset, it should be tested and evaluated using the natural balance of outcomes.

With these data challenges now behind us, the next chapter once again focuses on model building, and although data preparation is a substantial component of building better learners—after all, garbage in leads to garbage out—there is much more we can do to enhance the learning process itself. However, such techniques will require more than an off-the-shelf algorithm; they will require creativity and determination to maximize the learners' potential.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



14

Building Better Learners

When a sports team falls short of meeting its goal—whether it is to obtain an Olympic gold medal, a league championship, or a world record time—it must search for possible improvements. Imagine that you’re the team’s coach. How would you spend your practice sessions? Perhaps you’d direct the athletes to train harder or train differently in order to maximize every bit of their potential. You might also focus on teamwork to use each athlete’s strengths and weaknesses more smartly.

Now imagine that you’re training a championship machine learning algorithm. Perhaps you hope to compete in machine learning competitions or maybe you simply need to outperform business competitors. Where do you begin? Despite the different context, the strategies for improving a sports team’s performance are like those used for improving the performance of statistical learners. As the coach, it is your job to find the combination of training techniques and teamwork skills that allow the machine learning project to meet your performance goals.

This chapter builds on the material covered throughout this book to introduce techniques that improve the predictive ability of learning algorithms. You will learn:

- Techniques for automating model performance tuning by systematically searching for the optimal set of training conditions
- Methods for combining models into groups that use teamwork to tackle tough learning tasks
- How to use and differentiate among popular variants of decision trees that have become popular due to their impressive performance

None of these methods will be successful for every problem. Yet looking at the winning entries to machine learning competitions, you'll likely find at least one of them has been employed. To be competitive, you too will need to add these skills to your repertoire.

Tuning stock models for better performance

Some machine learning tasks are well suited to be solved by the stock models presented in prior chapters. For these tasks, it may not be necessary to spend much time iterating and refining the model, because it may perform well enough without additional effort. On the other hand, many real-world tasks are inherently more difficult. For these tasks, the underlying concepts to be learned tend to be extremely complex, requiring an understanding of many subtle relationships, or the problem may be affected by substantial amounts of random variability, which makes it difficult to find the signal within the noise.

Developing models that perform extremely well on these types of challenging problems is every bit an art as it is a science. Sometimes a bit of intuition is helpful when trying to identify areas where performance can be improved. In other cases, finding improvements will require a brute-force, trial-and-error approach. Of course, this is one of the strengths of using machines that never tire and never become bored; searching for numerous potential improvements can be made easier by automated programs. As we will see, however, human effort and computing time are not always fungible, and creating a finely-tuned learning algorithm can come with its own costs.

We attempted a difficult machine learning problem in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, as we attempted to predict bank loans that were likely to enter default. Although we were able to achieve a respectable classification accuracy of 82 percent, upon more careful examination in *Chapter 10, Evaluating Model Performance*, we realized that the accuracy statistic was a bit misleading. The kappa statistic—a better measure of performance for unbalanced outcomes—was only about 0.294 as measured via 10-fold **cross-validation (CV)**, which suggested that the model was performing somewhat poorly, despite the high accuracy. In this section, we'll revisit the credit scoring model to see whether we can improve the results.



To follow along with the examples, download the `credit.csv` file from the Packt Publishing website and save it to your R working directory. Load the file into R using the following command: `credit <- read.csv("credit.csv")`.

You may recall that we first used a stock C5.0 decision tree to build the classifier for the credit data and later attempted to improve the classifier's performance by adjusting the `trials` option to increase the number of boosting iterations.

By changing the number of iterations from the default value of 1 up to the value of 10, we were able to increase the model’s accuracy. As defined in *Chapter 11, Being Successful with Machine Learning*, these model options, known as hyperparameters, are not learned automatically from the data but are instead set before training. The process of testing various hyperparameter settings to achieve a better model fit is thus called **hyperparameter tuning**, and strategies for tuning range from simple ad hoc trial and error to more rigorous and systematic iteration.

Hyperparameter tuning is not limited to decision trees. For instance, we tuned k-NN models when we searched for the best value of k . We also tuned neural networks and support vector machines as we adjusted the number of nodes and the number of hidden layers, or chose different kernel functions. Most machine learning algorithms allow the adjustment of at least one hyperparameter, and the most sophisticated models offer many ways to tweak the model fit. Although this allows the model to be tailored closely to the learning task, the complexity of the many options can be daunting. A more systematic approach is warranted.

Determining the scope of hyperparameter tuning

When performing hyperparameter tuning, it is important to put bounds on the scope to prevent the search from proceeding endlessly. The computer provides the muscle, but it is up to the human to dictate where to look and for how long. Even as computing power is growing and cloud computing costs are shrinking, the search can easily get out of hand when sifting through nearly endless combinations of values. A narrow or shallow tuning scope may last long enough to grab a cup of coffee, while a wide or deep scope may give you time to get a good night of sleep—or more!

Time and money are often fungible, as you may be able to buy time in the form of additional computing resources or by enlisting additional team members to build models faster or in parallel. Even so, taking this for granted can lead to ruin in the form of budget overruns or missed deadlines because it is easy for the scope to balloon quickly when work proceeds down countless tangents and dead ends without a plan. To avoid such pitfalls, it is wise to strategize about the breadth and depth of the tuning process beforehand.

You might start by thinking about tuning as a process much like playing the classic board game *Battleship*. In this game, your opponent has placed a fleet of battleships on a two-dimensional grid, which is hidden out of your view. Your goal is to destroy the opponent’s fleet by guessing the coordinates of all their ships before they do the same to yours. Because the ships are known sizes and shapes, a smart player will begin by broadly probing the search grid in a checkerboard pattern but quickly focus on a specific target once it has been hit.

This is a better strategy than guessing coordinates at random or iterating across each coordinate systematically, both of which are inefficient in comparison.



Figure 14.1: The hunt for the optimal machine learning hyperparameters can be much like playing the classic Battleship board game

Similarly, there are methods for tuning that are more efficient than systematic iteration over endless values and combinations of values. With experience, you will develop an intuition for how to proceed, but for the first few attempts, it may be useful to think intentionally about the process. The following general strategy, listed as a series of steps, can be adapted to your machine learning project, computing and staffing resources, and work style:

1. **Replicate the real-world evaluation criteria:** To find the single best set of model hyperparameters, it is important that the models are evaluated using the same criteria as will be used in deployment. This may mean choosing an evaluation metric that mirrors the final, real-world metric, or it may involve writing a function that simulates the deployment environment.
2. **Consider the resource usage for one iteration:** As tuning will be iterating many times on the same algorithm, you should have an estimate of the time and computing resources needed for a single iteration. If it takes one hour to train a single model, it will take 100 hours or more for 100 iterations. If the computer memory is already at its limit, it is likely that you will exceed the limit during tuning. If this is a problem, you will need to invest in additional computing power, run the experiment in parallel, or reduce the size of the dataset via random sampling.

3. **Begin with a shallow search to probe for patterns:** The initial tuning process should be interactive and shallow. It is intended to develop your own understanding of what options and values are important. When probing a single hyperparameter, keep increasing or decreasing its setting in reasonable increments until the performance stops improving (or starts decreasing). Depending on the option, this may be increments of one, multiples of five or ten, or incrementally small fractions, such as 0.1, 0.01, 0.001, and so on. When tuning two or more hyperparameters, it may help to focus on one at a time and keep the other values static. This is a more efficient approach than testing all possible combinations of settings, but may ultimately miss important combinations that would have been discovered if all combinations were tested.
4. **Narrow in on the optimal set of hyperparameter values:** Once you have a sense of a range of values suspected to contain the optimal settings, you can reduce the increments between the tested values and test a narrower range with greater precision or test a greater number of combinations of values. The previous step should have already resulted in a reasonable set of hyperparameters, so this step should only improve and never detract from the model’s performance; it can be stopped at any time.
5. **Determine a reasonable stopping point:** Deciding when to stop the tuning process is easier said than done—the thrill of the hunt and the possibility of a slightly better model can lead to a stubborn desire to keep going! Sometimes, the stopping point is a project deadline when time is running out. In other cases, the work can only stop once the desired performance level has been reached. In any case, because the only way to guarantee finding the optimal hyperparameter values is to test an infinite number of possibilities, rather than working toward burnout, you will need to define the point at which performance is “good enough” to stop the process.

Figure 14.2 illustrates the process of homing in on hyperparameter values for single-parameter tuning. Five potential values (1, 2, 3, 4, and 5) denoted by solid circles were evaluated in the initial pass, and the accuracy was highest when the hyperparameter was set to 3. To check whether an even better hyperparameter setting might exist, eight additional values (from 2.2 to 3.8 in increments of 0.2, denoted by vertical tick marks) were tested within the range between 2 and 4, which led to the discovery of a higher accuracy when the hyperparameter was set to 3.2. If time allows, one could test even more values in a narrower range around this value to possibly find an even better setting.

Tuning One Hyperparameter:

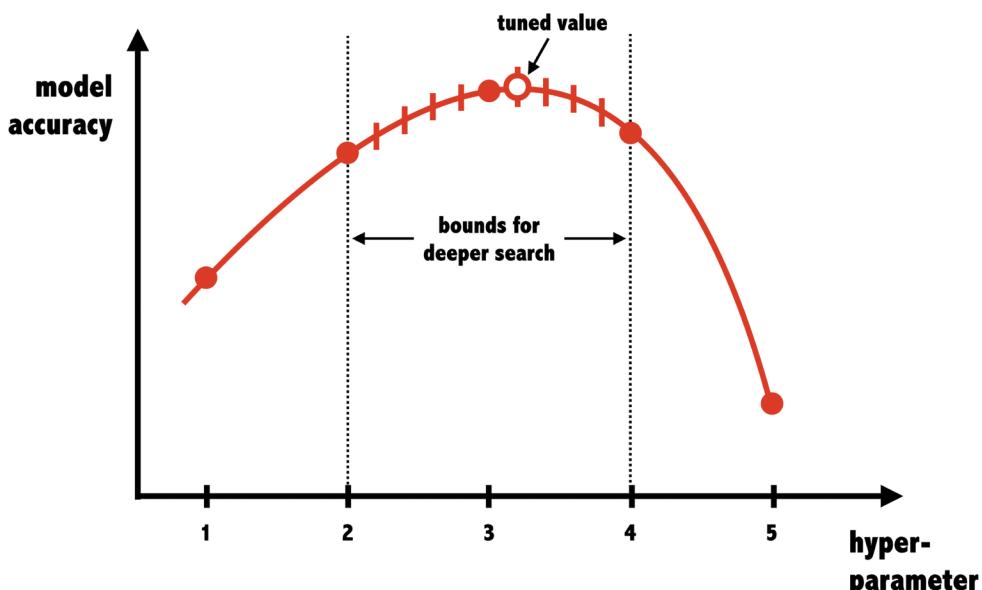


Figure 14.2: Strategies for parameter tuning home in on the optimal value by searching broadly and then narrowly

Tuning two or more hyperparameters is more complicated, because the optimal value of one parameter may depend on the value of the others. Constructing a visualization like the one depicted in *Figure 14.3* may help in understanding how to find the best combinations of parameters; within hot spots where certain combinations of values result in better model performance, one might test more values in narrower and narrower ranges:

Tuning Two Hyperparameters:

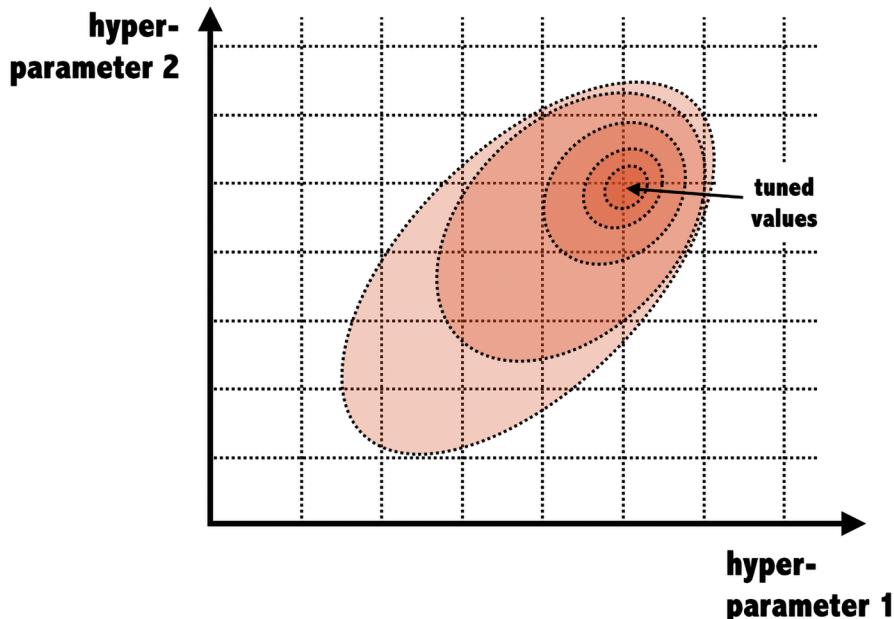


Figure 14.3: Tuning strategies become more challenging as more hyperparameters are added, as the model's best performance depends on combinations of values

This **Battleship-style grid search**, in which hyperparameters and combinations of hyperparameters are tested systematically, is not the only approach to tuning, although it may be the most widely used. A more intelligent approach called **Bayesian optimization** treats the tuning process as a learning problem that can be solved using modeling. This approach is included in some automated machine learning software but is outside the scope of this book. Instead, for the remainder of this section, we will focus on applying the idea of grid search to our real-world dataset.

Example – using caret for automated tuning

Thankfully, we can use R to conduct the iterative search through many possible hyperparameter values and combinations of values to find the best set. This approach is a relatively easy yet sometimes computationally expensive brute-force method of optimizing a learning algorithm's performance.

The `caret` package, which was used previously in *Chapter 10, Evaluating Model Performance*, provides tools to assist with this form of automated tuning. The core tuning functionality is provided by a `train()` function that serves as a standardized interface for over 200 different machine learning models for both classification and numeric prediction tasks. Using this function, it is possible to automate the search for optimal models using a choice of evaluation methods and metrics.

Automated parameter tuning with `caret` will require you to consider three questions:

- What type of machine learning algorithm (and specific R implementation of this algorithm) should be trained on the data?
- Which hyperparameters can be adjusted for this algorithm, and how extensively should they be tuned to find the optimal settings?
- What criterion should be used to evaluate the candidate models to identify the best overall set of tuning values?

Answering the first question involves finding a match between the machine learning task and one of the many models available to the `caret` package. This requires a general understanding of the types of machine learning models, which you may already have if you've been working through this book chronologically. It can also help to work through a process of elimination. Nearly half of the models can be eliminated depending on whether the task is classification or numeric prediction; others can be excluded based on the format of the training data or the need to avoid black box models, and so on. In any case, there's also no reason you can't create several highly tuned models and compare them across the set.

Addressing the second question is a matter largely dictated by the choice of model since each algorithm utilizes its own set of hyperparameters. The available tuning options for the predictive models covered in this book are listed in the following table. Keep in mind that although some models have additional options not shown, only those listed in the table are supported by `caret` for automatic tuning.

Model	Learning Task	Method Name	Hyperparameters
<i>k</i> -Nearest Neighbors	Classification	knn	k
Naive Bayes	Classification	nb	fL, usekernel
Decision Trees	Classification	C5.0	model, trials, winnow
OneR Rule Learner	Classification	OneR	None
RIPPER Rule Learner	Classification	JRip	NumOpt
Linear Regression	Regression	lm	None
Regression Trees	Regression	rpart	cp
Model Trees	Regression	M5	pruned, smoothed, rules
Neural Networks	Dual Use	nnet	size, decay
Support Vector Machines (Linear Kernel)	Dual Use	svmLinear	C
Support Vector Machines (Radial Basis Kernel)	Dual Use	svmRadial	C, sigma
Random Forests	Dual Use	rf	mtry
Gradient Boosting Machines (GBM)	Dual Use	gbm	n.trees, interaction.depth, shrinkage, n.minobsinnode
XGBoost (XGB)	Dual Use	xgboost	eta, max_depth, colsample_bytree, subsample, nrounds, gamma, min_child_weight



For a complete list of the models and corresponding tuning options covered by caret, refer to the table provided by package author Max Kuhn at <http://topepo.github.io/caret/available-models.html>.

If you ever forget the tuning parameters for a particular model, the `modelLookup()` function can be used to find them. Simply supply the method name as illustrated for the C5.0 model:

```
> modelLookup("C5.0")
```

	model parameter		label	forReg	forClass	probModel
1	C5.0	trials # Boosting Iterations	FALSE	TRUE	TRUE	
2	C5.0	model	Model Type	FALSE	TRUE	TRUE
3	C5.0	winnow	Winnow	FALSE	TRUE	TRUE

The goal of automatic tuning is to iterate over the set of candidate models comprising the search grid of potential parameter combinations. As it is impractical to search every conceivable combination, only a subset of possibilities is used to construct the grid. By default, caret searches, at most, three values for each of the model's p hyperparameters, which means that, at most, 3^p candidate models will be tested. For example, by default, the automatic tuning of k-nearest neighbors will compare $3^1 = 3$ candidate models with k=5, k=7, and k=9. Similarly, tuning a decision tree will result in a comparison of up to 27 different candidate models, comprising the grid of $3^3 = 27$ combinations of `model`, `trials`, and `winnow` settings. In practice, however, only 12 models are tested. This is because `model` and `winnow` can only take two values (tree versus rules and TRUE versus FALSE, respectively), which makes the grid size $3 * 2 * 2 = 12$.



Since the default search grid may not be ideal for your learning problem, caret allows you to provide a custom search grid defined by a simple command, which we will cover later.

The third and final step in automatic model tuning involves identifying the best model among the candidates. This uses the methods discussed in *Chapter 10, Evaluating Model Performance*, including the choice of resampling strategy for creating training and test datasets, and the use of model performance statistics to measure the predictive accuracy. All the resampling strategies and many of the performance statistics we've learned are supported by caret. These include statistics such as accuracy and kappa for classifiers and R-squared or **root-mean-square error (RMSE)** for numeric models. Cost-sensitive measures like sensitivity, specificity, and AUC can also be used if desired.

By default, caret will select the candidate model with the best value of the desired performance measure. Because this practice sometimes results in the selection of models that achieve minor performance improvements via large increases in model complexity, alternative model selection functions are provided. These alternatives allow us to choose simpler models that are still reasonably close to the best model, which may be desirable in the case where a bit of predictive performance is worth sacrificing for an improvement in computational efficiency.

Given the wide variety of options in the `caret` tuning process, it is helpful that many of the function's defaults are reasonable. For instance, without specifying the settings manually, `caret` uses prediction accuracy or RMSE on a bootstrap sample to choose the best performer for classification and numeric prediction models, respectively. Similarly, it will automatically define a limited grid to search. These defaults allow us to start with a simple tuning process and learn to tweak the `train()` function to design a wide variety of experiments of our choosing.

Creating a simple tuned model

To illustrate the process of tuning a model, let's begin by observing what happens when we attempt to tune the credit scoring model using the `caret` package's default settings. The simplest way to tune a learner requires only that you specify a model type via the `method` parameter. Since we used C5.0 decision trees previously with the `credit` model, we'll continue our work by optimizing this learner. The basic `train()` command for tuning a C5.0 decision tree using the default settings is as follows:

```
> library(caret)
> set.seed(300)
> m <- train(default ~ ., data = credit, method = "C5.0")
```

First, the `set.seed()` function is used to initialize R's random number generator to a set starting position. You may recall that we used this function in several prior chapters. By setting the `seed` parameter (in this case, to the arbitrary number 300), the random numbers will follow a predefined sequence. This allows simulations that use random sampling to be repeated with identical results—a very helpful feature if you are sharing code or attempting to replicate a prior result.

Next, we define a tree as `default ~ .` using the R formula interface. This models a loan default status (yes or no) using all the other features in the `credit` dataset. The parameter `method = "C5.0"` tells the function to use the C5.0 decision tree algorithm.

After you've entered the preceding command, depending upon your computer's capabilities, there may be a significant delay as the tuning process occurs. Even though this is a small dataset, a substantial amount of calculation must occur. R must repeatedly generate random bootstrap samples of data, build decision trees, compute performance statistics, and evaluate the result. Because there are 12 candidate models with varying hyperparameter values to be evaluated, and 25 bootstrap samples per candidate model to compute an average performance measure, there are $25 \times 12 = 300$ decision tree models being built using C5.0—and this doesn't even count the additional decision trees being built when the boosting trials are set!

A list named `m` stores the result of the `train()` experiment, and the command `str(m)` will display the associated results, but the contents can be substantial. Instead, simply type the name of the object for a condensed summary of the results. For instance, typing `m` yields the following output (note that numbered labels have been added for clarity):

1000 samples
16 predictor
2 classes: 'no', 'yes' 1

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 1000, 1000, 1000, 1000, 1000, 1000, ... 2
Resampling results across tuning parameters:

model	winnow	trials	Accuracy	Kappa
rules	FALSE	1	0.6918852	0.2749843
rules	FALSE	10	0.7144478	0.3112184
rules	FALSE	20	0.7270463	0.3344054
rules	TRUE	1	0.6947856	0.2752201
rules	TRUE	10	0.7144334	0.3150388
rules	TRUE	20	0.7268154	0.3393965
tree	FALSE	1	0.6909682	0.2569822
tree	FALSE	10	0.7256597	0.2996244
tree	FALSE	20	0.7322016	0.3157605
tree	TRUE	1	0.6920217	0.2604578
tree	TRUE	10	0.7279264	0.3099733
tree	TRUE	20	0.7299631	0.3131074

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were trials = 20, model = tree and winnow = FALSE. 4

Figure 14.4: The results of a caret experiment are separated into four components, as annotated in this figure

The labels highlight four main components in the output:

1. **A brief description of the input dataset:** If you are familiar with your data and have applied the `train()` function correctly, this information should not be surprising.
2. **A report of the preprocessing and resampling methods applied:** Here we see that 25 bootstrap samples, each including 1,000 examples, were used to train the models.
3. **A list of the candidate models evaluated:** In this section, we can confirm that 12 different models were tested, based on the combinations of three C5.0 hyperparameters: `model`, `trials`, and `winnow`. The average accuracy and kappa statistics for each candidate model are also shown.

4. **The choice of best model:** As the footnote describes, the model with the best accuracy (in other words, “largest”) was selected. This was the C5.0 model that used a decision tree with the settings `winnow = FALSE` and `trials = 20`.

After identifying the best model, the `train()` function uses the tuned hyperparameters to build a model on the full input dataset, which is stored in `m` as `m$finalModel`. In most cases, you will not need to work directly with the `finalModel` sub-object. Instead, simply use the `predict()` function with the `m` object as follows:

```
> p <- predict(m, credit)
```

The resulting vector of predictions works as expected, allowing us to create a confusion matrix that compares the predicted and actual values:

```
> table(p, credit$default)
```

p	no	yes
no	700	2
yes	0	298

Of the 1,000 examples used for training the final model, only two were misclassified, for an accuracy of 99.8 percent. However, it is very important to note that since the model was built on both the training and test data, this accuracy is optimistic and thus should not be viewed as indicative of performance on unseen data. The bootstrap accuracy estimate of 72.996 percent, which can be found in the last row of section three of the `train()` output in *Figure 14.4*, is a far more realistic estimate of future accuracy.

In addition to automatic hyperparameter tuning, using the `caret` package’s `train()` and `predict()` functions also offers a pair of benefits beyond the functions found in the stock packages.

First, any data preparation steps applied by the `train()` function will be similarly applied to the data used for generating predictions. This includes transformations like centering and scaling, as well as the imputation of missing values. Allowing `caret` to handle the data preparation will ensure that the steps that contributed to the best model’s performance will remain in place when the model is deployed.

Second, the `predict()` function provides a standardized interface for obtaining predicted class values and predicted class probabilities, even for model types that ordinarily would require additional steps to obtain this information. For a classification model, the predicted classes are provided by default:

```
> head(predict(m, credit))
```

```
[1] no yes no no yes no  
Levels: no yes
```

To obtain the estimated probabilities for each class, use the `type = "prob"` parameter:

```
> head(predict(m, credit, type = "prob"))
```

	no	yes
1	0.9606970	0.03930299
2	0.1388444	0.86115560
3	1.0000000	0.00000000
4	0.7720279	0.22797207
5	0.2948061	0.70519387
6	0.8583715	0.14162853

Even in cases where the underlying model refers to the prediction probabilities using a different string (for example, "raw" for a `naiveBayes` model), the `predict()` function will translate `type = "prob"` to the appropriate parameter setting automatically.

Customizing the tuning process

The decision tree we created previously demonstrates the `caret` package's ability to produce an optimized model with minimal intervention. The default settings allow optimized models to be created easily. However, it is also possible to change the default settings as desired, which may assist with unlocking the upper echelon of performance. Before the tuning process begins, it's worth answering a series of questions that will help guide the setup of the `caret` experiment:

- How long does it take for one iteration? In other words, how long does it take to train a single instance of the model being tuned?
- Given the time it takes to train a single instance, how long will it take to perform the model evaluation using the chosen resampling method? For example, 10-fold CV will require 10 times as much time as training a single model.

- How much time are you willing to spend on tuning? Based on this number, one can determine the total number of hyperparameter values that can be tested. For instance, if it takes one minute to evaluate a model using 10-fold CV, then 60 hyperparameter settings can be tested per hour.

Using time as the key limiting factor will help put bounds on the tuning process and prevent you from chasing better and better performance endlessly.

Once you've decided how much time to spend on the trials, it is easy to customize the process to your liking. To illustrate this flexibility, let's modify our work on the credit decision tree to mirror the process we used in *Chapter 10, Evaluating Model Performance*. In that chapter, we estimated the kappa statistic using 10-fold CV. We'll do the same here, using kappa to tune the boosting trials for the C5.0 decision tree algorithm and find the optimal setting for our data. Note that decision tree boosting was first covered in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, and will also be covered in greater detail later in this chapter.

The `trainControl()` function is used to create a set of configuration options known as a **control object**. This object guides the `train()` function and allows for the selection of model evaluation criteria such as the resampling strategy and the measure used for choosing the best model. Although this function can be used to modify nearly every aspect of a caret tuning experiment, we'll focus on two important parameters: `method` and `selectionFunction`.



If you're eager for more details about the control object, you can use the `?trainControl` command for a list of all the parameters.

When using the `trainControl()` function, the `method` parameter sets the resampling method, such as holdout sampling or k-fold CV. The following table lists the possible `method` values, as well as any additional parameters for adjusting the sample size and the number of iterations. Although the default options for these resampling methods follow popular conventions, you may choose to adjust these depending on the size of your dataset and the complexity of your model.

Resampling method	Method name	Additional options and default values
<i>Holdout sampling</i>	<code>LGOCV</code>	<code>p = 0.75</code> (training data proportion)
<i>k-fold CV</i>	<code>cv</code>	<code>number = 10</code> (number of folds)
<i>Repeated k-fold CV</i>	<code>repeatedcv</code>	<code>number = 10</code> (number of folds) <code>repeats = 10</code> (number of iterations)

<i>Bootstrap sampling</i>	boot	number = 25 (resampling iterations)
<i>0.632 bootstrap</i>	boot632	number = 25 (resampling iterations)
<i>Leave-one-out CV</i>	LOOCV	<i>None</i>

The `selectionFunction` parameter is used to specify the function that will choose the optimal model among the candidates. Three such functions are included. The `best` function simply chooses the candidate with the best value on the specified performance measure. This is used by default. The other two functions are used to choose the most parsimonious, or simplest, model that is within a certain threshold of the best model's performance. The `oneSE` function chooses the simplest candidate within one standard error of the best performance, and `tolerance` uses the simplest candidate within a user-specified percentage.



Some subjectivity is involved with the `caret` package's ranking of models by simplicity. For information on how models are ranked, see the help page for the selection functions by typing `?best` at the R command prompt.

To create a control object named `ctrl` that uses 10-fold CV and the `oneSE` selection function, use the following command, noting that `number = 10` is included only for clarity; since this is the default value for `method = "cv"`, it could have been omitted:

```
> ctrl <- trainControl(method = "cv", number = 10,
                      selectionFunction = "oneSE")
```

We'll use the result of this function shortly.

In the meantime, the next step in setting up our experiment is to create the search grid for hyperparameter tuning. The grid must include a column named for each hyperparameter in the desired model, regardless of whether it will be tuned. It must also include a row for each desired combination of values to test. Since we are using a C5.0 decision tree, this means we'll need columns named `model`, `trials`, and `winnow`, corresponding to the three options that can be tuned. For other machine learning models, refer to the table presented earlier in this chapter or use the `modelLookup()` function to find the hyperparameters as described previously.

Rather than filling the grid data frame cell by cell—a tedious task if there are many possible combinations of values—we can use the `expand.grid()` function, which creates data frames from the combinations of all values supplied. For example, suppose we would like to hold constant `model = "tree"` and `winnow = FALSE` while searching eight different values of `trials`.

This can be created as:

```
> grid <- expand.grid(model = "tree",
                         trials = c(1, 5, 10, 15, 20, 25, 30, 35),
                         winnow = FALSE)
```

The resulting `grid` data frame contains $1^*8^*1 = 8$ rows:

```
> grid
```

	model	trials	winnow
1	tree	1	FALSE
2	tree	5	FALSE
3	tree	10	FALSE
4	tree	15	FALSE
5	tree	20	FALSE
6	tree	25	FALSE
7	tree	30	FALSE
8	tree	35	FALSE

The `train()` function will build a candidate model for evaluation using each `grid` row's combination of model parameters.

Given the search grid and the control object created previously, we are ready to run a thoroughly customized `train()` experiment. As before, we'll set the random seed to the arbitrary number 300 in order to ensure repeatable results. But this time, we'll pass our control object and tuning grid while adding a parameter `metric = "Kappa"`, indicating the statistic to be used by the model evaluation function—in this case, "oneSE". The full set of commands is as follows:

```
> set.seed(300)
> m <- train(default ~ ., data = credit, method = "C5.0",
               metric = "Kappa",
               trControl = ctrl,
               tuneGrid = grid)
```

This results in an object that we can view by typing its name:

```
> m
```

```
C5.0
```

```

1000 samples
16 predictor
2 classes: 'no', 'yes'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results across tuning parameters:

  trials  Accuracy   Kappa
    1      0.710     0.2859380
    5      0.726     0.3256082
   10      0.725     0.3054657
   15      0.726     0.3204938
   20      0.733     0.3292403
   25      0.732     0.3308708
   30      0.733     0.3298968
   35      0.738     0.3449912

Tuning parameter 'model' was held constant at a value of tree

Tuning parameter 'winnow' was held constant at a value of FALSE
Kappa was used to select the optimal model using the one SE rule.
The final values used for the model were trials = 5, model = tree
and winnow = FALSE.

```

Although the output is similar to the automatically tuned model, there are a few notable differences. Because 10-fold CV was used, the sample size to build each candidate model was reduced to 900 rather than the 1,000 used in the bootstrap. Furthermore, eight candidate models were tested rather than the 12 in the prior experiment. Lastly, because `model` and `winnow` were held constant, their values are no longer shown in the results; instead, they are listed as a footnote.

The best model here differs quite significantly from the prior experiment. Before, the best model used `trials = 20`, whereas here, it used `trials = 1`. This change is because we used the `oneSE` function rather than the `best` function to select the optimal model. Even though the model with `trials = 35` obtained the best kappa, the single-trial model offers reasonably close performance with a much simpler algorithm.



Due to the large number of configuration parameters, caret can seem overwhelming at first. Don't let this deter you—there is no easier way to test the performance of models using 10-fold CV. Instead, think of the experiment as defined by two parts: a `trainControl()` object that dictates the testing criteria, and a tuning grid that determines what model parameters to evaluate. Supply these to the `train()` function and with a bit of computing time, your experiment will be complete!

Of course, tuning is just one possibility for building better learners. In the next section, you will discover that in addition to buffing up a single learner to make it stronger, it is also possible to combine several weaker models to form a more powerful team.

Improving model performance with ensembles

Just as the best sports teams have players with complementary rather than overlapping skillsets, some of the best machine learning algorithms utilize teams of complementary models. Since a model brings a unique bias to a learning task, it may readily learn one subset of examples but have trouble with another. Therefore, by intelligently using the talents of several diverse team members, it is possible to create a strong team of multiple weak learners.

This technique of combining and managing the predictions of multiple models falls into a wider set of **meta-learning methods**, which are techniques that involve learning how to learn. This includes anything from simple algorithms that gradually improve performance by iterating over design decisions—for instance, the automated parameter tuning used earlier in this chapter—to highly complex algorithms that use concepts borrowed from evolutionary biology and genetics for self-modifying and adapting to learning tasks.

Suppose you were a contestant on a television trivia show that allowed you to choose a panel of five friends to assist you with answering the final question for the million-dollar prize. Most people would try to stack the panel with a diverse set of subject matter experts. A panel containing professors of literature, science, history, and art, along with a current pop-culture expert, would be safely well-rounded. Given their breadth of knowledge, it would be unlikely that a question would stump the group.

The meta-learning approach that utilizes a similar principle of creating a varied team of experts is known as an **ensemble**. For the remainder of this chapter, we'll focus on meta-learning only as it pertains to ensembling—the task of modeling a relationship between the predictions of several models and the desired outcome. The teamwork-based methods covered here are quite powerful and are used often to build more effective classifiers.

Understanding ensemble learning

All ensemble methods are based on the idea that by combining multiple weaker learners, a stronger learner is created. Ensembles contain two or more machine learning models, which can be of the same type, such as several decision trees, or of different types, such as a decision tree and a neural network. Though there are myriad ways to construct an ensemble, they tend to fall into several general categories, which can be distinguished, in large part, by the answers to two questions:

- How are the ensemble's models chosen and trained?
- How are the models' predictions combined to make a single final prediction?

When answering these questions, it can be helpful to imagine the ensemble in terms of the following process diagram, which encompasses nearly all ensembling approaches:

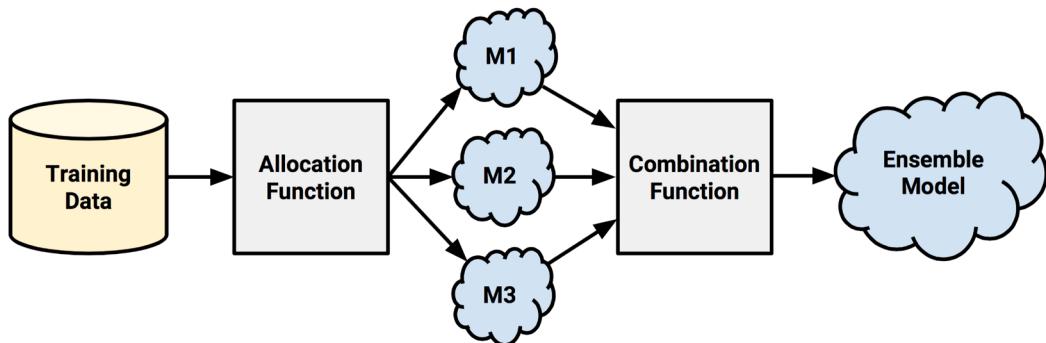


Figure 14.5: Ensembles combine multiple weaker models into a single stronger model

In this design pattern, input training data is used to build several models. The **allocation function** dictates how much and what subsets of the training data each model receives. Do they each receive the full training dataset or merely a sample? Do they each receive every feature or a subset of features? The decisions made here will shape the training of the weaker learners that comprise the stronger ensemble.

Just as you'd want a variety of experts to advise your appearance on a television trivia game show, ensembles depend on a **diverse** set of classifiers, which means that they have uncorrelated classifications but still perform better than random chance. In other words, each classifier must be making an independent prediction, but each must also be doing more than merely guessing.

Diversity can be added to the ensemble by including a variety of machine learning techniques, such as an ensemble that groups a decision tree, a neural network, and a logistic regression model.

Alternatively, the allocation function itself can also be a source of diversity by acting as a **data manipulator** and artificially varying the input data to bias the resulting learners, even if they use the same learning algorithm. As we will see in practice later, the allocation and data manipulation processes may be automated or included as part of the ensembling algorithm itself, or they may be performed by hand as part of the data engineering and model-building process. Overall, modes of increasing the ensemble's diversity generally fall into five categories:

- Using assorted base learning algorithms
- Manipulating the training sample by taking different samples at random, often by using bootstrapping
- Manipulating a single learning algorithm by using different hyperparameter settings
- Changing how the target feature is represented, such as representing an outcome as binary, categorical, or numeric
- Partitioning the training data into subgroups that represent different patterns to be learned; for instance, one might stratify the examples by key features, and let models in the ensemble become experts on different subsets of the training data

For instance, in an ensemble of decision trees, the allocation function might use bootstrap sampling to construct unique training datasets for each tree, or it may pass each one a different subset of features. On the other hand, if the ensemble already includes a diverse set of algorithms—such as a neural network, a decision tree, and a k-NN classifier—then the allocation function might pass the training data on to each algorithm relatively unchanged.

After the ensemble's models are trained, they can be used to generate predictions on future data, but this set of multiple predictions must be reconciled somehow to generate a single final prediction. The **combination function** is the step in the ensembling process that takes each of these predictions and combines them into a single authoritative prediction for the set. Of course, because some of the models may disagree on the predicted value, the function must somehow blend or unify the information from the learners. The combination function is also known as a **composer** due to its work synthesizing the final prediction.

There are two main strategies for merging or composing final predictions. The simpler of the two approaches involves **weighting methods**, which assign a score to each prediction that dictates how heavily it will factor into the final prediction. These range from a simple majority vote in which each classifier is weighted evenly, to more complex performance-based methods that grant more authority to some models than others if they have proven to be more reliable on past data.

The second approach uses more complex meta-learning methods, such as the model stacking technique, which will be covered in depth later in this chapter. These use the initial set of predictions from the weak learners to train a secondary machine learning algorithm to make the final prediction—a process that is analogous to a committee making recommendations to a leader that makes the final decision.

Ensembling methods are used to gain better performance than what is possible using only a single learning algorithm—the primary goal of the ensemble is to turn a group of weaker learners into a stronger, unified team. Still, there are many additional benefits, some of which may be surprising. These suggest additional reasons why one might turn to an ensemble, even outside of a machine learning competition environment:

- **The use of independent ensembles allows work in parallel:** Training independent classifiers separately means that work can be divided across multiple people. This allows more rapid iteration and may increase creativity. Each team member builds their best model, and the results can be easily combined into an ensemble at the end.
- **Improved performance on massive or minuscule datasets:** Many algorithms run into memory or complexity limits when an extremely large set of features or examples are used. An ensemble of independent models can be fed subsets of features or examples, which are more computationally efficient to train than a single full model, and importantly, can often be run in parallel using distributed computing methods. On the other side of the spectrum, ensembles also do well on the smallest datasets because resampling methods like bootstrapping are inherently part of the allocation function of many ensemble designs.
- **The ability to synthesize data from distinct domains:** Since there is no one-size-fits-all learning algorithm, and each learning algorithm has its own biases and heuristics, the ensemble’s ability to incorporate evidence from multiple types of learners is increasingly important for modeling the most challenging learning tasks relying on data drawn from diverse domains.
- **A more nuanced understanding of difficult learning tasks:** Real-world phenomena are often extremely complex, with many interacting intricacies. Methods like ensembles, which divide the task into smaller modeled portions, are more able to capture subtle patterns that a single model might miss. Some learners in the set can go narrower and deeper to learn a specific subset of the most challenging cases.

None of these benefits would be very helpful if you weren't able to easily apply ensemble methods in R, and there are many packages available to do just that. Let's look at several of the most popular ensemble methods and how they can be used to improve the performance of the credit model we've been working on.

Popular ensemble-based algorithms

Thankfully, using teams of machine learners to improve the predictive performance doesn't mean you'll need to train each ensemble member separately by hand, although this option does exist, as you will learn later in this chapter. Instead, there are ensemble-based algorithms that manipulate the allocation function to train a very large number of simpler models in a single step automatically. In this way, an ensemble that includes a hundred learners or more can be trained with no more human time and input than training a single learner. As easily as one might build a single decision tree model, it is possible to build an ensemble with hundreds of such trees and harness the power of teamwork. Although it would be tempting to assume this is a magic bullet, such power, of course, comes with downsides such as loss of interpretability and a less diverse set of base algorithms from which to choose. This will be apparent in the sections that follow, which cover the evolution of two decades' worth of popular ensembling algorithms—all of which, not coincidentally, are based on decision trees.

Bagging

One of the first ensemble methods to gain widespread acceptance used a technique called **bootstrap aggregating** or **bagging** for short. As described by Leo Breiman in the mid-1990s, bagging begins by generating several new training datasets using bootstrap sampling on the original training data. These datasets are then used to generate a set of models using a single learning algorithm. The models' predictions are combined using voting for classification and averaging for numeric prediction.



For additional information on bagging, refer to *Bagging predictors*. Breiman L., *Machine Learning*, 1996, Vol. 24, pp. 123-140.

Although bagging is a relatively simple ensemble, it can perform quite well if it is used with relatively **unstable** learners, that is, those generating models that tend to change substantially when the input data changes only slightly. Unstable models are essential for ensuring the ensemble's diversity despite only minor variations across the bootstrap training datasets.

For this reason, bagging is most often used with decision trees, which have the tendency to vary dramatically given minor changes in input data.

The `ipred` package offers a classic implementation of bagged decision trees. To train the model, the `bagging()` function works similarly to many of the models used previously. The `nbagg` parameter is used to control the number of decision trees voting in the ensemble, with a default value of 25. Depending on the difficulty of the learning task and the amount of training data, increasing this number may improve the model's performance, up to a limit. The downside is that this creates additional computational expense, and a large number of trees may take some time to train.

After installing the `ipred` package, we can create the ensemble as follows. We'll stick to the default value of 25 decision trees:

```
> library(ipred)
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> set.seed(123)
> mybag <- bagging(default ~ ., data = credit, nbagg = 25)
```

The resulting `mybag` model works as expected in concert with the `predict()` function:

```
> credit_pred <- predict(mybag, credit)
> table(credit_pred, credit$default)
```

credit_pred	no	yes
no	699	4
yes	1	296

Given the preceding results, the model seems to have fit the data extremely well—*too well*, probably, as the results are based only on the training data and thus may reflect overfitting rather than true performance on future unseen data. To obtain a better estimate of future performance, we can use the bagged decision tree method in the `caret` package to obtain a 10-fold CV estimate of accuracy and kappa. Note that the method name for the `ipred` bagging function is `treebag`:

```
> library(caret)
> credit <- read.csv("credit.csv")
> set.seed(300)
> ctrl <- trainControl(method = "cv", number = 10)
> train(default ~ ., data = credit, method = "treebag",
       trControl = ctrl)
```

Bagged CART

```
1000 samples
16 predictor
2 classes: 'no', 'yes'
```

```
No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results:
```

Accuracy	Kappa
0.732	0.3319334

The kappa statistic of 0.33 for this model suggests that the bagged tree model performs roughly as well as the C5.0 decision tree we tuned earlier in this chapter, which had a kappa statistic ranging from 0.32 to 0.34, depending on the tuning parameters. Keep this performance in mind as you read the next section, and consider the differences between the simple bagging technique and the more complex methods that build upon it.

Boosting

Another common ensemble-based method is called **boosting** because it improves or “boosts” the performance of weak learners to attain the performance of stronger learners. This method is based largely on the work of Robert Schapire and Yoav Freund, who have published extensively on the topic since the 1990s.



For additional information on boosting, refer to *Boosting: Foundations and Algorithms*, Schapire, RE, Freund, Y, Cambridge, MA: The MIT Press, 2012.

Like bagging, boosting uses ensembles of models trained on resampled data and a vote to determine the final prediction. There are two key distinctions. First, the resampled datasets in boosting are constructed specifically to generate complementary learners. This means that the work cannot occur in parallel, as the ensemble’s models are no longer independent from one another. Second, rather than giving each learner an equal vote, boosting gives each learner a vote that is weighted based on its past performance. Models that perform better have greater influence over the ensemble’s final prediction.

Boosting will result in performance that is often somewhat better and certainly no worse than the best model in the ensemble. Since the models in the ensemble are purposely built to be complementary, it is possible to increase ensemble performance to an arbitrary threshold simply by adding additional classifiers to the group, assuming that each additional classifier performs better than random chance. Given the obvious utility of this finding, boosting is thought to be one of the most significant discoveries in machine learning.



Although boosting can create a model that meets an arbitrarily low error rate, this may not always be reasonable in practice. One reason for this is that the performance gains are incrementally smaller as additional learners are gained, making some thresholds practically infeasible. Additionally, the pursuit of pure accuracy may result in the model being overfitted to the training data and not generalizable to unseen data.

A boosting algorithm called **AdaBoost**, short for **adaptive boosting**, was proposed by Freund and Schapire in 1997. The algorithm is based on the idea of generating weak learners that iteratively learn a larger portion of the difficult-to-classify examples in the training data by paying more attention (that is, giving more weight) to often misclassified examples.

Beginning from an unweighted dataset, the first classifier attempts to model the outcome. Examples that the classifier predicted correctly will be less likely to appear in the training dataset for the following classifier, and conversely, the difficult-to-classify examples will appear more frequently. As additional rounds of weak learners are added, they are trained on data with successively more difficult examples. The process continues until the desired overall error rate is reached or performance no longer improves. At that point, each classifier's vote is weighted according to its accuracy on the training data on which it was built.

Though boosting principles can be applied to nearly any type of model, the principles are most often used with decision trees. We already applied the boosting technique earlier in this chapter, as well as in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, as a method to improve the performance of a C5.0 decision tree. With C5.0, boosting can be enabled by simply setting a `trials` parameter to an integer value greater than one.

The **AdaBoost.M1** algorithm provides a standalone implementation of AdaBoost for classification with trees. The algorithm can be found in the `adabag` package.



For more information about the `adabag` package, refer to *adabag: An R Package for Classification with Boosting and Bagging*, Alfaro, E., Gamez, M., Garcia, N., *Journal of Statistical Software*, 2013, Vol. 54, pp. 1-35.

Let's create an `AdaBoost.M1` classifier for the credit data. The general syntax for this algorithm is similar to other modeling techniques:

```
> library(adabag)
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> set.seed(300)
> m_adaboost <- boosting(default ~ ., data = credit)
```

As usual, the `predict()` function is applied to the resulting object to make predictions:

```
> p_adaboost <- predict(m_adaboost, credit)
```

Departing from convention, rather than returning a vector of predictions, this returns an object with information about the model. The predictions are stored in a sub-object called `class`:

```
> head(p_adaboost$class)
[1] "no"  "yes" "no"  "no"  "yes" "no"
```

A confusion matrix can be found in the `confusion` sub-object:

```
> p_adaboost$confusion
          Observed Class
Predicted Class  no yes
      no    700   0
      yes    0 300
```

Before you get your hopes up about the perfect accuracy, note that the preceding confusion matrix is based on the model's performance on the training data. Since boosting allows the error rate to be reduced to an arbitrarily low level, the learner simply continued until it made no more errors. This likely resulted in overfitting on the training dataset.

For a more accurate assessment of performance on unseen data, we need to use another evaluation method. The `adabag` package provides a simple function to use 10-fold CV:

```
> set.seed(300)
> adaboost_cv <- boosting.cv(default ~ ., data = credit)
```

Depending on your computer's capabilities, this may take some time to run, during which it will log each iteration to the screen—on a recent MacBook Pro computer, it took about a minute. After it completes, we can view a more reasonable confusion matrix:

```
> adaboost_cv$confusion
          Observed Class

Predicted Class  no  yes
      no    598  160
      yes   102  140
```

We can find the kappa statistic using the `vcd` package, as demonstrated in *Chapter 10, Evaluating Model Performance*:

```
> library(vcd)
> Kappa(adaboost_cv$confusion)

      value     ASE     z Pr(>|z|)
Unweighted 0.3397 0.03255 10.44 1.676e-25
Weighted    0.3397 0.03255 10.44 1.676e-25
```

With a kappa of 0.3397, the boosted model is slightly outperforming the bagged decision trees, which had a kappa of around 0.3319. Let's see how boosting compares to another ensemble method.



Note that the prior results were obtained using R version 4.2.3 on a Windows PC and verified on Linux. At the time this was written, slightly different results are obtained using R 4.2.3 for Apple silicon on a recent MacBook Pro. Also note that the AdaBoost.M1 algorithm can be tuned with `caret` by specifying `method = "AdaBoost.M1"`.

Random forests

Yet another tree-based ensemble-based method, called **random forests**, builds upon the principles of bagging but adds additional diversity to the decision trees by only allowing the algorithm to choose from a randomly selected subset of features each time it attempts to split. Beginning at the root node, the random forest algorithm might only be allowed to choose from a small number of features selected at random from the full set of predictors; at each subsequent split, a different random subset is provided. As is the case for bagging, once the ensemble of trees (the forest) is generated, the algorithm performs a simple vote to make the final prediction.



For more detail on how random forests are constructed, refer to *Random Forests, Breiman L, Machine Learning, 2001, Vol. 45, pp. 5-32*. Note that the phrase “random forests” is trademarked by Breiman and Cutler but is used colloquially to refer to any type of decision tree ensemble. A pedant would use the more general term **decision tree forests** except when referring to their specific implementation.

The fact that each tree is built on different and randomly selected sets of features helps ensure that each tree in the ensemble is unique. It is even possible that two trees in the forest may have been built from completely different sets of features. Random feature selection limits the decision tree’s greedy heuristic from picking the same low-hanging fruit each time the tree is grown, which may help the algorithm discover subtle patterns that the standard tree-growing method may miss. On the other hand, the potential for overfitting is limited given that each tree has just one vote of many in the forest.

Given these strengths, it is no surprise that the random forest algorithm quickly grew to become one of the most popular learning algorithms—only recently has its hype been surpassed by a newer ensemble method, which you will learn about shortly. Random forests combine versatility and power into a single machine learning approach and are not especially prone to overfitting or underfitting. Because the tree-growing algorithm uses only a small, random portion of the full feature set, random forests can handle extremely large datasets, where the so-called curse of dimensionality might cause other models to fail. At the same time, its predictive performance on most learning tasks is as good as, if not better than, all but the most sophisticated methods. The following table summarizes the strengths and weaknesses of random forest models:

Strengths	Weaknesses
<ul style="list-style-type: none">An all-purpose model that performs well on most problems, including both classification and numeric predictionCan handle noisy or missing data as well as categorical or continuous featuresSelects only the most important featuresCan be used on data with an extremely large number of features or examples	<ul style="list-style-type: none">Unlike a decision tree, the model is not easily interpretableMay struggle with categorical features with very large numbers of levelsCannot be extensively tuned if greater performance is desired

Their strong performance combined with the ease of use makes random forests a terrific place to begin most real-world machine learning projects. The algorithm also provides a solid benchmark for other comparisons with highly tuned models, as well as the other, more complex approaches you will learn about later.

For a hands-on demonstration of random forests, we'll apply the technique to the credit-scoring data we've been using in this chapter. Although there are several packages with random forest implementations in R, the aptly named `randomForest` package is perhaps the simplest, while the `ranger` package offers much better performance on large datasets. Both are supported by the `caret` package for experimentation and automated parameter tuning. The syntax for training a model with `randomForest` is as follows:

Random forest syntax

Using the `randomForest()` function in the `randomForest` package

Building the classifier:

```
m <- randomForest(train, class, ntree = 500, mtry = sqrt(p))
```

- `train` is a data frame containing training data
- `class` is a factor vector with the class for each row in the training data
- `ntree` is an integer specifying the number of trees to grow
- `mtry` is an optional integer specifying the number of features to randomly select at each split (uses `sqrt(p)` by default, where `p` is the number of features in the data)

The function will return a random forest object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "response")
```

- `m` is a model trained by the `randomForest()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier
- `type` is either `"response"`, `"prob"`, or `"votes"` and is used to indicate whether the predictions vector should contain the predicted class, the predicted probabilities, or a matrix of vote counts, respectively

The function will return predictions according to the value of the `type` parameter.

Example:

```
credit_model <- randomForest(credit_train, loan_default)
credit_prediction <- predict(credit_model, credit_test)
```

Figure 14.6: Random forest syntax

By default, the `randomForest()` function creates an ensemble of 500 decision trees that each consider `sqrt(p)` random features at each split, where `p` is the number of features in the training dataset and `sqrt()` refers to R's square root function. For example, since the credit data has 16 features, each of the 500 decision trees would be allowed to consider only $\sqrt{16} = 4$ predictors each time the algorithm attempts to split.

Whether or not these default `ntree` and `mtry` parameters are appropriate depends on the nature of the learning task and training data. Generally, more complex learning problems and larger datasets (both more features as well as more examples) warrant a larger number of trees, though this needs to be balanced with the computational expense of training more trees. Once the `ntree` parameter is set to a sufficiently large value, the `mtry` parameter can be tuned to determine the best setting; however, the default tends to work well in practice. Assuming the number of trees is large enough, the number of randomly selected features can be surprisingly low before performance is degraded—but trying a few values is still a good practice. Ideally, the number of trees should be set large enough such that each feature has a chance of appearing in several models.

Let's see how the default `randomForest()` parameters work with the credit data. We'll train the model just as we have done with other learners. As usual, the `set.seed()` function ensures that the result can be replicated:

```
> library(randomForest)
> set.seed(300)
> rf <- randomForest(default ~ ., data = credit)
```

For a summary of model performance, we can simply type the resulting object's name:

```
> rf
```

```
Call:
randomForest(formula = default ~ ., data = credit)
Type of random forest: classification
Number of trees: 500
No. of variables tried at each split: 4

OOB estimate of error rate: 23.3%
Confusion matrix:
 no yes class.error
no 638 62 0.08857143
yes 171 129 0.57000000
```

The output shows that the random forest included 500 trees and tried four variables at each split, as expected. At first glance, you might be alarmed at the seemingly poor performance according to the confusion matrix—the error rate of 23.3 percent is far worse than the resubstitution error of any of the other ensemble methods so far. However, this confusion matrix does not show a resubstitution error. Instead, it reflects the **out-of-bag error rate** (listed in the output as `OOB estimate of error rate`), which, unlike a resubstitution error, is an unbiased estimate of the test set error. This means that it should be a fair estimate of future performance.

The out-of-bag estimate is computed using a clever technique during the construction of the random forest. Essentially, any example not selected for a single tree's bootstrap sample can be used to test the model's performance on unseen data. At the end of the forest construction, for each of the 1,000 examples in the dataset, any trees that did not use the example in training are allowed to make a prediction. These predictions are tallied, and a vote is taken to determine the single final prediction for the example. The total error rate of such predictions across all 1,000 examples becomes the out-of-bag error rate. Because each prediction uses only a subset of the forest, it is not equivalent to a true validation or test set estimation, but it is a reasonable substitute.



In *Chapter 10, Evaluating Model Performance*, it was stated that any given example has a 63.2 percent chance of being included in a bootstrap sample. This implies that an average of 36.8 percent of the 500 trees in the random forest voted for each of the 1,000 examples in the out-of-bag estimate.

To calculate the kappa statistic on the out-of-bag predictions, we can use the function in the `vcd` package as follows. The code applies the `Kappa()` function to the first two rows and columns of the `confusion` object, which stores the confusion matrix of the out-of-bag predictions for the `rf` random forest model object:

```
> library(vcd)
> Kappa(rf$confusion[1:2,1:2])
```

	value	ASE	z	$\text{Pr}(> z)$
Unweighted	0.381	0.03215	11.85	2.197e-32
Weighted	0.381	0.03215	11.85	2.197e-32

With a kappa statistic of `0.381`, the random forest is our best-performing model yet. Its performance was better than the bagged decision tree ensemble, which had a kappa of about `0.332`, as well as the AdaBoost.M1 model, which had a kappa of about `0.340`.

The `ranger` package, as mentioned previously, is a substantially faster implementation of the random forest algorithm. For a dataset as small as the credit dataset, optimizing for computational efficiency may be less important than ease of use, and by default, `ranger` sacrifices some conveniences in order to increase speed and reduce the memory footprint. Consequently, although the `ranger` function is nearly identical to `randomForest()` in syntax, in practice, you may find that it breaks existing code or takes a bit of digging through the help pages.

To recreate the previous model using `ranger`, we simply change the function name:

```
> library(ranger)
> set.seed(300)
> m_ranger <- ranger(default ~ ., data = credit)
```

The resulting model has quite a similar out-of-bag prediction error:

```
> m_ranger
```

```
Ranger result

Call:
ranger(default ~ ., data = credit)

Type:           Classification
Number of trees:      500
Sample size:        1000
Number of independent variables: 16
Mtry:              4
Target node size:    1
Variable importance mode: none
Splitrule:          gini
OOB prediction error: 23.10 %
```

We can compute kappa much as before while noting the slight difference in how the model's confusion matrix sub-object was named:

```
> Kappa(m_ranger$confusion.matrix)
      value     ASE      z Pr(>|z|)
Unweighted 0.381 0.0321 11.87 1.676e-32
Weighted   0.381 0.0321 11.87 1.676e-32
```

The kappa value is 0.381, which is the same as the result from the earlier random forest model. Note that this is coincidental, as the two algorithms are not guaranteed to produce identical results.



As with AdaBoost, the prior results were obtained using R version 4.2.3 on a Windows PC and verified on Linux. At the time this was written, slightly different results are obtained using R 4.2.3 for Apple silicon on a recent MacBook Pro.

Gradient boosting

Gradient boosting is an evolution of the boosting algorithm based on the finding that it is possible to treat the boosting process as an optimization problem to be solved using the gradient descent technique. We first encountered gradient descent in *Chapter 7, Black-Box Methods – Neural Networks and Support Vector Machines*, where it was introduced as a solution to optimize the weights in a neural network. You may recall that a cost function—essentially, the prediction error—relates the input values to the target. Then, by systematically analyzing how changes to the weights affect the cost, it is possible to find the set of weights that minimizes the cost. Gradient boosting treats the process of boosting in much the same way, with the weak learners in the ensemble being treated as the parameters to optimize. Models using this technique are termed **gradient boosting machines** or **generalized boosting models**—both of which can be abbreviated as **GBMs**.



For more on GBMs, see *Greedy Function Approximation: A Gradient Boosting Machine*, Friedman JH, 2001, *Annals of Statistics* 29(5):1189–1232.

The following table summarizes the strengths and weaknesses of GBMs. In short, gradient boosting is extremely powerful and can produce some of the most accurate models but may require tuning to find the balance between over- and underfitting.

Strengths	Weaknesses
<ul style="list-style-type: none"> An all-purpose classifier that can perform extremely well on both classification and numeric prediction Can achieve even better performance than random forests Performs well on large datasets 	<ul style="list-style-type: none"> May require tuning to match the performance of the random forest algorithm and more extensive tuning to exceed its performance Because there are several hyperparameters to tune, finding the best combination requires many iterations and more computing power

We'll use the `gbm()` function in the `gbm` package for creating GBMs for both classification and numeric prediction. You'll need to install and load this package to your R session if you haven't already. As the following box shows, the syntax is like the machine learning functions used previously, but it has several new parameters that may need to be adjusted. These parameters control the complexity of the model and the balance between over- and underfitting. Without tuning, the GBM may not perform as well as simpler methods, but it generally can surpass the performance of most other methods once parameter values have been optimized.

Gradient Boosting Machine (GBM) syntax

Using the `gbm()` function in the `gbm` package

Building the classifier:

```
m <- gbm(target ~ predictors, data = mydata,
           distribution = "bernoulli", n.trees = 100, interaction.depth = 1,
           n.minobsinnode = 10, shrinkage = 0.1)
```

- `target` is the outcome in the `mydata` data frame to be modeled
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` specifies the data frame in which the `target` and `predictors` variables can be found
- `distribution` is the form of target variable; can generally omit this to allow `gbm()` to determine it automatically
- `n.trees` is the number of boosting iterations (total number of trees to fit)
- `interaction.depth` is an integer specifying the maximum depth of each decision tree; deeper trees allow more interactions among predictors
- `n.minobsinnode` is an integer specifying the minimum number of observations in the trees' leaf nodes; smaller or larger values may lead to over- or under-fitting
- `shrinkage` refers to the learning rate, usually between 0.001 and 0.1; smaller values require more trees but may result in better models

The function will return a `gbm` object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "link")
```

- `m` is a model trained by the `gbm()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier
- `type` is either `"link"` or `"response"` and for classification models is used to indicate whether the predictions vector should contain the predicted class or the predicted probabilities, respectively. For numeric prediction this is unnecessary.

The function will return predictions according to the value of the `type` parameter.

Example:

```
credit_model <- gbm(default ~ ., data = credit_train)
credit_prediction <- predict(credit_model, credit_test,
                           type = "response")
```

Figure 14.7: Gradient boosting machine (GBM) syntax

We can train a simple GBM to predict loan defaults on the credit dataset as follows. For simplicity, we set `stringsAsFactors = TRUE` to avoid recoding the predictors, but then the target default feature must be converted back to a binary outcome, as the `gbm()` function requires this for binary classification. We'll create a random sample for training and testing, then apply the `gbm()` function to the training data, leaving the parameters set to their defaults:

```
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> credit$default <- ifelse(credit$default == "yes", 1, 0)
> set.seed(123)
> train_sample <- sample(1000, 900)
> credit_train <- credit[train_sample, ]
> credit_test <- credit[-train_sample, ]
> library(gbm)
> set.seed(300)
> m_gbm <- gbm(default ~ ., data = credit_train)
```

Typing the name of the model provides some basic information about the GBM process:

```
> m_gbm

gbm(formula = default ~ ., data = credit_train)
A gradient boosted model with 59 bernoulli loss function.
100 iterations were performed.
There were 16 predictors of which 14 had non-zero influence.
```

More importantly, we can evaluate the model on the test set. Note that we need to convert the predictions to binary, as they are given as probabilities. If the probability of loan default is greater than 50 percent, we will predict default, otherwise, we predict non-default. The table shows the agreement between the predicted and actual values:

```
> p_gbm <- predict(m_gbm, credit_test, type = "response")
> p_gbm_c <- ifelse(p_gbm > 0.50, 1, 0)
> table(credit_test$default, p_gbm_c)

    p_gbm_c
1
0 60 5
1 21 14
```

To measure the performance, we'll apply the `Kappa()` function to this table:

```
> library(vcd)
> Kappa(table(credit_test$default, p_gbm_c))
```

	value	ASE	z	Pr(> z)
Unweighted	0.3612	0.09529	3.79	0.0001504
Weighted	0.3612	0.09529	3.79	0.0001504

The resulting kappa value of about `0.361` is better than what was obtained with the boosted decision tree, but worse than the random forest model. Perhaps with a bit of tuning, we can get this higher.

We'll use the `caret` package to tune the GBM model and obtain a more robust performance measure. Recall that tuning needs a search grid, which we can define for GBM as follows. This will test three values for three of the `gbm()` function parameters and one value for the remaining parameter, which results in $3 * 3 * 3 * 1 = 27$ models to evaluate:

```
> grid_gbm <- expand.grid(
  n.trees = c(100, 150, 200),
  interaction.depth = c(1, 2, 3),
  shrinkage = c(0.01, 0.1, 0.3),
  n.minobsinnode = 10
)
```

Next, we set the `trainControl` object to select the best model from a 10-fold CV experiment:

```
> library(caret)
> ctrl <- trainControl(method = "cv", number = 10,
  selectionFunction = "best")
```

Lastly, we read in the `credit` dataset and supply the required objects to the `caret()` function while specifying the `gbm` method and the `Kappa` performance metric. Depending on the capabilities of your computer, this may take a few minutes to run:

```
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> set.seed(300)
> m_gbm_c <- train(default ~ ., data = credit, method = "gbm",
  trControl = ctrl, tuneGrid = grid_gbm,
  metric = "Kappa",
  verbose = FALSE)
```

Typing the name of the object shows the results of the experiment. Note that some lines of output have been omitted for brevity, but the full output contains 27 rows—one for each model evaluated:

```
> m_gbm_c
```

Stochastic Gradient Boosting

```
1000 samples  
16 predictor  
2 classes: 'no', 'yes'
```

```
No pre-processing
```

```
Resampling: Cross-Validated (10 fold)
```

```
Summary of sample sizes: 900, 900, 900, 900, 900, ...
```

```
Resampling results across tuning parameters:
```

shrinkage	interaction.depth	n.trees	Accuracy	Kappa
0.10	1	100	0.737	0.269966697
0.10	1	150	0.738	0.295886773
0.10	1	200	0.742	0.320157816
0.10	2	100	0.747	0.327928587
0.10	2	150	0.750	0.347848347
0.10	2	200	0.759	0.380641164
0.10	3	100	0.747	0.342691964
0.10	3	150	0.748	0.356836684
0.10	3	200	0.764	0.394578005

```
Tuning parameter 'n.minobsinnode' was held constant at a value of 10
```

```
Kappa was used to select the optimal model using the largest value.
```

```
The final values used for the model were n.trees = 200,
```

```
interaction.depth = 3, shrinkage = 0.1 and n.minobsinnode = 10.
```

From the output, we can see that the best GBM model had a kappa of 0.394, which exceeds the random forest trained previously. With additional tuning, it may be possible to bring the kappa up even higher. Or, as you will see in the next section, a more intensive form of boosting can be employed in the pursuit of even better performance.

Extreme gradient boosting with XGBoost

A cutting-edge implementation of the gradient boosting technique can be found in the **XGBoost** algorithm (<https://xgboost.ai>), which takes boosting to the “extreme” by improving the algorithm’s efficiency and performance. In the time since the algorithm was introduced in 2014, XGBoost has been found on top of the leaderboards of many machine learning competitions. In fact, according to the algorithm’s authors, among 29 winning solutions on Kaggle in 2015, a total of 17 used the XGBoost algorithm. Likewise, in the 2015 KDD Cup (described in *Chapter 11, Being Successful with Machine Learning*), all of the top 10 winners used XGBoost. Today, the algorithm is still the champion for traditional machine learning problems involving classification and numeric prediction, whereas its closest challenger, deep neural networks, tends to win only on unstructured data, such as image, audio, and text processing.



For more information on XGBoost, see *XGBoost: A Scalable Tree Boosting System*, Chen T and Guestrin C, 2016. <https://arxiv.org/abs/1603.02754>.

The great power of the XGBoost algorithm comes with the downside that the algorithm is not quite as easy to use and requires substantially more tuning than other methods examined so far. On the other hand, its performance ceiling tends to be higher than any other approach. The strengths and weaknesses of XGBoost are found in the following table:

Strengths	Weaknesses
<ul style="list-style-type: none">• An all-purpose classifier that can perform extremely well on both classification and numeric prediction• Perhaps undisputedly, the current champion of performance on traditional learning problems; wins virtually every machine learning competition on structured data• Highly scalable, performs well on large datasets, and can be run in parallel on distributed computing platforms	<ul style="list-style-type: none">• More challenging to use than other functions, as it relies on external frameworks that do not use native R data structures• Requires extensive tuning of a large set of hyperparameters that can be difficult to understand without a strong math background• Because there are many tuning parameters, finding the best combination requires many iterations and more computing power• Results in a “black box” model that is nearly impossible to interpret without explainability tools

To apply the algorithm, we'll use the `xgboost()` function in the `xgboost` package, which provides an R interface to the XGBoost framework. Entire books could be written about this framework, as it includes features for many types of machine learning tasks, and is highly extensible and adaptable to many high-performance computing environments. For more information about the XGBoost framework, see the excellent documentation on the web at <https://xgboost.readthedocs.io>. Our work will focus on a narrow slice of its functionality, as shown in the following syntax box, which is much denser than those for other algorithms due to a large increase in complexity and hyperparameters that may be tuned:

XGBoost (XGB) syntax	
Using the <code>xgboost()</code> function in the <code>xgboost</code> package	
Setting model parameters:	
<pre>params.xgb = list(objective = "binary:logistic", max_depth = 6, eta = 0.3, gamma = 0, colsample_bytree = 1, min_child_weight = 1, subsample = 1)</pre> <ul style="list-style-type: none"> • <code>objective</code> is determined by the target to be modeled; "binary:logistic" is for binary classification; use "multi:softprob" for categorical outcomes, "reg:squarederror" for regression, or "count:poisson" for count data • <code>max_depth</code> is between 0 and infinity and defines the maximum depth of any tree; larger values can find more specific patterns but risk overfitting • <code>eta</code> is between 0 and 1 and affects the learning rate; low values limit overfitting, but increase training time and will require more boosting iterations (<code>nrounds</code>) • <code>gamma</code> is between 0 and 1 and governs whether the algorithm will continue splitting; lower values can find more specific patterns but risk overfitting • <code>colsample_bytree</code> is between 0 and 1 and defines the % of features that will be selected at random for each tree; use <code>colsample_bynode</code> to instead select random features at each split • <code>min_child_weight</code> is between 0 and infinity and is analogous to the minimum examples needed to split; small values find more specific patterns but may overfit • <code>subsample</code> is between 0 and 1 and defines the % of randomly selected examples for each iteration; small values may prevent overfitting but require more iterations 	
Note that the above includes only a subset of parameters and hyperparameters. See ?xgboost or https://xgboost.readthedocs.io/en/latest/parameter.html for the full list.	
Building the classifier:	
<pre>m <- xgboost(params, data = mydata, label = mylabels, rounds = n)</pre> <ul style="list-style-type: none"> • <code>params</code> is a list of XGBoost hyperparameters (as defined above) • <code>data</code> is a matrix or sparse matrix with the features to be used for training • <code>label</code> is a vector of target values to be used for training • <code>rounds</code> is the maximum number of boosting iterations 	
The function will return an <code>xgb</code> object that can be used to make predictions.	
Making predictions:	
<pre>p <- predict(m, test)</pre> <ul style="list-style-type: none"> • <code>m</code> is a model trained by the <code>xgboost()</code> function • <code>test</code> is a data frame containing test data in the same form as the training data 	
The function returns predicted probabilities (for classifiers) or values (for numeric models).	

Figure 14.8: XGBoost (XGB) syntax

One of the challenges with using XGBoost in R is its need to use data in matrix format rather than R's preferred formats of tibbles or data frames. Because XGBoost is designed for extremely large datasets, it can also use sparse matrices, such as those discussed in previous chapters. You may recall that a sparse matrix only stores non-zero values, which makes it more memory-efficient than traditional matrices when many feature values are zeros.

Data in matrix form is often sparse because factors are typically one-hot or dummy coded during the transition between the data frame and matrix. These encodings create additional columns for additional levels of the factor, and all columns are set to zero except the one "hot" value that indicates the level for the given example. In the case of dummy coding, one feature level is left out of the transformation, so it results in one fewer column than one-hot; the missing level can be indicated by the presence of zeros in all of the $n - 1$ columns.



One-hot and dummy coding generally produce the same results, with the exception that statistics-based models like regression require dummy coding and will present errors or warning messages if one-hot is used instead.

Let's begin by reading the `credit.csv` file and creating a sparse matrix of data from the `credit` data frame. The `Matrix` package provides a function to perform this task, which uses the R formula interface to determine the columns to include in the matrix. Here, the formula `~ . -default` tells the function to use all features except `default`, which we don't want in the matrix, as this is our target feature for prediction:

```
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> library(Matrix)
> credit_matrix <- sparse.model.matrix(~ . -default, data = credit)
```

To confirm our work, let's check the dimensions of the matrix:

```
> dim(credit_matrix)
```

```
[1] 1000   36
```

We still have 1,000 rows, but the columns have increased from 16 features in the original data frame to 36 in the sparse matrix. This is due to the dummy coding that was applied automatically when converting to matrix form. We can see this if we examine the first five rows and 15 columns of the sparse matrix using the `print()` function:

```
> print(credit_matrix[1:5, 1:15])
```

```
5 x 15 sparse Matrix of class "dgCMatrix"
[[ suppressing 15 column names '(Intercept)', 'checking_balance > 200
DM', 'checking_balance1 - 200 DM' ... ]]

 1 1 . . . 6 . . . . . 1 . 1169
 2 1 . 1 . 48 1 . . . . . 1 . 5951
 3 1 . . 1 12 . . . . . 1 . . 2096
 4 1 . . . 42 1 . . . . . 1 . 7882
 5 1 . . . 24 . . 1 . 1 . . . 4870
```

The matrix is depicted with the dot (.) character indicating cells with zero values. The first column (1, 2, 3, 4, 5) is the row number and the second column (1, 1, 1, 1, 1) is a column for the intercept term, which was added automatically by the R formula interface. Two columns have numbers (6, 48, ...) and (1169, 5951, ...) that correspond to the numeric values of the `months_loan_duration` and `amount` features, respectively. All other columns are dummy-coded versions of factor variables. For instance, the third, fourth, and fifth columns reflect the `checking_balance` feature, with a 1 in the third column indicating a value of '`> 200 DM`', a 1 in the fourth column indicating '`1 - 200 DM`', and a 1 in the fifth column indicating the '`unknown`' feature value. Rows showing the sequence `. . .` in columns 3, 4, and 5 fall into the reference category, which was the '`< 0 DM`' feature level.

Since we are not building a regression model, the intercept column full of 1 values is useless for this analysis and can be removed from the matrix:

```
> credit_matrix <- credit_matrix[, -1]
```

Next, we'll split the matrix at random into training and test sets using a 90-10 split as we've done before:

```
> set.seed(12345)
> train_ids <- sample(1000, 900)
> credit_train <- credit_matrix[train_ids, ]
> credit_test <- credit_matrix[-train_ids, ]
```

To confirm the work was done correctly, we'll check the dimensions of these matrices:

```
> dim(credit_train)
```

```
[1] 900 35
```

```
> dim(credit_test)
```

```
[1] 100 35
```

As expected, the training set has 900 rows and 35 columns, and the test set has 100 rows and a matching set of columns.

Lastly, we'll create training and test vectors of labels for `default`, the target to be predicted. These are transformed from factors to binary 1 or 0 values using an `ifelse()` function so that they can be used to train and evaluate the XGBoost model, respectively:

```
> credit_train_labels <-  
  ifelse(credit[train_ids, c("default")] == "yes", 1, 0)  
> credit_test_labels <-  
  ifelse(credit[-train_ids, c("default")] == "yes", 1, 0)
```

We're now ready to start building the model. After installing the `xgboost` package, we'll load the library and start to define the hyperparameters for training. Without knowing where else to begin, we'll set the values to their defaults:

```
> library(xgboost)  
> params.xgb <- list(objective = "binary:logistic",  
  max_depth = 6,  
  eta = 0.3,  
  gamma = 0,  
  colsample_bytree = 1,  
  min_child_weight = 1,  
  subsample = 1)
```

Next, after setting the random seed, we'll train the model, supplying our parameters object as well as the matrix of training data and the target labels. The `nrounds` parameter determines the number of boosting iterations. Without a better guess, we'll set this to 100, which is a common starting point due to empirical evidence suggesting that results tend to improve very little beyond this value. Lastly, the `verbose` and `print_every_n` options are used to turn on diagnostic output and display the progress after every 10 boosting iterations:

```
> set.seed(555)  
> xgb_credit <- xgboost(params = params.xgb,  
  data = credit_train,  
  label = credit_train_labels,  
  nrounds = 100,  
  verbose = 1,  
  print_every_n = 10)
```

The output should appear as the training is completed, showing that all 100 iterations occurred and the training error (labeled `train-logloss`) continued to decline with additional rounds of boosting:

```
[1] train-logloss:0.586271
[11] train-logloss:0.317767
[21] train-logloss:0.223844
[31] train-logloss:0.179252
[41] train-logloss:0.135629
[51] train-logloss:0.108353
[61] train-logloss:0.090580
[71] train-logloss:0.077314
[81] train-logloss:0.065995
[91] train-logloss:0.057018
[100] train-logloss:0.050837
```

Knowing whether additional iterations would help the model performance or result in overfitting is something we can determine via tuning later. Before doing so, let's look at the performance of this trained model on the test set, which we held out earlier. First, the `predict()` function obtains the predicted probability of loan default for each row of test data:

```
> prob_default <- predict(xgb_credit, credit_test)
```

Then, we use `ifelse()` to predict a default (value 1) if the probability of a default is at least 0.50, or non-default (value 0) otherwise:

```
> pred_default <- ifelse(prob_default > 0.50, 1, 0)
```

Comparing the predicted to actual values, we find an accuracy of $(62 + 14) / 100 = 76$ percent:

```
> table(pred_default, credit_test_labels)
      credit_test_labels
pred_default  0  1
              0 62 13
              1 11 14
```

On the other hand, the kappa statistic suggests there is still room to improve:

```
> library(vcd)
> Kappa(table(pred_default, credit_test_labels))
```

	value	ASE	z	Pr(> z)
Unweighted	0.3766	0.1041	3.618	0.0002967
Weighted	0.3766	0.1041	3.618	0.0002967

The value of 0.3766 is a bit lower than the 0.394 we obtained with the GBM model, so perhaps a bit of hyperparameter tuning can help. For this, we'll use `caret`, starting with a tuning grid comprising a variety of options for each of the hyperparameters:

```
> grid_xgb <- expand.grid(
  eta = c(0.3, 0.4),
  max_depth = c(1, 2, 3),
  colsample_bytree = c(0.6, 0.8),
  subsample = c(0.50, 0.75, 1.00),
  nrounds = c(50, 100, 150),
  gamma = c(0, 1),
  min_child_weight = 1
)
```

The resulting grid contains $2 * 3 * 2 * 3 * 3 * 2 * 1 = 216$ different combinations of `xgboost` hyperparameter values. We'll evaluate each of these potential models in `caret` using 10-fold CV, as we've done for other models. Note that the `verbosity` parameter is set to zero so that the `xgboost()` function output is suppressed for the many iterations:

```
> library(caret)
> ctrl <- trainControl(method = "cv", number = 10,
  selectionFunction = "best")
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> set.seed(300)
> m_xgb <- train(default ~ ., data = credit, method = "xgbTree",
  trControl = ctrl, tuneGrid = grid_xgb,
  metric = "Kappa", verbosity = 0)
```

Depending on the capabilities of your computer, the experiment may take a few minutes to complete, but once it finishes, typing `m_xgb` will provide the results of all 216 models tested. We can also obtain the best model directly as follows:

```
> m_xgb$bestTune
```

nrounds	max_depth	eta	gamma	colsample_bytree
50	3	0.4	1	0.6

```
min_child_weight subsample
1
```

The kappa value for this model can be found using the `max()` function to find the highest value as follows:

```
> max(m_xgb$results["Kappa"])
[1] 0.4062946
```

The kappa value of `0.406` is our best-performing model so far, exceeding the `0.394` of the GBM model and the `0.381` of the random forest. The fact that XGBoost required so little effort to train—with a bit of fine-tuning—yet still surpassed other powerful techniques provides examples of why it always seems to win machine learning competitions. Yet, with even more tuning, it may be possible to go higher still! Leaving that as an exercise to you, the reader, we'll now turn our attention to the question of why all of these popular ensembles seem to focus exclusively on decision tree-based methods.

Why are tree-based ensembles so popular?

After reading the prior sections, you would not be the first person to wonder why ensembling algorithms seem to always be built upon decision trees. Although trees are not required for building an ensemble, there are several reasons why they are especially well-suited for this process. You may have noted some of them already:

- Ensembles work best with diversity, and because decision trees are not robust to small changes in the data, random sampling the same training data can easily create a diverse set of tree-based models
- Because of the greedy “divide-and-conquer” based algorithm, decision trees are computationally efficient and perform relatively well despite this fact
- Decision trees can be grown purposely large or small to overfit and underfit as needed
- Decision trees can automatically ignore irrelevant features, which reduces the negative impact of the “curse of dimensionality”
- Decision trees can be used for numeric prediction as well as classification

Based on these characteristics, it is not difficult to see why we've ended up with a wealth of tree-based ensembling approaches such as bagging, boosting, and random forests. The distinctions among them are subtle but important.

The following table may help contrast the tree-based ensembling algorithms covered in this chapter:

Ensembling Algorithm	Allocation Function	Combination Function	Other Notes
Bagging	Provides each learner with a bootstrap sample of the training data	The learners are combined using a vote for classification or a weighted average for numeric prediction	Uses an independent ensemble — the learners can be run in parallel
Boosting	The first learner is given a random sample; subsequent samples are weighted to have more difficult-to-predict cases	The learners' predictions are combined as above but weighted according to their performance on training data	Uses a dependent ensemble — each tree in the sequence receives data that earlier trees found challenging
Random Forest	Like bagging, each tree receives a bootstrap sample of training data; however, features are also randomly selected for each tree split	Similar to bagging	Similar to bagging, but the added diversity via random feature selection allows additional benefits for larger ensembles
Gradient Boosting Machine (GBM)	Conceptually similar to boosting	Similar to boosting, but there are many more learners and they comprise a complex mathematical function	Uses gradient descent to make a more efficient boosting algorithm; the trees are generally not very deep (decision tree “stumps”) but there are many more of them; requires more tuning

eXtreme Gradient Boosting (XGB)	Similar to GBM	Similar to GBM	Similar to GBM but more extreme; uses optimized data structures, parallel processing, and heuristics to create a very performant boosting algorithm; tuning is essential
---------------------------------	----------------	----------------	--

To be able to distinguish among these approaches reveals a deep understanding of several aspects of ensembling. Additionally, the most recent techniques, such as random forests and gradient boosting, are among the best-performing learning algorithms and are being used as off-the-shelf solutions to solve some of the most challenging business problems. This may help explain why companies hiring data scientists and machine learning engineers often ask candidates to describe or compare these algorithms as part of the interview process. Thus, even though tree-based ensembling algorithms are not the only approach to machine learning, it is important to be aware of their potential uses. However, as the next section describes, trees aren't the only approach to building a diverse ensemble.

Stacking models for meta-learning

Rather than using a canned ensembling method like bagging, boosting, or random forests, there are situations in which a tailored approach to ensembling is warranted. Although these tree-based ensembling techniques combine hundreds or even thousands of learners into a single, stronger learner, the process is not much different than training a traditional machine learning algorithm, and suffers some of the same limitations, albeit to a lesser degree. Being based on decision trees that have been weakly trained and minimally tuned may, in some cases, put a ceiling on the ensemble's performance relative to one composed of a more diverse set of learning algorithms that have been extensively tuned with the benefit of human intelligence. Furthermore, although it is possible to parallelize tree-based ensembles like random forests and XGB, this only parallelizes the computer's effort—not the human effort of model building.

Indeed, it is possible to increase an ensemble's diversity by not only adding additional learning algorithms but by distributing the work of model building to additional human teams working in parallel. In fact, many of the world's competition-winning models were built by taking other teams' best models and ensembling them together.

This type of ensemble is conceptually quite simple, offering performance boosts that would be otherwise unobtainable, but can become complex in practice. Getting the implementation details correct is crucial to avoid disastrous levels of overfitting. Done correctly, the ensemble will perform at least as well as the strongest model in the ensemble, and often substantially better.

Examining **receiver operating characteristic (ROC)** curves, as introduced in *Chapter 10, Evaluating Model Performance*, provides a simple method to determine whether two or more models would benefit from ensembling. If two models have intersecting ROC curves, their **convex hull**—the outermost boundary that would be obtained by stretching an imaginary rubber band around the curves—represents a hypothetical model that can be obtained by interpolating, or combining, the predictions from these models. As depicted in *Figure 14.9*, two ROC curves with identical **area under the curve (AUC)** values of 0.70 might create a new model with an AUC of 0.72 when paired in an ensemble:

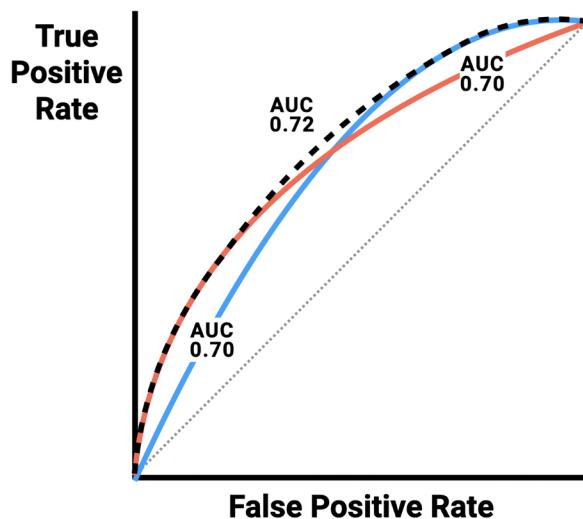


Figure 14.9: When two or more ROC curves intersect, their convex hull represents a potentially better classifier that can be generated by combining their predictions in an ensemble

Because this form of ensembling is performed largely by hand, a human needs to provide the allocation and combination functions for the models in the ensemble. In their simplest form, these can be implemented quite pragmatically. For example, suppose that the same training dataset has been given to three different teams. This is the allocation function. These teams can use this dataset however they see fit to build the best possible model using evaluation criteria of their choosing.

Next, each team is given the test set, and their models are used to make predictions, which must be combined into a single, final prediction. The combination function can take multiple different forms: the groups could vote, the predictions could be averaged, or the predictions could be weighted according to how well each group performed in the past. Even the simple approach of choosing one group at random is a viable strategy, assuming each group performs better than all others at least once in a while. Of course, even more intelligent approaches are possible, as you will soon learn.

Understanding model stacking and blending

Some of the most sophisticated custom ensembles apply machine learning to learn a combination function for the final prediction. Essentially, it is trying to learn which models can and cannot be trusted. This arbiter learner may realize that one model in the ensemble is a poor performer and shouldn't be trusted or that another deserves more weight in the ensemble. The arbiter function may also learn more complex patterns. For example, suppose that when models $M1$ and $M2$ agree on the outcome, the prediction is almost always accurate, but otherwise $M3$ is generally more accurate than either of the two. In this case, an additional arbiter model could learn to ignore the vote of $M1$ and $M2$ except when they agree. This process of using the predictions of several models to train a final model is called **stacking**.

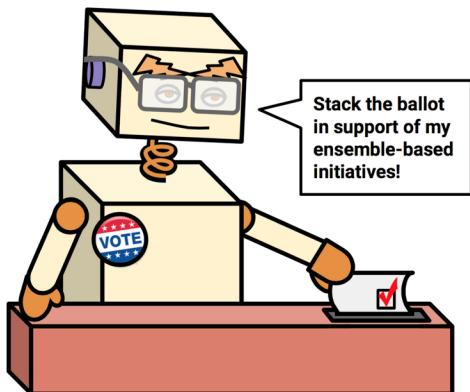


Figure 14.10: Stacking is a sophisticated ensemble that uses an arbiter learning algorithm to combine the predictions of a set of learners and make a final prediction

More broadly, stacking falls within a methodology known as **stacked generalization**. As formally defined, the stack is constructed using first-level models that have been trained via CV, and a second-level model or **meta-model** that is trained using the predictions for the out-of-fold samples—the examples the model does not see during training but is tested on during the CV process.

For example, suppose three first-level models are included in the stack and each one is trained using 10-fold CV. If the training dataset includes 1,000 rows, each of the three first-stage models is trained on 900 rows and tested on 100 rows ten times. The 100-row test sets, when combined, comprise the entire training dataset.

As all three models have made a prediction for every row of the training data, a new table can be constructed with four columns and 1,000 rows: the first three columns represent the predictions for the three models and column four represents the true value of the target. Note that because the predictions made for each of these 100 rows were made on the other 900 rows, all 1,000 rows are predictions on unseen data. This allows the second-stage meta-model, which is often a regression or logistic regression model, to learn which first-stage models perform better by training using the predicted values as the predictors of the true value. This process of finding the optimal combination of learners is sometimes called **super learning**, and the resulting model may be called a **super learner**. This process is often performed by machine learning software or packages, which train numerous learning algorithms in parallel and stack them together automatically.

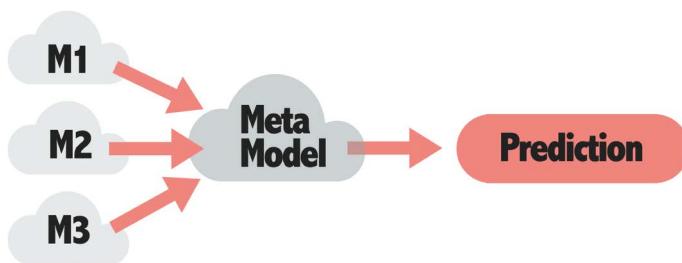
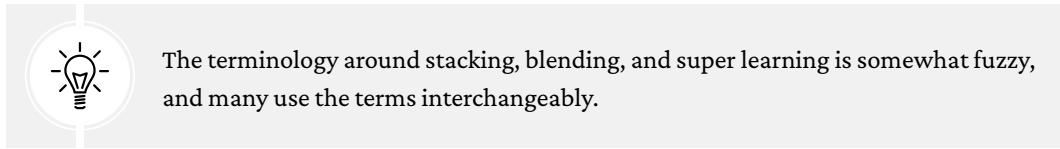


Figure 14.11: In a stacked ensemble, the second-stage meta-model or “super learner” learns from the predictions of the first-stage models on out-of-fold samples

For a more hands-on approach, a special case of stacked generalization called **blending** or **holdout stacking** provides a simplified way to implement stacking by replacing CV with a holdout sample. This allows the work to be distributed across teams more easily by merely dividing the training data into a training set for the first-level models and using a holdout set for the second-level meta-learner. It may also be less prone to overfitting the CV “information leak” described in *Chapter 11, Being Successful with Machine Learning*. Thus, even though it is a simple approach, it can be quite effective; blending is often what competition-winning teams do when they take other models and ensemble them together for better results.



Practical methods for blending and stacking in R

To perform blending in R requires a careful roadmap, as getting the details wrong can lead to extreme overfitting and models that perform no better than random guessing. The following figure illustrates the process. Begin by imagining that you are tasked with predicting loan defaults and have access to one million rows of past data. Immediately, you should partition the dataset into training and test sets; of course, the test set should be kept in a vault for evaluating the ensemble later. Assume the training set is 750,000 rows and the test set is 250,000 rows. The training set must then be divided yet again to create datasets for training the level one models and the level two meta-learner. The exact proportions are somewhat arbitrary, but it is customary to use a smaller set for the second-stage model—sometimes as low as ten percent. As *Figure 14.12* depicts, we might use 500,000 rows for level one and 250,000 rows for level two:

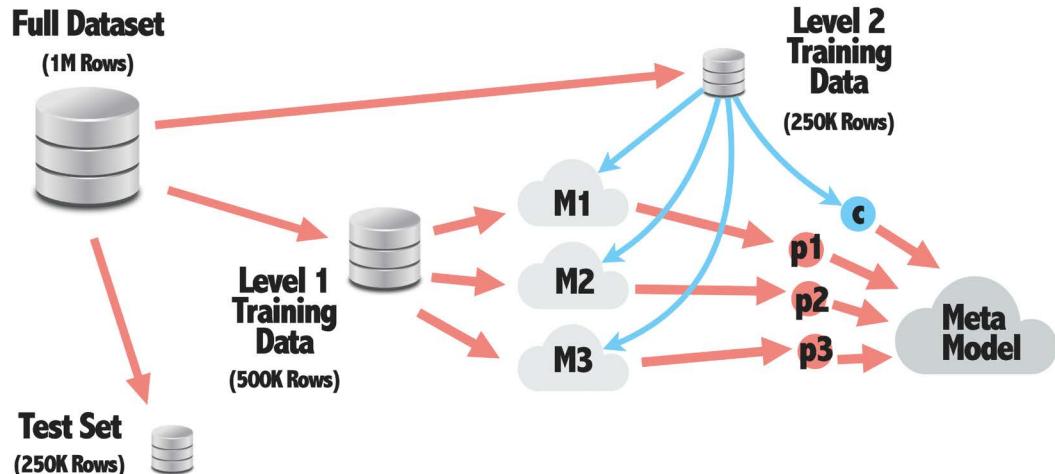


Figure 14.12: The full training dataset must be divided into distinct subsets for training the level one and level two models

The 500,000-row level one training dataset is used to train the first-level models exactly as we have done many times throughout this book. The M_1 , M_2 , and M_3 models may use any learning algorithm, and the work of building these models can even be distributed across different teams working independently.

There is no need for the models or teams to use the same set of features from the training data or the same form of feature engineering, assuming each team's feature engineering pipeline can be replicated or automated in the future when the ensemble is to be deployed. The important thing is that $M1$, $M2$, and $M3$ should be able to take a dataset with identical features and produce a prediction for each row.

The 250,000-row level two training dataset is then fed into the $M1$, $M2$, and $M3$ models after being processed through their associated feature engineering pipelines, and three vectors of 250,000 predictions are obtained. These vectors are labeled $p1$, $p2$, and $p3$ in the diagram. When combined with the 250,000 true values of the target (labeled c in the diagram) obtained from the level two training dataset, a four-column data frame is produced, as depicted in *Figure 14.13*:

M ₁ Prediction	M ₂ Prediction	M ₃ Prediction	Actual Value
Yes	Yes	Yes	Yes
No	No	Yes	Yes
No	Yes	Yes	Yes
Yes	No	No	No

Figure 14.13: The dataset used to train the meta-model is composed of the predictions from the first-level models and the actual target value from the level two training data

This type of data frame is used to create a meta-model, typically using regression or logistic regression, which predicts the actual target value (c in *Figure 14.12*) using the predictions of $M1$, $M2$, and $M3$ ($p1$, $p2$, and $p3$ in *Figure 14.12*) as predictors. In an R formula, this might be specified in a form like $c \sim p1 + p2 + p3$, which results in a model that weighs the input from three different predictions to make its own final prediction.

To estimate the future performance of this final meta-model, we must use the 250,000-row test set, which, as illustrated in *Figure 14.12* previously, was held out during the training process. As shown in *Figure 14.14*, the test dataset is then fed to the $M1$, $M2$, and $M3$ models and their associated feature engineering pipelines, and much like in the previous step, three vectors of 250,000 predictions are obtained. However, rather than $p1$, $p2$, and $p3$ being used to train a meta-model, they are now used as predictors for the existing meta-model to obtain a final prediction (labeled $p4$) for each of the 250,000 test cases. This vector can be compared to the 250,000 true values of the target in the test set to perform the performance evaluation and obtain an unbiased estimate of the ensemble's future performance.

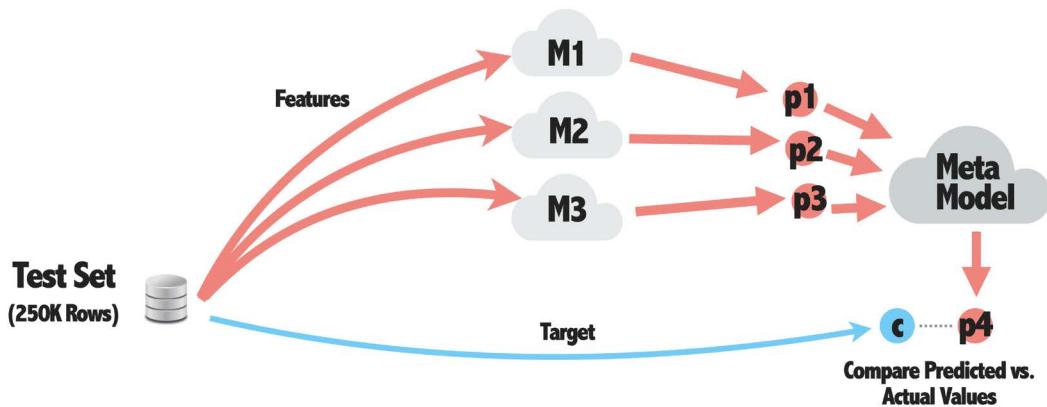


Figure 14.14: To obtain an unbiased estimate of the ensemble's future performance, the test set is used to generate predictions for the level one models, which are then used to obtain the meta-model's final predictions

The above methodology is flexible to create other interesting types of ensembles. *Figure 14.15* illustrates a blended ensemble that combines models trained on completely different subsets of features. Specifically, it envisions a learning task in which Twitter profile data is used to make a prediction about the user—perhaps their gender or whether they would be interested in purchasing a particular product:

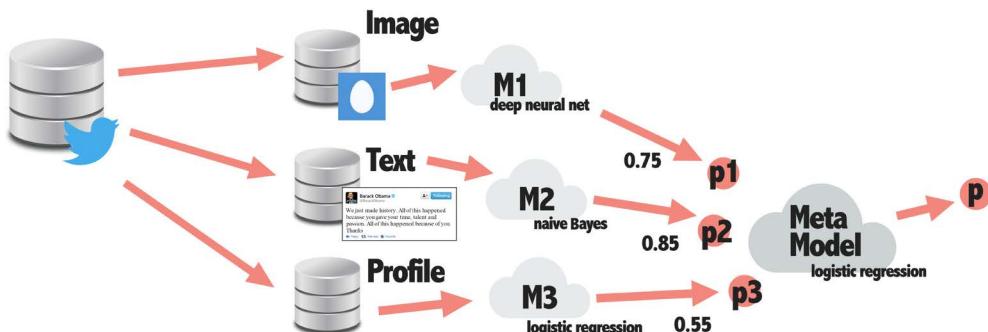


Figure 14.15: The stack's first-level models can be trained on different features in the training set, while the second-level model is trained on their predictions

The first model receives the profile's picture and trains a deep learning neural network with the image data to predict the outcome. Model two receives a set of tweets for the user and uses a text-based model like Naive Bayes to predict the outcome. Lastly, model three is a more conventional model using a traditional data frame of demographic data like location, total number of tweets, last login date, and so on.

All three models are combined, and the meta-model can learn whether the image, text, or profile data is most helpful for predicting the gender or purchasing behavior. Alternatively, because the meta-model is a logistic regression model like M_3 , it would be possible to supply the profile data directly to the second-stage model and skip the construction of M_3 altogether.

Aside from constructing blended ensembles by hand as described here, there is a growing set of R packages to assist with this process. The `caretEnsemble` package can assist with ensemble models trained with the `caret` package and ensure that the stack's sampling is handled correctly for stacking or blending. The `SuperLearner` package provides an easy way to create a super learner; it can apply dozens of base algorithms to the same dataset and stack them together automatically. As an off-the-shelf algorithm, this may be useful for building a powerful ensemble with the least amount of effort.

Summary

After reading this chapter, you should now know the approaches that are used to win data mining and machine learning competitions. Automated tuning methods can assist with squeezing every bit of performance out of a single model. On the other hand, tremendous gains are possible by creating groups of machine learning models called ensembles, which work together to achieve greater performance than single models can by working alone. A variety of tree-based algorithms, including random forests and gradient boosting, provide the benefits of ensembles but can be trained as easily as a single model. On the other hand, learners can be stacked or blended into ensembles by hand, which allows the approach to be carefully tailored to a learning problem.

With a variety of options for improving the performance of a model, where should someone begin? There is no single best approach, but practitioners tend to fall into one of three camps. First, some begin with one of the more sophisticated ensembles such as random forests or XGBoost, and spend most of their time tuning and feature engineering to achieve the highest possible performance for this model. A second group might try a variety of approaches, then collect the models into a single stacked or blended ensemble to create a more powerful learner. The third approach might be described as “throw everything at the computer and see what sticks.” This attempts to feed the learning algorithm as much data as possible and as quickly as possible, and is sometimes combined with automated feature engineering or dimensionality reduction techniques like those described in the previous chapters. With practice, you may be drawn to some of these ideas more than others, so feel free to use whichever works best for you.

Although this chapter was designed to help you prepare competition-ready models, note that your fellow competitors have access to the same techniques. You won't be able to get away with stagnancy; therefore, continue to add proprietary methods to your bag of tricks. Perhaps you can bring unique subject-matter expertise to the table, or perhaps your strengths include an eye for detail in data preparation. In any case, practice makes perfect, so take advantage of competitions to test, evaluate, and improve your machine learning skillset. In the next chapter—the last in this book—we'll look at ways to apply cutting-edge “big data” techniques to some highly specialized and difficult data tasks using R.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>



15

Making Use of Big Data

Although today's most exciting machine learning research is found in the realm of big data—computer vision, natural language processing, autonomous vehicles, and so on—most business applications are much smaller scale, using what might be termed, at best, “*medium*” data. As noted in *Chapter 12, Advanced Data Preparation*, true big data work requires access to datasets and computing facilities generally found only at very large tech companies or research universities. Even then, the actual job of using these resources is often primarily a feat of data engineering, which simplifies the data greatly before its use in conventional business applications.

The good news is that the headline-making research conducted at big data companies eventually trickles down and can be applied in simpler forms to more traditional machine learning tasks. This chapter covers a variety of approaches for making use of such big data methods in R. You will learn:

- How to borrow from the deep learning models developed at big data companies and apply them to conventional modeling tasks
- Strategies for reducing the complexity of large and unstructured big data formats like text and images so that they can be used for prediction
- Cutting-edge packages and approaches for accessing and modeling big datasets that may be too large to fit into memory

Despite R's reputation for being ill equipped for big data projects, the efforts of the R community have gradually transformed it into a tool capable of tackling a surprising number of advanced tasks. The goal of this chapter is to demonstrate R's ability to remain relevant in the era of deep learning and big data. Even though R is unlikely to be found at the heart of the biggest big data projects, and despite facing increasing competition from Python and cloud-based tools, R's strengths keep it on the desktops of many practicing data scientists.

Practical applications of deep learning

Deep learning has received a great deal of attention lately due to its successes in tackling machine learning tasks that have been notoriously difficult to solve with conventional methods. Using sophisticated neural networks to teach computers to think more like a human has allowed machines to catch up with or even surpass human performance on many tasks that humans once held a seemingly insurmountable lead. Perhaps more importantly, even if humans still perform better at certain tasks, the upsides of machine learning—workers that never tire, never get bored, and require no salary—turn even imperfect automatons into useful tools for many tasks.

Unfortunately, for those of us working outside of large technology companies and research organizations, it is not always easy to take advantage of deep learning methods. Training a deep learning model generally requires not only state-of-the-art computing hardware but also large volumes of training data. In fact, as mentioned in *Chapter 12, Advanced Data Preparation*, most practical machine learning applications in the business setting are in the so-called small or medium data regimes, and here deep learning methods might perform no better and possibly even worse than conventional machine learning approaches like regression and decision trees. Thus, many organizations that are investing heavily in deep learning are doing so as a result of hype rather than true need.

Even though some of the buzz around deep learning is surely based on its novelty as well as excitement from business leaders with visions of artificial intelligence replacing costly human workers, there are in fact practical applications of the technique that can be used in combination with, rather than as a replacement for, conventional machine learning methods. The purpose of this chapter is therefore not to provide a start-to-finish tutorial for building deep neural networks, but rather to show how deep learning's successes can be incorporated into conventional machine learning projects including those outside the big data regime.



Packt Publishing offers numerous resources on deep learning, such as *Hands-On Deep Learning with R: A practical guide to designing, building, and improving neural network models using R* (2020) by M. Pawlus and R. Devine, *Advanced Deep Learning with R* (2019) by B. Rai, and *Deep Learning with R Cookbook* (2020) by S. Gupta, R. A. Ansari, and D. Sarkar.

Beginning with deep learning

In recent years, with a new cohort of data science practitioners having been trained in the age of deep learning, a form of “generation gap” has developed in the machine learning community. Prior to the development of deep learning, the field was staffed primarily by those who were trained in statistics or computer science. Especially in the earliest years, machine learning practitioners carried with them the metaphorical baggage of their prior domain, and the software and algorithms they used fell into camps along party lines. Statisticians typically preferred regression-based techniques and software like R, whereas computer scientists favored iterative and heuristic-based algorithms like decision trees written in languages like Python and Java. Deep learning has blurred the line between these camps, and the next generation of data scientists may seem somewhat foreign to the prior generations as if they speak a different language.

The rift seems to have come out of nowhere, despite being able to see its origins clearly with the benefit of hindsight. As the famous author Ernest Hemingway once wrote, it happened “gradually, then suddenly.” Just as machine learning itself was only possible as the result of the simultaneous evolution of computing power, statistical methods, and available data, it makes sense that the next big evolutionary leap would arise out of a series of smaller evolutions in each of the same three components. Recalling the similar image presented in *Chapter 1, Introducing Machine Learning*, a revised cycle of advancement diagram depicting today’s state-of-the-art machine learning cycle illustrates the environment in which deep learning was developed:

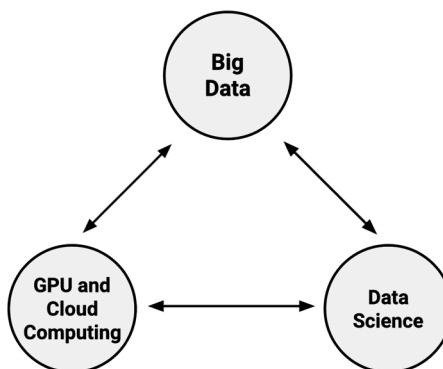
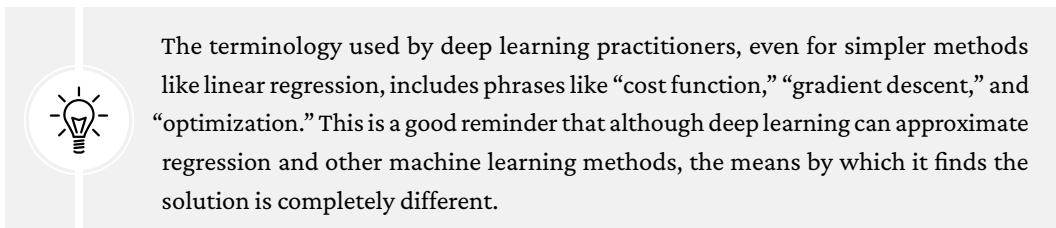


Figure 15.1: A combination of factors in the cycle of advancement led to the development of deep learning

It is no surprise that deep learning arose out of the big data era, while also being provided the requisite computing hardware—**graphics processing units (GPUs)** and cloud-based parallel processing tools, which will be covered later in this chapter—necessary to process datasets that are both very long and very wide. What is less obvious, and therefore easy to take for granted, is the academic and research environment that was also necessary for this evolution. Without a strong data science community comprising researchers whose expertise spans both statistics and computer science, in addition to applied data scientists motivated to solve practical business problems on large and complex real-world datasets, it is unlikely that deep learning would have arisen as it has. Stated differently, the fact that data science now exists as a focused academic discipline has undoubtedly accelerated the cycle of advancement. To borrow an analogy from science fiction, the system is much like a robot that becomes self-aware and learns much more quickly now that it has learned how to learn!

The rapid development of deep learning has contributed to the previously mentioned generation gap, but it is not the only factor. As you will soon learn, deep learning not only offers impressive performance on big data tasks, but it can also perform much like conventional learning methods on smaller tasks. This has led some to focus almost exclusively on the technique, much like earlier generations of machine learning practitioners focused exclusively on regression or decision trees. The fact that deep learning also utilizes specialized software tools and mathematical terminology to perform these tasks means that its practitioners are, in some cases, literally speaking another language to describe the same series of steps. As has been said many times before, “there is no free lunch” in the field of machine learning, so as you continue your machine learning journey, it is best to see it as one of many useful tools—and not the *only* tool for the job.



The terminology used by deep learning practitioners, even for simpler methods like linear regression, includes phrases like “cost function,” “gradient descent,” and “optimization.” This is a good reminder that although deep learning can approximate regression and other machine learning methods, the means by which it finds the solution is completely different.

Choosing appropriate tasks for deep learning

As mentioned in *Chapter 7, Black-Box Methods – Neural Networks and Support Vector Machines*, neural networks with at least one hidden layer can act as universal function approximators. Elaborating on this principle, one might say that given enough training data, a cleverly designed neural network can learn to mimic the output of any other function.

This implies that the conventional learning methods covered throughout this book can likewise be approximated by well-designed neural networks. In fact, it is quite trivial to design a neural network that almost exactly matches linear or logistic regression, and with a bit more work it is possible to approximate techniques like k-nearest neighbors and naive Bayes. Given enough data, a neural network can get closer and closer to the performance of even the best tree-based algorithms like random forests or gradient boosting machines.

Why not apply deep learning to every problem, then? Indeed, a neural network's ability to mimic all other learning approaches appears to be a violation of the “no free lunch” theorem, which, put simply, suggests that there is no machine learning algorithm that can perform best across all potential modeling tasks. There are a couple of key reasons why the theorem remains safe despite deep learning’s magic. First, the ability of a neural network to approximate a function is related to how much training data it has. In the small data regime, conventional techniques can perform better, especially when combined with careful feature engineering. Second, to reduce the amount of data the neural network needs for training, the network must have a topology that facilitates its ability to learn the underlying function. Of course, if the person building the model knows what topology to use, then it may be preferable to use the simpler conventional model in the first place.

People using deep learning for conventional learning tasks are likely to prefer the black box approach, which works in the big data regime. Big data, however, is not merely the presence of many rows of data, but also many features. Most conventional learning tasks, including those with many millions of rows of data, are in the medium data regime, where conventional learning algorithms still perform well. In this case, whether the neural network performs better will ultimately come down to the balance of overfitting and underfitting—a balance that is sometimes challenging to find with a neural network, as the method tends to somewhat easily overfit the training data.

Perhaps for this reason, deep learning is not well suited for racking up wins in machine learning competitions. If you ask a Kaggle Grandmaster, they are likely to tell you that neural networks don’t work on standard, real-life problems, and on traditional supervised learning tasks, gradient boosting wins. One can also see proof of this by browsing the leaderboards and noting the absence of deep learning. Perhaps a clever team will use neural networks for feature engineering and blend the deep learning model with other models in an ensemble to boost their performance, but even this is rare. Deep learning’s strengths are elsewhere. As a rule of thumb, tree-based ensemble methods win on structured, tabular data, while neural networks win on unstructured data, like image, sound, and text.

Reading recent news about research breakthroughs and technology startup companies, one is likely to encounter deep learning applications that utilize the method's unique ability to solve unconventional tasks. In general, these unconventional learning tasks fall into one of three categories: computer vision, natural language processing, or tasks involving unusual data formats like repeated measurements over time or having an exceptionally large number of interrelated predictors. A selection of specific successes for each category is listed in the following table:

Challenging machine learning tasks	Deep learning successes
Computer vision tasks involving classifying images found in still pictures or video data	<ul style="list-style-type: none"> • Identifying faces in security camera footage • Categorizing plants or animals for ecological monitoring • Diagnosing medical images such as X-rays, MRI, or CT scans • Measuring the activity of athletes on the sporting field • Autonomous driving
Natural language applications requiring an understanding of the meaning of words in context	<ul style="list-style-type: none"> • Processing social media posts to filter out fake news or hate speech • Monitoring Twitter or customer support emails for consumer sentiment or other marketing insights • Examining electronic health records for patients at risk of adverse outcomes or for eligibility for new treatments
Predictive analysis involving many repeated measurements or very large numbers of predictors	<ul style="list-style-type: none"> • Predicting the price of goods or equities in open markets • Estimating energy, resource, or healthcare utilization • Forecasting survival or other medical outcomes using insurance billing codes

Even though some people certainly do use deep learning for conventional learning problems, this chapter focuses only on tasks that cannot be solved via conventional modeling techniques. Deep learning is highly adept at tapping into the unstructured data types that characterize the big data era, such as images and text, which are extremely difficult to model with traditional approaches.

Unlocking these capabilities requires the use of specialized software and specialized data structures, which you will learn about in the next section.

The TensorFlow and Keras deep learning frameworks

Perhaps no software tool has contributed as much to the rapid growth in deep learning as **TensorFlow** (<https://www.tensorflow.org>), an open-source mathematical library developed at Google for advanced machine learning. TensorFlow provides a computing interface using directed graphs that “flow” data structures through a sequence of mathematical operations.



Packt Publishing offers many books on TensorFlow. To search the current offerings, visit <https://subscription.packtpub.com/search?query=tensorflow>.

The fundamental TensorFlow data structure is unsurprisingly known as a **tensor**, which is an array of zero or more dimensions. Building upon the simplest 0-D and 1-D tensors, which represent a single value and a sequence of values, respectively, adding additional dimensions allows more complex data structures to be represented. Note that because we typically analyze sets of structures, the first dimension is generally reserved to allow multiple objects to be stacked; the first dimension then refers to the batch or sample number for each structure. For example:

- A set of 1-D tensors, collecting feature values for a set of people, is a 2-D tensor analogous to a data frame in R: [person_id, feature_values]
- For measurements repeated over time, 2-D tensors can be stacked as a 3-D tensor: [person_id, time_sequence, feature_values]
- 2-D images are represented by a 4-D tensor, with the fourth dimension storing the color values for each pixel in the 2-D grid: [image_id, row, column, color_values]
- Video or animated images are represented in 5-D with an additional time dimension: [image_id, time_sequence, row, column, color_values]

Most tensors are rectangular matrices completely filled with numeric data, but more complex structures like ragged tensors and sparse tensors are available for use with text data.



For an in-depth look at TensorFlow’s tensor objects, the documentation is available at <https://www.tensorflow.org/guide/tensor>.

TensorFlow's graph, which can be more specifically termed a **dataflow graph**, uses nodes connected by directional arrows known as edges to represent dependencies between data structures, mathematical operations on these data structures, and the output. The nodes represent mathematical operations and the edges represent tensors flowing between operations. The graph aids in the parallelization of the work, since it is clear what steps must be completed in sequence versus those that may be completed simultaneously.

A dataflow graph can be visualized if desired, which produces something like the idealized graph depicted in *Figure 15.2*. Although this is a highly simplified representation and real-world TensorFlow graphs tend to be much more complex, the diagram here shows that after completing the first operation, the second and fourth operations can begin in parallel:

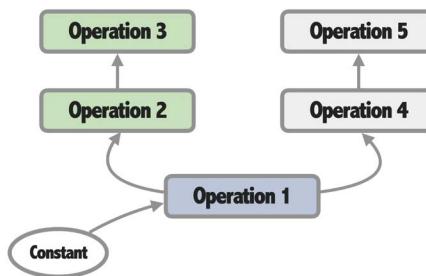


Figure 15.2: A simplified representation of a TensorFlow graph

As tensors flow through the graph, they are transformed by the series of operations represented by the nodes. The steps are defined by the person building the diagram, with each step moving closer to the end goal of accomplishing some sort of mathematical task. Some steps in the flow graph may apply simple mathematical transformations like normalization, smoothing, or bucketing to the data; others may attempt to train a model by iterating repeatedly while monitoring a **loss function**, which measures the fit of the model's predictions to the true values.

R interfaces to TensorFlow have been developed by the team at RStudio. The `tensorflow` package provides access to the core API, while the `tfestimators` package provides access to higher-level machine learning functionality. Note that TensorFlow's directed graph approach can be used to implement many different machine learning models, including some of those discussed in this book. However, to do so requires a thorough understanding of the matrix mathematics that defines each model, and thus is outside the scope of this text. For more information about these packages and RStudio's ability to interface with TensorFlow, visit <https://tensorflow.rstudio.com>.

Because TensorFlow relies so heavily on complex mathematical operations that must be programmed carefully by hand, the **Keras** library (<https://keras.io>) was developed to provide a higher-level interface to TensorFlow and allow deep neural networks to be built more easily. Keras was developed in Python and is typically paired with TensorFlow as the back-end computing engine. Using Keras, it is possible to do deep learning in just a few lines of code—even for challenging applications such as image classification, as you will discover in the example later in this chapter.



Packt Publishing offers numerous books and videos to learn Keras. To search current offerings, visit <https://subscription.packtpub.com/search?query=keras>.

The `keras` package, developed by RStudio founder J. J. Allaire, allows R to interface with Keras. Although there is very little code required to use the package, developing useful deep learning models from scratch requires extensive knowledge of neural networks as well as familiarity with TensorFlow and the Keras API. For these reasons, a tutorial is outside the scope of this book. Instead, refer to the RStudio TensorFlow documentation or the book *Deep Learning with R* (2018), which was co-authored by Francois Chollet and J. J. Allaire—the creators of Keras and the `keras` package, respectively. Given their credentials, there is no better place to begin learning about this tool.



Although the combination of Keras and TensorFlow is arguably the most popular deep learning toolkit, it is not the only tool for the task. The **PyTorch** framework developed at Facebook has rapidly gained popularity, especially in the academic research community, as an easy-to-use alternative. For more information, see <https://pytorch.org>.

TensorFlow's innovative idea to represent complex mathematical functions using a simple graph abstraction, combined with the Keras framework to make it easier to specify the network topology, has enabled the construction of deeper and more complex neural networks, such as those described in the next section. Keras even makes it easy to adapt pre-built neural networks to new tasks with no more than a few lines of code.

Understanding convolutional neural networks

Neural networks have been studied for over 60 years, and even though deep learning has only recently become widespread, the concept of a deep neural network has been known for decades. As first introduced in *Chapter 7, Black-Box Methods – Neural Networks and Support Vector Machines*, a **deep neural network (DNN)** is simply a neural network with more than one hidden layer.

This understates what deep learning is in practice, as typical DNNs are substantially more complex than the types of neural networks we've built previously. It's not enough to add a few extra nodes in a new hidden layer and call it "deep learning." Instead, typical DNNs use extremely complex but purposefully designed topologies to facilitate learning on big data, and in doing so are capable of human-like performance on challenging learning tasks.

A turning point for deep learning came in 2012, when a team called SuperVision used deep learning to win the ImageNet Large Scale Visual Recognition Challenge. This annual competition tests the ability to classify a subset of 10 million hand-labeled images across 10,000 categories of objects. In the early years of the competition, humans vastly outperformed computers, but the performance of the SuperVision model closed the gap significantly. Today, computers are nearly as good as humans at visual classification, and, in some specific cases, are even better. Humans tend to be better at identifying small, thin, or distorted items, while computers have a greater ability to distinguish among specific types of items such as dog breeds. Before long, it is likely that computers will outperform humans on both types of visual tasks.

An innovative network topology designed specifically for image recognition is responsible for the surge in performance. A **convolutional neural network (CNN)** is a deep feed-forward network used for visual tasks that independently learns the important distinguishing image features rather than requiring such feature engineering beforehand. For example, to classify road signs like "stop" or "yield," a traditional learning algorithm would require pre-engineered features like the shape and color of the sign. In contrast, a CNN requires only the raw data for each of the image pixels, and the network will learn how to distinguish important features like shape and color on its own.

Learning features like these is made possible due to the huge increase in dimensionality when using raw image data. A traditional learning algorithm would use one row per image, in a form like *(stop sign, red, octagon)*, while a CNN uses data in the form *(stop sign, x, y, color)*, where *x* and *y* are pixel coordinates and *color* is the color data for the given pixel. This may seem like an increase of only one dimension but note that even a very small image is made of many *(x, y)* combinations and color is often specified as a combination of RGB (*red, green, blue*) values. This means that a more accurate representation of a single row of training data would be:

$$(stop\ sign, x_1y_1r, x_1x_1g, x_1y_1b, x_2y_1r, x_2y_1g, x_2y_1b, \dots, x_ny_nr, x_nx_ng, x_ny_nb)$$

Each of the predictors refers to the level of red, green, or blue at the specified combination of *(x, y)*, and *r, g*, or *b* values. Thus, the dimensionality increases greatly, and the dataset becomes much wider as the image becomes larger.

A small 100×100 pixel image would have $100 \times 100 \times 3 = 30,000$ predictors. Even this is small compared to the SuperVision team, which used over 60 million parameters when it won the visual recognition challenge in 2012!

Chapter 12, Advanced Data Preparation, noted that if a model is overparameterized, it reaches an interpolation threshold at which there are enough parameters to memorize and perfectly classify all the training samples. The ImageNet Challenge dataset, which contained 10 million images, is much smaller than the 60 million parameters the winning team used. Intuitively, this makes sense; assuming there are no completely identical pictures in the database, at least one of the pixels will vary for every image. Thus, an algorithm could simply memorize every image to achieve the perfect classification of the training data. The problem is that the model will be evaluated on a dataset of unseen data, and thus the severe overfitting to the training data will lead to a massive generalization error.

The topology of a CNN prevents this from happening. We won't be diving too deeply into the black box of the CNN, but we will understand it as a series of layers in the following categories:

- **Convolutional layers** are placed early in the network and usually comprise the most computationally intensive step in the network because they are the only layers to process the raw image data directly; we can understand convolution as passing the raw data through a filter creating a set of tiles that represent small, overlapping portions of the full area
- **Pooling layers**, also known as **downsampling** or **subsampling** layers, gather the output signals from a cluster of neurons in one layer, and summarize them into a single neuron for the next layer, usually by taking the maximum or average value among those being summarized
- **Fully connected layers** are much like the layers in a traditional multilayer perceptron, and are used near the end of the CNN to build the model that makes predictions

The convolutional and pooling layers in the network serve the interrelated purposes of identifying important features of the images to be learned, as well as reducing the dimensionality of the dataset before hitting the fully connected layers that make predictions. In other words, the early stages of the network perform feature engineering, while the later stages use the constructed features to make predictions.



To better understand the layers in a CNN, see *An Interactive Node-Link Visualization of Convolutional Neural Networks* by Adam W. Harley at https://adamharley.com/nn_vis/. The interactive tool has you draw a number from zero to nine, which is then classified using a neural network. The 2D and 3D convolutional network visualizations clearly show how the digit you drew passes through the convolutional, downsampling, and fully connected layers before reaching the output layer where the prediction is made. Neural networks for general image classification work much the same way, but using a substantially larger and more complex network.

Transfer learning and fine tuning

Building a CNN from scratch requires a tremendous amount of data, expertise, and computing power. Thankfully, many large organizations that have access to data and computing resources have built image, text, and audio classification models, and have shared them with the data science community. Through a process of **transfer learning**, a deep learning model can be adapted from one context to another. Not only is it possible to apply the saved model to similar types of problems as it was trained on, but it may also be useful for problems outside the original domain. For instance, a neural network that was trained to recognize an endangered species of elephants in satellite photos may also be useful for identifying the position of tanks in infrared drone images taken above a war zone.

If the knowledge doesn't transfer directly to the new task, it is possible to hone a pre-trained neural network using additional training in a process known as **fine tuning**. Taking a model that was trained generally, such as a general image classification model that can identify 10,000 classes of objects, and fine tuning it to be good at identifying a single type of object not only reduces the amount of training data and computing power needed but also may offer improved generalization over a model trained on a single class of images.

Keras can be used for both transfer learning and fine tuning by downloading neural networks with pre-trained weights. A list of available pre-trained models is available at <https://keras.io/api/applications/> and an example of fine tuning an image processing model to better predict cats and dogs can be found at https://tensorflow.rstudio.com/guides/keras/transfer_learning. In the next section, we will apply a pre-trained image model to real-world images.

Example – classifying images using a pre-trained CNN in R

R may not be the right tool for the heaviest deep learning jobs, but with the right set of packages, we can apply pre-trained CNNs to perform tasks, such as image recognition, that conventional machine learning algorithms have trouble solving. The predictions generated by the R code can then be used directly for image recognition tasks like filtering obscene profile pictures, determining whether an image depicts a cat or a dog, or even identifying stop signs inside a simple autonomous vehicle. Perhaps more commonly, the predictions could be used as predictors in an ensemble that includes conventional machine learning models using tabular-structured data in addition to the deep learning neural network that consumes the unstructured image data. You may recall that *Chapter 14, Building Better Learners*, described a potential stacked ensemble that combined image, text, and traditional machine learning models in this way to predict elements of a Twitter user’s future behavior. The following pictures illustrate hypothetical Twitter profile pictures, which we will classify using a deep neural network shortly:



Figure 15.3: A pre-trained neural network can be used in R to identify the subject of images like these

First, before using a pre-trained model, it is important to consider the dataset that was used to train the neural network. Most publicly available image networks were trained on huge image databases comprising a variety of everyday objects and animals, such as cars, dogs, houses, various tools, and so on. This is appropriate if the desired task is to distinguish among everyday objects, but more specific tasks may require more specific training datasets. For instance, a facial recognition tool or an algorithm identifying stop signs would be more effectively trained on datasets of faces and road signs, respectively. With transfer learning, it is possible to fine-tune a deep neural network trained on a variety of images to be better at a more specific task—it could become very good at identifying pictures of cats, for example—but it is hard to imagine a network trained on faces or road signs ever becoming very good at identifying cats, even with additional tuning!

For this exercise, we will classify our small set of images using a CNN called **ResNet-50**, which is a 50-layer deep network that has been trained on a large and comprehensive variety of labeled images. This model, which was introduced by a group of researchers in 2015 as a state-of-the-art, competition-winning computer vision algorithm, has since been surpassed by more sophisticated approaches, but continues to be extremely popular and effective due to its ease of use and integration with tools like R and Keras.



For more information about ResNet-50, see *Deep Residual Learning for Image Recognition*, He, K., Zhang, X., Ren, S., and Sun, J., 2015, <https://arxiv.org/abs/1512.03385v1>.

The **ImageNet database** (<https://www.image-net.org>) that was used to train the ResNet-50 model is the same database used for the ImageNet Visual Recognition Challenge and has contributed greatly to computer vision since its introduction in 2010. Composed of over 14 million hand-labeled images and consuming many gigabytes of storage (or even terabytes in the case of the full, academic version), it is fortunate that there is no need to download this resource and train the model from scratch. Instead, we simply download the neural network weights for the ResNet-50 model that researchers trained on the full database, saving us a tremendous amount of computational expense.

To begin the process, we'll need to add the `tensorflow` and `keras` packages to R as well as the various dependencies. Most of these steps must only be performed once. The `devtools` package adds tools to develop R packages and use packages that are in active development, so we'll begin by installing and loading this as usual:

```
> install.packages("devtools")
> library(devtools)
```

Next, we'll use the `devtools` package to obtain the latest version of the `tensorflow` package from GitHub. Typically, we install packages from CRAN, but for packages in active development, it can be better to install directly from the latest source code. The command to install the `tensorflow` package from its GitHub path is:

```
> devtools:::install_github("rstudio/tensorflow")
```

This points R to the RStudio GitHub account, which stores the source code for the package. To read the documentation and see the code for yourself on the web, visit <https://github.com/rstudio/tensorflow> in a web browser.

After installing the tensorflow package, there are several dependencies that are required to begin using TensorFlow in R. In particular, the tensorflow package is merely an interface between R and TensorFlow, so we must first install TensorFlow itself. Perhaps ironically, Python and several of its packages are required for this. Thus, the R reticulate package (<https://rstudio.github.io/reticulate/>) is used to manage the interface between R and Python. As confusing as this seems, the complete installation process is driven by a single command from the tensorflow package as follows:

```
> library(tensorflow)  
> install_tensorflow()
```

While the command is running, you should see R working to install a large collection of Python tools and packages. If all goes well, you can proceed to install the Keras package from GitHub:

```
> devtools::install_github("rstudio/keras")
```

In the case of a problem, keep in mind that although the code for this example was tested on multiple platforms and R versions, it is quite possible for something to go wrong among the many dependencies required to have R interface with Python and TensorFlow. Don't be afraid to search the web for a particular error message or check the Packt Publishing GitHub repository for the updated R code for this chapter.

With the necessary packages installed, Keras can help load the ResNet-50 model with the weights trained on the ImageNet database:

```
> library(keras)  
> model <- application_resnet50(weights = 'imagenet')
```

Our 50-layer deep image classification model trained on millions of everyday images is now ready to start making predictions; however, the ease at which we loaded the model conceals the work to come.

The greater challenge with using a pre-trained model is transforming our unstructured image data, which we hope to classify, into the same structured format that it saw during training. ResNet-50 used square images of 224-by-224 pixels with each pixel reflecting a color composed of three channels, red, green, and blue, each having 255 levels of brightness. All images we hope to classify must be transformed from their original formats, such as PNG, GIF, or JPEG, into a 3-D tensor using this representation. We'll see this in practice using the previously depicted `cat.jpg`, `ice_cream.jpg`, and `pizza.jpg` files found in the R code folder for this chapter, but the process will work similarly for any image.

The `image_load()` function in the `keras` package will get the process started. Simply provide the file name and desired target dimensions as follows:

```
> img <- image_load("ice_cream.jpg", target_size = c(224,224))
```

This creates an image object, but we need one more command to convert it into a 3-D tensor:

```
> x <- image_to_array(img)
```

To prove it to ourselves, we can examine the dimensions and structure of the object as follows. As expected, the object is a numeric matrix with $224 \times 224 \times 3$ as the dimensions:

```
> dim(x)
```

```
[1] 224 224 3
```

The first few values in the matrix are all 255, which isn't very meaningful:

```
> str(x)
```

```
num [1:224, 1:224, 1:3] 255 255 255 255 255 255 255 255 255 255 ...
```

Let's do some investigation to better understand these data structures. Because of R's row, column matrix format, the matrix coordinates are (y, x) , with $(1, 1)$ representing the top-left pixel in the image and $(1, 224)$ the top-right pixel. To illustrate this, let's obtain each of the three color channels for a couple of pixels in the ice cream image:

```
> x[1, 224, 1:3]
```

```
[1] 253 253 255
```

```
> x[40, 145, 1:3]
```

```
[1] 149 23 34
```

The pixel at $(1, 224)$ has (r, g, b) colors of $(253, 253, 255)$, which corresponds to nearly the brightest possible white, while the pixel at $(40, 145)$ has color values $(149, 23, 34)$ translating to a dark red—a piece of strawberry in the ice cream. These coordinates are illustrated in the following figure:

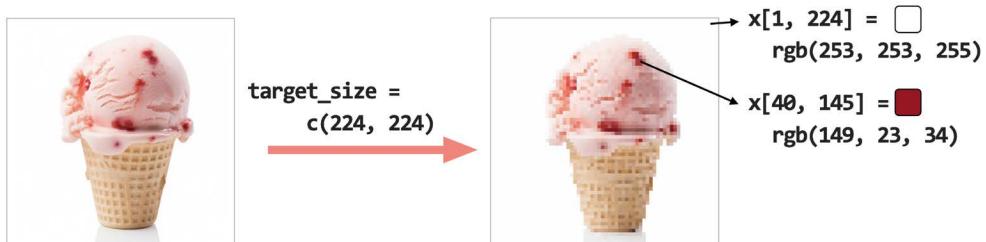


Figure 15.4: The picture of ice cream has been reduced from a matrix of 1,000x1,000 pixels to 224x224; each pixel in the image has three color channels

One additional complication is that the ResNet-50 model expects a four-dimensional tensor, with the fourth dimension representing the batch. With only one image to classify, we have no need for this parameter, so we'll simply assign it a constant value of 1 to create a matrix of $1 \times 224 \times 224 \times 3$. The command `c(1, dim(x))` defines the new matrix in this format, and then the `array_reshape()` function fills this matrix with the contents of `x` using the Python-style row-by-row ordering used by TensorFlow rather than the R-style column-by-column filling. The full command is as follows:

```
> x <- array_reshape(x, c(1, dim(x)))
```

To confirm that `x` has the correct dimensions, we can use the `dim()` command:

```
> dim(x)
```

```
[1] 1 224 224 3
```

Finally, we run the `imagenet_preprocess_input()` function to normalize the color values to match the ImageNet database:

```
> x <- imagenet_preprocess_input(x)
```

The primary function of this transformation is to zero-center each color with respect to the database, essentially treating each color value as being greater than or less than the average of ImageNet images on that color. For example, the red pixel in the ice cream at (40, 145) had color values of 149, 23, and 34 before; now, it has very different values:

```
> x[1, 40, 145, 1:3]
```

```
[1] -69.939 -93.779 25.320
```

Negative values indicate a color level less than the ImageNet average for that color, and positive values indicate higher. The preprocessing step also inverts the red-green-blue format to blue-green-red, so only the red channel is above the average ImageNet level, which is not terribly surprising, as a strawberry is quite red!

Now, let's see what the ResNet-50 network thinks is depicted in the image. We'll first use the `predict()` function on the model object and the image matrix, and then use the `keras` function `imagenet_decode_predictions()` to convert the network's predicted probabilities to text-based labels that categorize each of the ImageNet images. Because there are 1,000 categories of images in the ImageNet database, the `preds` object contains 1,000 predicted probabilities—one for each possibility. The decoding function allows us to limit the output to the top N most probable possibilities—ten, in this case:

```
> p_resnet50 <- predict(m_resnet50, x)
> c_resnet50 <- imagenet_decode_predictions(p_resnet50, top = 10)
```

The `c_resnet50` object is a list which contains the top ten predictions for our lone image. To see the predictions, we simply type the name of the list to discover that the network correctly identified the image as ice cream with about 99.6 percent probability:

```
> c_resnet50
[[1]]
  class_name class_description      score
1  n07614500          ice_cream 0.99612110853
2  n07836838    chocolate_sauce 0.00257453066
3  n07613480            trifle 0.00017260048
4  n07932039        eggnog 0.00011857488
5  n07930864            cup 0.00011558698
6  n07745940       strawberry 0.00010969469
7  n15075141   toilet_tissue 0.00006556125
8  n03314780    face_powder 0.00005355201
9  n03482405        hamper 0.00004582879
10 n04423845        thimble 0.00004054611
```

Although none of the other potential classifications had a predicted probability much greater than zero, some of the other top predictions make a bit of sense; it is not difficult to see why they were considered as possibilities. Eggnog is in the correct category of foods, while an ice cream cone might look a bit like a cup, or a thimble.

The model even listed strawberry as the sixth most likely option, which is the correct flavor of ice cream.

As an exercise, we'll do the same process with the other two images. The following sequence of steps uses the `lapply()` function to apply the image processing steps to the pair of images, each time creating a new list to supply to the subsequent function. The last step supplies the list containing two prepared image arrays to the `lapply()` function, which applies the `predict()` command to each image:

```
> img_list <- list("cat.jpg", "pizza.jpg")
> img_data <- lapply(img_list, image_load, target_size = c(224,224))
> img_arrrs <- lapply(img_data, image_to_array)
> img_resh <- lapply(img_arrrs, array_reshape, c(1, 224, 224, 3))
> img_prep <- lapply(img_resh, imagenet_preprocess_input)
> img_prob <- lapply(img_prep, predict, object = m_resnet50)
```

Finally, the `sapply()` function is used to apply the decoding function to each of the two sets of predictions, while simplifying the result. The `lapply()` function would also work here, but because `imagenet_decode_predictions()` returns a list, the result is a sub-list of length one within a list; `sapply()` recognizes that this is redundant and unnecessary, and will eliminate the additional level of hierarchy:

```
> img_classes <- sapply(img_prob, imagenet_decode_predictions,
                           top = 3)
```

Typing the name of the resulting object shows the top three predictions for each of the two images:

```
> img_classes
```

	[[1]]	[[2]]
	class_name class_description score	class_name class_description score
1	n02123045 tabby 0.63457680	n07873807 pizza 0.9890466332
2	n02124075 Egyptian_cat 0.08966244	n07684084 French_loaf 0.0083064679
3	n02123159 tiger_cat 0.06287414	n087747607 orange 0.0002433858

The ResNet-50 algorithm didn't merely classify the images correctly; it also correctly identified the cat picture as a tabby. This demonstrates the ability of neural networks to surpass human specificity for certain tasks; many or most people might have simply labeled the image as a cat, whereas the computer can determine the specific type of cat. On the other hand, humans remain better at identifying objects in less-than-optimal conditions. For example, a cat obscured by darkness or camouflaged in the weeds would present a greater challenge to a computer than to a human in most cases. Even so, the computer's ability to work tirelessly gives it a huge advantage for automating artificial intelligence tasks. As stated previously, applied to a large dataset such as Twitter profile images, the predictions from this type of computer vision model could be used in an ensemble model predicting countless different user behaviors.

Unsupervised learning and big data

The previous section illustrated how a deep neural network could be used to classify a limitless supply of input images as an instance of everyday creatures or objects. From another perspective, one might also understand this as a machine learning task that takes the highly dimensional input of image pixel data and reduces it to a lower-dimensional set of image labels. It is important to note, however, that the deep learning neural network is a supervised learning technique, which means that the machine can only learn what the humans tell it to learn—in other words, it can only learn from something that has been previously labeled.

The purpose of this section is to present useful applications of unsupervised learning techniques in the context of big data. These applications are in many ways similar to the techniques covered in *Chapter 9, Finding Groups of Data – Clustering with k-means*. However, where previous unsupervised learning techniques leaned heavily on humans to interpret the results, in the context of big data, the machine can go a step further than before and provide a deeper, richer understanding of the data and the implications of the connections the algorithm discovers.

To put this in practical terms, imagine a deep neural network that can learn to identify a cat without ever having been told what a cat is. Of course, without being given the label beforehand, the computer may not explicitly label it a “cat” *per se*, but it may understand that a cat has certain consistent relationships to other things that appear in pictures along with the cat: people, litter boxes, mice, balls of yarn—but rarely or never dogs! Such associations help form a conceptualization of cat as something that relates closely to people, litter boxes, and yarn, but is perhaps in opposition to another thing with four legs and a tail. Given enough pictures, it is possible the neural network could eventually associate its impression of cats with the English-language word “cat” by identifying cats near bags of cat food or in the internet’s countless cat-based memes!

Developing such a sophisticated model of cats would take more data and computing power than most machine learning practitioners have access to, but it is certainly possible to develop simpler models or to borrow from big data companies that do have access to such resources. These techniques provide yet another way to incorporate unstructured data sources into more conventional learning tasks, as the machine can reduce the complexity of big data into something much more digestible.

Representing highly dimensional concepts as embeddings

The things we encounter in everyday life can be described by a limitless number of attributes. Moreover, not only are there countless data points that can be used to describe each object, but the nature of human subjectivity makes it unlikely that any two people would describe an object in the same way. For example, if you ask a few people to describe typical horror films, one might say they imagine slasher films with blood and gore, another might think of zombie or vampire movies, and another one might think of spooky ghost stories and haunted houses. These descriptions could be represented using the following statements:

- *horror = killer + blood + gore*
- *horror = creepy + zombies + vampires*
- *horror = spooky + ghosts + haunted*

If we were to program these definitions into a computer, it could substitute any of the representations of horror for one another and thus use a wider general concept of “horror” rather than the more specific features like “gore,” “creepy,” or “spooky” to make predictions. For instance, a learning algorithm could discover that a social media user that writes any of these horror-related terms is more likely to click on an advertisement for the new *Scream* movie.

Unfortunately, if a user posts “I just love a good scary movie!” or “The Halloween season is my favorite time of year!” then the algorithm will be unable to relate the text to the prior conceptualizations of horror, and thus will be unable to realize that a horror movie advertisement should be displayed. This is likewise true for any of the hundreds of horror-related keywords that the computer had not previously seen verbatim, including many that will seem obvious to a human observer, like witches, demons, graveyards, spiders, skeletons, and so on. What is needed is a way to generalize the concept of horror to the almost limitless number of ways that it can be described.

An **embedding** is a mathematical concept referring to the ability to represent a higher-dimensional vector using fewer dimensions; in machine learning, the embedding is purposefully constructed such that dimensions that are correlated in the high-dimensional space are positioned more closely in the lower-dimensional space.

If the embedding is constructed well, the low-dimensional space will retain the semantics, or meaning, of the higher dimensions while being a more compact representation that can be used for classification tasks. The core challenge of creating an embedding is the unsupervised learning task of modeling the semantic meaning embedded in highly dimensional unstructured or semi-structured datasets.

Humans are quite adept at constructing low-dimensional representations of concepts, as we do this intuitively whenever we assign labels to objects or phenomena that are similar in broad strokes but may vary in the details. We do this when we label movies as comedy, science fiction, or horror; when we talk about categories of music like hip-hop, pop, or rock and roll; or when we create taxonomies of foods, animals, or illnesses. In *Chapter 9, Finding Groups of Data – Clustering with k-means*, we saw how the machine learning process of clustering can mimic this human process of labeling by grouping diverse but similar items through a process of “unsupervised classification.” However, even though this approach reduces the dimensionality of a dataset, it requires a structured dataset with the same specific features for each example before it can associate like-with-like. For something unstructured like a textual description of a movie, the features are too numerous and sparse to allow clustering.

Instead, what if we wanted to mimic the human ability to learn via association? Specifically, a human can watch a series of movies and associate like-with-like without having specific measurable features for each movie; we can classify one set of movies as horror films without needing to see the identical clichéd storyline or to count the number of screams each film elicited. The trick is that a human doesn’t need a concrete definition of “horror” because we intuit it as a concept relative to the others in a set. Just like a cat suddenly jumping into frame can be used as slapstick humor in a comedy movie or paired with suspenseful music to provoke a jump scare, the semantic meaning of horror is always determined by its context.

In much the same way, learning algorithms can construct embeddings via context. Each of the thousands of movies that Hollywood has produced can be understood relative to others, and without studying exactly what features the movies *Halloween* and *Night of the Living Dead* have in common, an algorithm can observe that they appear in similar contexts and are somewhat substitutable for one another across contexts. This notion of substitutability is the basis for most embedding algorithms, and has indeed been used to construct embeddings for use in movie recommendation algorithms and other domains. In the next section, we’ll see how a popular language embedding algorithm uses substitutability to discover the semantic meanings of words.

Understanding word embeddings

If there are roughly a million words in the English language, the feature space for a language-based model would be roughly a million dimensions wide even before phrases and word order are considered! This clearly would be far too large and sparse for most conventional learning algorithms to find a meaningful signal. A bag-of-words approach, as described in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*, might work with enough computing power, but it would also require a tremendous amount of training data to associate words with the desired outcome. What if, instead, we could use a language embedding that has been pre-trained on big data?

To illustrate the upside of this alternative approach, let's imagine the machine learning task of deciding whether to display an advertisement for a lunchtime café to users posting on a social media website. Consider the following posts made by hypothetical users:

- I ate bacon and eggs in the morning for the most important meal of the day!
- I am going to grab a quick sandwich this afternoon before hitting the gym.
- Can anyone provide restaurant recommendations for my date tonight?

For a naive Bayes approach, we would first need many of these types of sentences, but because the algorithm is a supervised learner, we would also need a target feature that indicates whether or not the user writing the sentence is interested in purchasing lunch from the café. We could then train the model to recognize which words are predictive of buying lunch.

In comparison, a human reading these sentences can easily form a reasonable guess as to which of the three users is most likely to be interested in buying lunch today. The human's guess is not based on being trained to predict lunch buying behavior specifically, but rather is based on an understanding of the embedded meaning in each of the sentences' words. In other words, because a human understands the meaning of the users' words, it becomes unnecessary to guess their behavior as we can instead just listen to what they are telling us they plan to do.



The most effective language models do more than merely look at the meaning of words; they also look at the meaning of words in relation to others. The use of grammar and phrasing can completely change the implications of a sentence. For example, the sentence "I skipped breakfast today, so I can stuff myself at lunch" is very different from "I stuffed myself at breakfast, so I need to skip lunch today" despite containing almost exactly the same words!

Ignoring for now how this might be constructed, suppose we have a very simple language embedding that captures the meaning of all English-language words in two dimensions: a “lunch” dimension that measures how related a term is to lunch, and a “food” dimension that indicates whether the term is related to food. In this model, the semantic meaning that was once delivered by unique and specific terms like “soup” and “salad” is instead represented by the position of these concepts in the 2-D space, as illustrated for a selection of words in *Figure 15.5*:

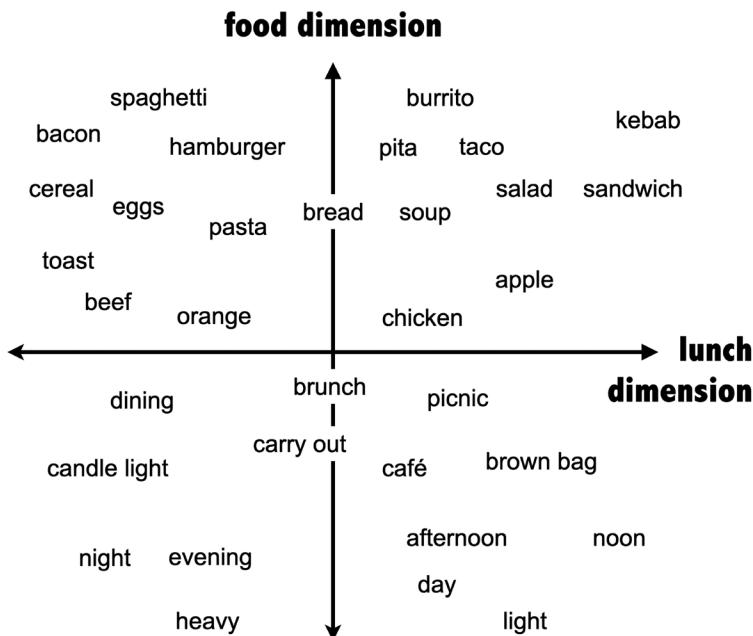


Figure 15.5: A very simple embedding reduces the highly dimensional meaning of various words into two dimensions that a machine can use to understand the subjective concepts of “food” and “lunch”

The embedding itself is a mapping of a word to coordinates in a lower-dimensional space. Thus, a lookup function can provide the values for a specific word. For instance, using the 2-D word embedding above, we can obtain coordinates for a selection of terms that may have appeared in social media posts:

- $f(\text{sandwich}) = (0.97, 0.54)$
- $f(\text{bacon}) = (-0.88, 0.75)$
- $f(\text{apple}) = (0.63, 0.25)$
- $f(\text{orange}) = (-0.38, 0.13)$

Terms with higher values of the first dimension are more specifically related to lunch (and lunch alone) while lower values indicate terms that are specifically not related to lunch. For example, the word “sandwich” has a high lunch value while “bacon” has a low lunch value, due to their close association with lunch and breakfast, respectively. In much the same way, terms with higher or lower values of the second dimension are more or less likely to be foods. The words “orange” and “apple” can both be foods, but the former can also represent a color while the latter can represent a computer, so they are near the middle of the food dimension. In contrast, the words “bacon” and “sandwich” are higher in this dimension but are lower than “burrito” or “spaghetti” due to their meanings outside of the culinary context; someone can “bring home the bacon” (that is, they can earn money) or an item can be “sandwiched” between other items.

An interesting and useful property of this type of embedding is that words can be related to one another via simple mathematics and nearest-neighbor-style distance calculations. In the 2-D plot, we can observe this property by examining the terms that are mirror images across the horizontal or vertical axis or those that are close neighbors. This leads to observations such as:

- An apple is a more lunch-related version of an orange
- Beef is like chicken, but not as associated with lunch
- Pitas and tacos are somewhat similar, as are kebabs and sandwiches
- Soup and salad are closely related and are the lunch versions of eggs and pasta
- Heavy and light are opposites with respect to lunch, as are afternoon and evening
- The term “brown bag” is lunch-ish like “apple” but less food-ish

Although this is a simple, contrived example, the word embeddings developed using big data have similar mathematical properties—albeit with a much higher number of dimensions. As you will soon see, these additional dimensions allow additional aspects of word meaning to be modeled, and enrich the embedding far beyond the “lunch” and “food” dimensions illustrated so far.

Example – using word2vec for understanding text in R

The previous sections introduced the idea of embedding as a means of encoding a highly dimensional concept in a lower-dimensional space. We’ve also learned that, conceptually, the process involves training a computer to learn about the substitutability of various terms by applying a human-like process of learning by association. But so far, we haven’t explored the algorithm that performs this feat. There are several such methods, which have been developed by big data companies or research universities and shared with the public.

Perhaps one of the most widely used word embedding techniques is **word2vec**, which was published in 2013 by a team of researchers at Google and, as the name suggests, literally transforms words to vectors. According to the authors, it is not a single algorithm so much as it is a collection of methods that can be used for natural language processing tasks. Although there have been many new methods published in the time since word2vec was published, it remains popular and is well studied even today. Understanding the full scope of word2vec is outside the scope of this chapter, but understanding some of its key components will provide a foundation upon which many other natural language processing techniques can be understood.



For a deep dive into the word2vec approach, see *Efficient Estimation of Word Representations in Vector Space* by Mikolov, T., Chen, K., Corrado, G., and Dean, J., 2013 at <https://arxiv.org/abs/1301.3781>. Another early but widely used approach for word embeddings is called the **GloVe algorithm**, which was published in 2014 by a team at Stanford University and uses a similar set of methods. For more information on GloVe, see <https://nlp.stanford.edu/projects/glove/>.

Consider a computer attempting to learn from reading a large corpus of text such as a web page or textbook. To begin learning which words are associated and are substitutable for one another, the computer will need a formal definition of “context” to limit the scope to something more reasonable than the entire text, particularly if the text is large. To this end, the word2vec technique defines a **window size** parameter that dictates how many words of context will be used when attempting to understand a single word. A smaller window size guarantees a tight association between words in context, but because related words can appear much later in the sentence, making the window too small may lead to missing important relationships among words and ideas. Balance is required, because making the window too large can pull in unrelated ideas much earlier or later in the text. Typically, the window is set to approximately the length of a sentence, or about five to ten words, with useless stop words like “and,” “but,” and “the” excluded.

Given contexts comprising approximately sentence-length sets of words, the word2vec process proceeds with one of two methodologies. The **continuous bag-of-words (CBOW)** methodology trains a model to predict each word from its context; the **skip-gram** approach does the inverse and attempts to guess the surrounding contextual words when provided with a single input word. Although the underlying process is nearly identical for both approaches, there are mathematical nuances that lead to different results depending on which one is used.

Because we are merely understanding the methods conceptually, it suffices to say that the CBOW methodology tends to create embeddings favoring words that are nearly identical replacements or true synonyms for one another, such as “apple” and “apples” or “burger” and “hamburger,” while the skip-gram method favors terms that are conceptually similar, like “apple” and “fruit” or “burger” and “fries.”

For both CBOW and skip-gram, the process of developing the embedding is similar and can be understood as follows. Beginning from a sentence like “an apple is a fruit I eat for lunch,” a model is constructed that attempts to relate a word like “apple” to its context, like “fruit,” “eat,” and “lunch.” By iterating over huge volumes of such sentences—like “a banana is a fruit people eat for breakfast” or “an orange is both a fruit and a color” and so on—the values of the embedding can be determined, such that the embedding minimizes the prediction error between the word and its context. Words that appear consistently in similar contexts will thus have similar values for the embedding and can therefore be treated as similar, interchangeable concepts:

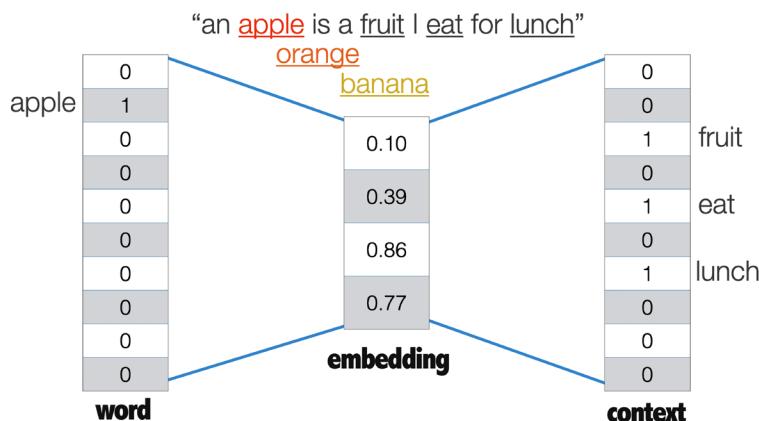


Figure 15.6: The word2vec process creates an embedding that relates each term to its context

Technically speaking, the word2vec approach is not considered “deep learning” even though it is in many ways analogous to deep learning. As depicted in the figure that follows, the embedding itself can be imagined as a hidden layer in a neural network, here represented with four nodes. In the CBOW approach, the input layer is a one-hot encoding of the input term, with one node for each possible word in the vocabulary, but only a single node with a value of 1 and the remaining nodes set to 0 values. The output layer also has one node per term in the vocabulary but can have multiple “1” values—each representing a word appearing in the context of the input term.

Note that for the skip-gram approach, this arrangement would be reversed:

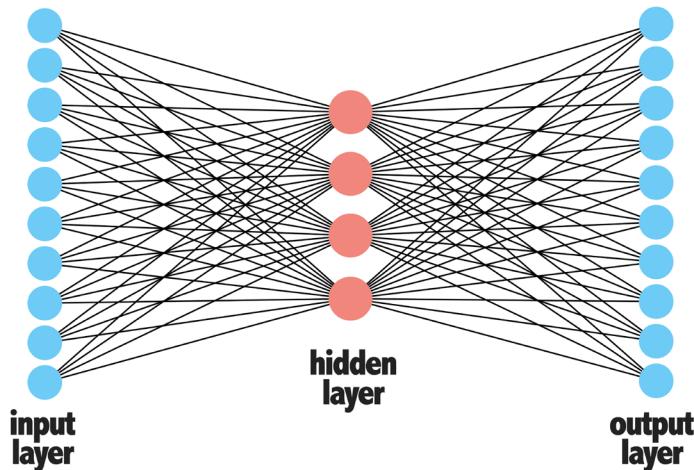


Figure 15.7: Developing an embedding involves training a model in a process analogous to deep learning

Varying the number of nodes in the hidden layer affects the complexity of the network as well as the depth of the model’s semantic understanding of each term. A greater number of nodes leads to a richer understanding of each term in its context but becomes much more computationally expensive to train and requires much more training data. Each additional node adds an additional dimension from which each term can be distinguished. Too few nodes and the model will have insufficient dimensionality to capture the many nuances of how each term can be used—the word “orange” as a color versus “orange” as a food, for instance—but using too many dimensions may increase the risk of the model being distracted by noise, or worse, being useless for the embedding’s initial intended purpose of dimensionality reduction! As you will soon see firsthand, even though the embeddings presented so far used just a few dimensions for simplicity and illustrative purposes, actual word embeddings used in practice typically have hundreds of dimensions and require huge amounts of training data and computational power to train.

In R, installing the `word2vec` package by Jan Wijffels will provide a wrapper for the C++ implementation of the `word2vec` algorithm. If desired, the package can train a word embedding if given a corpus of text data, but it is often preferable to use pre-trained embeddings that can be downloaded from the web. Here, we’ll use an embedding that was trained using a Google News archive consisting of 100 billion written words.

The resulting embedding contains 300-dimensional vectors for 3 million words and simple phrases, and is available for download at the Google word2vec project page as follows: <https://code.google.com/archive/p/word2vec/>. To follow along with the example, look for the link to the GoogleNews-vectors-negative300.bin.gz file, then download, unzip, and save the file to your R project folder before proceeding.



As a word of warning, the Google News embedding is quite large at about 1.5 GB compressed (3.4 GB after unzipping) and unfortunately cannot be distributed with the code for this chapter. Furthermore, the file can be somewhat hard to find on the project website. Try a find command (*Ctrl + F* or *Command + F*) in your web browser to search the page for the file name if needed. Depending on your platform, you may need an additional program to unzip files with the Gzip compression algorithm (.gz file extension).

As shown in the code that follows, to read the Google News embedding into R, we'll load the word2vec package and use the `read.word2vec()` function. Ensure you have downloaded and installed the word2vec package and Google News embedding before attempting this step:

```
> library(word2vec)
> m_w2v <- read.word2vec(file = "GoogleNews-vectors-negative300.bin",
                           normalize = TRUE)
```

If the embedding loaded correctly, the `str()` command will show details about this pre-trained model:

```
> str(m_w2v)
```

```
List of 4
$ model      :<externalptr>
$ model_path: chr "GoogleNews-vectors-negative300.bin"
$ dim        : int 300
$ vocabulary: num 3e+06
- attr(*, "class")= chr "word2vec"
```

As expected, the embedding has 300 dimensions for each of the 3 million terms. We can obtain these dimensions for a term (or terms) using `predict()` as a lookup function on the model object. The `type = "embedding"` parameter requests the embedding vector for the term, as opposed to the most similar terms, which will be demonstrated shortly.

Here, we'll request the word vectors for a few terms related to breakfast, lunch, and dinner:

```
> foods <- predict(m_w2v, c("cereal", "bacon", "eggs",
+                           "sandwich", "salad", "steak", "spaghetti"),
+                     type = "embedding")

> meals <- predict(m_w2v, c("breakfast", "lunch", "dinner"),
+                     type = "embedding")
```

The previous commands created matrices named `foods` and `meals`, with rows reflecting the terms and columns representing the 300 dimensions of the embedding. We can examine the first few values of a single word vector for *cereal* as follows:

```
> head(foods["cereal", ])

[1] -1.1961552  0.7056815 -0.4154012  3.3832674  0.1438890 -0.2777683
```

Alternatively, we can examine the first few columns for all foods:

```
> foods[, 1:5]

      [,1]      [,2]      [,3]      [,4]      [,5]
cereal -1.1961552  0.7056815 -0.4154012  3.383267  0.1438890
bacon  -0.4791541 -0.8049789  0.5749849  2.278036  1.2266345
eggs   -1.0626601  0.3271616  0.3689792  1.456238 -0.3345411
sandwich -0.7829969 -0.3914984  0.7379323  2.996794 -0.2267311
salad   -0.6817439  0.9336928  0.6224619  2.647933  0.6866841
steak    -1.5433296  0.4492917  0.2944511  2.030697 -0.5102126
spaghetti -0.2083995 -0.6843739 -0.4476731  3.828377 -1.3121454
```

Although we have no idea what each of the five dimensions represents (nor any of the remaining 295 dimensions not shown), we would expect similar, more substitutable foods and concepts to be closer neighbors in the 300-dimensional space. We can take advantage of this to measure the relatedness of the foods to the three main meals of the day using the `word2vec_similarity()` function as follows:

```
> word2vec_similarity(foods, meals)

      breakfast     lunch     dinner
cereal  0.6042315 0.5326227 0.3473523
bacon   0.6586656 0.5594635 0.5982034
eggs    0.4939182 0.4477274 0.4690089
```

```
sandwich 0.6928092 0.7046211 0.5999536  
salad     0.6797127 0.6867730 0.6821324  
steak     0.6580227 0.6383550 0.7106042  
spaghetti 0.6301417 0.6122567 0.6742931
```

In this output, higher values indicate greater similarity between the foods and each of the three mealtimes, according to the 300-dimension word embedding. Unsurprisingly, breakfast foods like cereal, bacon, and eggs are closer to the word *breakfast* than they are to *lunch* or *dinner*. Sandwiches and salads are closest to lunch, while steak and spaghetti are closest to dinner.



Although it was not used in the previous example, it is a popular convention to use the **cosine similarity** measure, which considers only the direction of the compared vectors, rather than the default Euclidean distance-like measure, which considers both direction and magnitude. The cosine similarity can be obtained by specifying `type = "cosine"` when calling the `word2vec_similarity()` function. Here, it is not likely to substantially affect the results because the Google News vectors were normalized when they were loaded into R.

For a more practical application of word2vec concepts, let's revisit the hypothetical social media posts presented earlier and attempt to determine whether to present the users with a breakfast, lunch, or dinner advertisement. We'll start by creating a `user_posts` character vector, which stores the raw text of each of the posts:

```
> user_posts = c(  
+   "I eat bacon and eggs in the morning for the most important meal of  
+   the day!",  
+   "I am going to grab a quick sandwich this afternoon before hitting the  
+   gym.",  
+   "Can anyone provide restaurant recommendations for my date tonight?"  
)
```

Importantly, there is a substantial hurdle we must pass before applying word2vec to each of the user posts; specifically, each post is a sentence composed of multiple terms, and word2vec is only designed to return vectors for single words. Unfortunately, there is no perfect solution to this problem, and choosing the correct solution may depend on the desired use case. For instance, if the application is intended merely to identify people that post about a particular subject, it may suffice to iterate over each word in the post and determine whether any of the words meet a similarity threshold.

More complex alternative solutions exist for solving the problem of applying word2vec to longer strings of text. A common but somewhat crude solution involves simply averaging the word2vec vectors across all words in the sentence, but this often results in poor results for much the same reason that mixing too many colors of paint results in an ugly shade of brown. As sentences grow longer, averaging across all words creates a muddy mess due to the fact that some words will inevitably have vectors in opposite directions and the resulting average is meaningless. Moreover, as sentences grow in complexity, it is more likely that word order and grammar will affect the meaning of the words in the sentence.

An approach called doc2vec attempts to address this by adapting the training of word2vec to longer blocks of text, called documents, which need not be full documents but may be paragraphs or sentences. The premise of doc2vec is to create an embedding for each document based on the words appearing in the document. Document vectors can then be compared to determine the overall similarity between two documents. In our case, the goal would be to compare whether two documents (that is, sentences) are conveying similar ideas—for instance, is a user’s post like other sentences that were about breakfast, lunch, or dinner?

Unfortunately, we do not have access to a doc2vec model to use this more sophisticated approach, but we can apply the word2vec package’s `doc2vec()` function to create a document vector for each user post and treat the document vector as if it were a single word. As stated previously, for longer sentences this may create a muddied vector, but because social media posts are often short and to the point, this issue may be mitigated.

We’ll begin by loading the `tm` package, which was introduced in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*, as a collection of tools for processing text data. The package provides a `stopwords()` function, which can be combined with its `removeWords()` function to remove unhelpful terms from social media posts. Then, the `txt_clean_word2vec()` function is used to prepare the posts for use with doc2vec:

```
> library(tm)
> user_posts_clean <- removeWords(user_posts, stopwords())
> user_posts_clean <- txt_clean_word2vec(user_posts_clean)
```

To see the result of this processing, let’s look at the first cleaned user post:

```
> user_posts_clean[1] # Look at the first cleaned user post
[1] "i eat bacon eggs morning important meal day"
```

As expected, the text has been standardized and all unhelpful words have been removed. We can then supply the posts to the `doc2vec()` function, along with the pre-trained Google News word2vec model as follows:

```
> post_vectors <- doc2vec(m_w2v, user_posts_clean)
```

The result of this operation is a matrix with three rows (one for each document) and 300 columns (one for each dimension in the embedding). The `str()` command shows the first few values of this matrix:

```
> str(post_vectors)
```

```
num [1:3, 1:300] -1.541 0.48 -0.825 -0.198 0.955 ...
```

We'll need to compare these pseudo-document vectors to the word vectors for breakfast, lunch, and dinner. These vectors were created previously using the `predict()` function and the word2vec model, but the code is repeated here for clarity:

```
> meals <- predict(m_w2v, c("breakfast", "lunch", "dinner"),
  type = "embedding")
```

Finally, we can compute the similarity between the two. Each row represents a user's post, and the column values indicate the similarity between that post's document vector and the corresponding term:

```
> word2vec_similarity(post_vectors, meals)
```

	breakfast	lunch	dinner
[1,]	0.7811638	0.7695733	0.7151590
[2,]	0.6262028	0.6700359	0.5391957
[3,]	0.5475215	0.5308735	0.5646606

Unsurprisingly, the user post about bacon and eggs is most similar to the word breakfast, while the post with sandwiches is most similar to lunch, and the evening date is most related to dinner. We could use the maximum similarity per row to determine whether to display a breakfast, lunch, or dinner advertisement to each user.

Document vectors can also be used directly as predictors in supervised machine learning tasks. For example, *Chapter 14, Building Better Learners*, described a theoretical model for predicting a Twitter user's gender or future purchasing behavior based on the user's basic profile data, profile picture, and social media post text.

The chapter proposed ensembling a traditional machine learning model with a deep learning model for the image data and a naive Bayes text model for the user posts. Alternatively, it is possible to use document vectors as is by treating the 300 dimensions as 300 individual predictors that the supervised learning algorithm can use to determine which are relevant to predicting the user's gender:

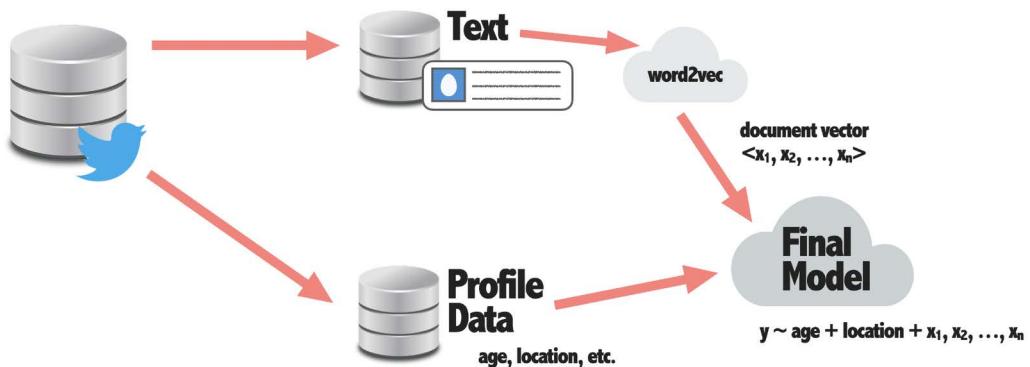


Figure 15.8: The values for a document vector resulting from unstructured text data can be used in a predictive model side by side with the more conventional predictors

This strategy of creating a document vector for an unstructured block of text and using the resultant embedding values as predictors for supervised learning is quite generalizable as a means of enhancing the performance of a conventional machine learning approach. Many datasets include unstructured text fields that go unused in conventional models due to their complexity or the inability to train a language model. However, a relatively simple transformation made possible by a pre-trained word embedding allows the text data to be used in the model alongside the other predictors. Thus, there is little excuse not to incorporate this approach and provide the learning algorithm with an infusion of big data the next time you encounter this type of machine learning task.

Visualizing highly dimensional data

Data exploration is one of the five key steps involved in any machine learning project, and thus is not immune to the so-called curse of dimensionality—the tendency of a project to become increasingly challenging as the number of features increases. Visualization techniques that work on simpler datasets may become useless as the number of dimensions grows unmanageable; for example, a scatterplot matrix may help identify relationships for a dozen or so features, but as the number grows bigger to dozens or hundreds of features, then what was once a helpful visualization may quickly turn into information overload.

Likewise, we can interpret a 2-D or even a three-dimensional plot without too much difficulty, but if we hope to understand the relationship among four or more dimensions, an entirely different approach is needed.

Though physics suggests there are ten or eleven dimensions of the universe, we only experience four, and only interact directly with three of them. Perhaps for this reason, our brains are attuned to understanding visuals in at most three dimensions; moreover, because most of our intellectual work is on 2-D surfaces like blackboards, whiteboards, paper, or computer screens, we are accustomed to seeing data represented in at most two dimensions. One day, as virtual or augmented reality computer interfaces become more prevalent, we may see an explosion of innovation in three-dimensional visualizations, but until that day comes, there is a need for tools that can aid the display of highly dimensional relationships in no more than two dimensions.

Reducing the dimensionality of a highly dimensional visualization to just two dimensions may seem like an impossibility, but the premise guiding the process is surprisingly straightforward: points that are closely positioned in the highly dimensional space need to be positioned closely in the 2-D space. If you are thinking that this idea sounds somewhat familiar, you would not be wrong; this is the same concept that guides embeddings, as described earlier in this chapter. The key difference is that while an embedding technique like word2vec reduces highly dimensional data down to a few hundred dimensions, embeddings for visualization must reduce the dimensionality even further to only two dimensions.

The limitations of using PCA for big data visualization

Principal component analysis (PCA), which was introduced in *Chapter 13, Challenging Data – Too Much, Too Little, Too Complex*, is one approach capable of reducing a highly dimensional dataset to two dimensions. You may recall that PCA works by expressing the covariance of multiple correlated attributes as a single vector. In this way, from the larger set of features, a smaller number of new features, called components, can be synthesized. If the number of components is set to two, a high-dimensional dataset can then be visualized in a simple scatterplot.

We'll apply this visualization technique to the 36-dimension social media profile dataset first introduced in *Chapter 9, Finding Groups of Data – Clustering with k-means*. The first few steps are straightforward; we use the tidyverse to read the data and select the 36 columns of interest, set the random seed to 123456 to ensure your results match the book, then use the `prcomp_irlba()` function from the `irlba` package to find the two principal components of the dataset:

```
> library(tidyverse)
> sns_terms <- read_csv("snsdata.csv") |> select(basketball:drugs)
```

```
> library(irlba)
> set.seed(123456)
> sns_pca <- sns_terms |>
  prcomp_irlba(n = 2, center = TRUE, scale = TRUE)
```

The `sns_pca$x` object contains a transformed version of the original dataset in which the 36 original dimensions have been reduced to 2. Because this is stored as a matrix, we'll first convert it to a data frame before piping it into a `ggplot()` function to create a scatterplot:

```
> library(ggplot2)
> as.data.frame(sns_pca$x) |>
  ggplot(aes(PC1, PC2)) + geom_point(size = 1, shape = 1)
```

The resulting visualization appears as follows:

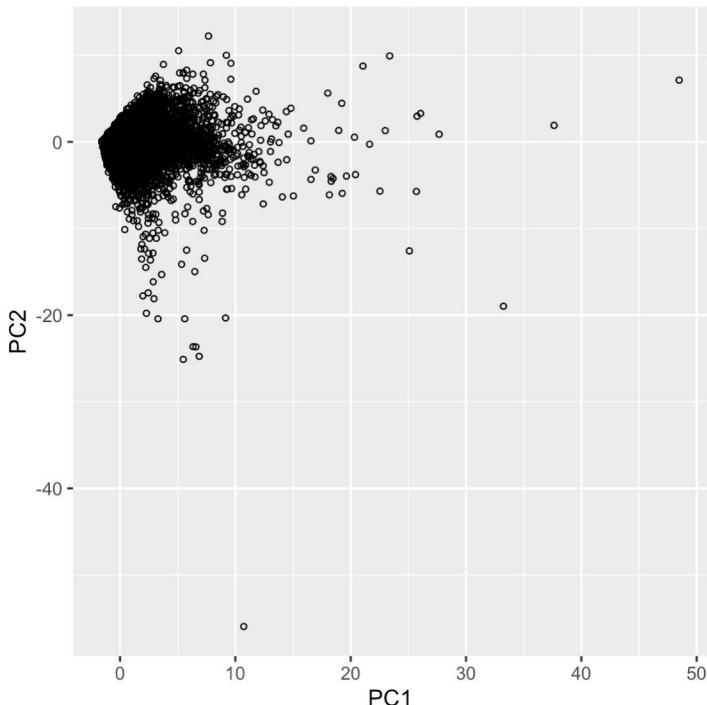


Figure 15.9: Principal component analysis (PCA) can be used to create 2-D visualizations of highly dimensional datasets, but the results are not always especially helpful

Unfortunately, this scatterplot reveals a limitation of using PCA for data exploration, which is that the two principal components often create little visual separation among the points in 2-D space. Based on our prior work in *Chapter 9, Finding Groups of Data – Clustering with k-means*, we know that there are clusters of social media users that use similar keywords on their social media profiles. These clusters ought to be visible as distinct groupings in the scatterplot, but instead, we see one large group of points and a scattering of apparent outliers around the perimeter. The disappointing result here is not specific to the dataset used here and is typical of PCA when used in this way. Thankfully, there is another algorithm that is better suited to data exploration, which will be introduced in the next section.

Understanding the t-SNE algorithm

The underlying math of the PCA technique utilizes covariance matrices to perform a linear dimensionality reduction, and the resulting principal components are intended to capture the overall variance of the dataset. The effect is like a compression algorithm that reduces the dimensionality of a dataset by eliminating redundant information. While this is obviously an important and useful attribute for a dimensionality reduction technique, it is less helpful for data visualization. As we observed in the previous section, this tendency of PCA to “compress” the dimensions may obscure important relationships in the data—the exact type of relationships we hope to discover when performing big data exploration.

A technique called **t-Distributed Stochastic Neighbor Embedding**, or **t-SNE** for short, is designed precisely as a tool for the visualization of high-dimensional datasets and thus addresses the previously mentioned shortcomings of PCA. The t-SNE approach was published in 2008 by Laurens van der Maaten, and it has quickly become a de facto standard for big data visualization for high-dimensional real-world datasets. Van der Maaten and others have published and presented numerous case studies contrasting PCA and t-SNE, and illustrating the strengths of the latter. However, because the math that drives t-SNE is highly complex, we will focus on understanding it conceptually and comparing it to other related methods covered previously.



For a deep dive into the mechanics of the t-SNE algorithm, see the original publication, *Visualizing Data using t-SNE*, van der Maaten, L. and Hinton, G., *Journal of Machine Learning Research* 9, 2008, pp. 2579-2606.

Just like with any technique for visualizing highly dimensional datasets, the goal of t-Distributed Stochastic Neighbor Embedding is to ensure that points or “neighbors” that are close in the high-dimensional space are positioned closely in the low-dimensional (2-D or 3-D) space.

The word *embedding* in the t-SNE name highlights the close connection between this and the more general task of constructing an embedding, as described in prior sections. However, as will be apparent shortly, t-SNE uses an approach unlike the deep learning analogue that is used for creating a word embedding. For starters, the word *stochastic* in the t-SNE name describes the non-deterministic nature of the algorithm, which implies that there is a relatively large degree of randomness in the output. But there are also more fundamental differences.

To begin to understand the t-SNE algorithm, imagine if the task were merely to reduce from three dimensions to two. In this case, if the data points were somehow depicted as small balls suspended in the air in three-dimensional space, and the same number of data points were placed randomly as flat discs on the ground in 2-D space, then a human could perform the dimensionality reduction by observing each ball in 3-D space, identifying its set of neighbors, and then carefully moving the discs in 2-D space to place neighbors closer together. Of course, this is more challenging than it sounds, because moving discs closer together and further apart in the flat space may inadvertently create or eliminate groupings relative to the 3-D space. For instance, moving point A to be closer to its neighbor point B may also move A closer to point C, when A and C should be distant according to the higher-dimensional space. For this reason, it would be important to iterate, observing each 3-D point's neighborhood and shifting its 2-D neighbors until the overall 2-D representation is relatively stable.

The same basic process can be performed algorithmically in a much larger number of dimensions using a series of mathematical steps. First, the similarity of each point in high-dimensional space is computed—traditionally, using the familiar metric of Euclidian distance as with k-means and k-nearest neighbors in earlier chapters. This similarity metric is used to define a conditional probability distribution stating that similar points are proportionally more probable to be neighbors in the high-dimensional space. Likewise, a similar distance metric and conditional probability distribution is defined for the low-dimensional space. With these two metrics defined, the algorithm must then optimize the entire system such that the overall error for the high- and low-dimensional probability distributions is minimized. Keep in mind that the two are inseparably linked by the fact they rely on the same set of examples; the coordinates are known for the high-dimensional space, so it is essentially solving for a way to transform the high-dimensional coordinates into a low-dimensional space while preserving the similarity as much as possible.

Given that the t-SNE algorithm is so different than PCA, it is no surprise that there are many differences in how they perform. An overall comparison of the two approaches is presented in the following table:

PCA	t-SNE
<ul style="list-style-type: none"> • Tends to compress the visualization • Global (overall) variance is depicted • Deterministic algorithm will produce the same result each run • Does not have hyperparameters to be set • Relatively fast (for datasets that can fit in memory) • Involves linear transformations • Useful as a general dimensionality reduction technique by creating additional principal components 	<ul style="list-style-type: none"> • Tends to cluster the visualization • Local variance is more apparent • Stochastic algorithm introduces randomness into the result • Result can be sensitive to hyperparameters • Relatively slow (but faster approximations exist) • Involves non-linear transformations • Typically used only as a data visualization technique (two or three dimensions)

As a rule of thumb, t-SNE is generally the more appropriate tool for big data visualization, but it is worth noting a few differences that can be weaknesses or present challenges in certain circumstances. First, we have observed that PCA can do a poor job at depicting natural clusters in the data, but t-SNE is so apt at presenting clusters that it can occasionally even form clusters in a dataset without these types of natural divisions. This fault is compounded by the fact that t-SNE is a non-deterministic algorithm that is often quite sensitive to the values of its hyperparameters; setting these parameters poorly is more likely to create false clusters or obscure real ones. Lastly, the t-SNE algorithm involves iterating repeatedly over a relatively slow process, but stopping too early often produces a poor result or creates a false sense of the dataset's structure; unfortunately, it is also possible that too many iterations will lead to the same problems!

These challenges are not listed here to imply that t-SNE is more work than it is worth, but rather to encourage treating the output with a degree of skepticism until it has been thoroughly explored. This may mean testing various hyperparameter combinations, or it may involve a qualitative examination of the visualization, such as investigating the identified clusters by hand in order to determine what features the neighborhood has in common. We'll see some of these potential pitfalls in practice in the next section, which applies t-SNE to a familiar real-world dataset.

Example – visualizing data’s natural clusters with t-SNE

To illustrate the ability of t-SNE to depict a dataset’s natural clusters, we’ll apply the method to the same 36-dimensional social media profile dataset used previously with PCA. Beginning as before, we’ll read the raw data into R using the tidyverse, but because t-SNE is somewhat computationally expensive, we use the `slice_sample()` command to limit the dataset to a random sample of 5,000 users. This is not strictly necessary but will speed up the execution time and make the visualization less dense and thus easier to read. Don’t forget to use the `set.seed(123)` command to ensure your results match those that follow:

```
> library(tidyverse)
> set.seed(123)
> sns_sample <- read_csv("snsdata.csv") |>
  slice_sample(n = 5000)
```

Even with a relatively small sample, the standard t-SNE implementation can still be rather slow. Instead, we will use a faster version called the **Barnes-Hut implementation**. The Barnes-Hut algorithm was originally developed to simulate the so-called “ n -body” problem—the complex system of gravitational relationships that arises among a set of n celestial bodies. Because every object exerts a force on every other object, exactly computing the net force for each body requires $n \times n = n^2$ calculations. This becomes computationally infeasible at an astronomical scale due to the scope of the universe and the virtually limitless numbers of objects within. Barnes-Hut simplifies this problem using a heuristic that treats more distant objects as a group identified by its center of mass, and only performs the exact calculations for objects closer than a threshold represented by the Greek letter *theta*. Larger values of theta drastically reduce the number of calculations needed to perform the simulation, while setting theta to zero performs the exact calculation.

Because the role of t-SNE can be imagined as an n -body problem of positioning points in space, with each point’s force of attraction to other points in the 2-D space based on how similar it is to the same points in the high-dimensional space, the Barnes-Hut simplification can be applied to simplify the computation of the system’s gravity-like forces. This provides a t-SNE implementation that is much faster and scales much better on large datasets.

The `Rtsne` package, which you should install if you have not done so already, provides a wrapper for the C++ implementation of Barnes-Hut t-SNE. It also includes other optimizations for use with very large-dimensional datasets. One of these optimizations includes an initial PCA step, which by default reduces the dataset to its first 50 principal components.

Admittedly, it may seem odd to use PCA as part of the t-SNE process, but the two have complementary strengths and weaknesses. While t-SNE tends to struggle with the curse of dimensionality, PCA is strong at dimensionality reduction; likewise, while PCA tends to obscure local variance, t-SNE highlights the data's natural structures. Using PCA to reduce the dimensionality and following this with the t-SNE process applies both techniques' strengths. In our case, with a dataset having only 36 dimensions, the PCA step does not meaningfully affect the result.

We'll begin by running a t-SNE process with the default parameters. After setting the random seed, the 5,000-row sample is piped into a `select()` command to choose only the 36 columns that measure the counts of various terms used on each user's profile. This is then piped into the `Rtsne()` function with `check_duplicates = FALSE` to prevent an error message that occurs when the dataset has duplicate rows. Duplicate rows are found in the social media dataset chiefly because there are many users who have counts of zero for all 36 terms. There is no reason that the t-SNE method cannot handle these duplicates, but including them may lead to unexpected or unsightly results in the visualization when the algorithm attempts to arrange such a tightly clustered set of points. For social media users, seeing this cluster will be helpful, so we will override the `Rtsne()` function's default as follows:

```
> library(Rtsne)
> set.seed(123)
> sns_tsne <- sns_sample |>
  select(basketball:drugs) |>
  Rtsne(check_duplicates = FALSE)
```



Piping a dataset into the `distinct()` function will eliminate duplicate rows and can be used prior to the `Rtsne()` command.

The 2-D representation of the 36-dimensional dataset is stored as a matrix named `Y` in the `sns_tsne` list object created by the `Rtsne()` function. This has 5,000 rows representing the social media users, and two columns representing the (x, y) coordinates of each user. After converting the matrix to a data frame, we can pipe these values into a `ggplot()` function to visualize the t-SNE result as follows:

```
> library(ggplot2)
> data.frame(sns_tsne$Y) |>
  ggplot(aes(X1, X2)) + geom_point(size = 2, shape = 1)
```

Displayed side by side with the earlier PCA visualization, it's remarkable to see the vast improvement in visual clarity that the t-SNE technique provides. Distinct clusters of users can be observed, reflecting these users' similarities in the 36-dimensional space:

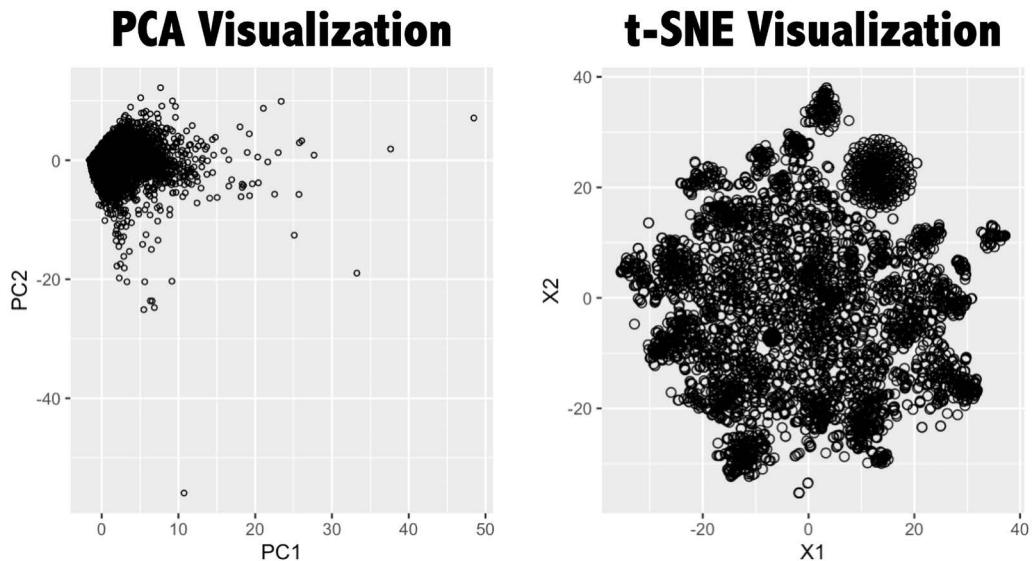


Figure 15.10: Compared to PCA, the t-SNE technique tends to create more useful visualizations that depict the data's natural clusters

Of course, it is somewhat unusual for a t-SNE visualization to work as nicely as this one did on the first try. If your results are disappointing, it is possible that merely setting a different random seed will generate better-looking results due to t-SNE's use of randomization. Additionally, the `perplexity` and `max_iter` parameters of the `Rtsne()` function can be adjusted to affect the size and density of the resulting plot. The perplexity governs the number of nearest neighbors to consider during the adjustment from high-to-low dimensions, and changing the maximum number of iterations (`max_iter`) up or down may lead the algorithm to arrive at a completely different solution.

Unfortunately, there are very few rules of thumb for tuning these parameters, and thus it often requires some trial and error to get things just right. The creator of t-SNE, Laurens van der Maaten, offers a few words of wisdom:



...one could say that a larger / denser dataset requires a larger perplexity. Typical values for the perplexity range between 5 and 50... [seeing a “ball” with uniformly distributed points] usually indicates you set your perplexity way too high. [If you continue to see bad results after tuning] maybe there is not very much nice structure in your data in the first place.

Source: <https://lvdmaaten.github.io/tsne/>

Be warned that the `Rtsne()` function parameters like `perplexity` and `max_iter` can drastically affect the amount of time it takes for the t-SNE algorithm to converge. If you’re not careful, you may need to force the process to quit rather than wait endlessly. Setting `verbose = TRUE` in the `Rtsne()` function call may provide insight into how the work has progressed.



For an outstanding treatment of t-SNE’s parameters and hyperparameters with interactive visualizations that show the impact of adjustments to each, see *How to Use t-SNE Effectively*, Wattenberg, M., Viégas, F., and Johnson, I., 2016, <https://distill.pub/2016/misread-tsne/>.

Because t-SNE is an unsupervised method, aside from the remarkably large and round cluster in the top right of the visualization—which we can reasonably assume is composed of identical users with no social media keywords in their profile—we have no idea what the other clusters represent. This being said, it is possible to probe the data to investigate the clusters by labeling points with different colors or shapes based on their underlying values.

For example, we can confirm the hypothesis about the top-right cluster by creating a categorical measure of how many keywords were used on each user’s page. The following tidyverse code begins by using `bind_cols()` to append the t-SNE coordinates onto the original dataset. Next, it uses the `rowwise()` function to change the behavior of `dplyr` so that the commands work on rows rather than columns. Thus, we can use the `sum()` function to count the number of terms each user had on their profile, using `c_across()` to select the columns with word counts. After using `ungroup()` to remove the rowwise behavior, this count is transformed into a two-outcome categorical variable using the `if_else()` function:

```
> sns_sample_tsne <- sns_sample |>  
  bind_cols(data.frame(sns_tsne$Y)) |> # add the t-SNE data
```

```
rowwise() |>
  mutate(n_terms = sum(c_across(basketball:drugs))) |>
  ungroup() |>
  mutate(`Terms Used` = if_else(n_terms > 0, "1+", "0"))
```

Using the result of this series of steps, we'll again plot the t-SNE data, but change the shape and color of the points according to the number of terms used:

```
> sns_sample_tsne |>
  ggplot(aes(X1, X2, shape = `Terms Used`, color = `Terms Used`)) +
  geom_point(size = 2) +
  scale_shape(solid = FALSE)
```

The resulting figure confirms our assumption, as the users with zero terms used in their social media profile (denoted by circles) comprise the dense cluster in the top right of the figure, while the users with one or more terms used (denoted by triangles) are scattered elsewhere in the plot:

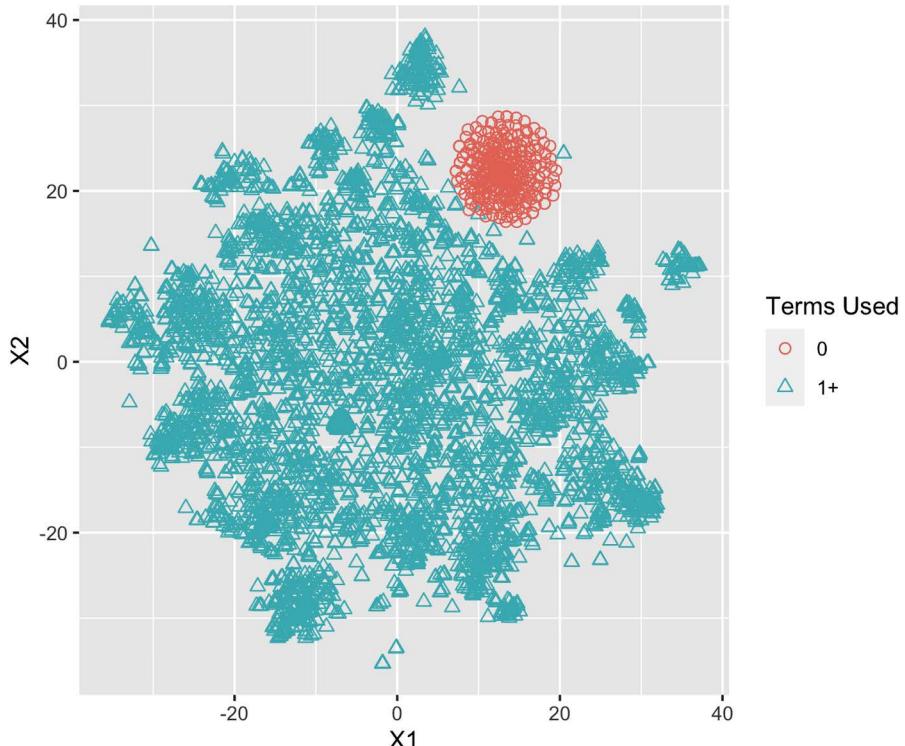


Figure 15.11: Adding color or changing the point style can help understand the clusters depicted in the t-SNE visualization

The t-SNE technique is more than just a tool to make pretty pictures, although it does tend to also do that well! For one, it may be helpful for determining the value of k to be used for k-means clustering. The t-SNE technique can also be used after clustering has been performed, with the points colored according to their cluster assignments to illustrate the clusters for presentation purposes. Stakeholders are more likely to trust a model with results that can be seen in a PowerPoint presentation. Similarly, t-SNE can be used to qualitatively gauge the performance of an embedding such as word2vec; if the embedding is meaningful, plotting the 300-dimensional vectors in 2-D space will reveal clusters of words with related meanings. With so many useful applications of t-SNE, it is no wonder that it has quickly become a popular tool in the data science toolkit.



For a fun application using both word2vec and t-SNE in which computers learned the meaning of emoji, see *emoji2vec: Learning Emoji Representations from their Description*, Eisner, B., Rocktäschel, T., Augenstein, I., Bošnjak, M., and Riedel, S., 2016, in *Proceedings of the 4th International Workshop on Natural Language Processing for Social Media at EMNLP 2016*.

While tools like word2vec and t-SNE provide means for understanding big data, they are of no use if R is unable to handle the workload. The remainder of this chapter will equip you with additional tools for loading, processing, and modeling such large data sources.

Adapting R to handle large datasets

Although the phrase “big data” means more than just the number of rows or the amount of memory a dataset consumes, sometimes working with a large volume of data can be a challenge in itself. Large datasets can cause computers to freeze or slow to a crawl when system memory runs out, or models cannot be built in a reasonable amount of time. Many real-world datasets are very large even if they are not truly “big,” and thus you are likely to encounter some of these issues on future projects. In doing so, you may find that the task of turning data into action is more difficult than it first appeared.

Thankfully, there are packages that make it easier to work with large datasets even while remaining in the R environment. We’ll begin by looking at the functionality that allows R to connect to databases and work with datasets that may exceed available system memory, as well as packages allowing R to work in parallel, and some that utilize modern machine learning frameworks in the cloud.

Querying data in SQL databases

Large datasets are often stored in a **database management system (DBMS)** such as Oracle, MySQL, PostgreSQL, Microsoft SQL, or SQLite. These systems allow the datasets to be accessed using the **Structured Query Language (SQL)**, a programming language designed to pull data from databases.

The tidy approach to managing database connections

RStudio version 1.1, which was released in 2017, introduced a graphical approach for connecting to databases. The **Connections** tab in the top-right portion of the interface provides the ability to interact with database connections found on your system. Upon clicking the **New Connection** button within this interface tab, you will see a window with the available connection options. The following screenshot depicts some of the possible connection types, but your own system is likely to have a different selection than those shown here:

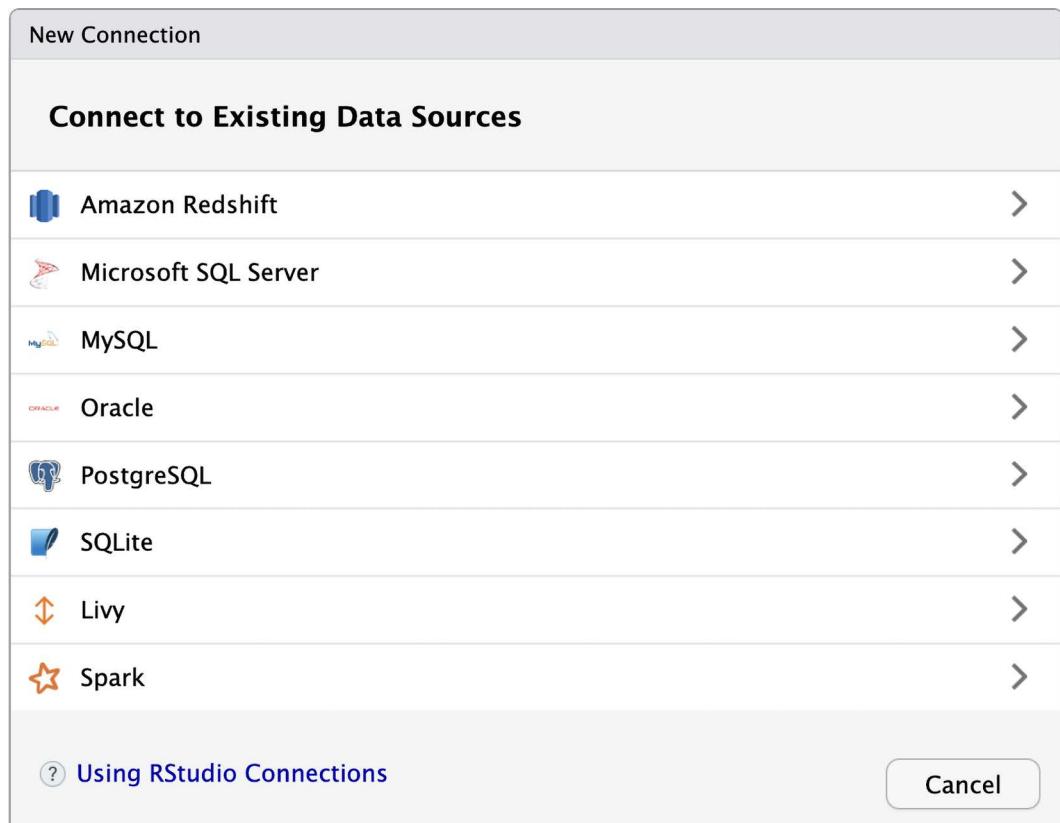


Figure 15.12: The “New Connection” button in RStudio v1.1 or greater opens an interface that will assist you with connecting to any predefined data sources

The creation of these connections is typically performed by a database administrator and is specific to the type of database as well as the operating system. For instance, on Microsoft Windows, you may need to install the appropriate database drivers as well as use the ODBC Data Source Administrator application; on macOS and Unix/Linux, you may need to install the drivers and edit an `odbc.ini` file. Complete documentation about the potential connection types and installation instructions is available at <https://solutions.posit.co/connections/db/>.

Behind the scenes, the graphical interface uses a variety of R packages to manage the connections to these data sources. At the core of this functionality is the `DBI` package, which provides a tidyverse-compliant front-end interface to the database. The `DBI` package also manages the back-end database driver, which must be provided by another R package. Such packages let R connect to Oracle (`ROracle`), MySQL (`RMySQL`), PostgreSQL (`RPostgreSQL`), and SQLite (`RSQLite`), among many others.

To illustrate this functionality, we'll use the `DBI` and `RSQLite` packages to connect to a SQLite database containing the credit dataset used previously. SQLite is a simple database that doesn't require running a server. It simply connects to a database file on a machine, which here is named `credit.sqlite3`. Before starting, be sure you've installed both required packages and saved the database file into your R working directory. After doing this, you can connect to the database using the following command:

```
> con <- dbConnect(RSQLite::SQLite(), "credit.sqlite3")
```

To prove the connection has succeeded, we can list the database tables to confirm the `credit` table exists as expected:

```
> dbListTables(con)
```

```
[1] "credit"
```

From here, we can send SQL query commands to the database and return records as R data frames. For instance, to return the loan applicants with an age of 45 years or greater, we would query the database as follows:

```
> res <- dbSendQuery(con, "SELECT * FROM credit WHERE age >= 45")
```

The entire result set can be fetched as a data frame using the following command:

```
> credit_age45 <- dbFetch(res)
```

To confirm that it worked, we'll examine the summary statistics, which confirm that the ages begin at 45 years:

```
> summary(credit_age45$age)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
45.00	48.00	52.00	53.98	60.00	75.00

When our work is done, it is advisable to clear the query result set and close the database connection to free these resources:

```
> dbClearResult(res)
> dbDisconnect(con)
```

In addition to SQLite and the database-specific R packages, the `odbc` package allows R to connect to many different types of databases using a single protocol known as the **Open Database Connectivity (ODBC)** standard. The ODBC standard can be used regardless of operating system or DBMS.

If you have previously connected to an ODBC database, you may have referred to it via its **data source name (DSN)**. You can use the DSN to create a database connection with a single line of R code:

```
> con <- dbConnect(odbc::odbc(), "my_data_source_name")
```

If you have a more complicated setup, or want to specify the connection properties manually, you can specify a full connection string as arguments to the DBI package `dbConnect()` function as follows:

```
> library(DBI)
> con <- dbConnect(odbc::odbc(),
  database = "my_database",
  uid = "my_username",
  pwd = "my_password",
  host = "my.server.address",
  port = 1234)
```

With the connection established, queries can be sent to the ODBC database and tables can be returned as data frames using the same functions that were used for the SQLite example previously.



Due to security and firewall settings, the instructions for configuring an ODBC network connection are highly specific to each situation. If you are having trouble setting up the connection, check with your database administrator. The Posit team (formerly known as RStudio) also provides helpful information at <https://solutions.posit.co/connections/db/best-practices/drivers/>.

Using a database backend for dplyr with dbplyr

Using the tidyverse’s `dplyr` functions with an external database is no more difficult than using it with a traditional data frame. The `dbplyr` package (short for “database plyr”) allows any database supported by the `DBI` package to be used transparently as a backend for `dplyr`. The connection allows tibble objects to be pulled from the database. Generally, one does not need to do more than merely install the `dbplyr` package, and `dplyr` can then take advantage of its functionality.

For example, let’s connect to the SQLite `credit.sqlite3` database used previously, then save its `credit` table as a tibble object using the `tbl()` function as follows:

```
> library(DBI)
> library(dplyr)
> con <- dbConnect(RSQLite::SQLite(), "credit.sqlite3")
> credit_tbl <- con |> tbl("credit")
```

Because `dplyr` has been routed through a database, the `credit_tbl` object here is not stored as a local R object, but rather is a table within a database. In spite of this, `credit_tbl` will act exactly like an ordinary tibble and will gain all the other benefits of the `dplyr` package, with the exception that the computational work will occur within the database rather than in R. This means that if the SQLite database were replaced with a database residing across a network on a more traditional SQL server, work could be offloaded to machines with more computational power rather than being performed on your local machine.

For example, to query the database and display the age summary statistics for credit applicants that are at least 45 years old, we can pipe the tibble through the following sequence of functions:

```
> library(dplyr)
> credit_tbl |>
  filter(age >= 45) |>
  select(age) |>
  collect() |>
  summary()
```

The result is as follows:

```
age
Min.    :45.00
1st Qu.:48.00
Median   :52.00
Mean     :53.98
3rd Qu.:60.00
Max.    :75.00
```

Note that the `dbplyr` functions are “lazy,” which means that no work is done in the database until it is necessary. Thus, the `collect()` function forces `dplyr` to retrieve the results from the “server” (in this case, a SQLite instance, but more typically a powerful database server) so that the summary statistics may be calculated. If the `collect()` statement is omitted, the code will fail as the `summary()` function cannot work directly with the database connection object.

Given a database connection, most `dplyr` commands will be translated seamlessly into SQL on the backend. To see how this works, we can ask `dbplyr` to show the SQL code that is generated for a series of `dplyr` steps. Let’s build a slightly more complex sequence of commands to show the average loan amount after filtering for ages 45 and older, and grouping by loan default status:

```
> credit_tbl |>
  filter(age >= 45) |>
  group_by(default) |>
  summarize(mean_amount = avg(amount))
```

The output shows that those that defaulted tended to request larger loan amounts on average:

```
# Source:  SQL [2 x 2]
# Database: sqlite 3.41.2 [/MLwR/Chapter 15/credit.sqlite3]
  default mean_amount
  <chr>      <dbl>
1 no          2709.
2 yes         4956.
```

Note that this looks different from a normal `dplyr` output and includes information about the database used, since the work was performed in the database rather than R. To see the SQL code that was generated to perform this analysis, simply pipe the steps into the `show_query()` function:

```
> credit_tbl |>
  filter(age >= 45) |>
  group_by(default) |>
  summarize(mean_amount = avg(amount)) |>
  show_query()
```

The output shows the SQL query that was run on the SQLite database:

```
<SQL>
SELECT `default`, avg(`amount`) AS `mean_amount`
FROM `credit`
WHERE (`age` >= 45.0)
GROUP BY `default`
```

Using the `dbplyr` functionality, the same R code that is used on smaller data frames can also be used to prepare larger datasets stored in SQL databases—the heavy lifting is done on the remote server, rather than your local laptop or desktop machine. In this way, learning the tidyverse suite of packages ensures your code will apply to any type of project from small to massive. Of course, there are even more ways to enable R to work with large datasets, as you will see in the sections that follow.

Doing work faster with parallel processing

In the early days of computing, computer processors always executed instructions in `serial`, which meant that they were limited to performing a single task at a time. In serial computing, the next instruction cannot be started until the previous instruction is complete:

Serial computing:

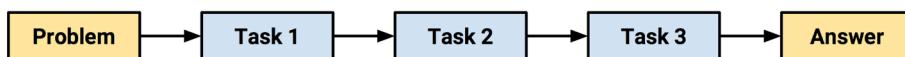


Figure 15.13: In serial computing, tasks cannot begin until prior tasks have been completed

Although it was widely known that many tasks could be completed more efficiently by completing steps simultaneously, the technology simply did not exist. This was addressed by the development of **parallel computing** methods, which use a set of two or more processors or computers to perform tasks simultaneously:

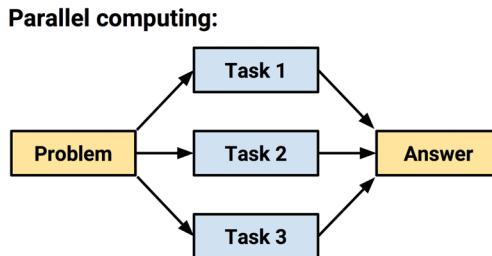


Figure 15.14: Parallel computing allows several tasks to occur simultaneously, which can speed up processing, but the results must be combined at the end

Many modern computers are designed for parallel computing. Even in the case that they have a single processor, they often have two or more cores that work in parallel. A core is essentially a processor within a processor, which allows computations to occur even if the other cores are busy with another task.

Networks of multiple computers called **clusters** can also be used for parallel computing. A large cluster may include a variety of hardware and be separated over large distances. In this case, the cluster is known as a **grid**. Taken to an extreme, a cluster or grid of hundreds or thousands of computers running commodity hardware could be a very powerful system. Cloud computing systems like Amazon Web Services (AWS) and Microsoft Azure make it easier than ever to use clusters for data science projects.

The catch, however, is that not every problem can be parallelized. Certain problems are more conducive to parallel execution than others. One might expect that adding 100 processors would result in 100 times the work being accomplished in the same amount of time (that is, the overall execution time would be 1/100), but this is typically not the case. The reason is that it takes effort to manage the workers. Work must be divided into equal, non-overlapping tasks, and each of the workers' results must be combined into one final answer.

So-called **embarrassingly parallel** problems are the ideal. These tasks are easy to reduce into non-overlapping blocks of work, and the results are easy to recombine. An example of an embarrassingly parallel machine learning task would be 10-fold cross-validation; once the 10 samples are divided, each of the 10 blocks of work is independent, meaning that they do not affect the others. As you will soon see, this task can be sped up quite dramatically using parallel computing.

Measuring R's execution time

Efforts to speed up R will be wasted if it is not possible to systematically measure how much time was saved. Although a stopwatch is one option, an easier solution is to wrap the offending code in a `system.time()` function.

For example, on the author's laptop, the `system.time()` function notes that it takes about 0.026 seconds to generate a million random numbers:

```
> system.time(rnorm(1000000))
```

user	system	elapsed
0.025	0.001	0.026

The same function can be used to evaluate improvement in performance, obtained with the methods that were just described or any R function.



For what it's worth, when the first edition of this book was published, generating a million random numbers took 0.130 seconds; the same took about 0.093 seconds for the second edition and 0.067 seconds for the third edition. Here, it takes only 0.026 seconds. Although I've used a slightly more powerful computer each time, this reduction of about 80 percent of the processing time over the course of about ten years illustrates just how quickly computer hardware and software are improving!

Enabling parallel processing in R

The `parallel` package, included with R version 2.14.0 and later, has lowered the entry barrier to deploying parallel algorithms by providing a standard framework for setting up worker processes that can complete tasks simultaneously. It does this by including components of the `multicore` and `snow` packages, which each take a different approach to multitasking.

If your computer is reasonably recent, you are likely to be able to use parallel processing. To determine the number of cores your machine has, use the `detectCores()` function as follows. Note that your output will differ depending on your hardware specifications:

```
> library(parallel)
> detectCores()
```

```
[1] 10
```

The `multicore` package was developed by Simon Urbanek and allows parallel processing on a single machine that has multiple processors or processor cores. It utilizes the multitasking capabilities of a computer's operating system to `fork`, or create a copy of, additional R sessions that share the same memory, and is perhaps the simplest way to get started with parallel processing in R.



Note that because the Microsoft Windows operating system does not support forking, the `multicore` example works only on macOS or Linux machines. For a Windows-ready solution, skip ahead to the next section on `foreach` and `doParallel`.

An easy way to get started with the `multicore` functionality is to use the `mclapply()` function, which is a multicore version of `lapply()`. For instance, the following blocks of code illustrate how the task of generating 10 million random numbers can be divided across 1, 2, 4, and 8 cores. The `unlist()` function is used to combine the parallel results (a list) into a single vector after each core has completed its chunk of work:

```
> system.time(l1 <- unlist(mclapply(1:10, function(x) {
  rnorm(10000000)}, mc.cores = 1)))
user  system elapsed
2.840  0.183  3.027

> system.time(l2 <- unlist(mclapply(1:10, function(x) {
  rnorm(10000000)}, mc.cores = 2)))
user  system elapsed
2.876  0.840  2.361

> system.time(l4 <- unlist(mclapply(1:10, function(x) {
  rnorm(10000000) }), mc.cores = 4)))
user  system elapsed
2.901  0.824  1.459

> system.time(l8 <- unlist(mclapply(1:10, function(x) {
  rnorm(10000000) }, mc.cores = 8)))
user  system elapsed
2.975  1.146  1.481
```

Notice how as the number of cores increases, the elapsed time decreases, though the benefit tapers off and may even be detrimental once too many cores have been added. Though this is a simple example, it can be adapted easily to many other tasks.

The `snow` package (Simple Network of Workstations) by Luke Tierney, A. J. Rossini, Na Li, and H. Sevcikova allows parallel computing on multicore or multiprocessor machines as well as on a network of multiple machines. It is slightly more difficult to use but offers much more power and flexibility. The `snow` functionality is included in the `parallel` package, so to set up a cluster on a single machine, use the `makeCluster()` function with the number of cores to be used:

```
> cl1 <- makeCluster(4)
```

Because `snow` communicates via network traffic, depending on your operating system, you may receive a message to approve access through your firewall.

To confirm the cluster is operational, we can ask each node to report back its hostname. The `clusterCall()` function executes a function on each machine in the cluster. In this case, we'll define a function that simply calls the `Sys.info()` function and returns the `nodename` parameter:

```
> clusterCall(cl1, function() { Sys.info()["nodename"] } )
```

```
[[1]]  
      nodename  
"Bretts-Macbook-Pro.local"  
  
[[2]]  
      nodename  
"Bretts-Macbook-Pro.local"  
  
[[3]]  
      nodename  
"Bretts-Macbook-Pro.local"  
  
[[4]]  
      nodename  
"Bretts-Macbook-Pro.local"
```

Unsurprisingly, since all four nodes are running on a single machine, they report back the same hostname. To have the four nodes run a different command, supply them with a unique parameter via the `clusterApply()` function. Here, we'll supply each node with a different letter. Each node will then perform a simple function on its letter in parallel:

```
> clusterApply(cl1, c('A', 'B', 'C', 'D'),  
              function(x) { paste("Cluster", x, "ready!") })
```

```
[[1]]  
[1] "Cluster A ready!"  
  
[[2]]  
[1] "Cluster B ready!"  
  
[[3]]  
[1] "Cluster C ready!"  
  
[[4]]  
[1] "Cluster D ready!"
```

When we're done with the cluster, it's important to terminate the processes it spawned. This will free up the resources each node is using:

```
> stopCluster(c11)
```

Using these simple commands, it is possible to speed up many machine learning tasks. For the largest big data problems, much more complex snow configurations are possible. For instance, you may attempt to configure a **Beowulf cluster**—a network of many consumer-grade machines. In academic and industry research settings with dedicated computing clusters, snow can use the Rmpi package to access these high-performance **message-passing interface (MPI)** servers. Working with such clusters requires knowledge of network configurations and computing hardware outside the scope of this book.



For a much more detailed introduction to snow, including some information on how to configure parallel computing on several computers over a network, see the following lecture by Luke Tierney: <http://homepage.stat.uiowa.edu/~luke/classes/295-hpc/notes/snow.pdf>.

Taking advantage of parallel with `foreach` and `doParallel`

The foreach package by Rich Calaway and Steve Weston provides perhaps the easiest way to get started with parallel computing, especially if you are running R on the Windows operating system, as some of the other packages are platform specific.

The core of the package is a foreach looping construct. If you have worked with other programming languages, this may be familiar. Essentially, it allows looping over a set of items, without explicitly counting the number of items; in other words, *for each item in the set, do something*.

If you’re thinking that R already provides a set of apply functions to loop over sets of items (for example, `apply()`, `lapply()`, `sapply()`, and so on), you are correct. However, the `foreach` loop has an additional benefit: iterations of the loop can be completed in parallel using a very simple syntax. Let’s see how this works.

Recall the command we’ve been using to generate millions of random numbers. To make this more challenging, let’s increase the count to a hundred million, which causes the process to take about 2.5 seconds:

```
> system.time(l1 <- rnorm(100000000))  
 user  system elapsed  
 2.466   0.080   2.546
```

After the `foreach` package has been installed, the same task can be expressed with a loop that combines four sets of 25,000,000 randomly generated numbers. The `.combine` parameter is an optional setting that tells `foreach` which function it should use to combine the final set of results from each loop iteration. In this case, since each iteration generates a set of random numbers, we simply use the `c()` concatenate function to create a single, combined vector:

```
> system.time(l4 <- foreach(i = 1:4, .combine = 'c')  
 %do% rnorm(25000000))  
  
 user  system elapsed  
 2.603   0.106   2.709
```

If you noticed that this function didn’t result in a speed improvement, good catch! In fact, the process was slower. The reason is that by default, the `foreach` package runs each loop iteration in serial, and the function adds a small amount of computational overhead to the process. The sister package `doParallel` provides a parallel backend for `foreach` that utilizes the `parallel` package included with R, described earlier in this chapter.

Before parallelizing this work, it is wise to confirm the number of cores available on your system as follows:

```
> detectCores()  
  
[1] 10
```

Your results will differ depending on your system capabilities.

Next, after installing and loading the `doParallel` package, simply register the desired number of cores and swap the `%do%` command with the `%dopar%` operator. Here, we only need at most four cores, as there are only four groups of random numbers to combine:

```
> library(doParallel)
> registerDoParallel(cores = 4)
> system.time(l4p <- foreach(i = 1:4, .combine = 'c')
  %dopar% rnorm(25000000))
```

user	system	elapsed
2.868	1.041	1.571

As shown in the output, this results in a performance increase, cutting the execution time by about 40 percent.



Warning: if the `cores` parameter is set to a number greater than the available cores on your system, or if the combined work exceeds the free memory on your computer, R may crash! In this case, the vector of random numbers is nearly a gigabyte of data, so systems with low RAM may be especially prone to crashing here.

To close the `doParallel` cluster, simply type:

```
> stopImplicitCluster()
```

Though the cluster will be closed automatically at the conclusion of the R session, it is better form to do so explicitly.

Training and evaluating models in parallel with `caret`

The `caret` package by Max Kuhn (covered previously in *Chapter 10, Evaluating Model Performance*, and *Chapter 14, Building Better Learners*) will transparently utilize a parallel backend if one has been registered with R using the `foreach` package described previously.

Let's look at a simple example in which we attempt to train a random forest model on the credit dataset. Without parallelization, the model takes about 65 seconds to train:

```
> library(caret)
> credit <- read.csv("credit.csv")
> system.time(train(default ~ ., data = credit, method = "rf",
  trControl = trainControl(allowParallel = FALSE)))
```

```
user  system elapsed
64.009  0.870 64.855
```

On the other hand, if we use the `doParallel` package to register eight cores to be used in parallel (be sure to lower this number if you have fewer than eight cores available), the model takes about 10 seconds to build—less than one-sixth of the time—and we didn’t need to change the remaining `caret` code:

```
> library(doParallel)
> registerDoParallel(cores = 8)
> system.time(train(default ~ ., data = credit, method = "rf"))
```

```
user  system elapsed
68.396  1.692 10.569
```

Many of the tasks involved in training and evaluating models, such as creating random samples and repeatedly testing predictions for 10-fold cross-validation, are embarrassingly parallel and ripe for performance improvements. With this in mind, it is wise to always register multiple cores before beginning a `caret` project.



Configuration instructions and a case study of the performance improvements for enabling parallel processing in `caret` are available on the project’s website: <https://topepo.github.io/caret/parallel-processing.html>.

Utilizing specialized hardware and algorithms

Base R has a reputation for being slow and memory inefficient, a reputation that is at least somewhat earned. These faults are largely unnoticed on a modern PC for datasets of many thousands of records, but datasets with millions of records or more can exceed the limits of what is currently possible with consumer-grade hardware. The problem is worsened if the dataset contains many features or if complex learning algorithms are being used.



CRAN has a high-performance computing task view that lists packages pushing the boundaries of what is possible in R at <http://cran.r-project.org/web/views/HighPerformanceComputing.html>.

Packages that extend R past the capabilities of the base package are being developed rapidly. These packages allow R to work faster, perhaps by spreading the work over additional computers or processors, by utilizing specialized computer hardware, or by providing machine learning optimized to big data problems.

Parallel computing with MapReduce concepts via Apache Spark

The **MapReduce** programming model was developed at Google to process its data on a large cluster of networked computers. MapReduce conceptualizes parallel programming as a two-step process:

- A **map** step, in which a problem is divided into smaller tasks that are distributed across the computers in the cluster
- A **reduce** step, in which the results of the small chunks of work are collected and synthesized into a solution to the original problem

A popular open-source alternative to the proprietary MapReduce framework is **Apache Hadoop**. The Hadoop software comprises the MapReduce concept plus a distributed filesystem capable of storing large amounts of data across a cluster of computers. Hadoop requires somewhat specialized programming skills to take advantage of its capabilities and to perform even basic machine learning tasks. Additionally, although Hadoop is excellent at working with extremely large amounts of data, it may not always be the fastest option because it keeps all data on disk rather than utilizing available memory.

Apache Spark is a cluster-computing framework for big data, offering solutions to these issues with Hadoop. Spark takes advantage of the cluster's available memory to process data approximately 100x faster than Hadoop. Additionally, it provides easy-to-use libraries for many common data processing, analysis, and modeling tasks. These include the SparkSQL data query language, the MLLib machine learning library, GraphX for graph and network analysis, and the Spark Streaming library for processing real-time data streams. For these reasons, Spark is perhaps the current standard for open-source big data processing.



Packt Publishing has published many books on Spark. To search their current offerings, visit <https://subscription.packtpub.com/search?query=spark>.

Apache Spark is often run remotely on a cloud-hosted cluster of virtual machines, but its benefits can also be seen running on your own hardware. In either case, the `sparklyr` package connects to the cluster and provides a `dplyr` interface for analyzing the data using Spark.

More detailed instructions for using Spark with R can be found at <https://spark.rstudio.com>, but the basic instructions for getting up and running are straightforward.

To illustrate the fundamentals, let's build a random forest model on the credit dataset to predict loan defaults. To begin, you'll need to install and load the `sparklyr` package. Then, the first time you use Spark, you'll need to run the `spark_install()` function, which downloads Spark onto your computer. Note that this is a sizeable download at about 220 megabytes, as it includes the full Spark environment:

```
> library(sparklyr)  
> spark_install()
```

Additionally, Spark itself requires a Java installation, which can be downloaded from <http://www.java.com> if you do not already have it. Once Spark and Java have been installed, you can instantiate a Spark cluster on your local machine using the following command:

```
> spark_cluster <- spark_connect(master = "local")
```

Next, we'll load the loan dataset from the `credit.csv` file on our local machine into the Spark instance, then use the Spark function `sdf_random_split()` to randomly assign 75 and 25 percent of the data to the training and test sets, respectively. The `seed` parameter is the random seed to ensure the results are identical each time this code is run:

```
> splits <- sdf_random_split(credit_spark,  
                                train = 0.75, test = 0.25,  
                                seed = 123)
```

Lastly, we'll pipe the training data into the random forest model function, make predictions, and use the classification evaluator to compute the AUC on the test set:

```
> credit_rf <- splits$train |>  
  ml_random_forest(default ~ .)  
> pred <- ml_predict(credit_rf, splits$test)  
> ml_binary_classification_evaluator(pred,  
  metric_name = "areaUnderROC")
```

```
[1] 0.7824574
```

We'll then disconnect from the cluster:

```
> spark_disconnect(spark_cluster)
```

With just a few lines of R code, we've built a random forest model using Spark that could expand to model millions of records. If even more computing power is needed, the code can be run in the cloud using a massively parallel Spark cluster simply by pointing the `spark_connect()` function to the correct hostname. The code can also be easily adapted to other modeling approaches using one of the supervised learning functions in the Spark Machine Learning Library (MLlib) listed at <https://spark.rstudio.com/mlib/>.



Perhaps the easiest way to get started using Spark is with Databricks, a cloud platform developed by the creators of Spark that makes it easy to manage and scale clusters via a web-based interface. The free "Community Edition" provides a small cluster for you to try tutorials or even experiment with your own data. Check it out at <https://databricks.com>.

Learning via distributed and scalable algorithms with H2O

The **H2O project** (<https://h2o.ai>) is a big data framework that provides fast in-memory implementations of machine learning algorithms, which can also operate in a cluster-computing environment. It includes functions for many of the methods covered in this book, including Naive Bayes, regression, deep neural networks, k-means clustering, ensemble methods, and random forests, among many others.

H2O uses heuristics to find approximate solutions to machine learning problems by iterating repeatedly over smaller chunks of the data. This gives the user the control to determine exactly how much of a massive dataset the learner should use. For some problems, a quick solution may be acceptable, but for others, the complete set may be required, which will require additional training time.

H2O is usually substantially faster and performs better on very massive datasets relative to Spark's machine learning functions, which are already much faster than base R. However, because Apache Spark is a commonly used cluster-computing and big data preparation environment, H2O can be run on Apache Spark using the **Sparkling Water** software. With Sparkling Water, data scientists have the best of both worlds—the benefits of Spark for data preparation, and the benefits of H2O for machine learning.

The `h2o` package provides functionality for accessing an H2O instance from within the R environment. A full tutorial on H2O is outside the scope of this book, and documentation is available at <http://docs.h2o.ai>, but the basics are straightforward.

To get started, be sure you have Java installed on your computer (<http://www.java.com>) and install the h2o package in R. Then, initialize a local H2O instance using the following code:

```
> library(h2o)
> h2o_instance <- h2o.init()
```

This starts an H2O server on your computer, which can be viewed via H2O Flow at <http://localhost:54321>. The H2O Flow web application allows you to administer and send commands to the H2O server, or even build and evaluate models using a simple, browser-based interface:

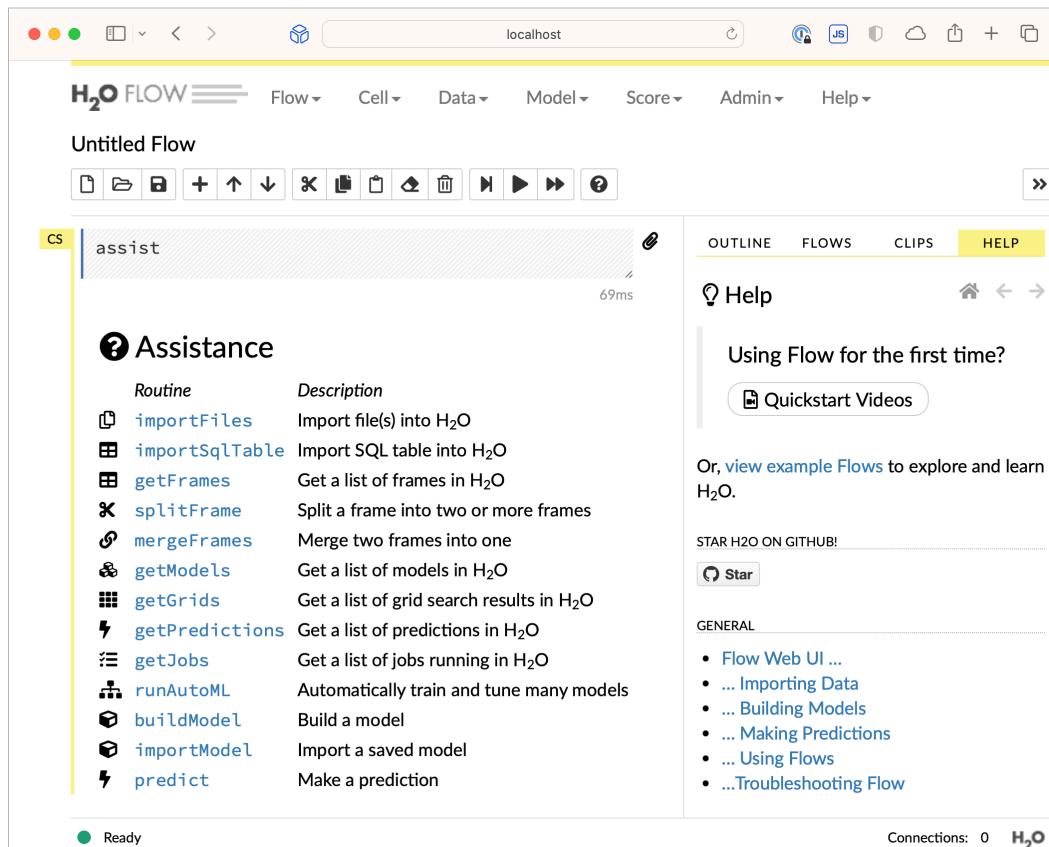


Figure 15.15: H2O Flow is a web application for interacting with the H2O instance

Although you could complete an analysis within this interface, let's go back to R and use H2O on the loan default data that we examined previously. First, we need to upload the `credit.csv` dataset to this instance using the following command:

```
> credit.hex <- h2o.uploadFile("credit.csv")
```

Note that the `.hex` extension is used to refer to an H2O data frame.

Next, we'll apply H2O's random forest implementation to this dataset using the following command:

```
> h2o.randomForest(y = "default",
                     training_frame = credit.hex,
                     ntrees = 500,
                     seed = 123)
```

The output of this command includes information on the out-of-bag estimates of model performance:

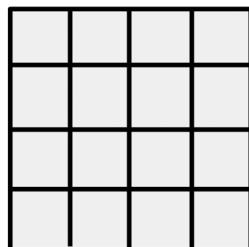
```
** Reported on training data. **
** Metrics reported on Out-Of-Bag training samples **
MSE:  0.1637001
RMSE:  0.4045987
LogLoss:  0.4956604
Mean Per-Class Error:  0.2835714
AUC:  0.7844833
AUCPR:  0.6195022
Gini:  0.5689667
R^2:  0.2204758
```

Although the credit dataset used here is not very large, the H2O code used here would scale to datasets of almost any size. Additionally, the code would be virtually unchanged if it were to be run in the cloud—simply point the `h2o.init()` function to the remote host.

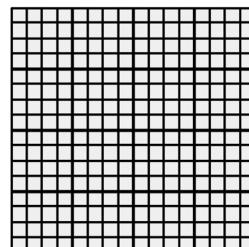
GPU computing

An alternative to parallel processing uses a computer's **graphics processing unit (GPU)** to increase the speed of mathematical calculations. A GPU is a specialized processor that is optimized for rapidly displaying images on a computer screen. Because a computer often needs to display complex 3D graphics (particularly for video games), many GPUs use hardware designed for parallel processing and extremely efficient matrix and vector calculations.

A side benefit is that they can be used to efficiently solve certain types of mathematical problems. As depicted in the following illustration, where a typical laptop or desktop computer processor may have on the order of 16 cores, a typical GPU may have thousands or even tens of thousands:



CPU with 16 cores



GPU with 1000+ cores

Figure 15.16: A graphics processing unit (GPU) has many times more cores than the typical central processing unit (CPU)

The downside of GPU computing is that it requires specific hardware that is not included with many computers. In most cases, a GPU from the manufacturer NVIDIA is required, as it provides a proprietary framework called **Complete Unified Device Architecture (CUDA)** that makes the GPU programmable using common languages such as C++.



For more information on NVIDIA's role in GPU computing, go to <https://www.nvidia.com/en-us/deep-learning-ai/solutions/machine-learning/>.

The `gputools` package by Josh Buckner, Mark Seligman, and Justin Wilson implements several R functions, such as matrix operations, clustering, and regression modeling using the NVIDIA CUDA toolkit. The package requires a CUDA 1.3 or higher GPU and the installation of the NVIDIA CUDA toolkit. This package was once the standard approach for GPU computing in R, but appears to have gone without an update since 2017 and has since been removed from the CRAN repository.

Instead, it appears that GPU work has transitioned to the TensorFlow mathematical library. The RStudio team provides information about using a local or cloud GPU on the following pages:

- https://tensorflow.rstudio.com/install/local_gpu
- https://tensorflow.rstudio.com/install/cloud_server_gpu

At the time of publication, a typical GPU used for deep learning is priced at several hundred US dollars for entry-level models and around \$1,000-\$3,000 for moderately priced units with greater performance. High-end units may cost many thousands of dollars.

Rather than spending this much up front, many people rent server time by the hour on cloud providers like AWS and Microsoft Azure, where it costs approximately \$1 per hour for a minimal GPU instance—just don’t forget to shut it down when your work completes, as it can get expensive quite quickly!

Summary

It is certainly an exciting time to be studying machine learning. Ongoing work on the relatively uncharted frontiers of parallel and distributed computing offers great potential for tapping the knowledge found in the deluge of big data. And the burgeoning data science community is facilitated by the free and open-source R programming language, which provides a very low barrier to entry—you simply need to be willing to learn.

The topics you have learned, in both this chapter as well as previous chapters, provide the foundation for understanding more advanced machine learning methods. It is now your responsibility to keep learning and adding tools to your arsenal. Along the way, be sure to keep in mind the no free lunch theorem—no learning algorithm rules them all, and they all have varying strengths and weaknesses. For this reason, there will always be a human element to machine learning, adding subject-specific knowledge and the ability to match the appropriate algorithm to the task at hand.

In the coming years, it will be interesting to see how the human side changes as the line between machine learning and human learning blurs. Services such as Amazon’s Mechanical Turk provide crowd-sourced intelligence, offering a cluster of human minds ready to perform simple tasks at a moment’s notice. Perhaps one day, just as we have used computers to perform tasks that human beings cannot do easily, computers will employ human beings to do the reverse. What interesting food for thought!

Join our book’s Discord space

Join our Discord community to meet like-minded people and learn alongside more than 4000 people at:

<https://packt.link/r>





packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

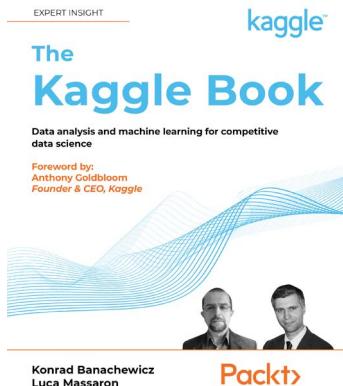
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



The Kaggle Book

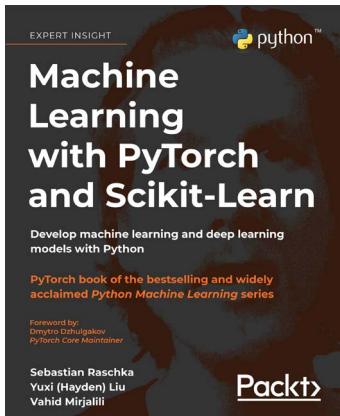
Konrad Banachewicz

Luca Marrson

ISBN: 978-1-80181-747-9

- Get acquainted with Kaggle as a competition platform
- Make the most of Kaggle Notebooks, Datasets, and Discussion forums
- Create a portfolio of projects and ideas to get further in your career
- Design k-fold and probabilistic validation schemes
- Get to grips with common and never-before-seen evaluation metrics

-
- Understand binary and multi-class classification and object detection
 - Approach NLP and time series tasks more effectively
 - Handle simulation and optimization competitions on Kaggle



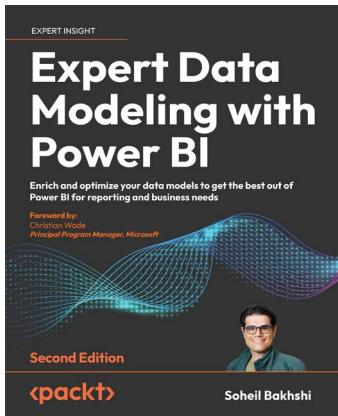
Machine Learning with PyTorch and Scikit-Learn

Dr. Sebastian Raschka

Yuxi (Hayden) Liu

ISBN: 978-1-80181-931-2

- Explore frameworks, models, and techniques for machines to ‘learn’ from data
- Use scikit-learn for machine learning and PyTorch for deep learning
- Train machine learning classifiers on images, text, and more
- Build and train neural networks, transformers, and boosting algorithms
- Discover best practices for evaluating and tuning models
- Predict continuous target outcomes using regression analysis
- Dig deeper into textual and social media data using sentiment analysis

**Expert Data Modeling with Power BI - Second Edition**

Soheil Bakhshi

ISBN: 978-1-80324-624-6

- Implement virtual tables and time intelligence functionalities in DAX to build a powerful model
- Identify Dimension and Fact tables and implement them in Power Query Editor
- Deal with advanced data preparation scenarios while building Star Schema
- Discover different hierarchies and their common pitfalls
- Understand complex data models and how to decrease the level of model complexity with different approaches
- Learn advanced data modeling techniques such as calculation groups, aggregations, incremental refresh, RLS/OLS, and more
- Get well-versed with datamarts and dataflows in PowerBI

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Machine Learning with R - Fourth Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

1-NN classification 89
1R (One Rule) algorithm 178-180
 strengths 179
 weakness 179
68-95-99.7 rule 72

A

abstraction 16
actionable rule 338
activation function 269
 Gaussian activation function 272
 linear activation function 272
 sigmoid activation function 270
 threshold activation function 269
 unit step activation function 269
active after-marketing 322
actuarial science 218
AdaBoost (adaptive boosting) 616
 AdaBoost.M1 algorithm 616
advanced data exploration
 data exploration roadmap,
 constructing 461-463
ggplot2, for visual data
 exploration 467-480

outliers, encountering 464-466
adversarial learning 30
agglomerative clustering 352
allocation function 610
Amazon Mechanical Turk
 URL 510
Amazon Web Services (AWS) 294, 698
antecedent 175
Apache Hadoop 706
Apache Spark 706
Apriori algorithm 317
Apriori principle
 used, for building set of rules 320, 321
Apriori property 318
area under ROC curve (AUC) 412, 413
 computing, in R 413
array 52
artificial intelligence (AI) 4
artificial neural network (ANN) 266, 267
 strength of concrete modeling example 281
artificial neuron 268, 269
association rules 316, 317
 Apriori algorithm 317-319
 frequently purchased groceries,
 identifying 321, 322

-
- set of rules, building with Apriori principle 320, 321
 - support and confidence, measuring 319, 320
 - autocorrelation** 438
 - auto insurance claims cost prediction example** 219
 - data, collecting 219, 220
 - data, exploring 220
 - data, preparing 220-223
 - improved regression model 233-235
 - insurance policyholder churn, predicting with logistic regression 238-244
 - interaction effects, adding to model 233
 - model performance, evaluating 230, 231
 - model performance, improving 232
 - model training, on data 227-229
 - nonlinear relationships, adding to model 232
 - predictions, with regression model 235-238
 - relationships among features, exploring 223
 - relationships among features, visualizing 224-226
 - automated feature engineering** 488
 - automated tuning, with caret**
 - example 598-600
 - simple tuned model, creating 601-604
 - tuning process, customizing 604-608
 - axis-parallel splits** 152
 - B**
 - backpropagation** 278
 - backward elimination technique** 541
 - bagging** 613
 - bag-of-words** 125
 - Barnes-Hut implementation** 686
 - Battleship-style grid search** 597
 - Bayesian methods** 110
 - concepts 110
 - conditional probability, computing 114-116
 - events 111
 - joint probability 112-114
 - probability 111
 - trials 111
 - Bayesian optimization** 597
 - Bayes' theorem**
 - used, for computing conditional probability 114-116
 - Beowulf cluster** 702
 - bias** 21
 - bias terms** 287
 - bias-variance tradeoff** 90
 - big data** 2, 666
 - big data regime 491
 - small data regime 491
 - variety 490
 - velocity 490
 - veracity 490
 - volume 490
 - bimodal** 75
 - binning** 122
 - bins** 122
 - biological neuron**
 - cell body 267
 - dendrites 267
 - synapse 267
 - bits** 155
 - bivariate relationships** 76
 - blending** 641
 - performing 642-645
 - practical methods 642-644

- boosting** 615, 616
bootstrap aggregating 613
bootstrap sampling 425-427
Boruta
 used, for feature selection 545-547
box-and-whisker plot 66
boxplot 66-68
branches 148
breast cancer, diagnosing with k-NN algorithm 95
 alternative values of k, testing 106, 107
 data, collecting 96
 data, exploring 96-98
 data, preparing 96-98
 model performance, evaluating 103, 104
 model performance, improving 104
 model, training on data 101-103
 numeric data, normalizing 98, 99
 test dataset, creating 100, 101
 training dataset, creating 100, 101
 z-score standardization 104-106
- C**
- C5.0 decision tree algorithm** 153, 154
 best split, selecting 154-156
 risky bank loans, identifying with 158
- cardinality** 462
- caret package** 704
 reference link 705
 used, for automated tuning 598-600
- CART algorithm** 245
- categorical feature** 26
 exploring 73, 74
- central processing unit (CPU)** 16
- central tendency**
 mean 62
 measuring 62, 74, 75
 median 63
 mode 74
- centroid** 360
- ChatGPT**
 URL 266
- chi-squared statistic** 157
- class** 27
- class-conditional independence** 119
- classification** 27
 with Naive Bayes algorithm 118-120
- classification rules** 175, 176
 1R algorithm 178-180
 from decision trees 182, 183
 greedy learners 183-185
 RIPPER algorithm 180-182
 separate and conquer 176-178
- clustering** 28, 348
 as machine learning task 348-350
- clustering algorithms**
 agglomeration function 351
 hierarchical methods 351
 model or density-based methods 351
 partition-based methods 351
 similarity metric 351
- clusters** 348, 698
- clusters, of clustering algorithms** 351
- Cohen's kappa coefficient** 393
- column-major order** 51
- combination function** 611
- comma-separated values (CSV) file** 55

Complete Unified Device Architecture (CUDA) 711
composer 611
Comprehensive R Archive Network (CRAN) 32
 Packages link 33
 Task Views link 33
 URL 32
conditional probability 114
 computing, with Bayes' theorem 114-117
confusion matrix 387, 388
consequent 175
contingency table 78
continuous bag-of-words (CBOW) methodology 672
control group 445
control object 605
convex hull 297
convolutional neural network (CNN) 275, 655-657
 convolutional layers 657
 fine tuning 658
 fully connected layers 657
 pooling layers 657
 pre-trained CNN example 659-666
 topology 657
 transfer learning 658
core points 355
corpus 127
correlation 77, 205-207
cosine similarity measure 677
cost matrix 173
cost per impression 364
covering algorithms 178
cross-tabulation 78-80

cross-validation 421
crosswalk table 510
Cubist algorithm 259
curse of dimensionality 535
cut points 122

D

data
 categorical features, exploring 73, 74
 exploring 59
 managing, with R 52
 numeric features, exploring 61, 62
 relationships, exploring 76
 structure, exploring 60, 61
data architects 491
database management system (DBMS) 692
data dictionary 60
data drift 447
data engineers 491
data exploration roadmap
 constructing 461-463
dataflow graph 654
data frame 47
 creating 48-50
data manipulator 611
data mining 3
data querying, SQL databases 692
 database backend, using for dplyr with dbplyr 695-697
 database connections, managing 692-694
data science 15, 452-454
 advanced data exploration, performing 460, 461
 R Markdown, using 458, 459
 R Notebooks, using 456-458

- datasets** 25
data source name (DSN) 694
data storage 16
dbplyr package 695
DBSCAN 355
deciles 65
decision nodes 148
decision tree forests 619
decision trees 148, 149
 classification rules 182
 pruning 157, 158
deep learning 275
 Keras 655
 practical applications 648
 tasks, selecting 650-652
 TensorFlow 653
 working with 649, 650
Deep Learning Playground
 reference link 278
deep neural network (DNN) 275, 655
delimiter 54
delta 439
dendrogram 351
 hypothetical dendrogram 352
density-based clustering 354
dependencies 33
dependent events 114
dependent variable 76, 198
descriptive model 28
Deutsche Mark (DM) 160
dimensionality reduction 535
discretization 567
discretize numeric features 122
distance function 88
divide and conquer 149-152
divisive clustering 352
document-term matrix (DTM) 133
doParallel package 703
downsampling layers 657
dplyr package 515
 data, piping 516-520
 data, preparing 515
 reference link 515
dummy coding 93, 94
dummy variable 78
- E**
- early stopping** 157
Eclat algorithm 343
elbow method 362
elbow point 362
elements 40
embarrassingly parallel problems 698
embedded methods 539-541
embedding 667
ensemble 29, 609
 model performance, improving with 609
ensemble-based algorithms 613
 bagging 613-615
 boosting 615-617
 extreme gradient boosting,
 with XGBoost 629-635
 gradient boosting 624-628
 random forests 618-623
 tree-based ensembles 636-638
ensemble learning 610-612
entropy 155
epoch 279

epoch, in backpropagation algorithm
 backward phase 279
 forward phase 279

error rate 389

Euclidean distance 88, 93

evaluation 22

exploding gradient problem 293

exploratory data analysis (EDA) 460

F

factor analysis 549

factors 43, 44

feature engineering 484
 external data, appending 509-511
 goals 496
 impact, of big data and deep learning 489-495
 in practice 496, 497
 insights hidden, finding in text 498-500
 neighbor's behavior, observing 501, 502
 new features, brainstorming 497, 498
 numeric ranges, transforming 500, 501
 performing 484
 related rows, utilizing 503, 504
 role of human and machine 485-488
 time series, decomposing 504-508

feature extraction 548
 performing 548
 principal component analysis 548-553

feature selection

applying 536-538
 Boruta, using 545-547
 embedded methods 540
 filter methods 538, 539
 stepwise regression, using 541-544
 wrapper methods 539

feature space 86

Featuretools
 URL 489

feedback network 276

feedforward networks 275

filter methods 538, 539

five-number summary 64

F-measure 403

focused sampling 580

folds 421

foreach package 702

forward selection technique 540

frequency table
 constructing 116

frequently purchased groceries
 data, collecting 322, 323
 data, exploring 323
 data model, training 331-335
 data, preparing 323
 identifying, with association rules 321, 322
 model performance, evaluating 335-339
 model performance, improving 339

future performance estimation 416
 bootstrap sampling 425-427
 cross-validation 421-425
 holdout method 417-420

G

gain ratio 157

Gaussian activation function 272

Gaussian RBF kernel 302

General Data Protection Regulation (GDPR) 13

generalizability 466

generalization 20

-
- generalized linear model (GLM)** 212
 canonical link function 213
 components 213-216
 exponential family 213
 family 213
 link function 213
- geom function** 469
- ggplot2**
 for visual data exploration 467-480
- Gini index** 157
- GloVe algorithm** 672
 reference link 672
- glyph** 303
- gmodels package**
 loading 34
 unloading 34
- GPU computing** 710, 711
- gradient boosting** 624-628
 with XGBoost 629-635
- gradient boosting models (GBMs)** 625
- gradient descent** 280
- graphics processing unit (GPU)** 710
- greedy learners** 183
- grid** 698
- H**
- H2O project** 708
 distributed and scalable algorithms,
 learning via 708-710
 reference link 708
- ham**
 examples 124
- harmonic mean** 403
- header line** 54
- heuristics** 20
- hierarchical clustering** 351
 dendrogram 352
- highly dimensional data**
 challenge 534
 PCA, limitations for big data
 visualization 681-683
 representing, as embeddings 667, 668
 t-SNE algorithm 683-685
 visualizing 680, 681
- histogram** 68-70
- holdout method** 417
- holdout stacking** 641
- hyper-confidence** 337
- hyper-lift** 337
- hyperparameters** 440
 scope, determining 593-597
 tuning 593
- hyperplanes** 294, 295
 classification with 295-297
 kernels, for nonlinear spaces 300-302
 linearly separable data 297, 298
 nonlinearly separable data 299, 300
- hypothesis testing** 199
- I**
- identity link** 215
- ImageNet database** 660
 URL 660
- imbalanced data** 579, 580
 balanced data, considering 587, 588
 strategies, for rebalancing data 580-583
 synthetic balanced dataset, generating with
 SMOTE 583, 584
- incremental reduced error pruning (IREP)
 algorithm** 181
- independent events** 113

-
- independent variable** 198
information gain 156
input data
 matching, to machine learning algorithms 31, 32
 types 24-26
instance-based learning 94
integrated development environment (IDE) 35
interaction 233
intercept 198
interpolation threshold 494
interquartile range (IQR) , 65
Interrater Reliability 396
item frequency plots 328, 329
Iterative Dichotomiser 3 (ID3) 153
- J**
- Java development kit (JDK)** 191
joint probability 112-114
- K**
- Kaggle**
 URL 432
kappa statistic 393-397
Keras 655
 URL 655
kernels
 for nonlinear spaces 300, 301
 Gaussian RBF kernel 302
 linear kernel 302
 polynomial kernel 302
 sigmoid kernel 302
kernel trick 300
- kernlab**
 reference link 306
k-fold cross-validation (k-fold CV) 421
k-means++ algorithm 358
k-means clustering algorithm 356, 357
 advantages 356
 disadvantages 356
 distance, used for cluster assignments 357-361
 distance, used for updating assignment 357-361
 number of clusters, selecting 362, 363
 teen market segments, finding 364
k-nearest neighbors algorithm (k-NN) 84-88, 356
 advantages 84
 appropriate k, selecting 90, 91
 data, preparing for usage 91-94
 disadvantages 84
 reason, for considering lazy learning 94, 95
 similarity, measuring with distance 88, 89
 used, for diagnosing breast cancer 95
knn() function 102
Knowledge Discovery and Data Mining (KDD) Cup
 reference link 432
knowledge representation 17
- L**
- lagged variables** 505
Laplace estimator 120-122
lazy learning algorithms 94
leaf nodes 148
leakage 437
learning rate 280

- leave-one-out method** 421
- left-hand side (LHS) rule** 316
- levels** 27
- LIBSVM library** 305
- likelihood** 115
 - table constructing 116
- linear activation function** 272
- linear kernel** 302
- linearly separable data** 295-298
- linear regression**
 - auto insurance claims cost prediction example 218
 - linear regression models 199
- list** 44
 - illustrating 45-47
- locally optimal solutions** 357
- logistic regression** 200, 216-218
- logit link function** 216
- log link function** 215
- loss function** 654
- LSTM neural networks** 276, 277
- lubridate package** 526
 - dates, cleaning 527-531
 - reference link 526
- M**
- machine learning** 2-4, 14
 - abstraction 15-20
 - data storage 14-16
 - ethics 10-14
 - evaluation 15, 22, 23
 - generalization 15, 20
 - limitations 8, 9
 - origins 2, 3
- skillsets, for practitioner** 430, 431
- successes** 7
- machine learning algorithms**
 - input data, matching to 31, 32
 - types 26-30
- machine learning model**
 - building 432-436
 - evaluations, conducting 439-442
 - predictions, avoiding 436-438
 - real-world impacts 443-448
 - trust building 448-452
- machine learning process, applying to real-world tasks**
 - data collection 23
 - data exploration 24
 - data preparation 24
 - model evaluation 24
 - model improvement 24
 - model training 24
- magrittr package**
 - reference link 516
- Manhattan distance** 89
- MapReduce** 706
 - map step 706
 - reduce step 706
- marginal likelihood** 115
- market basket analysis** 28
- matrix** 51
 - creating 51, 52
- matrix data** 25
- matrix inverse** 210
- matrix notation** 209
- Matthews correlation coefficient (MCC)** 397-399
- maximum likelihood estimation (MLE)** 218

-
- maximum margin hyperplane (MMH)** 296
- mean** 62
- mean absolute error (MAE)** 258
- mean imputation** 576
- median** 63
- message-passing interface (MPI) servers** 702
- meta-learners** 29
- meta-learning** 638, 639
- meta-learning methods** 612
- meta-model** 640
- Midjourney**
- URL 17
- mind map** 497
- min-max normalization** 92, 100
- missing data**
- data missing at random (MAR) 573
 - data missing completely at random (MCAR) 573
 - data missing not at random (MNAR) 574
 - handling 572, 573
 - multiple imputation 572
 - types 573
- missing value** 365
- imputation, performing 575, 576
- missing value indicator (MVI)** 576-578
- simple imputation with 576, 577
- missing value pattern (MVP)** 577, 578
- mixture modeling** 354
- mobile phone spam filtering example, Naive Bayes algorithm** 123
- data, collecting 124
 - data, exploring 125, 126
 - data, preparing 125, 126
 - indicator features, creating for frequent words 139, 140
- model performance, evaluating** 143, 144
- model performance, improving** 144, 145
- model, training on data** 141, 142
- test dataset, creating** 135, 136
- text data, cleaning** 126-132
- text data, standardizing** 126-132
- text data, visualizing** 136-139
- text documents, splitting text into words** 132-135
- training dataset, creating** 135, 136
- mode** 74
- model-based clustering** 354
- model decay** 447
- model drift** 448
- model explainability** 451
- model interpretability** 451
- model performance**
- association rules, saving to file 342
 - Eclat algorithm for greater efficiency, saving 343-345
 - improving, with ensemble 609
 - set of association rules, sorting 340
 - subsets of association rules, taking 341, 342
- model performance evaluation, for classification** 382
- classifier's predictions 383-387
 - confusion matrices, using 389-391
 - performance measures 391-393
 - visualizations, with ROC curves 404-408
- model performance evaluation metrics**
- F-measure 403
 - kappa statistic 393-397
 - Matthews correlation coefficient (MCC) 397-399
 - precision 401
 - recall 402

-
- sensitivity 400
 - specificity 400
 - model performance, improving** 170
 - costly errors, reducing 173-175
 - decision trees accuracy, boosting 170-173
 - models** 18
 - examples 18
 - model stacking** 640
 - for meta-learning 638, 639
 - model tree** 245
 - model tree algorithm (M5)** 246
 - m-of-n mapping** 564
 - multilayer perceptron (MLP)** 277
 - multimodal** 75
 - multinomial logistic regression model** 200
 - multiple linear regression** 199, 207-211
 - advantages 207
 - disadvantages 207
 - models 208
 - multivariate relationships** 76
 - N**
 - Naive Bayes algorithm** 110, 117
 - advantages 117
 - classification, performing with 118-120
 - disadvantages 117
 - Laplace estimator 120-122
 - mobile phone spam filtering example 123
 - numeric features, using 122, 123
 - nearest neighbor classification** 84
 - used, for computer vision applications 84
 - used, for identifying patterns 84
 - used, for recommendation systems 84
 - nested cross-validation** , 425
 - network topology** 273
 - characteristics 273
 - hidden layers 274
 - information travel direction 275-277
 - input nodes 273
 - layers 273
 - multilayer network 274
 - number of layers 273-275
 - number of nodes, in each layer 277, 278
 - output node 273
 - single-layer network 273
 - neural networks** 266
 - activation function 269
 - artificial neuron 269
 - biological neuron 267, 268
 - characteristics 269
 - network topology 269, 273
 - training algorithm 269
 - training, with backpropagation 278-280
 - neurons** 266
 - nodes** 266
 - No Free Lunch theorem** 22
 - URL 22
 - noisy data** 22
 - nominal feature** 26
 - nonlinear kernels**
 - advantages 301
 - disadvantages 301
 - nonlinearly separable data** 299, 300
 - non-parametric learning methods** 94
 - normal distribution** 71
 - numeric data** 26
 - numeric features**
 - boxplots 66-68
 - central tendency, measuring 62, 63

exploring 61, 62
 histograms 68-70
 numeric data 70, 71
 spread, measuring 64-66, 71, 72
 using, with Naive Bayes 122, 123
numeric prediction 27
NVIDIA CUDA toolkit 711

O

one-hot encoding 93
one-of-n mapping 564
OpenAI
 URL 5
Open Database Connectivity (ODBC)
 standard 694
optical character recognition (OCR), with
 SVMs 302
 data collection 303
 data, exploring 304, 305
 data, preparing 304, 305
 model performance, evaluating 308-310
 model performance, improving 310
 model training, on data 305-307
 SVM cost parameter accuracy,
 identifying 311-313
 SVM kernel function, changing 310
ordinal feature 26
ordinary least squares estimation 203-205
ordinary least squares (OLS) 203
outliers 63
out-of-bag error rate 622
overfitting 23
overparameterization 495
oversampling 580

P

package 32
parallel computing 697, 698
 enabling, in R 699-702
 model, evaluating 704
 model training 704
 R's execution time, measuring 699
 with `foreach` and `doParallel` 702, 703
 with MapReduce, via Apache Spark 706-708
parameters 439
partition-based clustering 353
pattern discovery 28
PCA loadings 557
Pearson correlation coefficient 205
Pearson's chi-squared test 80
percentiles 65
personas 376
**poisonous mushrooms identification, with
 rule learners** 185
 data, collecting 186
 data, exploring 186, 187
 data, preparing 186, 187
 model performance, evaluating 189, 190
 model performance, improving 190-194
 model, training on data 187-189
Poisson regression 200
polynomial kernel 302
Posit 36
 URL 35
Posit Cloud
 reference link 191
posterior probability 115
post-privacy era 13

- post-pruning** 157
precision 402
prediction accuracy 389
predictive model 26
principal component analysis (PCA) 548-553, 681
 used, for reducing highly dimensional social media data 553-561
 versus t-SNE algorithm 684
principal components 549
prior probability 115
probability 111
ROC
 reference link 413
proof-of-concept (POC) 445
proxy measure 498
pseudorandom number generator 162
pure 155
purity 155
PyTorch framework 655
- Q**
- quadratic optimization** 297
quantiles 65
quartiles 65
quintiles 65
- R**
- R** 36
 observations 36, 37
radial basis function (RBF) network 272
random-access memory (RAM) 16
random forests 618-623
- random sample** 161
R data structures 40
 array 52
 data frames 47-50
 dataset formats, importing with RStudio 56-58
 datasets, importing from CSV files 54, 55
 datasets, saving 54, 56
 factors 43, 44
 list 44-47
 loading 53
 managing 52
 matrix 51
 removing 54
 saving 52
 vectors 40-42
- readr package** 513
 reference link 513
 used, for reading rectangular files faster 513
- readxl package** 514
 reference link 514
 used, for reading rectangular files faster 514
- recall** 402
- receiver operating characteristic (ROC) curve** 404
 area under ROC curve (AUC) 412, 413
 AUC, computing in R 413-416
 comparing, across models 409-411
 creating 406-408, 413, 414
 performance visualizations 404, 405
- recency, frequency, monetary value (RFM) analysis** 239
- record linkage** 510
- rectified linear unit (ReLU)** 290
- recurrent neural network (RNN)** 276
- recursive partitioning** 149

- regression** 198-200
 - correlations 205-207
 - generalized linear model (GLM) 213-215
 - logistic regression 216-218
 - multiple linear regression 207-211
 - ordinary least squares estimation 203-205
 - simple linear regression 200-202
- regression analysis** 199
 - use cases 199
- regression trees** 245, 246
 - advantages 246
 - disadvantages 246
 - example 247, 248
 - wine quality estimation example 248
- regular expression (regex)** 131, 523
 - reference link 524
- reinforcement learning** 29, 30
- relationships, data**
 - bivariate relationships 76
 - examining 78-81
 - exploring 76
 - multivariate relationships 76
 - scatterplot visualization 76
 - two-way cross-tabulation 78
 - visualizing 76, 77
- repeated holdout** 421
- repeated k-fold CV** 425
- resampling techniques**
 - illustrating 581-583
- residuals** 203
- ResNet-50**
 - reference link 660
- resubstitution error** 416
- return on this investment (ROI)** 443
- RFM analysis** 438
- R, for large datasets** 691
 - algorithms, utilizing 705
 - data querying, in SQL databases 692
 - parallel processing 697, 698
 - specialized hardware, utilizing 705
- right-hand side (RHS) rule** 316
- RIPPER algorithm** 180, 181
- risky bank loans identification, with C5.0 decision trees** 158
 - data, collecting 159
 - data, exploring 159, 160
 - data, preparing 159, 160
 - model performance, evaluating 169
 - model performance, improving 170
 - model, training on data 163-168
 - random training, creating 161-163
 - test datasets, creating 161-163
- R Markdown**
 - using 458-460
- R Notebooks**
 - using 456-458
- Root-Mean-Squared Error (RMSE)** 600
- root node** 148
- rote learning** 94
- R packages**
 - installing 33, 34
 - loading 34
 - reference link 33
 - unloading 34
- R reticulate package** 661
 - reference link 661
- RStudio** 34-36
 - installing 35
 - used, for importing dataset formats 56-58
- rule learners** 175

S

sampling with replacement process 425
scatterplot 76
 using 77
scree plot 555
seed value 161
segmentation analysis 28
self-supervised learning 29
semi-supervised learning 29
sensitivity 400
sensitivity/specificity plot 405
sentiment analysis 499
separate and conquer 176
serial computing 697
Shapley Additive Explanations (SHAP) 452
 reference link 452
sigmoid activation function 270
sigmoid kernel 302
simple linear regression 199-202
simply multiple regression 199
skew 70
skip-gram approach 672
slack variable 299
slope 198
slope-intercept form 198
SmoothReLU 290
SMOTE algorithm
 applying, in R 584-587
 synthetic balanced dataset, generating
 with 583, 584
SnowballC package 131
social networking service (SNS) 364
softplus 290

spam
 examples 124
Sparkling Water 708
Spark Machine Learning Library (MLlib)
 reference link 708
sparse categorical data
 remapping 563-567
sparse data
 identifying 562, 563
 using 562
sparse matrix 133, 324
 creating, for transaction data 324-327
 plotting 330, 331
sparse numeric data
 binning 567-571
specificity 400
spread, measuring 64
 five-number summary 64
 quartiles 65, 66
 standard deviation 71, 72
 variance 71
squashing functions 272
stacked generalization 640
stacking 640
 practical methods 642-645
standard deviation 71, 72
standard deviation reduction (SDR) 247
statistical hypothesis testing 199
statistical power of analysis 576
stemming 131
stepwise regression 541
 used, for feature selection 541-544
stock models
 hyperparameter tuning, performing 593-597
 tuning 592

stop words 129
stratified random sampling 420
strength of concrete modeling example, ANN 281
 data collection 281
 data, exploring 282, 283
 data, preparing 282, 283
 model performance, evaluating 287, 288
 model performance, improving 288-293
 model training, on data 284-287
stringr package 520
 reference link 520
 text, transforming 520-526
structured data 25
Structured Query Language (SQL) 692
subsampling layers 657
subset() function
 keywords 342
 operators 342
subtree raising 158
subtree replacement 158
summary statistics 61
sum of the squared errors (SSE) 203, 287
sunk cost fallacy 443
super learning 641
supervised learning 27
SuperVision 656
support vector machine (SVM) 294
support vectors 296, 297
SVMlight
 reference link 306
synthetic balanced dataset
 generating, with SMOTE 583, 584
synthetic generation 583

T

tab-separated values (TSV) 56
tabular data file 54
target feature 26
target leakage 437
teen market segments, finding with k-means clustering 364
 data, collecting 364, 365
 data, exploring 365
 data, preparing 365, 366
 missing values, dummy coding 367
 missing values, inputting 368-370
 model performance, evaluating 373-376
 model performance, improving 377-379
 model, training on data 370-373
tensor 653
TensorFlow 653
 dataflow graph 654
 URL 653
term-document matrix (TDM) 133
terminal nodes 148
test dataset 22, 417
themis package
 URL 585
threshold activation function 269
tibble 512
 tidy table structure, making 512, 513
tidyverse 511
 reference link 511
time series data 437
tokenization 132
training 19
training dataset 22, 417

tree-based ensembling algorithms 636-638
trivial rule 339
t-SNE algorithm 683-685
 natural cluster data visualization
 example 686-691
 versus PCA 684
Tukey outliers 464
Turing test 266
 reference link 266
two-way cross-tabulation 78

U

undersampling 580
uniform distribution 70
unimodal 75
unit of analysis 25
unit of observation 24
unit step activation function 269
universal function approximator 278
unstable learners 613
unstructured data 25
unsupervised classification 348
unsupervised learning 28, 666

V

validation dataset 418
vanishing gradient problem 293
variable 61
variable's distribution 70
variance 71
vectors 40-42
Venn diagram 113

visual data exploration
 ggplot2 visualization 467-480
Visualizing Categorical Data (VCD) package 396
Voronoi diagram 359

W

Ward's method 353
weighting methods 611
window size parameter 672
winemaking 248
wine quality estimation example 248
 data, collecting 249
 data, exploring 250, 251
 data, preparing 250, 251
 decision trees, visualizing 255, 256
 model performance, evaluating 257
 model performance, improving 259-261
 model performance, measuring with
 MAE 257, 258
 model training, on data 252-254
word2vec 672
word cloud 136
word embeddings 669-671
 example 671-680
wrapper methods 539

X

XGBoost algorithm 629
 URL 629

Z

ZeroR 179
z-score 92
 z-score standardization 92, 104, 106

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/978-1-80107-132-1>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

