

DATA SCIENCE SERIES

# TREE-BASED METHODS FOR STATISTICAL LEARNING IN R



BRANDON M. GREENWELL



CRC Press  
Taylor & Francis Group

A CHAPMAN & HALL BOOK

# Tree-Based Methods for Statistical Learning in R

The book follows up most ideas and mathematical concepts with code-based examples in the R statistical language; with an emphasis on using as few external packages as possible. For example, users will be exposed to writing their own random forest and gradient tree boosting functions using simple for loops and basic tree fitting software (like **rpart** and **party/partykit**), and more. The core chapters also end with a detailed section on relevant software in both R and other opensource alternatives (e.g., Python, Spark, and Julia), and example usage on real data sets. While the book mostly uses R, it is meant to be equally accessible and useful to non-R programmers.

Consumers of this book will have gained a solid foundation (and appreciation) for tree-based methods and how they can be used to solve practical problems and challenges data scientists often face in applied work.

## Features:

- Thorough coverage, from the ground up, of tree-based methods (e.g., CART, conditional inference trees, bagging, boosting, and random forests).
- A companion website containing additional supplementary material and the code to reproduce every example and figure in the book.
- A companion R package, called **treemisc**, which contains several data sets and functions used throughout the book (e.g., there's an implementation of gradient tree boosting with LAD loss that shows how to perform the line search step by updating the terminal node estimates of a fitted **rpart** tree).
- Interesting examples that are of practical use; for example, how to construct partial dependence plots from a fitted model in Spark MLlib (using only Spark operations), or post-processing tree ensembles via the LASSO to reduce the number of trees while maintaining, or even improving performance.

## **CHAPMAN & HALL/CRC DATA SCIENCE SERIES**

Reflecting the interdisciplinary nature of the field, this book series brings together researchers, practitioners, and instructors from statistics, computer science, machine learning, and analytics. The series will publish cutting-edge research, industry applications, and textbooks in data science.

The inclusion of concrete examples, applications, and methods is highly encouraged. The scope of the series includes titles in the areas of machine learning, pattern recognition, predictive analytics, business analytics, Big Data, visualization, programming, software, learning analytics, data wrangling, interactive graphics, and reproducible research.

### Published Titles

#### **An Introduction to IoT Analytics**

*Harry G. Perros*

#### **Data Analytics**

A Small Data Approach

*Shuai Huang and Houtao Deng*

#### **Public Policy Analytics**

Code and Context for Data Science in Government

*Ken Steif*

#### **Supervised Machine Learning for Text Analysis in R**

*Emil Hvitfeldt and Julia Silge*

#### **Massive Graph Analytics**

*Edited by David Bader*

#### **Data Science**

An Introduction

*Tiffany-Anne Timbers, Trevor Campbell and Melissa Lee*

#### **Tree-Based Methods for Statistical Learning in R**

*Brandon M. Greenwell*

#### **Urban Informatics**

Using Big Data to Understand and Serve Communities

*Daniel T. O'Brien*

For more information about this series, please visit: <https://www.routledge.com/Chapman--HallCRC-Data-Science-Series/book-series/CHDSS>

# Tree-Based Methods for Statistical Learning in R

Brandon M. Greenwell



CRC Press  
Taylor & Francis Group  
Boca Raton London New York

---

CRC Press is an imprint of the  
Taylor & Francis Group, an **informa** business  
A CHAPMAN & HALL BOOK

First edition published 2022  
by CRC Press  
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press  
4 Park Square, Milton Park, Abingdon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

© 2022 Brandon M. Greenwell

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access [www.copyright.com](http://www.copyright.com) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact [mpkbookspermissions@tandf.co.uk](mailto:mpkbookspermissions@tandf.co.uk)

*Trademark notice:* Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

---

#### Library of Congress Cataloging-in-Publication Data

---

Names: Greenwell, Brandon M., author.

Title: Tree-Based Methods for Statistical Learning in R / Brandon M. Greenwell.

Description: First edition. | Boca Raton, FL : CRC Press, 2022. | Series: Chapman & Hall/CRC data science series | Includes bibliographical references and index.

Identifiers: LCCN 2021059606 (print) | LCCN 2021059607 (ebook) | ISBN 9780367532468 (hbk) | ISBN 9780367543822 (pbk) | ISBN 9781003089032 (ebk)

Subjects: LCSH: Decision making--Mathematics. | Decision trees--Data processing. | Trees (Graph theory)--Data processing. | R (Computer program language)

Classification: LCC T57.95 .G73 2022 (print) | LCC T57.95 (ebook) | DDC 658.4/03--dc23/eng/20220217

LC record available at <https://lccn.loc.gov/2021059606>

LC ebook record available at <https://lccn.loc.gov/2021059607>

---

ISBN: 978-0-367-53246-8 (hbk)

ISBN: 978-0-367-54382-2 (pbk)

ISBN: 978-1-003-08903-2 (ebk)

DOI: [10.1201/9781003089032](https://doi.org/10.1201/9781003089032)

Typeset in LM Roman  
by KnowledgeWorks Global Ltd.

*Publisher's note:* This book has been prepared from camera-ready copy provided by the authors.

To my son, Oliver.





Taylor & Francis  
Taylor & Francis Group  
<http://taylorandfrancis.com>

---

# *Contents*

---

<b>Preface</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Select topics in statistical and machine learning . . . . .	2
1.1.1 Statistical jargon and conventions . . . . .	3
1.1.2 Supervised learning . . . . .	4
1.1.2.1 Description . . . . .	5
1.1.2.2 Prediction . . . . .	6
1.1.2.3 Classification vs. regression . . . . .	7
1.1.2.4 Discrimination vs. prediction . . . . .	7
1.1.2.5 The bias-variance tradeoff . . . . .	8
1.1.3 Unsupervised learning . . . . .	10
1.2 Why trees? . . . . .	10
1.2.1 A brief history of decision trees . . . . .	12
1.2.2 The anatomy of a simple decision tree . . . . .	14
1.2.2.1 Example: survival on the Titanic . . . . .	15
1.3 Why R? . . . . .	17
1.3.1 No really, why R? . . . . .	17
1.3.2 Software information and conventions . . . . .	19
1.4 Some example data sets . . . . .	20
1.4.1 Swiss banknotes . . . . .	21
1.4.2 New York air quality measurements . . . . .	21
1.4.3 The Friedman 1 benchmark problem . . . . .	23
1.4.4 Mushroom edibility . . . . .	24
1.4.5 Spam or ham? . . . . .	25
1.4.6 Employee attrition . . . . .	28
1.4.7 Predicting home prices in Ames, Iowa . . . . .	29
1.4.8 Wine quality ratings . . . . .	30
1.4.9 Mayo Clinic primary biliary cholangitis study . . . . .	31
1.5 There ain't no such thing as a free lunch . . . . .	35
1.6 Outline of this book . . . . .	35
<b>I Decision trees</b>	<b>37</b>
<b>2 Binary recursive partitioning with CART</b>	<b>39</b>
2.1 Introduction . . . . .	39

2.2	Classification trees . . . . .	41
2.2.1	Splits on ordered variables . . . . .	43
2.2.1.1	So which is it in practice, Gini or entropy? .	47
2.2.2	Example: Swiss banknotes . . . . .	48
2.2.3	Fitted values and predictions . . . . .	51
2.2.4	Class priors and misclassification costs . . . . .	52
2.2.4.1	Altered priors . . . . .	54
2.2.4.2	Example: employee attrition . . . . .	55
2.3	Regression trees . . . . .	58
2.3.1	Example: New York air quality measurements . . . . .	59
2.4	Categorical splits . . . . .	62
2.4.1	Example: mushroom edibility . . . . .	64
2.4.2	Be wary of categoricals with high cardinality . . . . .	67
2.4.3	To encode, or not to encode? . . . . .	68
2.5	Building a decision tree . . . . .	69
2.5.1	Cost-complexity pruning . . . . .	71
2.5.1.1	Example: mushroom edibility . . . . .	74
2.5.2	Cross-validation . . . . .	77
2.5.2.1	The 1-SE rule . . . . .	78
2.6	Hyperparameters and tuning . . . . .	78
2.7	Missing data and surrogate splits . . . . .	78
2.7.1	Other missing value strategies . . . . .	80
2.8	Variable importance . . . . .	82
2.9	Software and examples . . . . .	83
2.9.1	Example: Swiss banknotes . . . . .	84
2.9.2	Example: mushroom edibility . . . . .	88
2.9.3	Example: predicting home prices . . . . .	96
2.9.4	Example: employee attrition . . . . .	100
2.9.5	Example: letter image recognition . . . . .	103
2.10	Discussion . . . . .	105
2.10.1	Advantages of CART . . . . .	105
2.10.2	Disadvantages of CART . . . . .	106
2.11	Recommended reading . . . . .	108
3	<b>Conditional inference trees</b>	111
3.1	Introduction . . . . .	111
3.2	Early attempts at unbiased recursive partitioning . . . . .	112
3.3	A quick digression into conditional inference . . . . .	114
3.3.1	Example: $\mathbf{X}$ and $\mathbf{Y}$ are both univariate continuous .	117
3.3.2	Example: $\mathbf{X}$ and $\mathbf{Y}$ are both nominal categorical .	118
3.3.3	Which test statistic should you use? . . . . .	120
3.4	Conditional inference trees . . . . .	121
3.4.1	Selecting the splitting variable . . . . .	121
3.4.1.1	Example: New York air quality measurements	123
3.4.1.2	Example: Swiss banknotes . . . . .	124

3.4.2	Finding the optimal split point . . . . .	125
3.4.2.1	Example: New York air quality measurements	126
3.4.3	Pruning . . . . .	128
3.4.4	Missing values . . . . .	128
3.4.5	Choice of $\alpha$ , $g()$ , and $h()$ . . . . .	128
3.4.6	Fitted values and predictions . . . . .	131
3.4.7	Imbalanced classes . . . . .	131
3.4.8	Variable importance . . . . .	132
3.5	Software and examples . . . . .	132
3.5.1	Example: New York air quality measurements . . .	133
3.5.2	Example: wine quality ratings . . . . .	137
3.5.3	Example: Mayo Clinic liver transplant data . . . .	140
3.6	Final thoughts . . . . .	143
<b>4</b>	<b>The hitchhiker’s GUIDE to modern decision trees</b>	<b>147</b>
4.1	Introduction . . . . .	148
4.2	A GUIDE for regression . . . . .	150
4.2.1	Piecewise constant models . . . . .	150
4.2.1.1	Example: New York air quality measurements	152
4.2.2	Interaction tests . . . . .	153
4.2.3	Non-constant fits . . . . .	154
4.2.3.1	Example: predicting home prices . . . . .	155
4.2.3.2	Bootstrap bias correction . . . . .	157
4.3	A GUIDE for classification . . . . .	157
4.3.1	Linear/oblique splits . . . . .	157
4.3.1.1	Example: classifying the Palmer penguins .	158
4.3.2	Priors and misclassification costs . . . . .	161
4.3.3	Non-constant fits . . . . .	161
4.3.3.1	Kernel-based and $k$ -nearest neighbor fits .	162
4.4	Pruning . . . . .	162
4.5	Missing values . . . . .	163
4.6	Fitted values and predictions . . . . .	163
4.7	Variable importance . . . . .	163
4.8	Ensembles . . . . .	164
4.9	Software and examples . . . . .	165
4.9.1	Example: credit card default . . . . .	165
4.10	Final thoughts . . . . .	172
<b>II</b>	<b>Tree-based ensembles</b>	<b>177</b>
<b>5</b>	<b>Ensemble algorithms</b>	<b>179</b>
5.1	Bootstrap aggregating (bagging) . . . . .	181
5.1.1	When does bagging work? . . . . .	184
5.1.2	Bagging from scratch: classifying email spam . .	184
5.1.3	Sampling without replacement . . . . .	187

5.1.4	Hyperparameters and tuning . . . . .	187
5.1.5	Software . . . . .	188
5.2	Boosting . . . . .	188
5.2.1	AdaBoost.M1 for binary outcomes . . . . .	189
5.2.2	Boosting from scratch: classifying email spam . . . . .	190
5.2.3	Tuning . . . . .	192
5.2.4	Forward stagewise additive modeling and exponential loss . . . . .	192
5.2.5	Software . . . . .	194
5.3	Bagging or boosting: which should you use? . . . . .	195
5.4	Variable importance . . . . .	195
5.5	Importance sampled learning ensembles . . . . .	196
5.5.1	Example: post-processing a bagged tree ensemble . . . . .	197
5.6	Final thoughts . . . . .	202
<b>6</b>	<b>Peeking inside the “black box”: post-hoc interpretability</b>	<b>203</b>
6.1	Feature importance . . . . .	204
6.1.1	Permutation importance . . . . .	204
6.1.2	Software . . . . .	206
6.1.3	Example: predicting home prices . . . . .	206
6.2	Feature effects . . . . .	208
6.2.1	Partial dependence . . . . .	208
6.2.1.1	Classification problems . . . . .	209
6.2.2	Interaction effects . . . . .	210
6.2.3	Individual conditional expectations . . . . .	210
6.2.4	Software . . . . .	211
6.2.5	Example: predicting home prices . . . . .	211
6.2.6	Example: Edgar Anderson’s iris data . . . . .	215
6.3	Feature contributions . . . . .	217
6.3.1	Shapley values . . . . .	217
6.3.2	Explaining predictions with Shapley values . . . . .	219
6.3.2.1	Tree SHAP . . . . .	220
6.3.2.2	Monte Carlo-based Shapley explanations . . . . .	221
6.3.3	Software . . . . .	223
6.3.4	Example: predicting home prices . . . . .	223
6.4	Drawbacks of existing methods . . . . .	225
6.5	Final thoughts . . . . .	226
<b>7</b>	<b>Random forests</b>	<b>229</b>
7.1	Introduction . . . . .	229
7.2	The random forest algorithm . . . . .	229
7.2.1	Voting and probability estimation . . . . .	232
7.2.1.1	Example: Mease model simulation . . . . .	234
7.2.2	Subsampling (without replacement) . . . . .	236
7.2.3	Random forest from scratch: predicting home prices . . . . .	237

7.3	Out-of-bag (OOB) data . . . . .	239
7.4	Hyperparameters and tuning . . . . .	243
7.5	Variable importance . . . . .	245
7.5.1	Impurity-based importance . . . . .	245
7.5.2	OOB-based permutation importance . . . . .	247
7.5.2.1	Holdout permutation importance . . . . .	248
7.5.2.2	Conditional permutation importance . . . . .	249
7.6	Casewise proximities . . . . .	249
7.6.1	Detecting anomalies and outliers . . . . .	251
7.6.1.1	Example: Swiss banknotes . . . . .	251
7.6.2	Missing value imputation . . . . .	252
7.6.3	Unsupervised random forests . . . . .	253
7.6.3.1	Example: Swiss banknotes . . . . .	254
7.6.4	Case-specific random forests . . . . .	254
7.7	Prediction standard errors . . . . .	256
7.7.1	Example: predicting email spam . . . . .	257
7.8	Random forest extensions . . . . .	258
7.8.1	Oblique random forests . . . . .	258
7.8.2	Quantile regression forests . . . . .	259
7.8.2.1	Example: predicting home prices (with prediction intervals) . . . . .	260
7.8.3	Rotation forests and random rotation forests . . . . .	261
7.8.3.1	Random rotation forests . . . . .	263
7.8.3.2	Example: Gaussian mixture data . . . . .	264
7.8.4	Extremely randomized trees . . . . .	267
7.8.5	Anomaly detection with isolation forests . . . . .	269
7.8.5.1	Extended isolation forests . . . . .	271
7.8.5.2	Example: detecting credit card fraud . . . . .	271
7.9	Software and examples . . . . .	276
7.9.1	Example: mushroom edibility . . . . .	277
7.9.2	Example: “deforesting” a random forest . . . . .	277
7.9.3	Example: survival on the Titanic . . . . .	283
7.9.3.1	Missing value imputation . . . . .	284
7.9.3.2	Analyzing the imputed data sets . . . . .	287
7.9.4	Example: class imbalance (the good, the bad, and the ugly) . . . . .	294
7.9.5	Example: partial dependence with Spark MLlib . . . . .	300
7.10	Final thoughts . . . . .	306
<b>8</b>	<b>Gradient boosting machines</b>	<b>309</b>
8.1	Steepest descent (a brief overview) . . . . .	310
8.2	Gradient tree boosting . . . . .	311
8.2.0.1	Loss functions . . . . .	314
8.2.0.2	Always a regression tree? . . . . .	317
8.2.0.3	Priors and missclassification cost . . . . .	317

8.3	Hyperparameters and tuning . . . . .	317
8.3.1	Boosting-specific hyperparameters . . . . .	318
8.3.1.1	The number of trees in the ensemble: $B$ . . .	318
8.3.1.2	Regularization and shrinkage . . . . .	319
8.3.1.3	Example: predicting ALS progression . . . . .	320
8.3.2	Tree-specific hyperparameters . . . . .	321
8.3.3	A simple tuning strategy . . . . .	322
8.4	Stochastic gradient boosting . . . . .	322
8.4.1	Column subsampling . . . . .	323
8.5	Gradient tree boosting from scratch . . . . .	323
8.5.1	Example: predicting home prices . . . . .	326
8.6	Interpretability . . . . .	327
8.6.1	Faster partial dependence with the recursion method .	328
8.6.1.1	Example: predicting email spam . . . . .	329
8.6.2	Monotonic constraints . . . . .	329
8.6.2.1	Example: bank marketing data . . . . .	330
8.7	Specialized topics . . . . .	332
8.7.1	Level-wise vs. leaf-wise tree induction . . . . .	332
8.7.2	Histogram binning . . . . .	333
8.7.3	Explainable boosting machines . . . . .	333
8.7.4	Probabilistic regression via natural gradient boosting	334
8.8	Specialized implementations . . . . .	335
8.8.1	eXtreme Gradient Boosting: XGBoost . . . . .	335
8.8.2	Light Gradient Boosting Machine: LightGBM . . .	337
8.8.3	CatBoost . . . . .	338
8.9	Software and examples . . . . .	339
8.9.1	Example: Mayo Clinic liver transplant data . . . .	339
8.9.2	Example: probabilistic predictions with NGBoost (in Python) . . . . .	346
8.9.3	Example: post-processing GBMs with the LASSO .	347
8.9.4	Example: direct marketing campaigns with XGBoost	351
8.10	Final thoughts . . . . .	356
	<b>Bibliography</b>	<b>359</b>
	<b>Index</b>	<b>381</b>

---

# Preface

---

Welcome to *Tree-based methods for statistical learning (with examples in R)*. Tree-based methods, as viewed in this book, refer to a broad family of algorithms that rely on *decision trees*, of which this book attempts to provide a thorough treatment. This is not a general statistical or machine learning book, nor is it an R book. Consequently, some familiarity with both would be useful, but I've tried to keep the core material as accessible and practical as possible to a broad audience (even if you're not an R programmer or master of statistical and machine learning). That being said, I'm a firm believer in learning by doing, and in understanding concept through code examples. To that end, almost every major section in this book is followed-up by general programming examples to help further drive the material home. Therefore, this book necessarily involves a lot of code snippets.

---

## Who is this book for?

This book is primarily aimed at researchers and practitioners who want to go beyond a fundamental understanding of tree-based methods, such as decision trees and tree-based ensembles. It could also serve as a useful supplementary text for a graduate level course on statistical and machine learning. Some parts of the book necessarily involve more math and notation than others, but where possible, I try to use code to make the concepts more comprehensible. For example, [Chapter 3](#) involves a bit of linear algebra and intimidating matrix notation, but the math-oriented sections can often be skipped without sacrificing too much in the way of understanding the core concepts; the adjacent code examples should also help drive the main concepts home by connecting the math to simple coding logic.

Nonetheless, this book does assume some familiarity with the basics of statistical and machine learning, as well as the R programming language. Useful references and resources are provided in the introductory material in [Chapter 1](#). While I try to provide sufficient detail and background where possible, some topics could only be given cursory treatment, though, whenever possible, I try to point the more ambitious reader in the right direction in terms of references.

---

## Companion website

There is a companion website for this book located at

<https://bgreenwell.github.io/treebook/>.

This is where I plan to include chapter exercises, code to reproduce most of the examples and figures in the book, errata, and various supplementary material.

Contributions from the community are more than welcome! If you notice something is missing from the website (e.g., the code to reproduce one of the figures or examples) or notice an issue in the book (e.g., typos or problems with the material), please don't hesitate to reach out. A good place to report such problems is the companion website's GitHub issues tab located at

<https://github.com/bgreenwell/treebook/issues>.

Even if it's a section of the material you found confusing or hard to understand, I want to hear about it!

---

## The `treemisc` package

Along with the companion website, there's also a companion R package, called `treemisc` [Greenwell, 2021c], that houses a number of the data sets and functions used throughout this book. Installation instructions and documentation can be found in the package's GitHub repository at

<https://github.com/bgreenwell/treemisc>.

---

## Colorblindness

This book contains many visuals in color. I have tried as much as possible to keep every figure colorblind friendly. For the most part, I use the Okabe-Ito color palette, designed by Masataka Okabe and Kei Ito (<https://jfly.uni-koeln.de/color/>), which is available in R ( $\geq 4.0.0$ ); see `?grDevices::palette.colors` for details. If you find any of the visuals hard to read (whether due to color blindness or not) please consider reporting it so that it can be corrected in the next available printing/version.

---

---

## Acknowledgments

I'm extremely grateful to Bradley Boehmke, who back in 2016 asked me to help him write "Hands-On Machine Learning with R" [Boehmke and Greenwell, 2020]. Without that experience, I would not have had the confidence (nor the skill or patience) to prepare this book on my own. Thank you, Brad.

Also, a huge thanks to Alex Gutman and Jay Cunningham, who both agreed to provide feedback on an earlier draft of this book. Their reviews and attention to detail have ultimately led to a much improved presentation of the material. Thank you both.

Lastly, I cannot express how much I owe to my wonderful wife Jennifer, and our three kids: Julia, Lillian, and Oliver. You help inspire all I do and keep me sane, and I truly appreciate you putting up with me while I worked on this book.

Brandon M. Greenwell



Taylor & Francis  
Taylor & Francis Group  
<http://taylorandfrancis.com>

# 1

---

## Introduction

---

“Here is a new game,” said Scrooge. “One half-hour, Spirit, only one!”

It was a game called Yes and No, where Scrooge’s nephew had to think of something, and the rest must find out what; he only answering to their questions yes or no, as the case was. The brisk fire of questioning to which he was exposed elicited from him that he was thinking of an animal, a live animal, rather a disagreeable animal, a savage animal, an animal that growled and grunted sometimes, and talked sometimes, and lived in London, and walked about the streets, and wasn’t made a show of, and wasn’t led by anybody, and didn’t live in a menagerie, and was never killed in a market, and was not a horse, or an ass, or a cow, or a bull, or a tiger, or a dog, or a pig, or a cat, or a bear. At every fresh question that was put to him, this nephew burst into a fresh roar of laughter; and was so inexpressibly tickled, that he was obliged to get up off the sofa, and stamp. At last the plump sister, falling into a similar state, cried out:

“I have found it out! I know what it is, Fred! I know what it is!”

“What is it?” cried Fred.

“It’s your uncle Scro-o-o-o-oge!”

Charles Dickens

A Christmas Carol

---

Ever play a game called *twenty questions*? If you have, then you should have no trouble understanding the basics of how decision trees work. A decision tree is essentially a set of sequential yes or no questions regarding the available

features in an attempt to make an accurate prediction (or classification). For example, “Is systolic blood pressure less than 120 mm Hg?” or “Does it have three leaves?”. The answer to each question determines the next follow-up question. For example, in trying to determine whether or not a particular plant in your backyard is poison ivy, if you answered yes to whether or not it has three leaves<sup>a</sup>, the next question you (or the decision tree) might ask is “Does it have notched leaves?”, and so forth. The overall idea is to ask a series of good and intelligent questions using the available data that will hopefully lead to an accurate prediction. What mostly differs between the various decision tree algorithms I’ll discuss in this book is the nature of the questions asked and how they’re determined given a set of training observations.

Compared to other nonparametric algorithms, there’s also a bit more transparency in how decision trees make predictions. For example, in classifying poison ivy, a decision tree might use a simple rule such as, “If it has three leaves AND and the leaves are notched, then it’s likely poison ivy.” Being able to interpret the output from a model is crucial in understanding how a model makes predictions and conveying the results to others.

But we don’t often just ask random questions, whether we’re playing twenty questions or trying to determine if a particular plant is poison ivy. Our inquiries tend to have a hierarchy, in that we often start with the most general questions that we think will narrow down the possibilities the most, then follow up with more refined questions to home in on the answer. Decision trees work in a similar way in that the first handful of questions tends to be the most important, while the questions further down the tree are just smaller refinements to further improve accuracy.

I’ll return to talking about trees in [Section 1.2](#). Next, I’ll turn the discussion to some basic (but important) ideas in statistical and machine learning.

---

## 1.1 Select topics in statistical and machine learning

This section is intended to provide a (very) brief overview of a handful of topics from statistical and machine learning that will be useful to know for some of the material to come. Select topics, like the *bias-variance tradeoff* and *right censoring*, are important to several areas and examples in this book, and so I’ll spend the next few sections highlighting some of these important ideas. This is by no means intended to act as a primer, or even just a basic introduction to statistical and machine learning in general, but rather to highlight key topics

<sup>a</sup>“Leaves of three, let it be.” is a common rule of thumb for avoiding contact with poison ivy.

that will help introduce more advanced topics later in the book. This book does assume, however, that readers have at least some general background or exposure to common topics in statistics and machine learning (like hypothesis testing, cross-validation, and hyperparameter tuning). If you’re looking for a more thorough overview of statistical and machine learning, I’d suggest starting with James et al. [2021]. For a deeper dive, go with Hastie et al. [2009]. Both books are freely available for download, if you choose not to purchase a hard copy. Harrell [2015] and Matloff [2017], while more statistical in nature, offer valuable takes on several concepts fundamental to statistical and machine learning, and I highly recommend each.

### 1.1.1 Statistical jargon and conventions

To start, let’s introduce ourselves to some of the notation used throughout the book; additional mathematical notation will be introduced when necessary in the chapters that follow. Since this book is primarily concerned with fitting tree-based models for prediction and description, I’ll often be talking about independent variables (what we use to predict) and dependent variables (what we want to predict). The independent variables are referred to as either features or predictors (maybe even as covariates<sup>b</sup> or regressors at one point or another). The dependent variable is referred to as the response, target, or outcome variable. Generic features are denoted by  $x$  or  $x_1, x_2$ , and so on, and the response using  $y$ . In most cases, bold symbols typically refer to matrices (usually uppercase Latin letters) or column vectors (usually lowercase Latin letters). For example,  $\mathbf{X}$  typically represents an  $N \times p$  matrix of  $p$  features from a data set with  $N$  rows (or observations/records);  $\mathbf{x}_i$  denotes the  $i$ -th row of  $\mathbf{X}$ , whereas  $\mathbf{x}$  (or sometimes  $\mathbf{x}_0$ ) refers to the  $p$  predictor values of a (single) generic observation.

As far as variable types go, this book is mainly concerned with three:

- Nominal categorical (i.e., categorical where the order of categories doesn’t matter). Examples include gender, eye color, zip code, or blood type.
- Ordered categorical (i.e., categorical where the order of categories matters). Examples include socioeconomic status (e.g., low < middle < high), age range (e.g., [0-10yrs.] < [11-20yrs.] < ...), or satisfaction rating (e.g., not satisfied < somewhat satisfied < very satisfied). Ordered categorical variables are sometimes referred to as ordinal.
- Ordered numeric. Examples include age or temperature measured on a continuum, height, weight, or concentration.

---

<sup>b</sup>Technically, a covariate refers to a predictor that we think is related to the response, but whose effect is not of direct interest (e.g., we may not care to interpret its effect, but we include it to improve the overall model; think *analysis of covariance*).

I'll often refer to ordered numeric variables as either numeric or continuous, and both ordered categorical and numeric variables collectively as ordered variables; note that ordered categorical variables can arbitrarily be mapped to integers as long as the original ordering is preserved (for example, low < med < high  $\rightarrow$  1 < 2 < 3).

Many tree-based algorithms only make the distinction between ordered and nominal variables. Categorical variables, whether ordered or nominal, will often be referred to as *factors* (e.g., temperature was recorded as an ordered factor with levels: freezing < cold < warm < hot). There's also the concept of *censored variables* (usually the response), but I'll defer discussion of censored outcomes to the example in [Section 1.4.9](#).

The *learning sample*, also called the *training data*, is often denoted by  $\mathbf{d}_{trn} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ , where  $N$  is the sample size, and  $\mathbf{x}_i$  is the  $i$ -th row of training features (e.g.,  $x_{1i}, x_{2i}, \dots$ ).

On the rare occasion where I'm referring to a *random variable*, I'll typically use an upper case Latin letter (e.g.,  $Y$  or  $X$ ) or the lowercase Greek letter  $\epsilon$ ; other Greek letters, like  $\beta$  or  $\theta$ , will generally represent the fixed, but unknown parameters of a model. The operators  $E$  and  $V$  will denote the expected value (i.e., mean) and variance of a random variable, respectively. Several probability distributions are also used throughout this book and are denoted using a mix of capital letters (sometimes in a calligraphic font) and Greek letters, for example:

- $\mathcal{U}(0, 1)$  represents a continuous uniform distribution over the interval  $[0, 1]$ .
- $N(\mu, \sigma)$  represents a normal (or Gaussian) distribution with mean  $\mu$  and standard deviation  $\sigma$ .
- $\chi_\nu^2$  represents a chi-squared distribution with  $\nu$  degrees of freedom.

To say that  $X_1, X_2, \dots, X_p$  are a random sample from some arbitrary distribution  $\mathcal{D}$ , parameterized by  $\boldsymbol{\theta}$ , I'll write  $\{X_i\}_{i=1}^N \stackrel{iid}{\sim} \mathcal{D}(\boldsymbol{\theta})$ ; the *iid* stands for *independently and identically distributed*.

Whenever possible, I try to emphasize words and terms you may be unfamiliar with using *italicized text*, and I encourage you to “google” them for more details, but not knowing them should not distract you from the fundamental ideas presented throughout this book.

### 1.1.2 Supervised learning

Supervised learning can often be thought of as an exercise in *function approximation*. For simplicity, we often assume that the response variable,  $Y$ , is

related to a set of predictors,  $\mathbf{x}$ , through a model with additive error:

$$Y = f(\mathbf{x}) + \epsilon, \quad (1.1)$$

where  $\epsilon$  is a random variable with mean zero (i.e.,  $E(\epsilon) = 0$ ) and is assumed to be independent of  $\mathbf{x}$ . Note that the response is also a random variable here since it is a function of  $\epsilon$ . The function  $f(\mathbf{x})$  is fixed and represents the *systematic* part of the relationship between  $Y$  and  $\mathbf{x}$ . As is almost always the case, the true relationship between  $Y$  and  $\mathbf{x}$  is often statistical in nature (i.e., not deterministic) and the additive error helps to capture the non-deterministic aspect of this relationship (e.g., unobserved predictors, measurement error, etc.).

Since we assume  $E(\epsilon) = 0$ , it turns out that  $f(\mathbf{x})$  can be viewed as a *conditional expectation*:

$$E(Y|\mathbf{x}) = f(\mathbf{x}),$$

where we can interpret  $f(\mathbf{x})$  as the mean response for all observations with predictor values equal to  $\mathbf{x}$ . In the case of *J-class classification* ([Section 1.1.2.3](#)), we can still view  $f(\mathbf{x})$  as a conditional mean; in this case it's the conditional proportion corresponding to a particular class:  $E(Y = j|\mathbf{x})$ , which can be interpreted as an estimate of  $\Pr(Y = j|\mathbf{x})$ —the probability that  $Y$  belongs to class  $j$  given a particular set of predictor values  $\mathbf{x}$ . In this sense, class probability estimation is really a regression problem.

The term “supervised” in supervised learning refers to the fact that we use labeled training data<sup>c</sup>  $\{y_i, \mathbf{x}_i\}_{i=1}^N$  and an algorithm to learn a reasonable mapping between the observed response values,  $y_i$ , and a set of predictor values,  $\mathbf{x}_i$ . Without a labeled response column, the task would be *unsupervised* ([Section 1.1.3](#)), and the analytic goal would be different.

An estimate  $\hat{f}(\mathbf{x})$  of  $f(\mathbf{x})$  can be used for either *description* or *prediction* (or both). I'll briefly discuss the meaning of each in turn next.

### 1.1.2.1 Description

In supervised learning, descriptive tasks are often concerned with determining which features have the most impact on  $\hat{f}(\mathbf{x})$  and how. For example, in supervised learning problems, we are often interested in determining

- which predictors are the most “important” for prediction (feature importance);
- the marginal impact of each predictor (or a subset of the important ones) on the predicted outcome (feature effects).

---

<sup>c</sup>The labels here are provided by the response values  $\{y_i\}_{i=1}^N$ .

For example, in the Ames housing data ([Section 1.4.7](#)), we may be interested in determining which predictors are most influential on the predicted sale price in a fitted model. We may also be interested in how a particular feature (e.g., overall house size) functionally relates to the predicted sale price from a fitted model.

Questions like these are relatively straightforward to glean from simpler models, like an *additive linear model* or a simple decision tree. However, this type of information is often hidden in more complicated nonparametric models—like *neural networks* (NNs) and *support vector machines* (SVMs)—which unfortunately has given rise to the term “black box” models. In [Chapter 6](#), I’ll look at several model-agnostic techniques that can be helpful in extracting relevant descriptive information from any supervised learning model.

### 1.1.2.2 Prediction

As the name implies, prediction tasks are concerned with predicting future or unobserved outcomes. For example, we may be interested in predicting the sale price for a new home given a set of relevant features. This could, in theory, be a useful starting point in setting the listing price for a home, or trying to help infer whether or not a particular house is under- or over-valued. Great care must be taken in such problems, however, as the outcome variable (the sale price of homes, in this case) can be complex in nature and the available data may not be enough to adequately capture sudden changes in the distribution of future response values; a bit more on this in [Section 1.4.7](#).

It should be stressed that prediction and description often go hand in hand. Description helps provide transparency in how a model’s predictions are generated. Transparency helps reveal potential issues and biases and therefore can help increase trust or distrust in a model’s predictions. Would you feel comfortable putting a model into production if you did not have some understanding as to how different subsets of features contribute to the model’s predictions?

Single decision trees, while often great descriptors, seldom make for good predictors, at least when compared to more contemporary techniques. Nonetheless, as we’ll see in [Part I](#) of this book, single decision trees are sometimes the right tool for the job, but it just so happens that more accurate decision trees tend to be harder to interpret. This is especially true for the decision trees discussed in [Chapter 4](#), which are flexible enough to achieve good performance, but often pay a price in interpretability.

### 1.1.2.3 Classification vs. regression

Supervised learning tasks generally fall into one of two categories: *classification* or *regression*. Regression is used in a very general sense here, and often refers to any supervised learning task with an ordered outcome. Examples of ordered outcomes might be sale price or wine quality on a scale of 0–10 (essentially, an ordered category).

In classification, the response is categorical and the objective is to “classify” new observations into one of  $J$  possible categories. In the mushroom classification example (Section 1.4.4), for instance, the goal is to classify new mushrooms as either edible or poisonous on the basis of simple observational attributes about each (like the color and odor of each mushroom). In this example,  $J = 2$  (edible or poisonous) and the task is one of *binary classification*. When  $J > 2$ , the task is referred to as *multiclass classification*.

### 1.1.2.4 Discrimination vs. prediction

Pure classification is almost never the goal, as we are usually not interested in directly classifying observations into one of  $J$  categories. Instead, interest often lies in estimating the conditional probability of class membership. That is to say, it is often far more informative to estimate  $\{\Pr(Y = j|\mathbf{x}_0)\}_{j=1}^J$  as opposed to predicting the class membership of some observation  $\mathbf{x}_0$ . Even when the term “classification” is used, the underlying goal is usually that of estimating class membership probabilities conditional on the feature values<sup>d</sup>.

Frank Harrell, a prominent biostatistician, couldn’t have said it better:

---

It is important to distinguish prediction and classification. In many decision making contexts, classification represents a premature decision, because classification combines prediction and decision making and usurps the decision maker in specifying costs of wrong decisions. The classification rule must be reformulated if costs/utilities or sampling criteria change. Predictions are separate from decisions and can be used by any decision maker.

Classification is best used with non-stochastic/deterministic outcomes that occur frequently, and not when two individuals with identical inputs can easily have different outcomes. For the latter, modeling tendencies (i.e., probabilities) is key.

---

<sup>d</sup>The term “classification” is actually abused quite often in practice (and in this book), as it is often used in situations where the true goal is class probability estimation.

Classification should be used when outcomes are distinct and predictors are strong enough to provide, for all subjects, a probability near 1.0 for one of the outcomes.

Frank Harrell

<https://www.fharrell.com/post/classification/>

---

A related issue, known as *class imbalance* is discussed in [Section 7.9.4](#).

### 1.1.2.5 The bias-variance tradeoff

The terms *overfitting* and *underfitting* are used throughout this book (more so the former), but what do they mean? Overfitting occurs when your model is too complex, and has gone past any signal in the data and is starting to fit the noise. Underfitting, on the other hand, refers to when a model is too simple and does not adequately capture any of the signal in the data. In both cases, the model will not generalize well to new data.

A model that is overfitting the learning sample often exhibits lower bias but has higher variance when compared to a model that is underfitting, which often exhibits higher bias but lower variance. This tradeoff is more specifically referred to as the bias-variance tradeoff. Excellent discussions of this topic can be found in Matloff [2017, Sec. 1.11] and Hastie et al. [2009, [Sec. 7.2–7.3](#)]; the latter provides more of a theoretical view.

For the additive error model (1.1) with constant variance  $\sigma^2$ , Hastie et al. [2009] show that the mean square prediction error for an arbitrary observation  $\mathbf{x}_0$  can be decomposed into

$$\mathrm{E} \left[ \left( Y - \hat{f}(\mathbf{x}_0) \right)^2 | \mathbf{x} = \mathbf{x}_0 \right] = \sigma^2 + \text{Bias}^2 \left[ \hat{f}(\mathbf{x}_0) \right] + \text{V} \left[ \hat{f}(\mathbf{x}_0) \right],$$

where  $\sigma^2$  represents irreducible error that we cannot do anything about, regardless of how well we estimate  $f(\mathbf{x}_0)$ . Normally, increasing the complexity of  $\hat{f}$  will cause the squared bias (middle term) to decrease and the variance (last term) to increase, and vice versa.

For illustration, consider the data displayed in [Figure 1.1](#), which consists of a random sample of  $N = 100$  observations from a simple quadratic model with additive Gaussian noise:

$$Y = 1 + 0.5X^2 + \epsilon,$$

where  $X \sim \mathcal{U}(0, 1)$ , and  $\epsilon \sim \mathcal{N}(0, \sigma = 0.1)$ . [Figure 1.1](#) also shows the fitted mean response from three linear models: a simple linear regression model

(left), a polynomial model polynomial regression of degree two (middle), and a polynomial model polynomial regression of degree 20. Clearly, the simple linear and 20-th degree polynomial models polynomial regression are underfitting and overfitting, respectively, while the quadratic model (the correct fit) provides the best tradeoff.

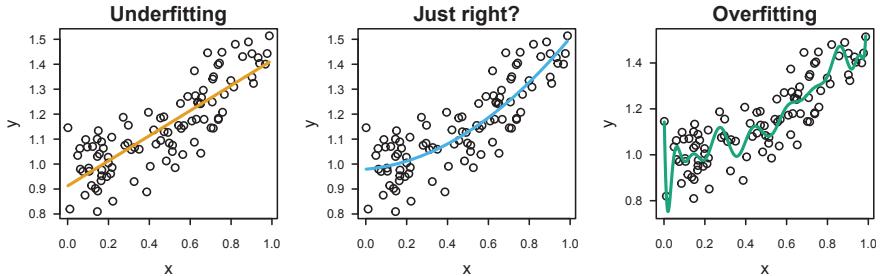


FIGURE 1.1: Fitted mean response from three linear models applied to the quadratic example. Left: a simple linear model (i.e., degree one polynomial). Middle: a quadratic model (i.e., the correct model). Right: a 20-th degree polynomial model.

The same tradeoff applies equally to classification as well. In probabilistic classification, for example, the MSE between the true and predicted class probabilities can also be decomposed into parts due to irreducible error, squared bias, and variance; see Manning et al. [2008, pp. 308–314] for details.

To visually illustrate the bias-variance tradeoff for classification, Figure 1.2 shows  $N = 500$  observations generated from the simple “twonorm” benchmark problem; see `?mlbench::mlbench.twonorm` in R for details and references. Here, the classes correspond to two bivariate normal distributions with mean vectors  $\mu_1 = (\sqrt{2}, \sqrt{2})^\top$  (yellow points) and  $\mu_2 = -\mu_1$  (blue points) and unit covariance matrix (i.e.,  $\Sigma = I_2$ ). Figure 1.2 also shows two different *decision boundaries*<sup>e</sup>. The dashed line corresponds to the optimal or *Bayes decision boundary* (i.e., the best we can do in this problem), which in this case is linear. The second decision boundary (solid line) corresponds to a simple *k-nearest neighbor* (*k*-NN) classifier with  $k = 1$ <sup>f</sup>. Clearly, the 1-NN model is overfitting and produces an overly complex decision boundary (e.g., the three little islands) compared to the optimal linear boundary.

<sup>e</sup>In two dimensions, the decision boundary from a classifier is the boundary that separates the predictor space into disjoint sets, one for each class. They can be useful for understanding and comparing the flexibility and performance of different classifiers.

<sup>f</sup>A 1-NN model classifies a new observation according to the class of its nearest neighbor in the learning sample; in this case, “nearest neighbor” is defined as the closest observation in the training set as measured by the Euclidean distance.

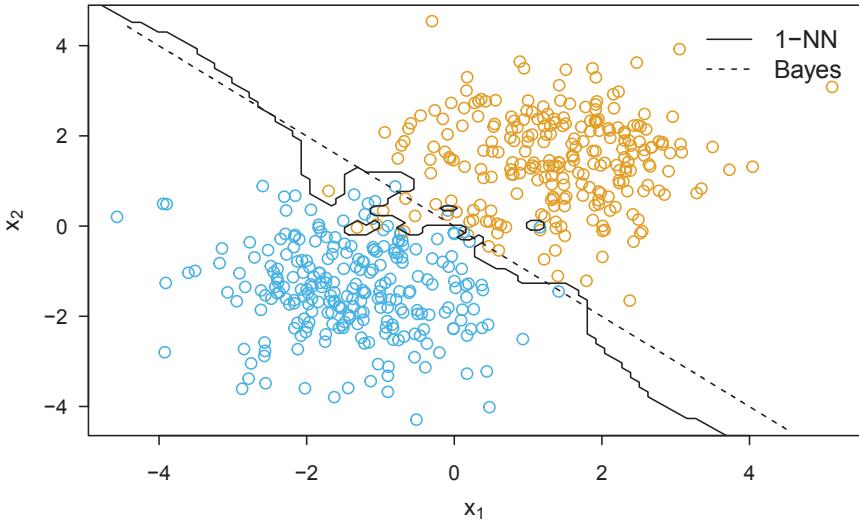


FIGURE 1.2: Decision boundaries for the twonorm benchmark example. In this case, the Bayes decision boundary is linear (dashed line). Also shown is the decision boundary from a 1-NN model (solid line), which is clearly overfitting; notice the three little islands.

### 1.1.3 Unsupervised learning

In *unsupervised learning*, there is no working response  $y$ ; hence, the learning sample comprises just features  $\{\mathbf{x}_i\}_{i=1}^N$ . The general goal is to determine whether not the data form any “interesting groups;” for example, whether or not the data form natural *clusters*. A useful application is in detecting *outliers* or *anomalies*. An example of using decision trees for the purpose of anomaly detection is given in Section 7.8.5. Another example, related to the identification of potentially mislabeled response values, is given in Section 7.6.1.

## 1.2 Why trees?

There are a number of great modeling tools available to data scientists. But what makes a modeling tool good? Is being able to achieve competitive accuracy all that matters? Of course not. According to the late Leo Breiman, a good modeling tool, at a minimum, should:

- be applicable to both classification (binary and multiclass) and regression;
- be competitive in terms of accuracy;

- be capable of handling large data sets;
- handle missing values effectively.

Additionally, Leo also believed that good modeling tools should be able to:

- tell you which predictors are important;
- tell you how the predictors functionally relate to the response;
- tell you if and how the predictors interact;
- tell you how the data cluster;
- tell you if there are any novel cases or outliers.

As we'll see throughout this book, tree-based methods naturally check a lot of these boxes. While individual trees are particularly capable of handling a wide range of problems, their main disadvantage is that they don't often perform as well compared to more complex models, like NNs or SVMs; however, in **Part II** of this book, I'll discuss powerful ways to combine decision trees into *ensembles* that are quite competitive with current state-of-the-art algorithms, while still adhering to several of the features of a good modeling tool outlined above.

**Table 1.1** compares several characteristics of different statistical learning algorithms; this is a modified recreation of Table 10.1 from Hastie et al. [2009, p. 351]. MARS, which stands for *multivariate adaptive regression splines* [Friedman, 1991], can be thought of as an extension of *linear models* that automatically handles variable selection, nonlinear relationships, and interactions between predictors<sup>g</sup>; MARSis discussed in the online supplementary material on the book website. Note how tree-based methods tend to have a lot of desirable properties. The idea of tree-based ensembles is hopefully to retain many of these useful properties while improving predictive performance.

**Table 1.1** is not in full agreement with Table 10.1 from Hastie et al. [2009, p. 351], on which it is based. For example, individual trees receive a ● for predictive performance in **Table 1.1**, indicating that they can be fairly accurate in certain problems, especially modern decision tree algorithms, like GUIDE ([Chapter 4](#)).

Trees also make great *off-the-shelf* modeling tools. The term off-the-shelf, at least as it's used here, implies a procedure that can be usefully applied to a wide range of data sets without requiring much in the way of pre-processing

---

<sup>g</sup>The terms “MARS” is actually trademarked and licensed to Salford Systems (which was acquired by Minitab in 2017), hence, open source implementations often go by different names. For example, a popular open source implementation of MARSis the fantastic **earth** package [Milborrow, 2021a] in R.

TABLE 1.1: Comparison of several characteristics between different statistical learning algorithms. This is a modified recreation of Table 10.1 from Hastie et al. [2009, p. 351]; here, TBEs stands for “tree-based ensembles”,  $\text{✗}$  means “bad”,  $\checkmark$  means “good”, and  $\text{🟡}$  means “so-so”.

Characteristic	NNs	SVMs	MARS	Trees	TBEs
Can naturally handle data of mixed type	$\text{✗}$	$\text{✗}$	$\checkmark$	$\checkmark$	$\checkmark$
Can naturally handle missing values	$\text{✗}$	$\text{✗}$	$\checkmark$	$\checkmark$	$\checkmark$
Robust to outliers in the predictors	$\text{✗}$	$\text{✗}$	$\text{✗}$	$\checkmark$	$\checkmark$
Insensitive to monotone transformations of the predictors	$\text{✗}$	$\text{✗}$	$\text{✗}$	$\checkmark$	$\checkmark$
Scales well with large $N$	$\text{✗}$	$\text{✗}$	$\checkmark$	$\checkmark$	$\checkmark$
Ability to deal with irrelevant features	$\text{✗}$	$\text{✗}$	$\checkmark$	$\checkmark$	$\checkmark$
Ability to extract linear combinations of the predictors	$\checkmark$	$\checkmark$	$\text{✗}$	$\text{✗}$	$\text{✗}$
Descriptive power (interpretability)	$\text{✗}$	$\text{✗}$	$\text{🟡}$	$\text{🟡}$	$\text{🟡}$
Predictive power	$\checkmark$	$\checkmark$	$\text{🟡}$	$\text{🟡}$	$\checkmark$

or careful tuning. You can generally just hit “Run” and get something useful. THAT IS NOT TO SAY THAT YOU SHOULD NOT PUT TIME AND EFFORT INTO CLEANING UP YOUR DATA AND CAREFULLY TUNING THESE MODELS. Rather, trees can work seamlessly in rather messy data situations (e.g., outliers, missing values, skewness, mixed data types, etc.) without requiring the level of pre-processing necessary for other algorithms to “just work” (e.g., neural networks). For example, even if I’m not using a decision tree for the final model, I will often use it as a first pass as it can give me a quick and dirty picture of the data, and any serious issues (which can easily be missed in the exploratory phase) will often be highlighted (e.g., extreme *target leakage* or accidentally leaving in an ID column). In other words, trees make great exploratory tools, especially when dealing with potentially messy data.

### 1.2.1 A brief history of decision trees

Decision trees have a long and rich history in both statistics and computer science, and have been around for many decades. However, decision trees arguably got their true start in the social sciences. Motivated by the need

for finding interaction effects in complex survey data, Morgan and Sonquist [1963] developed and published the first decision tree algorithm for regression called *automatic interaction and detection* (AID). Starting at the root node, AID recursively partitions the data into two homogeneous subgroups, called *child nodes*, by maximizing the between-node sum of squares, similar to the process described in [Section 2.3](#). AID continues successively bisecting each resulting child node until the reduction in the within-node sum of squares is less than some prespecified threshold.

Messenger and Mandell [1972] extended AID to classification in their *theta automatic interaction detection* (THAID) algorithm. The *theta criterion* used in THAID to choose splits maximizes the sum of the number of observations in each modal category.

The *chi-square automatic interaction detection* (CHAID) algorithm, introduced in Kass [1980], improved upon AID by countering some of its initial criticisms; CHAID was originally developed for classification and later extended to also handle regression problems. Similar to the decision tree algorithms discussed in [Chapters 3–4](#), CHAID employs statistical tests and stopping rules to select the splitting variables and split points. In particular, CHAID relies on chi-squared tests, which require discretizing ordered variables into bins. Compared to AID and THAID, CHAID was unique in that it allowed *multiway splits* (which typically require larger sample sizes, otherwise the child nodes can become too small rather quickly) and included a separate category for missing values.

Despite the novelty of AID, THAID, and CHAID, it wasn't until Breiman et al. [1984] introduced the more general *classification and regression tree* (CART) algorithm<sup>h</sup>, that tree-based algorithms started to catch on in the statistical community. CART-like decision trees are the topic of [Chapter 2](#). A similar tree-based algorithm, called C4.5 [Quinlan, 1993], which evolved into the current C5.0 algorithm<sup>i</sup>, has become very similar to CART in many regards; hence, I focus on CART in this book and discuss the details of C4.5/C5.0 in the online supplementary material.

CART helped generate renewed interest in partitioning methods, and we've seen that evolution unfold over the last several decades. While the history is rife with advancements, the first part of this book will focus on three of the most important tree-based algorithms:

**Chapter 2:** Classification and regression trees (CART).

**Chapter 3:** Conditional inference trees (CTree).

---

<sup>h</sup>Like MARS, the term “CART” is also trademarked and licensed to Salford Systems, which is now part of Minitab.

<sup>i</sup>C50 is now open source and available in the R package **C50** [Kuhn and Quinlan, 2021].

**Chapter 4:** Generalized, unbiased, interaction detection, and estimation (GUIDE).

GUIDE itself has a rather long and interesting history, and evolved out of several earlier well-known tree algorithms—like QUEST [Loh and Shih, 1997] and CRUISE [Kim and Loh, 2001]<sup>j</sup>. If you’re interested in a more thorough overview of the history of tree-based algorithms, I highly encourage you to read Loh [2014].

### 1.2.2 The anatomy of a simple decision tree

In this section, I’ll look at the basic parts of a typical decision tree (perhaps tree topology would’ve been a cooler section header).

A typical (binary) decision tree is displayed in [Figure 1.3](#). The tree is made up of *nodes* and *branches*; the path between two consecutive branches is called an *edge*. The nodes are the points at which a branch occurs. Here, we have three *internal nodes*, labeled  $\mathcal{I}_1$ ,  $\mathcal{I}_2$ , and  $\mathcal{I}_3$ , and five terminal (or leaf) nodes, labeled  $\mathcal{L}_1$ ,  $\mathcal{L}_2$ ,  $\mathcal{L}_3$ ,  $\mathcal{L}_4$ , and  $\mathcal{L}_5$ . The tree is binary because it only uses two-way splits; that is, at each node, a split results in only two branches, labeled “Yes” and “No”<sup>k</sup>. The path taken at each internal node depends on whether or not the corresponding split condition is satisfied. For example, an observation with  $x_1 = 0.33$  and  $x_5 = 1.19$  would find itself in terminal node  $\mathcal{L}_2$ , regardless of the values of the other predictors.

The split conditions (or just splits) for an ordered predictor  $x$  have the form  $x < c$  vs.  $x \geq c$ , where  $c$  is in the domain of  $x$  (typically the midpoint between two consecutive  $x$  values in the learning sample); note that the same type of splits are used for ordered factors since we just need to preserve the natural ordering of the categories (e.g.,  $x <$  medium vs.  $x \geq$  medium)<sup>1</sup>. Splits on (unordered) categorical variables have the form  $x \in S_1$  vs.  $x \in S_2$ , where  $S = S_1 \cup S_2$  is the full set of unique categories in  $x$ .

Each tree begins with a *root node* containing the entire learning sample. Starting with the root node, the training data are split into two non-overlapping groups, one going left, and the other right, depending on whether or not the first split condition is satisfied by each observation. The process is repeated on each subgroup (or child node) until each observation reaches a terminal node.

<sup>j</sup>For those interested, QUEST stands for *quick, unbiased, and efficient statistical tree* and CRUISE stands for *classification rule with unbiased interaction selection and estimation*.

<sup>k</sup>Some decision tree algorithms allow multiway (i.e.,  $> 2$ ) splits, but none are really discussed in this book.

<sup>1</sup>For splits on ordered variables, it generally doesn’t matter whether the left branches in a tree correspond to  $x < c$  or  $x \geq c$ , as long as you’re consistent.

At each node in a tree, we can compute several quantities that may be of interest, for example:

- the number (or fraction) of observations from the learning sample in that node;
- for classification models, the number of observations classified correctly (or incorrectly) in that node (as determined by the majority class in that node).
- for probabilistic models, the proportion of observations in that node belonging to each class;
- the current fitted or predicted value if this were a leaf or terminal node (e.g., the mean response or class proportions).

The terminal nodes themselves are usually characterized by a statistical summary of the training response values in each, like the sample mean for regression, a frequency table for classification, or the Kaplan-Meier estimator/curve for censored outcomes (see the example in [Section 1.4.9](#)). These summaries can be used to produce fitted values and predictions for new observations<sup>m</sup>. For example, if the response is continuous (i.e., a regression tree), then new observations that occupy a terminal node might be assigned a predicted value equal to the mean response of the training observations defining that node.

### 1.2.2.1 Example: survival on the Titanic

To further illustrate, let's look at the tree diagram in [Figure 1.4](#). This CART-like decision tree was constructed using the well-known Titanic data set and is trying to separate passengers who survived from those who didn't using readily available information about each. The data, which I'll revisit in [Section 7.9.3](#), contain  $N = 1,309$  observations (i.e., passengers) on the following six variables:

- **survived**: binary indicator of passenger survival (the response);
- **pclass**: integer specifying passenger class (i.e., 1–3);
- **age**: passenger age in years;
- **sex**: factor giving the sex of each passenger (i.e., male/female);
- **sibsp**: integer specifying the number of siblings/spouses aboard;
- **parch**: integer specifying the number of parents/children aboard.

The variable **pclass** is commonly treated as nominal categorical, but here the natural ordering has been taken into account. The tree split the passengers into six relatively homogeneous groups (terminal nodes) based on four of the above

---

<sup>m</sup>Fitted values are just the predicted values for each observation in the learning sample.

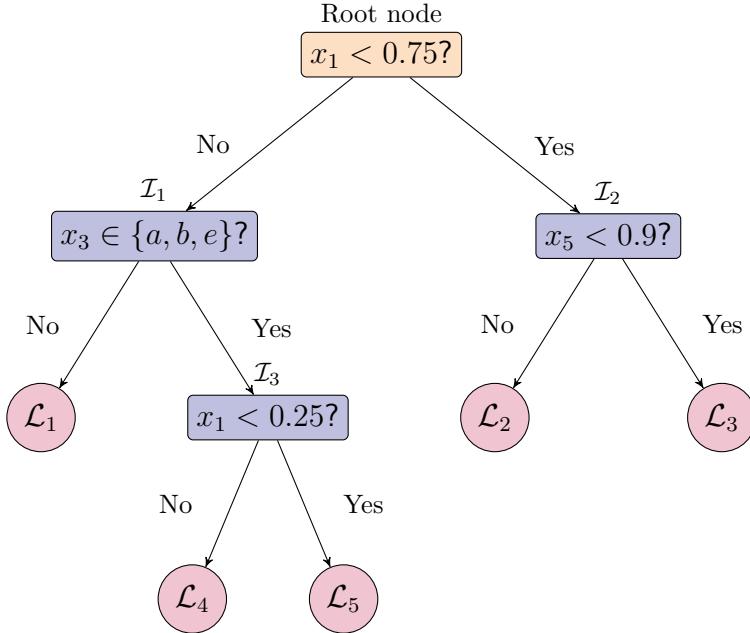


FIGURE 1.3: A basic (binary) decision tree with four splits.

five available features; `parch` is the only variable not selected to partition the data. The terminal nodes (nodes 6–11) each contain a node summary giving the proportion of surviving passengers in that node. As we’ll see in later chapters, these proportions can be used as class probability estimates.

For example, the tree diagram estimates that first and second class female passengers had a 93% chance of survival; the percentage displayed in the bottom of each node corresponds to the fraction of training observations used to define that node. Given what you know about the ill-fated Titanic, does the tree diagram make sense to you? Does it appear that women and children were given priority and had a higher chance of survival (i.e., “women and children first”)? Perhaps, unless you were a third-class passenger.

In [Part I](#) of this book, I’ll look at how several popular decision tree algorithms choose which variables to split on (splitters) and how each split condition is determined (e.g., `age < 9.5`). [Part II](#) of this book will then look at how to improve the accuracy and generalization performance of a single tree by combining several hundred or thousand individual trees together.

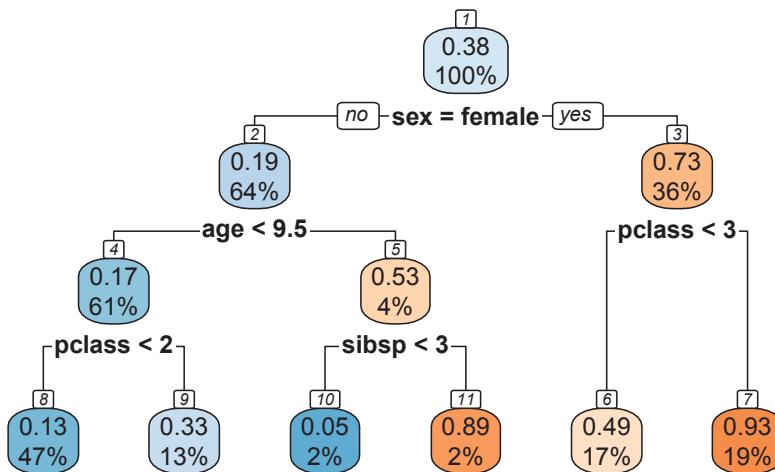


FIGURE 1.4: Decision tree diagram summarizing survival probability aboard the ill-fated Titanic.

---

## 1.3 Why R?

Why not?

### 1.3.1 No really, why R?

I grew up on R (and SAS), but I've chosen it for this book primarily for one reason: it currently provides the best support and access to a wide range of tree-based methods (both classic and modern-day)<sup>n</sup>. For example, I'm not aware of any non-R open source implementation of tree-based methods that provides full support for nominal categorical variables without requiring them to be encoded numerically<sup>o</sup>. Another good example is *conditional inference trees* [Hothorn et al., 2006c], the topic of [Chapter 3](#), which are only implemented in R (as far as I'm aware). Nonetheless, there is at least one Python example included in this book!

<sup>n</sup>This book was also written in R (and LaTeX) using the wonderful **knitr** package Xie [2021].

<sup>o</sup>There's been some progress in areas of scikit-learn and other open source software that will be discussed in [Chapter 8](#).

While I have a strong appreciation for the power of base R (i.e., the core language), I'm extremely appreciative of, and often rely on, the amazing ecosystem of contributed packages. Nonetheless, I've decided to keep the use of external packages to a minimum (aside from those packages related to the core tree-based methods discussed in this book), and instead rely on vanilla R programming as much as possible. This choice was made for several (highly opinionated) reasons:

- it will make the book easier to maintain going forward, as the code examples will (hopefully) continue to work for many years to come without much modification;
- using standard R programming constructs (e.g., `for` loops and their apply-style functional replacements, like `lapply()`) will make the material easier to comprehend for non-R programmers, and easier to translate to other open source languages, like Julia and Python;
- it emphasizes the basic concepts of the methods being introduced, rather than focusing on current best coding practices and cool packages, of which there are many. (Please don't send me hate mail for using `sapply()` instead of `vapply()` or the family of map functions available in `purrr` [Henry and Wickham, 2020].)

Note that I've tried to be as aggressive as possible in terms of commenting upon the various code snippets scattered throughout this book and the online supplementary material. You should pay careful attention to these comments as they often link a particular line or section of code to a specific step in an algorithm, try to explain a hacky approach I'm using, and so on.

Each chapter includes one or more software sections, which highlight both R and non-R implementations of the relevant algorithms under discussion. Additionally, each chapter also contains R-specific software example sections (usually at the end of each chapter), which demonstrate use of relevant tree-specific software on actual data (either simulated or real).

There are many great resources for learning R, but I would argue that the online manual, "An Introduction to R", which can be found at

<https://cran.r-project.org/doc/manuals/r-release/R-intro.html>,

is a great place to start. Book-wise, I think that Matloff [2011] and Wickham [2019] are some of the best resources for learning R (note that the latter is freely available to read online). If you're interested in a more hands-on introduction to statistical and machine learning with R, I'll happily self-promote Boehmke and Greenwell [2020].

### 1.3.2 Software information and conventions

Each chapter contains at least one software section, which points to relevant implementations of the ideas discussed in the corresponding chapter. And while this book focuses on R, these sections will also allude to additional implementations in other open source programming languages, like Python and Julia. Furthermore, several code snippets are contained throughout the book to help solidify certain concepts (mostly in R).

Be warned, I occasionally use dots in variable and function names—old R programming habits die hard. Package names are in bold text (e.g., **rpart**), inline code and function names are in typewriter font (e.g., `sapply()`), and file names are in sans serif font (e.g., `path/to/filename.txt`). In situations where it may not be obvious which package a function belongs to, I'll use the notation `foo::bar()`, where `bar()` is the name of a function in package `foo`.

I often allude to the documentation and help pages for specific R functions. For example, you can view the documentation for function `foo()` in package `bar` by typing `?foo::bar` or `help("foo", package = "bar")` at the R console. It's a good idea to read these help pages as they will often provide more useful details, further references, and example usage. For base R functions—that is, functions available in R's `base` package—I omit the package name (e.g., `?kronecker`). I also make heavy use of R's `apply()`-family of functions throughout the book, often for brevity and to avoid longer code snippets based on `for` loops. If you're unfamiliar with these, I encourage you to start with the help pages for both `apply()` and `lapply()`.

R package vignettes (when available) often provide more in-depth details on specific functionality available in a particular package. You can browse any available vignettes for a CRAN package, say `foo`, by visiting the package's homepage on CRAN at

<https://cran.r-project.org/package=foo>.

You can also use the `utils` package to view package vignettes during an active R session. For example, the vignettes accompanying the R package `rpart` [Therneau and Atkinson, 2019], which is heavily used in Chapter 2, can be found at <https://CRAN.R-project.org/package=rpart> or by typing `utils::vignette("bar", package = "foo")` at the R console.

There's a myriad of R packages available for fitting tree-based models, and this book only covers a handful. If you're not familiar with CRAN's task views, you should be. They provide useful guidance on which packages on CRAN are relevant to a certain topic (e.g., machine learning). The task view on statistical and machine learning, for example, which can be found at

<https://cran.r-project.org/web/views/MachineLearning.html>,

lists several R packages useful for fitting tree-based models across a wide variety of situations. For instance, it lists **RWeka** [Hornik, 2021] as providing an

open source interface to the J4.8-variant of C4.5 and M5 (see the online supplementary material on the book website). A brief description of all available task views can be found at <https://cran.r-project.org/web/views/>.

Keep in mind that the focus of this book is to help you build a deeper understanding of tree-based methods, it is not a programming book. Nonetheless, writing, running, and experimenting with code is one of the best ways to learn this subject, in my opinion.

This book uses a couple of graphical parameters and themes for plotting that are set behind the scene. So don't fret if your plots don't look exactly the same when running the code. This book uses a mix of base R and **ggplot2** [Wickham et al., 2021a] graphics, though, I think there's a **lattice** [Sarkar, 2021] graphic or two floating around somewhere. For **ggplot2**-based graphics, I use the `theme_bw()` theme, which can be set at the top level (i.e., for all plots) using `theme_set(theme_bw())`. Most of the base R graphics in this book use the following `par()` settings (see `?graphics::par` for details on each argument):

```
par(
  mar = c(4, 4, 0.1, 0.1), # may be different for a handful of figures
  cex.lab = 0.95,
  cex.axis = 0.8,
  mgp = c(2, 0.7, 0),
  tcl = -0.3,
  las = 1
)
```

Some of the base R graphics in this book use a slightly different setting for the `mar` argument (e.g., to make room for plots that also have a top axis, like Figure 8.12 on page 349).

---

## 1.4 Some example data sets

The examples in this book make use of several data sets, both real and simulated, and both small and large. Many of the data sets are available in the **treemisc** package that accompanies this book (or another R package), but many are also available for download from the book's website:

<https://bgreenwell.github.io/treebook/datasets.html>.

In this section, I'll introduce a handful of the data sets used in the examples throughout this book. Some of these data sets are pretty common, and are

often used in other texts or articles to illustrate concepts or compare and benchmark performance.

### 1.4.1 Swiss banknotes

The Swiss banknote data [Flury and Riedwyl, 1988] contain measurements from 200 Swiss 1000-franc banknotes: 100 genuine and 100 counterfeit. There are six available predictors, each giving the length (in mm) of a different dimension for each bill (e.g., the length of the diagonal). The response variable is a 0/1 indicator for whether or not the bill was genuine/counterfeit. This is a small data set that will be useful when exploring how some classification trees are constructed. The code snippet below generates a simple scatterplot matrix of the data, which is displayed in [Figure 1.5](#):

```
bn <- treemisc::banknote
cols <- palette.colors(3, palette = "Okabe-Ito")
pairs(bn[, 1L:6L], col = adjustcolor(cols[bn$y + 2], alpha.f = 0.5),
      pch = c(1, 2)[bn$y + 1], cex = 0.7)
```

Note how good some of the features are at discriminating between the two classes (e.g., `top` and `diagonal`). This is a small data set that will be used to illustrate fundamental concepts in decision tree building in [Chapters 2–3](#).

### 1.4.2 New York air quality measurements

The New York air quality data contain daily air quality measurements in New York from May through September of 1973 (153 days). The data are conveniently available in R's built-in `datasets` package; see `?datasets::airquality` for details and the original source. The main variables include:

- `Ozone`: the mean ozone (in parts per billion) from 1300 to 1500 hours at Roosevelt Island;
- `Solar.R`: the solar radiation (in Langleys) in the frequency band 4000–7700 Angstroms from 0800 to 1200 hours at Central Park;
- `Wind`: the average wind speed (in miles per hour) at 0700 and 1000 hours at LaGuardia Airport;
- `Temp`: the maximum daily temperature (in degrees Fahrenheit) at La Guardia Airport.

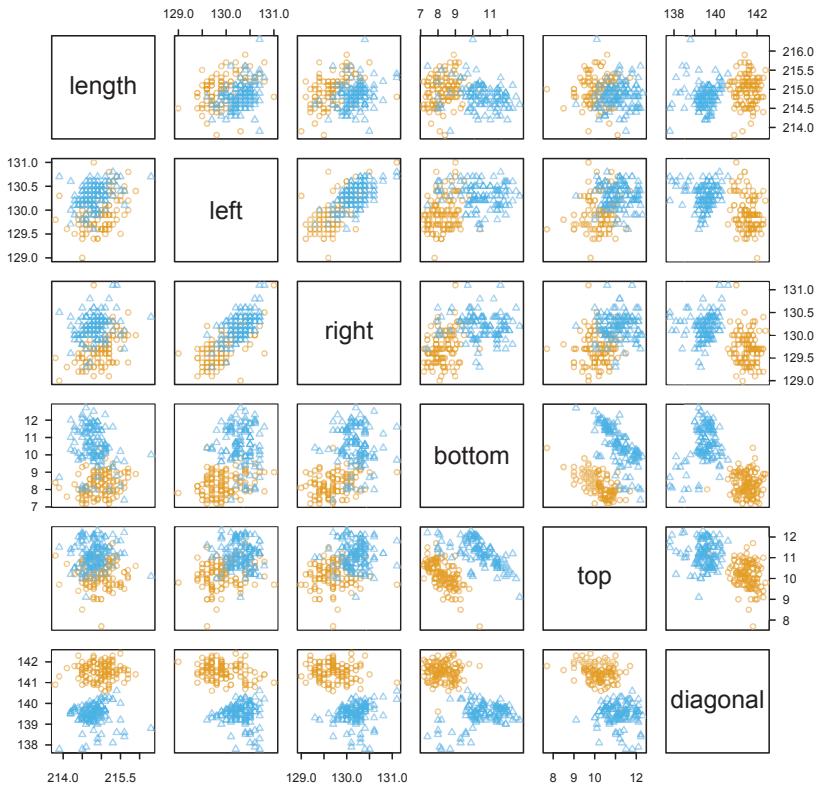


FIGURE 1.5: Scatterplot matrix of the Swiss banknote data. The black circles and orange triangles correspond to genuine and counterfeit banknotes, respectively.

The month (1–12) and day of the month (1–31) are also available in the columns `Month` and `Day`, respectively. In these data, `Ozone` is treated as a response variable.

This is another small data set that will be useful when exploring how some regression trees are constructed. A simple scatterplot matrix of the data is constructed below; see Figure 1.6. The upper diagonal scatterplots each contain a *LOWESS smooth*<sup>P</sup> of the data (red curve). Note that there's a relatively strong nonlinear relationship between `Ozone` and both `Temp` and `Wind`, compared to the others.

<sup>P</sup>A LOWESS smoother is a nonparametric smooth based on locally-weighted polynomial regression; see [Cleveland, 1979] for details.

```

aq <- datasets::airquality
color <- adjustcolor("forestgreen", alpha.f = 0.5)
ps <- function(x, y, ...) { # custom panel function
  panel.smooth(x, y, col = color, col.smooth = "black",
    cex = 0.7, lwd = 2)
}
pairs(aq, cex = 0.7, upper.panel = ps, col = color)

```

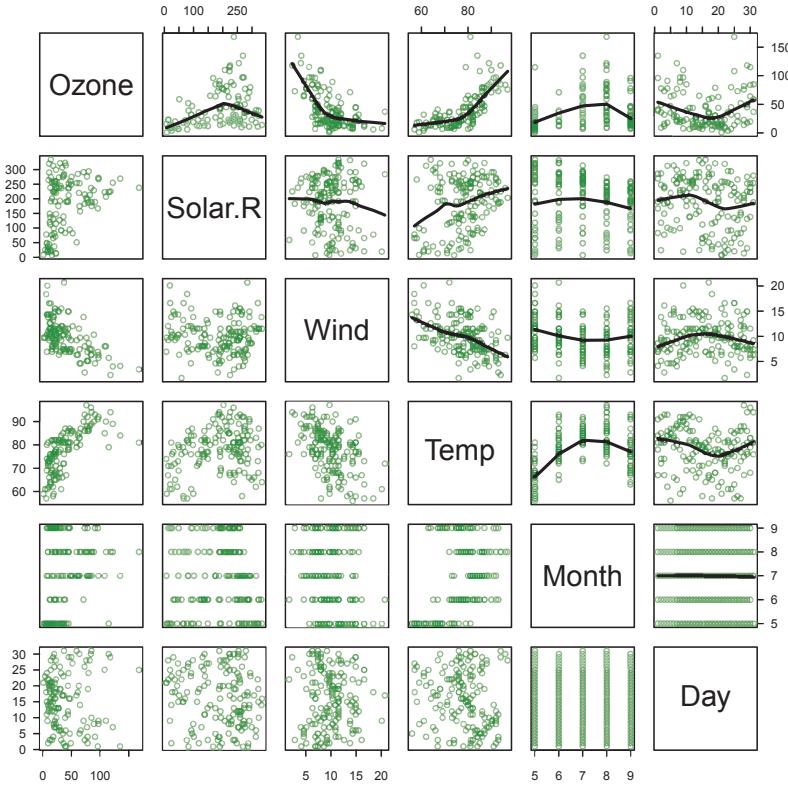


FIGURE 1.6: Scatterplot matrix of the New York air quality data. Each black curve in the upper panel represents a LOWESS smoother.

#### 1.4.3 The Friedman 1 benchmark problem

The Friedman 1 benchmark problem [Breiman, 1996a, Friedman, 1991] uses simulated regression data with 10 input features according to:

$$Y = 10 \sin(\pi X_1 X_2) + 20(X_3 - 0.5)^2 + 10X_4 + 5X_5 + \epsilon, \quad (1.2)$$

where  $\epsilon \sim \mathcal{N}(0, \sigma)$  and the input features are all independent uniform random variables on the interval  $[0, 1]$ :  $\{X_j\}_{j=1}^1 \stackrel{iid}{\sim} \mathcal{U}(0, 1)$ . Notice how  $X_6 - X_{10}$  are unrelated to the response  $Y$ .

These data can be generated in R using the `mlbench.friedman1` function from package `mlbench` [Leisch and Dimitriadou., 2021]. Here, I'll use the `gen_friedman1` function from package `treemisc`, which allows you to generate any number of features  $\geq 5$ ; similar to the `make_friedman1` function in scikit-learn's `sklearn.datasets` module for Python. See `?treemisc::gen_friedman1` for details. Below, I generate a sample of  $N = 5$  observations from (1.2) with only seven features (so it prints nicely):

```
set.seed(943) # for reproducibility
treemisc::gen_friedman1(5, nx = 7, sigma = 0.1)

#>      y     x1     x2     x3     x4     x5     x6     x7
#> 1 18.5 0.346 0.853 0.655 0.839 0.293 0.3408 0.0573
#> 2 13.7 0.442 0.691 0.214 0.108 0.543 0.1616 0.5055
#> 3 10.8 0.223 0.789 0.807 0.252 0.257 0.8595 0.7248
#> 4 18.9 0.859 0.520 0.891 0.129 0.936 0.0348 0.7105
#> 5 14.5 0.181 0.590 0.893 0.611 0.415 0.4104 0.2636
```

From (1.2), it should be clear that features  $X_1 - X_5$  are the most important! (The others don't influence  $Y$  at all.) Also, based on the form of the model, we'd expect  $X_4$  to be the most important feature, probably followed by  $X_1$  and  $X_2$  (both comparably important), with  $X_5$  probably being less important. The influence of  $X_3$  is harder to determine due to its quadratic nature, but it seems likely that this nonlinearity will suppress the variable's influence over its observed range (i.e.,  $[0, 1]$ ). Since the true nature of  $E(Y|\mathbf{x})$  is known, albeit somewhat complex (e.g., nonlinear relationships and an explicit interaction effect), these data are useful in testing out different model interpretability techniques (at least on numeric features), like those discussed in Section 6. Since these data are convenient to generate, I'll use them in a couple of small-scale simulations throughout this book.

#### 1.4.4 Mushroom edibility

The mushroom edibility data is one of my favorite data sets. It contains 8124 mushrooms described in terms of 22 different physical characteristics, like odor and spore print color. The response variable (`Ediblity`) is a binary indicator for whether or not each mushroom is `Edible` or `Poisonous`. The data are available from the UCI Machine Learning repository at <https://archive.ics.uci.edu/ml/datasets/Mushroom>.

[uci.edu/ml/datasets/mushroom](https://archive.ics.uci.edu/ml/datasets/mushroom), but can also be obtained from **treemisc**; see `?treemisc::mushroom` for details and the original source.

What's interesting about these data (at least to me) is that every single variable, both predictor and response, is categorical. These data will be helpful in illustrating how certain decision tree algorithms deal with categorical predictors when choosing splits. A *mosaic plot* showing the relationship between mushroom edibility and odor (one of the better discriminators between edible and poisonous mushrooms in this sample) is constructed below; see [Figure 1.7](#).

The area of each tile is proportional to the number of observations in the particular category. The mosaic plot indicates that the poisonous group is dominated by mushrooms with a strong or unpleasant odor. Hence, we might surmise that poisonous mushrooms tend to be associated with strong or unpleasant odors.

```
mushroom <- treemisc::mushroom
mosaicplot(~ Edibility + odor, data = mushroom, color = TRUE,
           las = 1, main = "", cex.axis = 0.6)
```

#### 1.4.5 Spam or ham?

These data refer to  $N = 4,601$  emails classified as either spam (i.e., junk email) or non-spam (i.e. “ham”) that were collected at Hewlett-Packard (HP) Labs. In addition to the class label, there are 57 predictors giving the relative frequency of certain words and characters in each email. For example, the column `charDollar` gives the relative frequency of dollar signs (\$) appearing in each email. The data are available from the UCI Machine Learning repository at

<https://archive.ics.uci.edu/ml/datasets/spambase>.

In R, the data can be loaded from the **kernlab** package [Karatzoglou et al., 2019]; see `?kernlab::spam` for further details.

Below, I load the data into R, check the frequency of spam and non-spam emails, then look at the average relative frequency of several different words and characters between each:

```
data(spam, package = "kernlab")

# Distribution of ham and spam
table(spam$type)

#>
#> nonspam     spam
#>    2788     1813
```

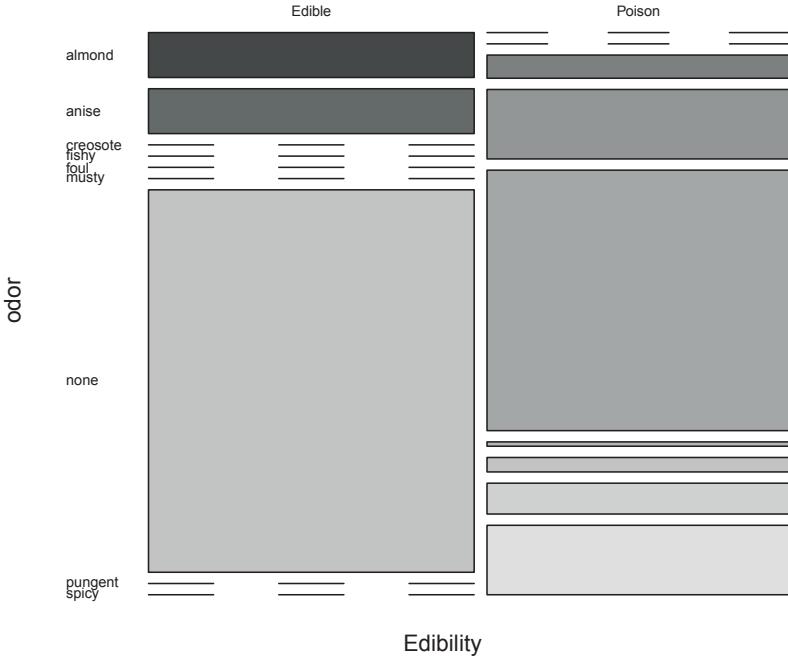


FIGURE 1.7: Mosaic plot visualizing the relationship between mushroom edibility and odor. The area of each tile is proportional to the number of observations in the particular category.

```
# Compute average relative frequency of different words and characters
aggregate(cbind(remove, charDollar, hp, parts, direct) ~ type,
           data = spam, FUN = mean)

#>      type remove charDollar      hp      parts direct
#> 1 nonspam 0.00938     0.0116 0.8955 0.01872 0.0831
#> 2      spam 0.27541     0.1745 0.0175 0.00471 0.0367
```

Notice how the first three variables show a much larger difference between spam and non-spam emails; we might expect these to be important predictors (at least compared to the other two) in classifying new HP emails as spam vs. non-spam. For example, given that these emails all came from Hewlett-Packard Labs, the fact that the non-spam emails contain a much higher relative frequency of the word `hp` makes sense (email spam was not as clever back in 1998).

As a preview of what's to come, the code chunk below fits a basic decision tree with three splits (i.e., it asks three yes or no questions) to a 70% random

sample of the data. It also takes into account the specified assumption that classifying a non-spam email as spam is five times more costly than classifying a spam email as non-spam. We'll learn all about **rpart** and the steps taken below in [Chapter 2](#).

```
library(rpart)
library(treemisc)

# Split into train/test sets using a 70/30 split
set.seed(852) # for reproducibility
id <- sample.int(nrow(spam), size = floor(0.7 * nrow(spam)))
spam.trn <- spam[id, ] # training data
spam.tst <- spam[-id, ] # test data

# Fit a simple classification tree
loss <- matrix(c(0, 1, 5, 0), nrow = 2) # misclassification costs
spam.cart <- rpart(type ~ ., data = spam.trn, cp = 0,
                    parms = list("loss" = loss))
cp <- spam.cart$cptable
cp <- cp[cp[, "nsplit"] == 3, "CP"] # CP associated with 3 splits
spam.cart.pruned <- prune(spam.cart, cp = cp) # grab smaller subtree

# Display tree diagram
tree_diagram(spam.cart.pruned)
```

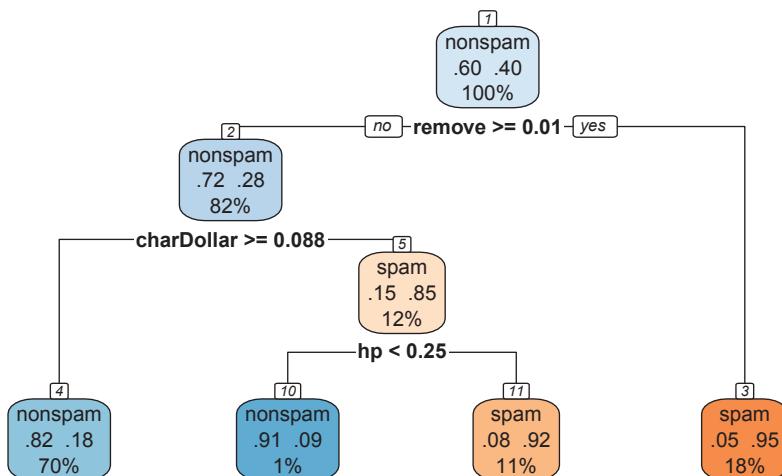


FIGURE 1.8: Decision tree diagram for a simple classification tree applied to the email spam learning sample.

The associated tree diagram is displayed in [Figure 1.8](#). This tree is too simple and underfits the training data (I'll re-analyze these data using an ensemble in [Chapter 5](#)). Nonetheless, simple decision trees can often be displayed as a small set of simple *rules*. As a set of mutually exclusive and exhaustive rules, the tree diagram in [Figure 1.8](#) translates to:

**Rule 1** (path to terminal node 3)  
 IF remove  $\geq 0.01$   
 THEN classification = **spam** (probability = 0.95)

**Rule 2** (path to terminal node 11)  
 IF remove  $< 0.01$  AND  
 | charDollar  $\geq 0.088$  AND  
 | hp  $< 0.25$   
 THEN classification = **spam** (probability = 0.92)

**Rule 3** (path to terminal node 10)  
 IF remove  $< 0.01$  AND  
 | charDollar  $\geq 0.088$  AND  
 | hp  $\geq 0.25$   
 THEN classification = **non-spam** (probability = 0.91)

**Rule 4** (path to terminal node 4)  
 IF remove  $< 0.01$  AND  
 | charDollar  $< 0.088$  AND  
 THEN classification = **non-spam** (probability = 0.82)

The first rule, for instance, states that if the relative frequency of the word “remove” is 0.01 or larger, then we would classify the email as spam with probability 0.95.

#### 1.4.6 Employee attrition

The employee attrition data contain (simulated) human resources analytics data of employees that stay and leave a particular company. The main objective with these data, according to the original source, is to “Uncover the factors that lead to employee attrition...” Such factors include age, job satisfaction, and commute distance. The response variable is **Attrition**, which is a binary indicator for whether or not the employee left (**Attrition** = Yes) or stayed (**Attrition** = No). The data are conveniently available via the R package **modeldata** [Kuhn, 2021]; they can also be obtained from the following IBM GitHub repository: <https://github.com/IBM/employee-attrition-aif360>. To load the data in R, use:

```
data(attrition, package = "modeldata")  
  
# Distribution of class outcomes
```

```
table(attrition$Attrition)

#>
#>   No   Yes
#> 1233  237
```

### 1.4.7 Predicting home prices in Ames, Iowa

The Ames housing data [De Cock, 2011], which are available in the R package **AmesHousing** [Kuhn, 2020], contain information from the Ames Assessor’s Office used in computing assessed values for individual residential properties sold in Ames, Iowa from 2006–2010; online documentation describing the data set can be found at <http://jse.amstat.org/v19n3/decock/DataDocumentation.txt>. These data are often used as a more contemporary replacement to the often cited—and ethically challenged [Carlisle, 2019]—Boston housing data [Harrison and Rubinfeld, 1978].

The data set contains  $N = 2,930$  observations on 81 variables. The response variable here is the final sale price of the home (**Sale\_Price**). The remaining 80 variables, which I’ll treat as predictors, are a mix of both ordered and categorical features.

In the code chunk below, I’ll load the data into R and split it into train/test sets using a 70/30 split, which I’ll use in several examples throughout this book (note that for plotting purposes, mostly to avoid large numbers on the  $y$ -axis, I’ll rescale the response by dividing by 1,000):

```
ames <- as.data.frame(AmesHousing::make_ames())
ames$Sale_Price <- ames$Sale_Price / 1000 # rescale response
set.seed(2101) # for reproducibility
trn.id <- sample.int(nrow(ames), size = floor(0.7 * nrow(ames)))
ames.trn <- ames[trn.id, ] # training data/learning sample
ames.tst <- ames[-trn.id, ] # test data
```

Figure 1.9 shows a scatterplot of sale price vs. above grade (ground) living area in square feet (**Gr\_Liv\_Area**) from the 70% learning sample. Above grade living area, as we’ll see in later chapters, is arguably one of the more important predictors in this data set (as you might expect). It is evident from this plot that *heteroscedasticity* is present, with variation in sale price increasing with home size. Linear models assume constant variance whenever relying on the usual normal theory standard errors and confidence intervals for interpretation. Outliers are another potential problem.

```
plot(Sale_Price ~ Gr_Liv_Area, data = ames.trn,
      col = adjustcolor(1, alpha.f = 0.5),
      xlab = "Above grade square footage",
      ylab = "Sale price / 1000")
```

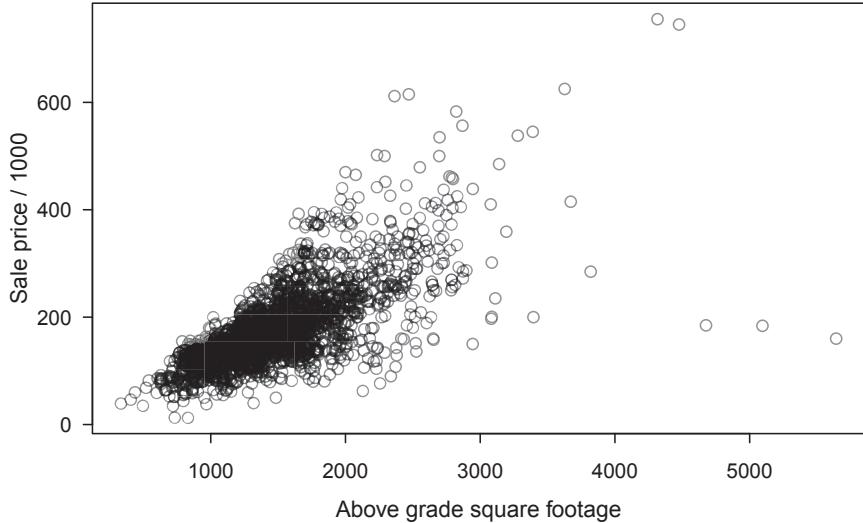


FIGURE 1.9: Scatterplot of sale price vs. above grade (ground) living area in square feet from the Ames housing training sample; here you can see five or so potential outliers.

Note that predictions based solely on these data should not be used alone in setting the sale price of a home. I mean, they could, but they would likely not perform well over time. There are many complexities involved in valuing a home, and housing markets change over time. With the data at hand, it can be hard to predict such changes, especially during the initial Covid-19 outbreak during which the majority of this book was written (many things became rather hard to predict and forecast). However, such a model could be a useful place to start, especially for descriptive purposes.

#### 1.4.8 Wine quality ratings

These data are related to red and white variants of the Portuguese “Vinho Verde” wine; for details, see Cortez et al. [2009]. Due to privacy and logistic issues, only physicochemical and sensory variables are available (e.g., there is no data about grape types, wine brand, wine selling price, etc.). The response variable here is the wine quality score (**quality**), which is an ordered integer in the range 0–10.

The data are available in the R package **treemisc** and can be used for classification or regression, but given the ordinal nature of the response, the latter is more appropriate; see `?treemisc::wine`. The data can also be downloaded from the UCI Machine Learning repository at <https://archive.ics.uci.edu/ml/datasets/Wine+Quality>.

[edu/ml/datasets/wine+quality](https://archive.ics.uci.edu/ml/datasets/wine+quality). Outlier detection algorithms could be used to detect the few excellent or poor wines. Also, it is not known if all the available predictors are relevant.

Below, I load the data into R and look at the distribution of quality scores by wine type (e.g., red or white):

```
wine <- treemisc::wine
xtabs(~ type + quality, data = wine)

#>      quality
#> type    3   4   5   6   7   8   9
#>   red    10  53  681  638  199  18   0
#>   white   20 163 1457 2198  880  175   5
```

Note that most wines (red or white) are mediocre and relatively few have very high or low scores. The response here, while truly an integer in the range 0–10, is often treated as binary by arbitrarily discretizing the ordered response into “low quality” and “high quality” wines. A more appropriate analysis, which utilizes the fact that the response is ordered, is given in [Section 3.5.2](#).

#### 1.4.9 Mayo Clinic primary biliary cholangitis study

This example concerns data from a study by the Mayo Clinic on *primary biliary cholangitis* (PBC) of the liver conducted between January 1974 and May 1984; follow-up continued through July 1986. PBC is an autoimmune disease leading to destruction of the small bile ducts in the liver. There were a total of  $N = 418$  patients whose *survival time* and *censoring indicator* were known (I’ll discuss what these mean briefly). The goal was to compare the drug *D-penicillamine* with a placebo in terms of survival probability. The drug was ultimately found to be ineffective; see, for example, Fleming and Harrington [1991, p. 2] and Ahn and Loh [1994] (the latter employs a tree-based analysis). An additional 16 potential covariates are included which I’ll investigate further as predictors in [Section 3.5.3](#).

Below, I load the data from the **survival** package [Therneau, 2021] and do some prep work. For starters, I’ll only consider the subset of patients who were randomized into the D-penicillamine and placebo groups; see `?survival::pbc` for details. Second, I’ll consider the small number of subjects who underwent liver transplant to be censored at the day of transplant<sup>q</sup>:

```
library(survival)

pbc2 <- pbc[!is.na(pbc$trt), ] # omit non-randomized subjects
pbc2$id <- NULL # remove ID column
```

---

<sup>q</sup>As mentioned in Harrell [2015, Sec. 8.9], liver transplantation was rather uncommon at the time the data were collected, so it still constitutes a natural history study for PBC.

```
# Consider transplant patients to be censored at day of transplant
pb2$status <- ifelse(pb2$status == 2, 1, 0)

# Look at frequency of death and censored observations
table(pb2$status)

#>
#>   0   1
#> 187 125
```

In this sample, 125 subjects died (i.e., experienced the event of interest) and the remaining 187 were considered censored (i.e., we only know they did not die before dropping out, receiving a transplant, or reaching the end of the study period).

In survival studies (like this one), the dependent variable of interest is often *time until some event occurs*; in this example, the event of interest is death. However, medical studies cannot go on forever, and sometimes subjects drop out or are otherwise lost to follow-up. In these situations, we may not have observed the event time, but we at least have some partial information. For example, some of the subjects may have survived beyond the study period, or perhaps some dropped out due to other circumstances. Regardless of the specific reason, we at least have some partial information on these subjects, which survival analysis (also referred to as *time-to-event* or *reliability analysis*) takes into account.

The scatterplot in [Figure 1.10](#) shows the survival times for the first ten subjects in the PBC data, with an indicator for whether or not each observation was censored. The first subject, for example, was recorded dead at  $t = 400$  days, while subject two was censored at  $t = 4,500$  days.

In survival analysis, the response variable typically has the form

$$Y = \min(T, C),$$

where  $T$  is the survival time and  $C$  is the *censoring time*. In this book, I'll only consider right censoring (the most common form of censoring), where  $T \geq Y$ . In this case, all we know is that the true event time is at least as large as the observed time<sup>r</sup>. For example, if we were studying the failure time of some motor in a machine, we might have observed a failure at time  $t = 56$  hours, or perhaps the study ended at  $t = 100$  hours, so all we know is that the true failure time would have occurred some time after that.

To indicate that a particular observation is censored, we can use a censoring indicator:

---

<sup>r</sup>Left censoring and interval censoring are other common forms of censoring.

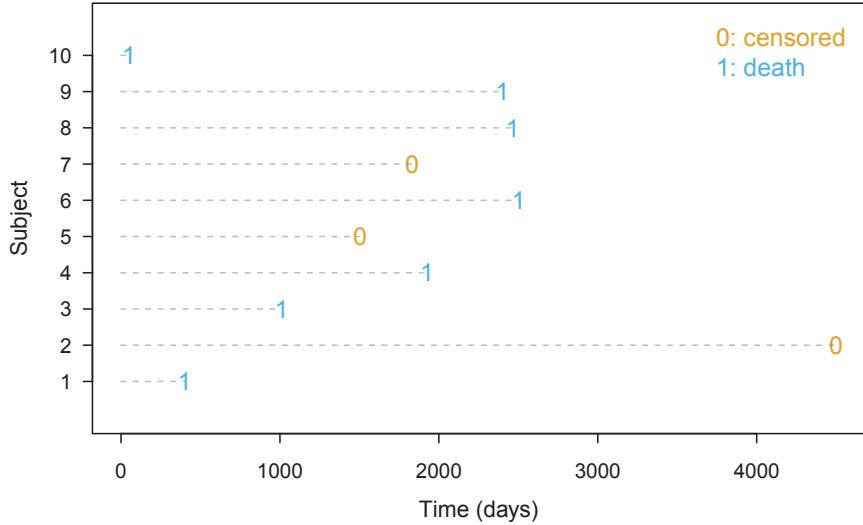


FIGURE 1.10: Survival times for the first ten (randomized) subjects in the Mayo Clinic PBC data.

$$\delta = \begin{cases} 1 & \text{if } T \leq C \\ 0 & \text{if } T > C \text{ (i.e., censored)} \end{cases},$$

where  $\delta = 1$  implies that we observed the true survival time and  $\delta = 0$  indicates a right censored observation (i.e., we only know the subject survived past time  $C$ ). A common cause for right censoring in medical studies is that the study ended before the event of interest (e.g., death) occurred or perhaps some of the individuals dropped out or were lost to follow-up; in either case, we only have partial information. As we'll see, several classes of decision tree algorithms can be extended to handle right censored outcomes. Examples are provided in [Sections 3.5.3](#) (single decision tree) and [8.9.1](#) (ensemble of decision trees).

A common summary of interest in survival studies is the *survival function*:

$$S(t) = \Pr(T > t), \quad (1.3)$$

which describes the probability of surviving longer than time  $t$ . The Kaplan-Meier (or product limit) estimator is a nonparametric statistic used for estimating the survival function in the presence of censoring (if there isn't any censoring, then we could just use the ordinary *empirical distribution function*).

The details are beyond the scope of this book, but the `survfit` function from package `survival` can do the heavy lifting for us.

In the code snippet below, I call `survfit` to estimate and plot the survival curves for both the drug and placebo groups; see Figure 1.11. Here, you can see that the estimated survival curves between the treatment and control group are similar, indicating that D-penicillamine is rather ineffective. The *log-rank test* can be used to test for differences between the survival distributions of two groups. Some decision tree algorithms for the analysis of survival data use the log-rank test to help partition the data; see, for example, Segal [1988] and Leblanc and Crowley [1993].

```
palette("Okabe-Ito")
plot(survfit(Surv(time, status) ~ trt, data = pbc2), col = 2:3,
    conf.int = FALSE, las = 1, xlab = "Days until death",
    ylab = "Estimated survival probability")
legend("bottomleft", legend = c("Penicillmain", "Placebo"),
    lty = 1, col = 2:3, text.col = 2:3, inset = 0.01, bty = "n")
palette("default")
```

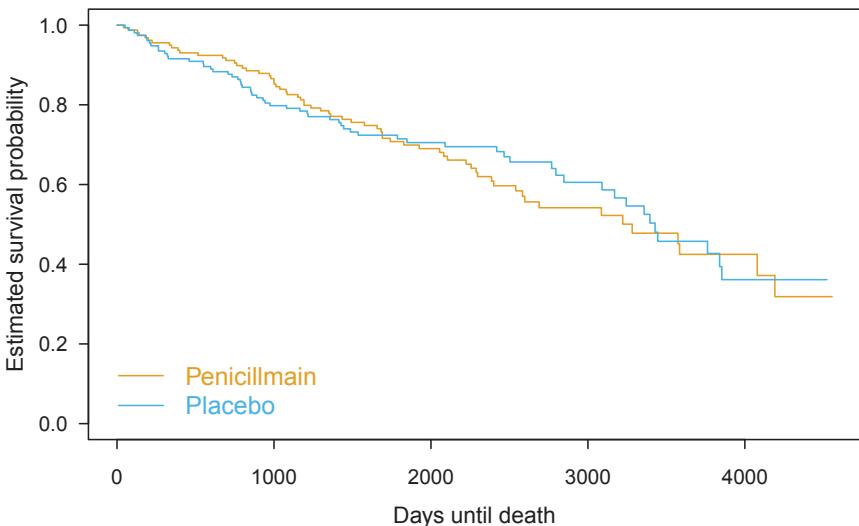


FIGURE 1.11: Kaplan-Meier estimate of the survival function for the randomized subjects in the Mayo Clinic PBC data by treatment group (i.e., drug vs. placebo). The median survival times are 3282 days (drug) and 3428 days (placebo).

In Section 3.5.3, we'll see how a simple tree-based analysis can estimate the survival function conditional on a set of predictors, denoted  $\hat{S}(t|\mathbf{x})$ , by partitioning the learning sample into non-overlapping groups with similar survival rates; here, we'll see further evidence that D-penicillamine was not effective

in improving survival. For a thorough overview of survival analysis, my gold standard has always been Klein and Moeschberger [2003].

---

## 1.5 There ain't no such thing as a free lunch

Too often, we see papers or hear arguments claiming that some cool new algorithm A is better than some existing algorithms B and C at doing D. This is mostly baloney, as any experienced statistician or modeler would tell you that no one procedure or algorithm is uniformly superior across all situations. That being said, you should not walk away from this book with the impression that tree-based methods are superior to any other algorithm or modeling tool. They are powerful and flexible tools for sure, but that doesn't always mean they're the right tool for the job. Consider them as simply another tool to include in your modeling and analysis toolbox.

---

## 1.6 Outline of this book

This book is about decision trees, both individual trees ([Part I](#)) and ensembles thereof ([Part II](#)). There are a large number of decision tree algorithms in existence, and entire books have even been dedicated to some. Consequently, I had to be quite selective in choosing the topics to present in detail in this book, which has mostly been guided by my experiences with tree-based methods over the years in both academics and industry. As mentioned in Loh [2014], “There are so many recursive partitioning algorithms in the literature that it is nowadays very hard to see the wood for the trees.”

I'll discuss some of the major, and most important tree-based algorithms in current use today. However, due to time and page constraints, several important algorithms and extensions didn't make the final cut, and are instead discussed in the (free) online supplementary material that can be found on the book website. These methods include:

- C5.0 [Kuhn and Johnson, 2013, Sec. 14.6], the successor to C4.5 [Quinlan, 1993], which is similar enough to CART that including it in a separate chapter would be largely redundant with [Chapter 2](#);
- MARS, which was briefly mentioned in [Section 1.2](#) (see [Table 1.1](#)), is essentially an extension of linear models (and CART) that automatically

handles variable selection, nonlinear relationships, and interactions between predictors;

- *rule-based models*, like Certifiable Optimal Rule ListS [Angelino et al., 2018], or CORELS for short, which are very much like decision trees, but with an emphasis on producing a small number of simple rules (i.e., short sequences of yes or no questions).

Decision trees remain one of the most flexible and practical tools in the data science toolbox, whether for description or prediction. While they are most commonly used for prediction problems in an ensemble (see [Chapters 5–8](#)), individual decision trees are still one of the most useful off-the-shelf analytic tools available (e.g., they can be used for missing value imputation, description, and variable ranking and selection, to name a few).

The rest of this book is split into two parts:

**Part I:** Individual decision trees. Common decision tree algorithms, like CART ([Chapter 2](#)), CTree ([Chapter 3](#)), and GUIDE ([Chapter 4](#)), are brought into focus. I'll discuss both the basics and the nitty-gritty details which are often glossed over in other texts, or buried in the literature. These algorithms form the building blocks upon which many current state-of-the-art prediction algorithms are built. Such algorithms are the focus of [Part II](#).

**Part II:** Decision tree ensembles. While [Part I](#) will highlight several useful decision tree algorithms, it will become apparent that individual trees rarely make good predictors, at least when compared to other popular algorithms, like neural networks and *random forests* ([Chapter 7](#)). Fortunately, we can often improve their performance by combining the predictions from several hundred or thousand individual trees together. There are several ways this can be accomplished, and [Chapter 5](#) presents two popular and general strategies: *bagging* and *boosting*. [Chapters 7–8](#) then dive deeper into specialized versions of bagging and boosting, respectively.

Each chapter contains numerous software examples that help solidify the main concepts, typically, only involving minimal package use and developing ideas from scratch. Tree-specific software and longer examples, however, are typically reserved for the end of each chapter, after the main ideas have been presented.

# Part I

# Decision trees



Taylor & Francis  
Taylor & Francis Group  
<http://taylorandfrancis.com>

# 2

---

## Binary recursive partitioning with CART

---

I'm always thinking one step ahead, like a carpenter that makes stairs.

Andy Bernard

The Office

---

This is arguably the most important chapter in the book. It is long, and rather involved, but serves as the foundation to more contemporary partitioning algorithms, like *conditional inference trees* CTree([Chapter 3](#)), *generalized, unbiased, interaction detection, and estimation* ([Chapter 4](#)), and tree-based ensembles, such as *random forests* ([Chapter 7](#)) and *gradient boosting machines* ([Chapter 8](#)).

---

### 2.1 Introduction

In this chapter, I'll discuss one of the most general (and powerful) tree-based algorithms in current practice: *binary recursive partitioning*. This treatment of the subject follows closely with the open source routines available in the **rpart** package [Therneau and Atkinson, 2019], the details of which can be found in the corresponding package vignettes which can be accessed directly from R using `browseVignettes("rpart")` (they can also be found on the package's CRAN landing page at <https://cran.r-project.org/package=rpart>). The **rpart** package, which is discussed in depth in [Section 2.9](#), is a modern

implementation of the *classification and regression tree* (CART)<sup>a</sup> procedures proposed in Breiman et al. [1984]. But don’t let the words “classification” and “regression” in the name CART fool you; the procedure is general enough to be applied to many different types of data (e.g., categorical, continuous, multivariate, count, and censored outcomes). However, the primary focus of this chapter will be on standard classification and regression.

[Figure 2.1](#) shows two separate scatterplots, each of which has been divided into three non-overlapping rectangular regions. The left plot contains  $N = 200$  Swiss banknotes ([Section 1.4.1](#)) that have been identified as either genuine (purple circles) or counterfeit (yellow triangles). The  $x$ -axis and  $y$ -axis correspond to the length (in mm) of the top and bottom edges of each bill, respectively. Clearly there’s some separation between the classes using just these two features. We could use these three regions to classify new banknotes as either genuine or counterfeit according to the majority class in whichever region they belong to. For example, any banknote that lands in Region 3 will be classified as counterfeit, since the majority of training observations that occupy it are counterfeit. In this way, the top and right edges of Region 2 form a decision boundary that can be used for classifying new Swiss banknotes.

Similarly, the right plot shows the relationship between temperature (degrees F), wind speed (mph), and ozone level (the response, measured in ppb) for the New York air quality data ([Section 1.4.2](#)); brighter points indicate higher ozone readings. The regions were selected in a way that tries to minimize the response variance within each, subject to some additional constraints. To predict the ozone level for a new data point  $\mathbf{x}$ , we could use the average response rate from whichever region  $\mathbf{x}$  falls in (i.e., the prediction surface is a *step function*).

This is the overall goal of CART, that is, to divide the feature space into non-overlapping rectangular regions that have similar response rates in each, which can then be used for description or prediction. For example, from a description standpoint, we can see that counterfeit Swiss banknotes tend to have abnormally longer top and bottom edges.

In more than two dimensions (i.e., more than two predictors), the disjoint regions are formed by *hyperrectangles*. Why rectangular regions? Rectangular regions are simpler and more computationally feasible to find; they also tend to yield a more interpretable model that can be represented using a convenient tree diagram. In particular, we want the resulting regions to be as homogeneous as possible with respect to the response variable. The challenge is in defining the regions. For example, how many regions should we use and where should we draw the lines? Obviously we could continue refining each

---

<sup>a</sup>As mentioned in [Section 1.2](#), the term “CART” is trademarked; hence, all the open source implementations go by other names. For brevity, I’ll use the acronym CART to refer to the broad class of implementations that follow the original ideas in Breiman et al. [1984], which includes `rpart` and scikit-learn’s `sklearn.tree` module scikit-learn.

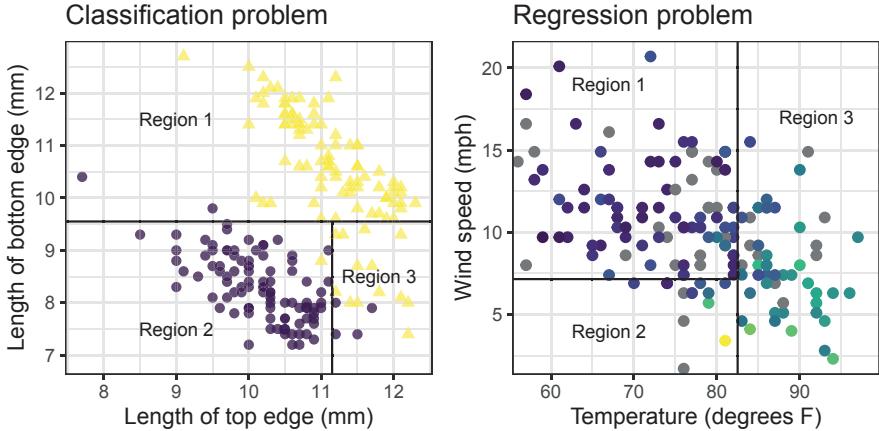


FIGURE 2.1: Scatterplots of two data sets split into three non-overlapping rectangular regions. The regions were selected so that the response values within each were as homogenous as possible. Left: a binary classification problem concerning 200 Swiss banknotes that have been identified as either genuine (purple circles) or counterfeit (yellow triangles). Right: a regression problem (brighter spots indicate higher average response rates within each bin).

region in the left side of Figure 2.1 by making more partitions, but this would eventually lead to overfitting.

The term “binary recursive partitioning” is quite descriptive of the general CART procedure, which I’ll discuss in detail in the next section for the classification case. The word *binary* refers to the binary (or two-way) nature of the splits used to construct the trees (i.e., each split partitions a set of observations into two non-overlapping subsets). The word *recursive* refers to the *greedy* nature of the algorithm in choosing splits sequentially (i.e., the algorithm does not look ahead to find splits that are globally optimal in any sense; it only tries to find the next best split). And of course, partitioning refers to the way splits attempt to partition a set of observations into non-overlapping subgroups with homogeneous response values.

## 2.2 Classification trees

The construction of *classification trees* (categorical outcome) and *regression trees* (continuous outcome) is very similar. However, classification trees involve some subtle nuances that are easy to overlook, so I’ll consider them in detail

first. To begin, let's go back to the Swiss banknote data from [Figure 2.1](#). As discussed in [Section 1.4.1](#), these data contain six continuous measurements on 200 Swiss 1000-franc banknotes: 100 genuine and 100 counterfeit. The goal is to use the six available features to classify new Swiss banknotes as either genuine or counterfeit.

The code chunk below loads the data into R and prints the first few observations:

```
head(bn <- treemisc::banknote) # load and peek at data

#>   length left right bottom  top diagonal y
#> 1    215   131    131    9.0  9.7     141  0
#> 2    215   130    130    8.1  9.5     142  0
#> 3    215   130    130    8.7  9.6     142  0
#> 4    215   130    130    7.5 10.4     142  0
#> 5    215   130    130   10.4  7.7     142  0
#> 6    216   131    130    9.0 10.1     141  0
```

A tree diagram representation of the Swiss banknote regions from [Figure 2.1](#) is displayed in [Figure 2.2](#). The bottom number in each node gives the fraction of observations that pass through that node (hence, the root node displays 100%). The values in the middle give the proportion of counterfeit and genuine banknotes, respectively, and the class printed at the top corresponds to the larger fraction (i.e., whichever class holds the majority in the node). The number above each node gives the corresponding node number. This is an example classification tree that can be used to classify new Swiss banknotes. For example, any Swiss banknote with `bottom >= 9.55` would be classified as counterfeit ( $y = 1$ ); note that the split points are rounded for display purposes in [Figure 2.2](#). The proportion of counterfeit bills in this node is 0.977 and can be used as an estimate of  $\Pr(Y = 1|x)$ ; but more on this later.

From this tree, we can construct three simple rules for classifying new Swiss banknotes using just the bottom and top length of each bill:

**Rule 1** (path to terminal node 2)  
 IF `bottom >= 9.55 (mm)`  
 THEN `classification = Counterfeit (probability = 0.977)`

**Rule 2** (path to terminal node 6)  
 IF `bottom < 9.55 (mm) AND top >= 11.15 (mm)`  
 THEN `classification = Counterfeit (probability = 0.765)`

**Rule 3** (path to terminal node 7)  
 IF `bottom < 9.55 (mm) AND top < 11.15 (mm)`  
 THEN `classification = Genuine (probability = 0.989)`

This tree was found using the CART algorithm as implemented in `rpart`; the corresponding R code is used in [Section 2.9.1](#). But how did CART determine

which features to split on and which split point to use for each? Since this is a binary classification problem, CART searched for the predictor/split combinations that “best” separated the genuine banknotes from the counterfeit ones (I’ll discuss how “best” is determined in the next section).

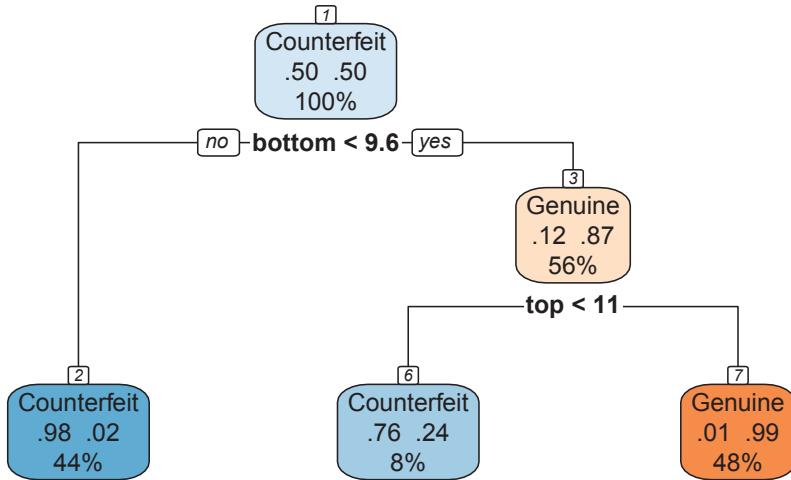


FIGURE 2.2: Example decision tree diagram for classifying Swiss banknotes as counterfeit or genuine.

### 2.2.1 Splits on ordered variables

Let’s first discuss in general how CART finds the “best” split for an ordered variable. A hypothetical split  $S$  of an arbitrary node  $A$  into left and right child nodes, denoted  $A_L$  and  $A_R$ , respectively, is shown in Figure 2.3. If  $A$  contains  $N$  observations, then  $S$  partitions  $A$  into subsets  $A_L$  and  $A_R$  with node sizes  $N_L$  and  $N_R$ , respectively; note that  $N_L + N_R = N$ . Since the splitting process we’re about to describe applies to any node in a tree, we can assume without loss of generality that  $A$  is the root node, which contains the entire learning sample (that is, all of the training data that will be used in constructing the tree). For now, I’ll assume that all of the features are ordered, which includes both continuous and ordered categorical variables (I’ll discuss splits for nominal categorical features in Section 2.4). The first step is to partition the root node in a way that “best separates” the individual class labels into two child nodes; I’ll discuss ways to measure how well a particular split separates the class labels momentarily.

The split  $S$  depicted in Figure 2.3 can be summarized via a 2-by-2 *contingency table* giving the number of observations from each class that go to the left or

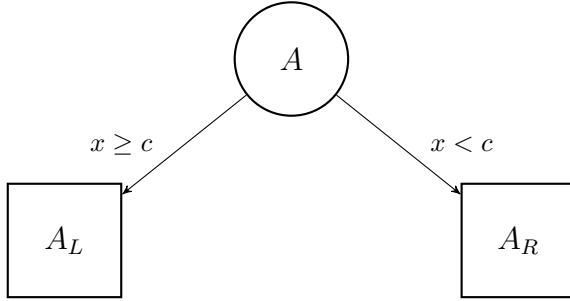


FIGURE 2.3: Hypothetical split for some parent node  $A$  into two child nodes using a continuous feature  $x$  with split point  $c$ .

TABLE 2.1: Confusion table summarizing the split  $S$  depicted in Figure 2.3.

	$y = 0$	$y = 1$	
$A_L : x \geq c$	$N_{0,A_L}$	$N_{1,A_L}$	$N_{A_L}$
$A_R : x < c$	$N_{0,A_R}$	$N_{1,A_R}$	$N_{A_R}$
	$N_{0,A}$	$N_{1,A}$	

right child node. Table 2.1 gives such a summary for a binary 0/1 outcome. For example,  $N_{0,A_L}$  is the number of observations belonging to class  $y = 0$  that went to the left child node. The row and column margins are also displayed.

CART takes a *greedy* approach to tree construction. At each step in the splitting process, CART uses an exhaustive search to look for the next best (i.e., locally optimal) split, which does not necessarily lead to a globally optimal tree structure. This offers a reasonable trade-off between simplicity and complexity—otherwise the algorithm would have to consider all future potential splits at each step, which would lead to a combinatorial explosion. Let's turn our attention now to how CART chooses to split a node.

Let's assume the outcome is binary with  $J = 2$  classes that are arbitrarily coded as 0/1 (e.g., for failure/success). For a continuous feature  $x$  with  $k$

distinct values, CART will consider  $k - 1$  potential splits.<sup>b</sup> Typically, the midpoints of any two consecutive unique values are used as potential split points; for example, if  $x$  has unique values  $\{1, 3, 7\}$  in the learning sample, then CART will consider  $\{2, 5\}$  for potential split points. With  $k - 1$  potential splits to consider, which one does CART choose to partition the data? Ideally, it'd be the split that gives the “best separation” of the class labels (e.g., genuine and counterfeit banknotes, or edible and poisonous mushrooms). So how do we define the goodness of a particular split? Enter *node impurity* measures.

Ideally, we want the two resulting child nodes,  $A_L$  and  $A_R$ , to be as homogeneous as possible with respect to the class labels (e.g., all 0s or all 1s, if possible). To that end, we'd like to construct some function  $i(A)$  that measures the *impurity* of a particular node  $A$ . At one extreme,  $A$  could be a *pure node*, that is, contain either all 0s or all 1s, in which case  $i(A) = 0$ . At the other extreme, the class labels in  $A$  are uniformly distributed (i.e., a 50/50 mix of 0s and 1s)—this is a worst-case scenario and the worst split possible. In this situation, the impurity function,  $i(A)$ , should be at a maximum.

Two common measures of node impurity used in CART are the *Gini index* and *cross-entropy* (or just *entropy* for short). For a response with  $J$  classes, these are defined as:

$$i(A) = \begin{cases} \sum_{j=1}^J p_j(A)(1 - p_j(A)) & \text{Gini index} \\ -\sum_{j=1}^J p_j(A) \log(p_j(A)) & \text{Cross-entropy} \end{cases}, \quad (2.1)$$

where  $p_j(A)$  is the expected proportion of observations in  $A$  that belong to class  $j$ ; note that  $i(A)$  is a function of the  $p_j(A)$  ( $j = 1, 2, \dots, J$ ). To avoid problems with  $\log(0)$  in (2.1), we define  $0 \log(0) \equiv 0$ .

Another splitting measure, called the *twoing splitting rule* [Breiman et al., 1984, pp. 104–106], is only implemented in proprietary software (at least I'm not aware of any open source implementations of CART that support it). The twoing method tends to generate more balanced splits than the Gini or cross-entropy methods. For a binary response, the twoing criterion is equivalent to the Gini index. See Breiman [1996c] for additional details.

Before continuing, we need to introduce some more notation. Let  $N$  be the number of observations in the learning sample and  $N_j$  be the number of observations in the learning sample that belong to class  $j$  (i.e.,  $\sum_{j=1}^J N_j = N$ ). Similarly, let  $N_A$  be the number of observations in node  $A$ , and  $N_{j,A}$  be the

---

<sup>b</sup>For large data sets,  $k$  may be too large, and approximate solutions can be used for scalability; for example, binning  $x$  by constructing a histogram on GPUs (Graphical Processing Units) [Zhang et al., 2017], which can then be used to quickly find a nearly optimal split.

number of observations in  $A$  that belong to class  $j$ . We can estimate  $p_j(A)$  with  $N_{j,A}/N_A$ , the proportion of observations in  $A$  that belong to class  $j$ .<sup>c</sup>

For binary 0/1 outcomes, if we let  $p = p_1(A)$  be the expected proportion of 1s in  $A$ , then (2.1) simplifies to

$$i(A) = \begin{cases} 2p(1-p) & \text{Gini index} \\ -p \log(p) - (1-p) \log(1-p) & \text{Cross-entropy} \end{cases}. \quad (2.2)$$

These are plotted in [Figure 2.4](#) below.

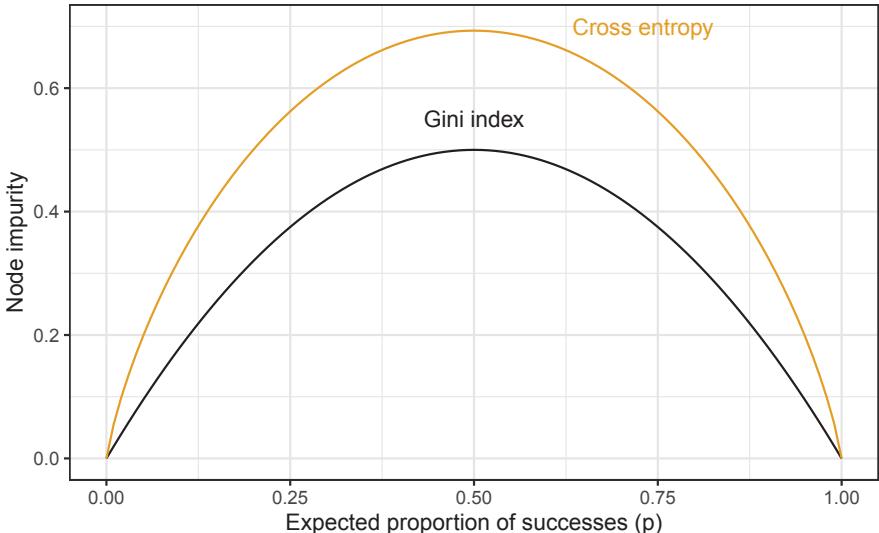


FIGURE 2.4: Common impurity measures for two-class classification problems, as a function of the the expected proportion of successes ( $p$ ).

You may wonder why I'm not considering *misclassification error* as a measure of impurity. As it turns out, misclassification error is not a useful impurity measure for deciding splits; see, for example, [Hastie et al. \[2009, Section 9.2.3\]](#). However, misclassification error can be a useful measure of the *risk* associated with a tree and is used in *decision tree pruning* ([Section 2.5](#)).

<sup>c</sup>Technically, we should use  $p_j(A) \propto \pi_j(N_{j,A}/N_j)$ , where  $\pi_j$  represents the true proportion of class  $j$  in the population of interest (called the prior for class  $j$ ), but I'll come back to this in [Section 2.2.4](#). For now, let's take  $\pi_j = N_j/N$ , the observed proportion of observations in the learning sample that belong to class  $j$ —this assumption is not always valid (e.g., when the data have been *downsampled*), but simplifies the formulas in this section, so I'll leave the complexities to [Section 2.2.4](#).

Now that we have some notion of node impurity, we can define a measure for the quality of a particular split. In essence, the quality of an ordered split  $S = \{x, c\}$  (see [Figure 2.3](#)), often called the *gain* of  $S$ , denoted  $\Delta\mathcal{I}(S, A)$ , is defined as the degree to which the two resulting child nodes,  $A_L$  and  $A_R$ , reduce the impurity of the parent node  $A$ :

$$\begin{aligned}\Delta\mathcal{I}(S, A) &= p(A)i(A) - [p(A_L)i(A_L) + p(A_R)i(A_R)] \\ &= p(A)i(A) - p(A_L)i(A_L) - p(A_R)i(A_R).\end{aligned}. \quad (2.3)$$

Here,  $p(A)$ ,  $p(A_L)$ , and  $p(A_R)$  correspond to the expected proportion of new observations in nodes  $A$ ,  $A_L$ , and  $A_R$ , respectively. For example, we can interpret  $p(A_L)$  as the probability of a case falling in the left child node  $A_L$ . If  $A$  is the root node, then  $p(A) = 1$ ; otherwise, we can estimate it with  $N_A/N$ . For the two-class problem (i.e.,  $J = 2$ ), we can estimate  $p(A_L)$  and  $p(A_R)$  with the corresponding proportion of training cases in  $A_L$  and  $A_R$ , respectively. For instance, we can estimate  $p(A_L)$  with  $N_{A_L}/N$ .

In essence, we want to find the split  $S$  (i.e.,  $x < c$  vs.  $x \geq c$ ) associated with the maximum gain:  $S_{best} = \arg \max_S \Delta\mathcal{I}(S, A)$ . To this end, CART performs an exhaustive search through all features and potential splits therein, and chooses the split with maximal gain to partition  $A$  into two child nodes. This process is then repeated recursively in each resulting child node until a *saturated tree* has been constructed (i.e., no more splits are possible) or some suitable stopping criteria have been met (e.g., the specified maximum depth of the tree has been reached). While CART's approach to choosing the best split seems complicated, we'll implement it in R from scratch and apply it to the Swiss banknote data set in [Section 2.2.2](#).

### 2.2.1.1 So which is it in practice, Gini or entropy?

For binary trees, Breiman [1996c] noted that the Gini index tends to prefer splits that put the most frequent class into one pure node, and the remaining classes into the other. Both entropy and the twoing splitting rules, on the other hand, put their emphasis on balancing the class sizes in the two child nodes. In problems with a small number of classes (i.e.,  $J = 2$ ), the Gini and entropy criteria tend to produce similar results.

Géron [2019, pp.183–184] echoes similar thoughts to Breiman's: “So should you use Gini impurity or entropy? The truth is, most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.”

## 2.2.2 Example: Swiss banknotes

Returning to the Swiss banknote example, our goal is to find the first split condition that “best” separates the genuine banknotes from the counterfeit ones. Here, we’ll restrict our attention to just two features: `top` and `bottom`, which give the length (in mm) of the top and bottom edge, respectively. (We’re restricting attention to these two features because, as we’ll see later, `diagonal` is too good a predictor and leads to a less interesting illustration of finding splits.) Since this is a classification problem, we can use cross-entropy or the Gini index to measure the goodness of each split; here, we’ll use the Gini index and leave implementing cross-entropy as an exercise for the reader.

A simple R function for computing the Gini index in the two-class case is given below. This function takes the binary target values as input, which are assumed to be coded as 0/1 (which corresponds to genuine/counterfeit, in this example); compare the function below to (2.2).

```
gini <- function(y) { # y should be coded as 0/1
  p <- mean(y) # proportion of successes (or 1s)
  2 * p * (1 - p) # Gini index
}
```

To find the optimal split  $S = \{x, c\}$ , where  $x$  is an ordered (but numeric) feature and  $c$  is in its domain, we need to search through every possible value of  $c$ . This can be done, for example, by searching through the midpoints of the sorted, unique values of  $x$ . For each split, we then need to compute the weighted impurity of the current (or parent) node, as well as the weighted impurities of the resulting left and right child nodes; then we find which split point resulted in the largest gain (2.3).

A simple R function, called `splits()`, for carrying out these steps is given below. Here, `node` is a data frame containing the observations in a particular node (i.e., a subset of the learning sample), while `x` and `y` give the column names in `node` corresponding to the (ordered numeric) feature of interest and the (binary or 0/1) target, respectively. The argument `n` specifies the number of observations in the learning sample; this is needed to compute the probabilities  $p(A)$ ,  $p(A_L)$ , and  $p(A_R)$  used in (2.3). The use of `drop = TRUE` in the definitions of the variables `left` and `right` ensures the results are coerced to the lowest possible dimension. The `drop` argument in subsetting arrays and matrices is used a lot in this book; for details, see `?`[`` and `?drop` for additional details.

```
splits <- function(node, x, y, n) { # y should be coded as 0/1
  xvals <- sort(unique(node[[x]])) # sorted, unique values
  xvals <- xvals[-length(xvals)] + diff(xvals) / 2 # midpoints
  res <- matrix(nrow = length(xvals), ncol = 2) # to store results
  colnames(res) <- c("cutpoint", "gain")
  for (i in seq_along(xvals)) { # loop through each midpoint
```

```

left <- node[node[[x]] >= xvals[i], y, drop = TRUE] # left child
right <- node[node[[x]] < xvals[i], y, drop = TRUE] # right child
p <- c(nrow(node), length(left), length(right)) / n # proportions
gain <- p[1L] * gini(node[[y]]) - # Equation (2.3)
      p[2L] * gini(left) - p[3L] * gini(right)
res[i, ] <- c(xvals[i], gain) # store split point and gain
}
res # return matrix of results
}

```

Let's test this function out on the full data set (i.e.,  $A$  is the root node) and find the optimal split point for `bottom`. To start, we'll find the gain that is associated with each possible split point and plot the results:

```

res <- splits(bn, x = "bottom", y = "y", n = nrow(bn))
head(res, n = 5) # peek at first five rows

#>      cutpoint      gain
#> [1,]    7.25 0.00761
#> [2,]    7.35 0.01020
#> [3,]    7.45 0.01948
#> [4,]    7.55 0.03045
#> [5,]    7.65 0.03616

plot(res, type = "b", col = 2, las = 1,
     xlab = "Split value for bottom edge length (mm)",
     ylab = "Gain") # Figure 2.5

```

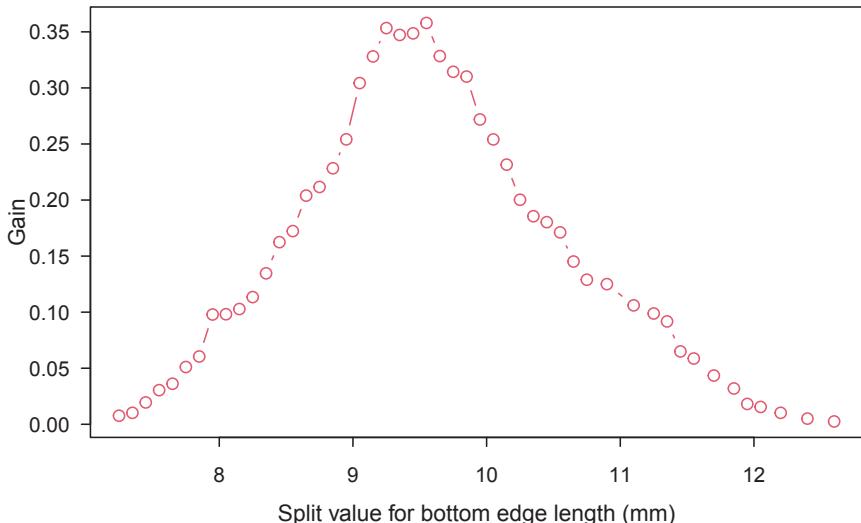


FIGURE 2.5: Reduction to the root node impurity as a function of the split value  $c$  for the bottom edge length (mm).

Figure 2.5 shows the split value  $c$  as a function of gain (or goodness of split). We can extract the exact cutpoint associated with the largest gain using

```
res[which.max(res[, "gain"])], ] # extract row with maximum gain

#> cutpoint      gain
#>    9.550     0.358
```

Here, we can see that the optimal split point for `bottom` is 9.55 mm. A typical tree algorithm based on an exhaustive search would do this for each feature and pick one feature with the largest overall gain. Since all the features in `banknote` are continuous, we can just apply `splits()` to each feature to see which predictor would be used to first split the training data (i.e., the root node). To make things easier, let's write a wrapper function that calls `splits()` for any number of features, finds the split point associated with the largest gain for each, and then returns the best predictor/cutpoint pair. This is accomplished by the `find_best_split()` function below:

```
find_best_split <- function(node, x, y, n) {
  res <- matrix(nrow = length(x), ncol = 2) # to store output
  rownames(res) <- x # set row names to feature names
  colnames(res) <- c("cutpoint", "gain") # column names
  for (xname in x) { # loop through each feature
    # Compute optimal split
    cutpoints <- splits(node, x = xname, y = y, n = n)
    res[xname, ] <- cutpoints[which.max(cutpoints[, "gain"])], ]
  }
  res[which.max(res[, "gain"])], , drop = FALSE
}
```

Now we're ready to start recursively partitioning the `banknote` data set. The code chunk below uses `find_best_split()` on the root node (i.e., the full learning sample) to find the best split between the features `top` and `bottom`:

```
features <- c("top", "bottom") # feature names
find_best_split(bn, x = features, y = "y", n = nrow(bn))

#>           cutpoint      gain
#> bottom      9.55  0.358
```

Using the Gini index, the best way to separate genuine bills from counterfeit ones, using only the lengths of the top and bottom edges, is to separate the banknotes according to whether or not `bottom`  $\geq 9.55$  (mm), which partitions the root node (i.e., full learning sample) into two relatively homogeneous subgroups (or child nodes):

```
left <- bn[bn$bottom >= 9.55, ] # left child node
right <- bn[bn$bottom < 9.55, ] # right child node

table(left$y) # class distribution in left child node
```

```
#>
#> 0 1
#> 2 86


```

It makes no difference which node we consider the left or right child node; here I chose them for consistency with the tree diagram from [Figure 2.2](#). Notice how the left child node is nearly pure, since 86 of the 88 observations (98%) in that node are counterfeit. While we could try to further partition this node using another split, it will likely lead to overfitting. The right node, on the other hand, is less homogeneous, with 14 of the 112 observations being counterfeit, and could potentially benefit from further splitting, as shown below:

```
find_best_split(right, x = features, y = "y", n = nrow(bn))
#>      cutpoint gain
#> top      11.1 0.082
```

The next best split used `top` with a split value of  $c = 11.15$  (mm) and a corresponding gain of 0.082. The resulting child nodes from this split are more homogenous but still not pure.

These two splits match the tree structure from [Figure 2.2](#), which was obtained using actual tree fitting software, but more on that later. Without any stopping criteria defined, the partitioning algorithm could continue splitting until all terminal nodes are pure (a saturated tree). In [Section 2.5](#), we'll discuss how to select an optimal number of splits (e.g., based on cross-validation). Saturated (or nearly full grown) trees are not generally useful on their own; however, in [Chapter 5](#), we'll discuss a simple ensemble technique for improving the performance of individual trees by aggregating the results from several hundred (or even thousand) saturated trees.

### 2.2.3 Fitted values and predictions

Fitted values and predictions for new observations are obtained by passing records down the tree and seeing which terminal nodes they fall in. Recall that every terminal node in a fitted tree comprises some subset of the original training instances. If  $A$  is a terminal node, then any observation  $\mathbf{x}$  (training or new) that lands in  $A$  would be assigned to the majority class in  $A$ :  $\arg \max_{j \in \{1, 2, \dots, J\}} N_{j,A}$ ; tie breaking can be handled in a number of ways (e.g., drawing straws). The predicted probability of  $\mathbf{x}$  belonging to class  $j$ , which is often of more interest (and more useful) than the classification of  $\mathbf{x}$ ,

is given by the proportion of training observations in  $A$  that belong to class  $j$ :

$$\widehat{\Pr}(Y = j | \mathbf{x}) = p_j(A) = N_{j,A}/N_A, \quad j = 1, 2, \dots, J.$$

In the Swiss banknote tree (Figure 2.2; p. 43), any Swiss banknote with `bottom >= 9.55` (mm) would be classified as counterfeit (since the majority of observations in the corresponding terminal node are counterfeit) with a predicted probability of  $86/(86 + 2) = 0.977$ ; note that the fitted probabilities in Figure 2.2 have been rounded to two decimal places, which is why they are not identical to the results we computed by hand in the previous section.

In summary, terminal nodes in a CART-like tree are summarized by a single statistic (or sometimes multiple statistics, like the individual class proportions for  $J$ -class classification), which is then used to obtain fitted values and predictions—all observations that are predicted to be in the same terminal node also receive the same prediction. In classification trees, terminal nodes can be summarized by the majority class or the individual class proportions which are then used to generate classifications or predicted class probabilities for each of the  $J$  classes, respectively. Similarly, the terminal nodes in a CART-like regression tree (Section 2.3) can be summarized by the mean or median response, typically the former.

## 2.2.4 Class priors and misclassification costs

In Section 2.2.3, I mentioned that classifying new observations is done via a majority vote.<sup>d</sup> Similarly, predicted class probabilities can be obtained using the observed class proportions in the terminal nodes. This is a reasonable thing to do if the data are a random sample from some population of interest and the observed frequencies of each target class reflect the true balance in the population. If the observed class frequencies are off (e.g., the data have been *downsampled*, *upsampled*, or the design used to collect the data intentionally over-sampled the minority class to get a representative sample), then it may be beneficial to reweight the observations in a way that reflects the true class proportions, especially when searching for the best splits.

The common but often misguided practice of artificially rebalancing the class labels is especially interesting. Frank Harrell, who we briefly met in Section 1.1.2.4, once wrote

<sup>d</sup>For more than two classes (i.e.,  $J > 2$ ), a *plurality* vote is used.

A special problem with classifiers illustrates an important issue. Users of machine classifiers know that a highly imbalanced sample with regard to a binary outcome variable  $y$  results in a strange classifier. For example, if the sample has 1,000 diseased patients and 1,000,000 non-diseased patients, the best classifier may classify everyone as non-diseased; you will be correct 0.999 of the time. 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. It is simply the case that a classifier trained to a 1/2 prevalence situation will not be applicable to a population with a 1/1,000 prevalence. The classifier would have to be re-trained on the new sample, and the patterns detected may change greatly.

Frank Harrell

<https://www.fharrell.com/post/classification/>

---

Fortunately, CART can flexibly handle imbalanced class labels without changing the learning sample. At a high level, we can assign specific unequal losses or penalties on a one-by-one basis to each type of misclassification error; in binary classification, there are two types of misclassification errors we can make: misclassify a 0 as a 1 (a *false positive*) or misclassify a 1 as a 0 (a false negative). The CART algorithm can account for these unequal losses or misclassification costs when deciding on splits and making predictions. Unfortunately, it seems that many practitioners are either unaware, or fail to take advantage of this feature.

Our discussion of splitting nodes in [Section 2.2.1](#) implicitly made several assumptions about the available data. For instance, estimating  $p_j(A)$  with  $N_{j,A}/N_A$ , the proportion of observations in node  $A$  that belong to class  $j$ , **assumes the training data are a random sample from some population of interest**. In particular, it assumes that the true prior probability of observing class  $j$ , denoted  $\pi_j$ , can be estimated with the observed proportion of class  $j$  observations in the training data; that is,  $\pi_j \approx N_j/N$ . If the observed class proportions are off (e.g., the data have been downsampled or the minority class has intentionally been over-sampled to over-represent rare cases), then  $N_{j,A}/N_A$  is no longer a reasonable estimate of  $p_j(A)$ . Instead, we should be using  $p_j(A) \propto \pi_j N_{j,A}/N_j$ , where we scale the  $\{p_j(A)\}_{j=1}^J$  to sum to one. Note that if we take  $\pi_j$  to be the observed class proportions, then  $\pi_j = N_j/N$  and

$p_j(A)$  reduces to the observed proportion of observations in  $A$  that belong to class  $j$ . Similarly, when determining the “best” split in [Section 2.2.1](#), we weighted the impurity of the two resulting child nodes,  $A_L$  and  $A_R$ , by the expected proportions of new observations going to each. If the data are not a random sample, then we should estimate  $p(A)$  with  $p(A) \approx \sum_{j=1}^J \pi_j N_{j,A}/N_j$ ; and similarly for  $p(A_L)$  and  $p(A_R)$ .

Again, if we take  $\pi_j$  to be the observed class proportions in the learning sample, like we assumed in [Section 2.2.1](#), then we can estimate  $p(A)$  with  $N_A/N$ , the proportion of observations in node  $A$ . However, this is not always realistic. Think about the Swiss banknote data. These data consist of a 50/50 split of both counterfeit and genuine banknotes, which is not likely to be representative of the true class distributions. Nonetheless, I can’t find any background information on how these data were collected. So, without additional information about the true class distributions, there’s not much we can do. The example given in [Section 2.9.5](#) demonstrates the use of CART with updated class prior information from historical data.

What’s important to remember is that the prior class probabilities,  $\{\pi_j\}_{j=1}^J$ , affect the choice of splits in a tree and how the terminal nodes are summarized (e.g., how fitted values and new predictions are computed).

Increasing/decreasing the prior probabilities for certain classes essentially tricks CART into attaching more/less importance to those classes. In other words, it will try harder to correctly predict the classes associated with higher priors at the expense of less accurately predicting the other ones; in this sense, the prior probabilities can be seen as a *tuning parameter* in decision tree construction, especially if you want to attach more importance to correctly classifying certain classes. However, in some cases, it may be more natural to think about the specific costs associated with certain misclassifications. For example, with binary outcomes, it is often the case that false positives are more severe than false negatives, or vice versa. In the mushroom classification example ([Section 1.4.4](#)), it would be far worst to misclassify a poisonous mushroom as edible (a false negative, assuming poisonous represents the positive class, or class of interest) than to misclassify an edible mushroom as poisonous (a false positive). The next section introduces a general strategy for incorporating unequal losses, called *altered priors*; a second strategy, called the *generalized Gini index*, is discussed in the “Introduction to Rpart” vignette; see `vignette("longintro", package = "rpart")` for details.

#### 2.2.4.1 Altered priors

Let  $\mathbf{L}$  be a  $J \times J$  loss matrix with entries  $L_{i,j}$  representing the loss (or cost) associated with misclassifying an  $i$  as a  $j$ . We can define the *risk* of a node  $A$  as

$$r(A) = \sum_{j=1}^J p_j(A) \times L_{j,\tau_A}, \quad (2.4)$$

where  $\tau_A$  is the class assigned to  $A$ , if  $A$  were a terminal node, such that this risk is minimized. Since  $p_j(A)$  depends on the prior class probabilities, risk is a function of both misclassification costs and class priors.

As a consequence, we can take misclassification costs into account by absorbing them into the priors for each class; this is referred to as the *altered priors* method. In particular, if

$$L_{i,j} = \begin{cases} L_i & i \neq j \\ 0 & i = j \end{cases}$$

then we can use the prior approach discussed above with the priors altered according to

$$\tilde{\pi}_i = \pi_j L_i / \sum_{j=1}^J \pi_j L_j, \quad (2.5)$$

where  $\pi_j$  is the prior (observed or specified) associated with class  $j$  ( $j = 1, 2, \dots, J$ ). This is always possible for binary classification (i.e.,  $J = 2$ ). For multiclass problems (i.e.,  $J \geq 3$ ), we can use (2.5) with  $L_i = \sum_{j=1}^J L_{i,j}$ .

For details and further discussion, see Berk [2008, pp. 122–128] or the “Introduction to Rpart” vignette in package **rpart** (use `vignette("longintro", package = "rpart")` at the R console).

#### 2.2.4.2 Example: employee attrition

To illustrate, let's walk through a detailed example using the employee attrition data set (Section 1.4.6). Figure 2.6 displays two classification trees fit to the employee attrition data, each with a max depth of two.<sup>e</sup> The only difference is that the tree on the left used the observed class priors  $\pi_{no} = 1233/1470 = 0.839$  and  $\pi_{yes} = 237/1470 = 0.161$  (i.e., it treats both types of misclassifications as equal). The tree on the right used altered priors based on the following loss (or misclassification cost) matrix:

---

<sup>e</sup>The depth of a decision tree is the maximum of the number of edges from the root node to each terminal node and is a common tuning parameter; see Section 8.3.2.

$$\mathbf{L} = \begin{matrix} & \text{No} & \text{Yes} \\ \text{No} & 0 & 1 \\ \text{Yes} & 8 & 0 \end{matrix},$$

where the rows represent the true class and the columns represent the predicted class. For example, we're saying that it is 8 times more costly to misclassify a Yes (employee will leave due to attrition) as a No (employee will not leave due to attrition) than it is to misclassify a No as a Yes. Using this loss matrix, we can compute the altered priors as follows:

$$\tilde{\pi}_{no} \propto (0 + 1) \pi_{no} = 1233/1470 = 0.839$$

$$\tilde{\pi}_{yes} \propto (8 + 0) \pi_{yes} = 8 (237/1470) = 1.290$$

Rescaling so that  $\tilde{\pi}_{no} + \tilde{\pi}_{yes} = 1$  gives  $\tilde{\pi}_{no} = 0.394$  and  $\tilde{\pi}_{yes} = 0.606$ . Notice how altering the priors resulted in a tree with different splits and node summaries.

The confusion matrix from each tree applied to the learning sample is shown in [Table 2.2](#). Altering the priors by specifying a higher cost for misclassifying the Yeses increased the number of true negatives (assuming No represents the positive class) from 48 to 233, albeit at the expense of decreasing the number of true negatives from 1212 to 163. Finding the right balance is application-specific and requires a lot of thought and collaboration with subject matter experts.

TABLE 2.2: Confusion matrix from the trees in [Figure 2.6](#).

		Observed class			
		Default priors		Altered priors	
		No	Yes	No	Yes
Predicted class	No	1212	189	163	4
	Yes	21	48	1070	233

The tree structure on the left of [Figure 2.6](#) uses the same calculations we worked through for the Swiss banknote example, so let's walk through some of the calculations for the tree on the right.

In any particular node A, we estimate  $p_{no}(A) \propto \tilde{\pi}_{no} \times N_{no,A}/N_{no}$  and  $p_{yes}(A) \propto \tilde{\pi}_{yes} \times N_{yes,A}/N_{yes}$ , which are rescaled to sum to one. For instance, if A is the root node, we have  $p_{no}(A) = \tilde{\pi}_{no} = 0.394$  since  $N_{no,A}/N_{no} = 1233/1233 = 1$ . Similarly,  $p_{yes}(A) = 0.606$ . We can then calculate the impurity of the root node using the Gini index:

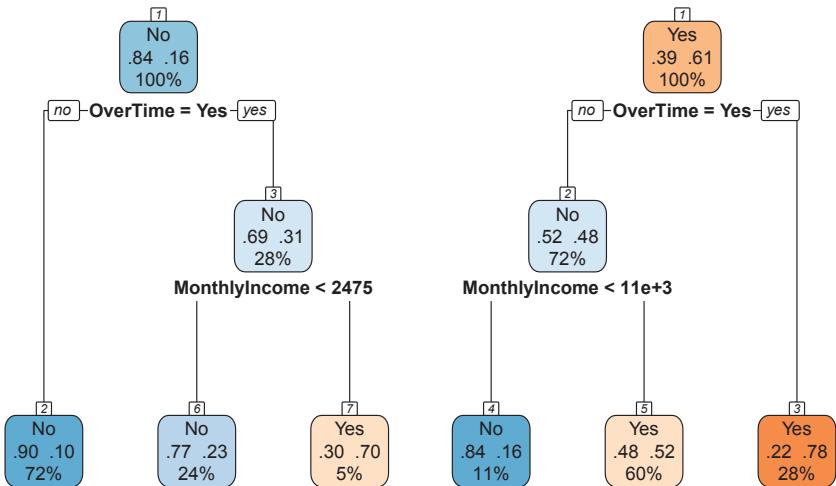


FIGURE 2.6: Decision trees for the employee attrition example. Left: default (i.e., observed) class priors. Right: altered class priors.

$$i(A) = 2 \times p_{no}(A) \times (1 - p_{no}(A)) = 0.478.$$

If we split the data according to `Overtime = Yes` (right branch) vs. `Overtime = No` (left branch), we have the following:

```
left <- attrition[attrition$OverTime == "No", ] # left child
right <- attrition[attrition$OverTime == "Yes", ] # right child
table(attrition$Attrition) # class frequencies

#>
#>   No   Yes
#> 1233  237

table(left$Attrition) # class frequencies (left node)

#>
#>   No   Yes
#>  944  110

table(right$Attrition) # class frequencies (right node)

#>
#>   No   Yes
#>  289  127
```

For the left child node, we have

$$\begin{aligned}
p(A_L) &= \tilde{\pi}_{no} \times (944/1233) + \tilde{\pi}_{yes} \times (110/237) = 0.583, \\
p_{no}(A_L) &= \tilde{\pi}_{no} \times (944/1233) / p(A_L) = 0.518, \\
p_{yes}(A_L) &= 1 - p_{no}(A_L) = 0.482, \\
i(A_L) &= 2 \times p_{no}(A_L) \times (1 - p_{no}(A_L)) = 0.499.
\end{aligned}$$

Similarly, for the right child node, we have:

$$\begin{aligned}
p(A_R) &= \tilde{\pi}_{no} \times (289/1233) + \tilde{\pi}_{yes} \times (127/237) = 0.417, \\
p_{no}(A_R) &= \tilde{\pi}_{no} \times (289/1233) / p(A_R) = 0.221, \\
p_{yes}(A_R) &= 1 - p_{no}(A_R) = 0.779, \\
i(A_R) &= 2 \times p_{no}(A_R) \times (1 - p_{no}(A_R)) = 0.345.
\end{aligned}$$

And the gain for this split is

$$p(A) \times i(A) - p(A_L) \times i(A_L) - p(A_R) \times i(A_R) = 0.043.$$

In [Section 2.9.4](#), we'll verify these calculations using open source tree software that follows the same CART-like procedure for altered priors.

This wraps our discussion of CART's search for the best split for an ordered variable in classification trees. Before discussing the search for splits on categorical features, I'll introduce the concept of a *regression tree*; that is, a decision tree with a continuous outcome.

## 2.3 Regression trees

Up to this point, our discussion of splitting nodes applies primarily to the case of CART-like classification trees. In CART, regression trees are constructed in nearly the same way as classification trees. The only real difference is that rather than finding the predictor/split combination that gives the greatest reduction in the within-node impurity, we look for the predictor/split combination that gives the greatest reduction in node *sum of squared errors (SSE)*:

$$\Delta \mathcal{I}(S, A) = SSE_A - (SSE_{A_L} + SSE_{A_R}), \quad (2.6)$$

where, for example,  $SSE_A = \sum_{i=1}^{N_A} (y_i - \bar{y})$  is the SSE within node  $A$ ; recall that  $N_A$  is the number of training records in node  $A$ . This is equivalent to choosing the split that maximizes the between-groups sum-of-squares in an *analysis of variance* (ANOVA); in fact, in **rpart**, this split rule is referred to as the "anova" method (see `?rpart::rpart`). Note the similarities and differences between Equations (2.3) and (2.6).

To speed up the search for the best split, open source implementations, like **rpart** and scikit-learn's **sklearn.tree** module, do not directly search for splits that maximize (2.6) directly, but rather an equivalent proxy that's more efficient to compute. For example, it can be shown that

$$SSE_A = SSE_{A_L} + SSE_{A_R} + \frac{N_{A_L} N_{A_R}}{N_A} (\bar{y}_L - \bar{y}_R)^2, \quad (2.7)$$

where  $\bar{y}_L$  and  $\bar{y}_R$  give the sample mean for the left and right child nodes of  $A$ , respectively. This implies that maximizing (2.6) is equivalent to maximizing the last term in (2.7), which makes sense, since we want the child nodes to be as different as possible (i.e., a greater difference in the mean responses).

In the regression case, we don't have to worry about priors or node probabilities. The terminal nodes are summarized by the mean response in each (the sample median is another possibility), and these are used for producing fitted values and predictions. For example, if a new observation  $x$  were to occupy some node terminal node  $A$ , then  $\hat{f}(x) = \sum_{i=1}^{N_A} y_{i,A}/N_A$ , where  $y_{i,A}$  denotes the  $i$ -th response value from the learning sample that resides in terminal node  $A$ .

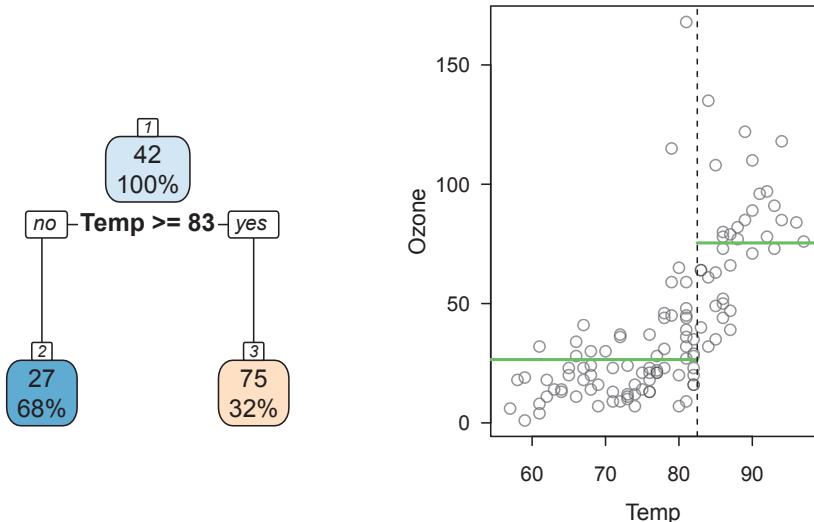
Aside from being useful in their own right, regression trees, as presented here, serve as the basic building blocks for *gradient tree boosting* ([Chapter 8](#)), one of the most powerful tree-based ensemble algorithms available.

### 2.3.1 Example: New York air quality measurements

Consider, for example, the **airquality** data frame introduced in [Section 1.4.2](#), which contains daily air quality measurements in New York from May to September of 1973. A regression tree with a single split was fit to the data and the corresponding tree diagram is displayed in the left side of [Figure 2.7](#). Here, the chosen splitter was temperature (in degrees Fahrenheit). Each node displays the predicted ozone concentration for all observations that fall in that node (top number) as well as the proportion of training observations in each (bottom number). According to this tree, the predicted ozone concentration is given by the simple rule:

$$\widehat{\text{ozone}} = \begin{cases} 26.544 & \text{if Temp} < 82.5 \\ 75.405 & \text{if Temp} \geq 82.5 \end{cases}.$$

The estimated regression surface is plotted in the right side of [Figure 2.7](#). Note that the estimated prediction surface from a regression tree is essentially a step function, which makes it hard for decision trees to capture arbitrarily smooth or linear response surfaces.



**FIGURE 2.7:** Decision stump predicting ozone concentration as a function of temperature. Left: tree diagram. Right: estimated regression function; a vertical dashed line is drawn at the split point  $c = 82.5$  (the tree diagram on the left rounded up to the nearest integer).

To manually find the first partition and reconstruct the tree in [Figure 2.7](#), we'll start by creating a simple function to calculate the within-node SSE. Note that these data contain a few missing values<sup>f</sup> (or NAs in R), so I set `na.rm = TRUE` in order to remove them before computing the results.

```
sse <- function(y, na.rm = TRUE) {
  sum((y - mean(y, na.rm = na.rm))^ 2, na.rm = na.rm)
}
```

Next, I'll modify the `splits()` function from [Section 2.2.2](#) to work for the regression case:

---

<sup>f</sup>CART is actually pretty clever in how it handles missing values in the predictors, but more on this in [Section 2.7](#).

```
splits.sse <- function(node, x, y) {
  xvals <- sort(unique(node[[x]])) # sorted, unique values
  xvals<- xvals[-length(xvals)] + diff(xvals) / 2 # midpoints
  res <- matrix(nrow = length(xvals), ncol = 2)
  colnames(res) <- c("cutpoint", "gain")
  for (i in seq_along(xvals)) { # loop through each feature
    left <- node[node[[x]] >= xvals[i], y, drop = TRUE] # left
    right <- node[node[[x]] < xvals[i], y, drop = TRUE] # right
    gain <- sse(node[[y]]) - sse(left) - sse(right) # Equation (2.6)
    res[i, ] <- c(xvals[i], gain) # store cutpoint and associated gain
  }
  res # return matrix of results
}
```

Before applying this function to the air quality data, I'll remove the 37 rows that have a missing response value. The possible split points for `Temp`, along with their associated gains, are displayed in [Figure 2.8](#). (To make the  $y$ -axis look nicer on the plot, the gain values were divided by 1,000.)

```
# Find optimal split for `Temp`
aq <- airquality[!is.na(airquality$Ozone), ]
res <- splits.sse(aq, x = "Temp", y = "Ozone")
res[which.max(res[, "gain"]), ]

#> cutpoint      gain
#>     82.5  60158.5

# Plot results
res[, "gain"] <- res[, "gain"] / 1000 # rescale for plotting
plot(res, type = "b", col = 2, las = 1,
     xlab = "Temperature split value (degrees Fahrenheit)",
     ylab = "Gain/1000")
abline(v = 82.5, lty = 2, col = 2)
```

To show that temperature is the best primary splitter for the root node, we can use `sapply()` to find the optimal cutpoint for all five features.:

```
features <- c("Solar.R", "Wind", "Temp", "Month", "Day")
sapply(features, FUN = function(xname) {
  res <- splits.sse(aq, x = xname, y = "Ozone")
  res[which.max(res[, "gain"]), ]
})

#>          Solar.R      Wind      Temp      Month      Day
#> cutpoint    153       6.6     82.5      6.5     24.5
#> gain        29721 50591.2 60158.5 14511.3 10282.8
```

Clearly, the split associated with the largest gain is `Temp`, followed by `Wind`, `Solar.R`, `Month`, and `Day`.

A regression tree in one predictor produces a step function, as was seen in the right side of [Figure 2.7](#). The same idea extends to higher dimensions as well.

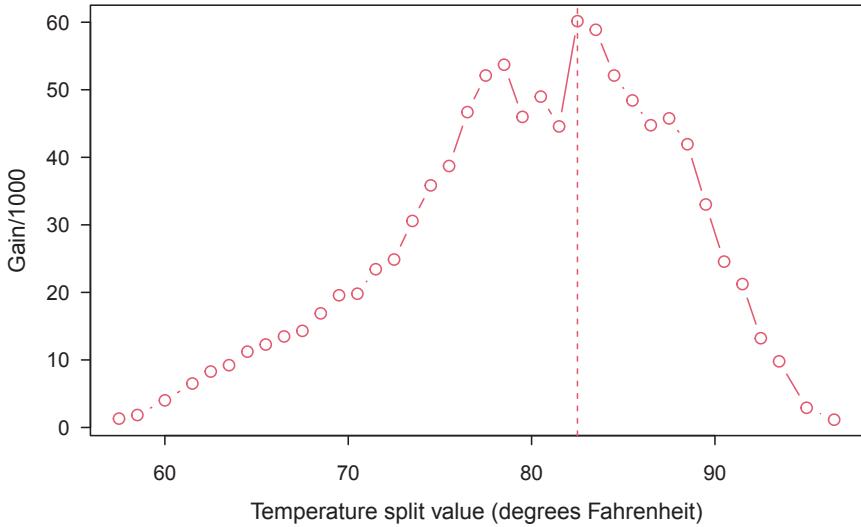


FIGURE 2.8: Potential split points for temperature as a function of gain. The maximum gain occurs at a temperature of 82.5 °F (the dashed vertical line).

For example, suppose we considered splitting on `Wind` next. Using the same procedures previously described, we would find that the next best partition occurs in the left child node using `Wind` with a cutpoint of 7.15 (mph). The corresponding tree diagram is displayed on the left side of [Figure 2.9](#). If we stop splitting here, the result is a regression tree in two features. The corresponding prediction function, displayed on the right side of [Figure 2.9](#), is a surface that's constant over each terminal node.

## 2.4 Categorical splits

Up to this point, we've only considered splits for ordered predictors, which have the form  $x < c$  vs.  $x \geq c$ , where  $c$  is in the domain of  $x$ . But what about splits involving nominal categorical features? If  $x$  is ordinal (i.e., an ordered category, like low < medium < high), then we can map its ordered categories to the integers  $1, 2, \dots, J$ , where  $J$  is the number of unique categories, and split as if  $x$  were originally numeric. If  $x$  is nominal (i.e., the order of the categories has no meaning), then we have to consider all possible ways to split

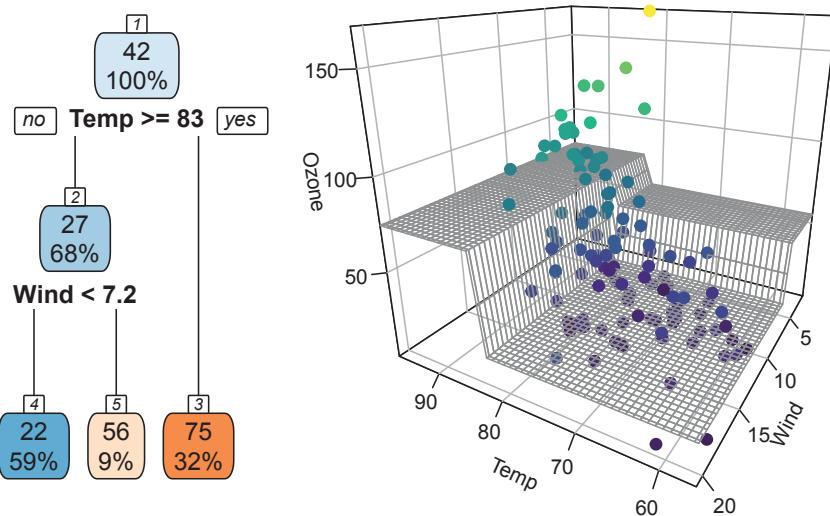


FIGURE 2.9: Regression tree diagram (left) and corresponding regression surface (right) for the air quality data. These are the same splits shown in [Figure 2.1](#).

$x$  into two mutually disjoint groups. For example, if  $x$  took on the categories  $\{a, b, c\}$ , then we could form a total three splits:

- $x \in \{a\}$  vs.  $x \in \{b, c\}$ ;
- $x \in \{b\}$  vs.  $x \in \{a, c\}$ ;
- $x \in \{c\}$  vs.  $x \in \{a, b\}$ .

For a nominal predictor with  $J$  categories, there are a total of  $2^{J-1} - 1$  potential splits to search through, which can be computationally prohibitive for large  $J$ ; for  $J \geq 21$ , we'd have to search more than a million splits! Fortunately, for ordered or binary outcomes, there is a computational shortcut that can be exploited for the splitting rules discussed in this chapter (i.e., Gini index, entropy, and SSE). This is discussed, for example, in Hastie et al. [2009, Sec. 9.2.4] and the “User Written Split Functions” vignette in package **rpart** (use `vignette("usercode", package = "rpart")` at the R console).

In short, the optimal split for a nominal predictor  $x$  at some node  $A$  can be found by first ordering the individual categories of  $x$  by their average response value—for example, the proportion of successes in the binary outcome case—and then finding the best split using this new ordinal variable.<sup>g</sup> This reduces

<sup>g</sup>This is equivalent to performing *mean/target encoding* [Micci-Barreca, 2001] prior to searching for the best split at each node; see [Section 2.4.3](#).

the total number of possible splits from  $2^{J-1} - 1$  to  $J - 1$ , an appreciable reduction in the total number of splits that must be searched. It will also still result in the optimal split when using the Gini index, cross-entropy, or SSE splitting rules discussed earlier. A proof for the Gini and entropy measures is provided in Ripley [1996, p. 218], with Chou [1991] providing a proof for a more general family of impurity measures. For multiclass problems (i.e.,  $J > 2$ ), no such computational shortcut exists, although efficient search methods have been proposed in Sleumer [1969] and Loh and Vanichsetakul [1988].

### 2.4.1 Example: mushroom edibility

To illustrate, let's return to the mushroom edibility example, which contains all categorical features and a binary response. A simple classification tree diagram for the data is shown in Figure 2.10. The tree contains two splits on the features `odor` and `spore.print.color`. Since the response (Edibility) is binary, we can use the shortcut approach to build the tree using the same process for ordered splits, as long as we apply the Gini or entropy splitting criterion; here, I'll use the Gini index since it's already built into our previously defined `find_best_split()` function.

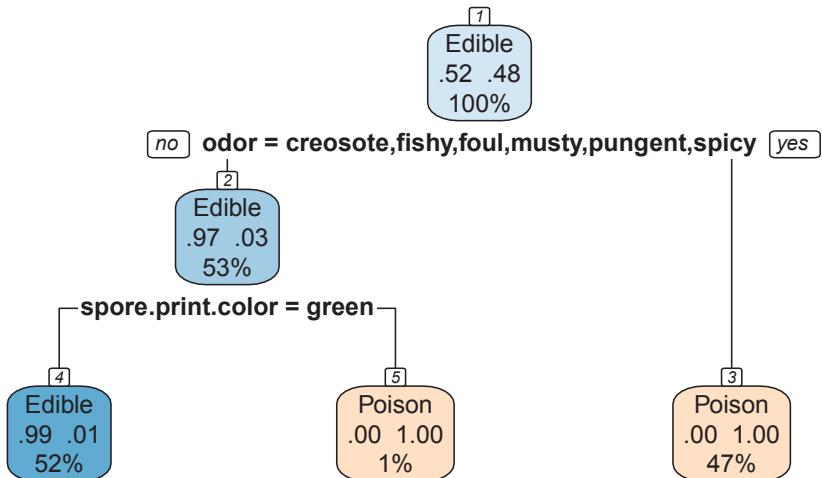


FIGURE 2.10: Example classification tree for determining the edibility of mushrooms.

For each mushroom attribute, the individual categories need to be mapped to the proportion of successes within each. For this example, I'll refer to the outcome class `Poison` as a success and re-encode the target as 0/1 for

`Edible/Poison`. I'll also remove the `veil.type` feature because it only takes on a single value (i.e., it has zero variance) and can contribute nothing to the partitioning:

```
m <- treemisc::mushroom # load mushroom data
m$veil.type <- NULL # remove useless feature
m$Edibility <- ifelse(m$Edibility == "Poison", 1, 0)
m2 <- m # make a copy of the original data
```

To illustrate the main idea, let's look at a frequency table for the `veil.color` predictor, which has four unique categories:

```
table(m2$veil.color)

#>
#> brown orange white yellow
#>    96     96   7924      8
```

We need to find the mean response within each category—in this case, the proportion of poisonous mushrooms—and then map those back to the original feature values. For instance we would re-encode all the values of "white" in `veil.color` as 0.493 because  $3908/7924 \approx 0.493$  of the mushrooms with `veil.color = "white"` are poisonous. This can be done in any number of ways, and here I'll write a simple function, called `ordinalize()`, that returns a list with two components: `map`, which contains the numeric value each category gets mapped to, and `encoded`, which contains the re-encoded feature values.

```
ordinalize <- function(x, y) { # convert nominal to ordered
  map <- tapply(y, INDEX = x, FUN = mean)
  list("mapping" = map, "encoded" = map[x])
}

# Check which numeric values `veil.color` gets mapped to
ordinalize(m2$veil.color, m2$Edibility)$map

#> brown orange white yellow
#> 0.000 0.000 0.493 1.000
```

Next, I'll write a simple `for` loop that uses `ordinalize()` to numerically re-encode each feature column in the `m2` data frame:

```
xnames <- setdiff(names(m2), "Edibility")
for (xname in xnames) { # mean/target encode each feature
  m2[[xname]] <- ordinalize(m2[[xname]], y = m2[["Edibility"]])$encoded
}

# Take a peek at the re-encoded data
m2[1L:8L, 1L:5L]

#>   Edibility cap.shape cap.surface cap.color bruises
#> 1           1        0.467       0.552      0.447    0.185
```

```
#> 2      0    0.467    0.552    0.627    0.185
#> 3      0    0.106    0.552    0.308    0.185
#> 4      1    0.467    0.536    0.308    0.185
#> 5      0    0.467    0.552    0.439    0.693
#> 6      0    0.467    0.536    0.627    0.185
#> 7      0    0.106    0.552    0.308    0.185
#> 8      0    0.106    0.536    0.308    0.185
```

Since all the categorical features have been re-encoded numerically, we can use our previously defined `find_best_split()` function to partition the data. Starting with the root node (i.e., the full learning sample), we obtain:

```
find_best_split(m2, x = xnames, y = "Edibility", n = nrow(m2))

#>      cutpoint gain
#> odor    0.517 0.471

# Summarize split
left <- m2[m2$odor >= 0.5170068, ]
right <- m2[m2$odor < 0.5170068, ]

table(left$Edibility) # non-pure node

#>
#>     1
#> 3796

table(right$Edibility) # pure node

#>
#>     0     1
#> 4208   120
```

The first split uses `odor`, with a mean/target encoded split point of  $c = 0.517$  and a corresponding gain of 0.471. Since the resulting right child node is pure (in this case, all poisonous), let's continue partitioning with the left one:

```
# Ordinalize left child node and find next best split
right.ord <- right
for (xname in xnames) { # mean/target encode each feature
  right.ord[[xname]] <-
    ordinalize(right.ord[[xname]],
               y = right.ord[["Edibility"]])$encoded
}

# Find best split in newly "ordinalized" predictors
find_best_split(right.ord, x = xnames, y = "Edibility", n = nrow(m2))

#>      cutpoint gain
#> spore.print.color    0.538 0.017
```

The next split is based on `spore.print.color`, with a mean/target encoded split point  $c = 0.538$  and a corresponding gain of 0.017, which is equivalent

to separating mushrooms based on whether or not they have a green spore print.

To map these splits back to their corresponding categories, we can look at the `$map` component from the output of `ordinalize()` on each split variable:

```
sort(ordinalize(m$odor, m$Edibility)$map)

#>    almond     anise      none creosote     fishy      foul
#> 0.000 0.000 0.034 1.000 1.000 1.000
#>  musty  pungent    spicy
#> 1.000 1.000 1.000

sort(ordinalize(right[["spore.print.color"]],
                 y = right[["Edibility"]])$map)

#>    black     brown      buff chocolate     orange
#> 0.0000 0.0000 0.0000 0.0000 0.0000
#>  purple   yellow     white     green
#> 0.0000 0.0000 0.0769 1.0000
```

For example, the split point for `odor` was 0.517 (the midpoint between 0.034 and 1.00), and every feature mapped to a re-encoded `odor` value  $\geq 0.517$  is used to construct the first partition; see the first split in [Figure 2.10](#). In [Section 2.9.2](#), we'll verify these results (e.g., the computed gain for both splits) using CART-like software in R.

## 2.4.2 Be wary of categoricals with high cardinality

One drawback of CART-like decision trees is that they tend to favor categorical features with high cardinality (i.e., large  $J$ ), even if they are mostly irrelevant.<sup>h</sup> For categorical features with large  $J$ , for example, there are so many potential splits that the tree is more likely to find a good split just by chance. Think about the extreme case where a nominal feature  $x$  is different and unique in every row of the learning sample, like a row ID column. The split variable selection bias in CART-like decision trees has been discussed plenty in the literature; see, for example, Breiman et al. [1984, p. 42], Segal [1988], and Hothorn et al. [2006c] (and the additional references therein)

To illustrate the issue, I added ten random categorical features (`cat1–cat10`) to the `airquality` data set from [Section 2.3.1](#), each with a cardinality of  $J = 26$  (they're just random letters from the alphabet). A default regression tree was fit to the data using `rpart`, and the resulting tree diagram is displayed in [Figure 2.11](#). Notice that all of the splits, aside from the first, use the completely irrelevant categorical features that were added! In [Section 2.5](#) we'll look at a

---

<sup>h</sup>This bias actually extends to any predictor with lots of potential split points, whether ordered or nominal.

general *pruning* technique that can be helpful in screening out pure noise variables.

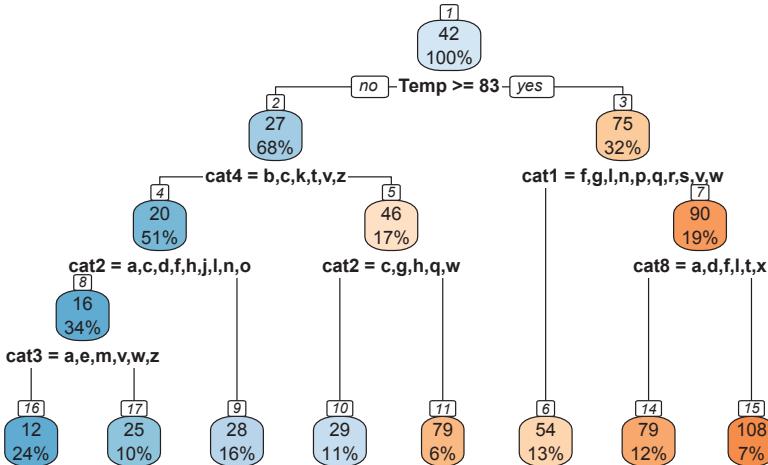


FIGURE 2.11: A decision tree fit to a copy of the air quality data set that includes ten completely random categorical features, each with cardinality 26.

In some cases, it's possible to reduce the number of potential categories to something more manageable—like lumping rare categories together, or combining categories into a smaller set of meaningful subgroups (e.g., combining zip or area codes into a smaller set of larger geographic areas).

The partitioning algorithms discussed in [Chapters 3–4](#) address the split selection bias issue more directly by separating the exhaustive search over all possible splits for each feature into two sequential steps, where the optimal split point is found only after a splitting variable has been selected.

### 2.4.3 To encode, or not to encode?

When dealing with categorical data, we are often concerned with how to encode such features. In linear models, for example, we often employ *dummy encoding* or *effect encoding*, depending on the task at hand. Similarly, *one-hot-encoding* (OHE), closely related to dummy encoding, is often used in general machine learning problems outside of (generalized) linear models. And there are plenty of other ways to encode categorical variables, depending on the algorithm and task at hand.

As you've already seen, decision trees can naturally handle variables of any type without special encoding, although we did see that a local form of mean/target encoding can be used to reduce the computational burden imposed by nominal categorical splits. Nonetheless, using an encoding strategy, like OHE, can sometimes improve the predictive performance or interpretability of a tree-based model; see Kuhn and Johnson [2013, Sec. 14.7] for a brief discussion on the use of OHE in tree-based methods. Further, some tree-based software, like Scikit-learn's `sklearn.tree` module, require all features to be numeric—forcing users to employ different encoding schemes for categorical features. See Boehmke and Greenwell [2020, [Chap. 3](#)] for details on different encoding strategies (with examples in R), and further references.

---

## 2.5 Building a decision tree

In the previous sections, we talked about the basics of splitting a node (i.e., partitioning some subset of the learning sample). Building a CART-like decision tree starts by splitting the root node, and then recursively applying the same splitting procedure to every resulting child node until a saturated tree is obtained (i.e., all terminal nodes are pure) or other stopping criteria are met. In essence, the partitioning stops when at least one of the following conditions are met:

- all the terminal nodes are pure;
- the specified maximum tree depth has been reached;
- the minimum number of observations that must exist in a node in order for a split to be attempted has been reached;
- no further splits are able to decrease the overall lack of fit by a specified factor;
- and so forth.

This often results in an overly complex tree structure that overfits the learning sample; that is, it has low bias, but high variance.

To illustrate, consider a random sample of size  $N = 500$ , generated from the following sine wave with Gaussian noise:

$$Y = \sin(X) + \epsilon,$$

where  $X \sim \mathcal{U}(0, 2\pi)$  and  $\epsilon \sim \mathcal{N}(0, \sigma = 0.3)$ . A scatterplot of the data, along with the true response function, is shown in [Figure 2.12](#).

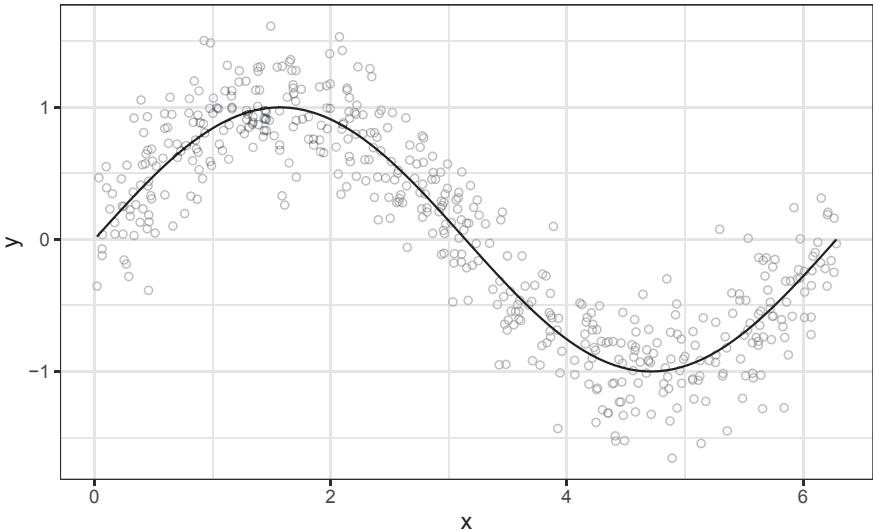


FIGURE 2.12: Data generated from a simple sine wave with Gaussian noise. The black curve shows the true mean response  $E(Y|X = x) = \sin(x)$ .

[Figure 2.13](#) shows the prediction function from two regression trees fit to the same data.<sup>i</sup> The tree on the left is too complex and has too many splits, and exhibits high variance, but low bias (i.e., it fits the current sample well, but the tree structure will vary wildly from one sample to the next because it's mostly fitting the noise here); unstable models, like this one are often referred to as *unstable learners* (more on this in [Section 5.1](#)). The tree on the right, which is a simple decision stump (i.e., a tree with only a single split), is too simple, and will also not be useful for prediction because it has extremely high bias, but low variance (i.e., it doesn't fit the data too well, but the tree structure will be more stable from sample to sample); such a weak performing model is often referred to as a *weak learner* (more on this in [Section 5.2](#)).

Neither tree is likely to be accurate when applied to a different sample from the same model; the ensemble methods discussed in [Part II](#) of this book can improve the performance of both weak and unstable learners. When using a single decision tree, however, the question we need to answer is, How complex should we make the tree? Ideally, we should have stopped splitting nodes at some *subtree* along the way, but where?

A rather careless approach is to build a tree by only splitting nodes that meet some threshold on prediction error. However, this is shortsighted because a low-quality split early on may lead to a very good split later in the tree. The standard approach to finding an optimal subtree—basically, determining when

---

<sup>i</sup>The associated tree diagrams are shown in the top left and bottom right of [Figure 2.14](#) (p. 73), respectively.

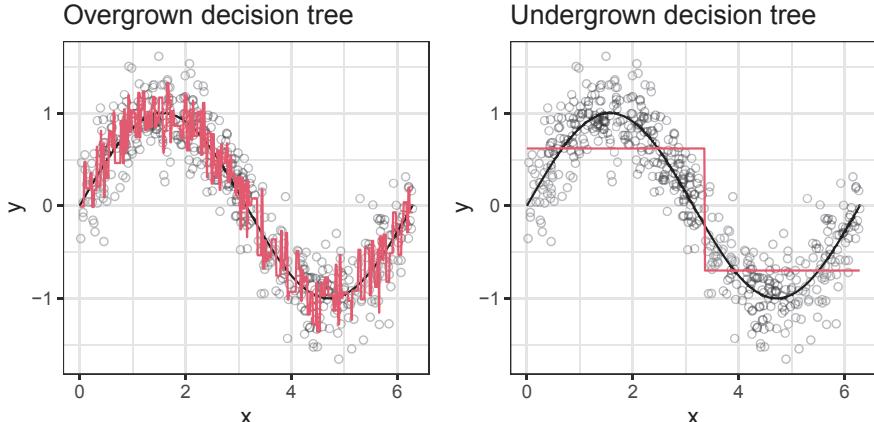


FIGURE 2.13: Regression trees applied to the sine wave example. Left: this tree is too complex (i.e., low bias and high variance). Right: this tree is too simple (i.e., high bias and low variance).

we should have stopped splitting nodes—is called *cost-complexity pruning*, or *weakest link pruning* [Breiman et al., 1984], or just pruning for short. Other pruning procedures are discussed in Ripley [1996, pp. 226–231] and Zhang and Singer [2010, pp. 44–49]. Pruning a decision tree is quite analogous to the process of *backward elimination* in multiple linear regression—start with a complex tree with too many splits, and *prune off* leaves whose contributions aren’t enough to offset the added complexity. The details are covered in the next section.

### 2.5.1 Cost-complexity pruning

The idea of pruning a decision tree is similar to the process of backward elimination in multiple linear regression. In essence, we build a large tree with too many splits, denoted  $\mathcal{T}_0$ , and then prune it back by collapsing internal nodes until we find some optimal subtree, denoted  $\mathcal{T}_{opt}$ , that meets a certain criterion, like having the smallest cross-validation error.

Let  $\{A_k\}_{k=1}^K$  be the terminal nodes of some tree  $\mathcal{T}$ , where  $|\mathcal{T}| = K$  is the number of terminal nodes, or size of  $\mathcal{T}$ . Recall that the overall goal of CART is to extract homogenous subgroups (i.e., terminal nodes). In this sense, the overall quality (or *risk*) of the tree depends on the quality of its terminal nodes. We define the risk of the tree to be  $R(\mathcal{T}) = \sum_{k=1}^K p(A_k) \times r(A_k)$ , where  $r(A_k)$  is some measure of the quality of the  $k$ -th terminal node; see (2.4) on page 55. For regression trees,  $R(\mathcal{T})$  is the error sum of squares (SSE). For classification trees based on the observed class priors and equal misclassification costs

(i.e.,  $L_{i,j} = 1$  for all  $i \neq j$ ),  $R(\mathcal{T})$  is simply the proportion of observations misclassified in the learning sample.

Building a tree to minimize  $R(\mathcal{T})$  will always lead to a saturated tree, resulting in a model with little or no bias but often high variance (i.e., overfitting the learning sample). Instead, we penalize the complexity (or size) of the tree by minimizing

$$R_\alpha(\mathcal{T}) = R(\mathcal{T}) + \alpha|\mathcal{T}|,$$

where  $\alpha \geq 0$  is a tuning parameter controlling the trade-off between the complexity of the tree,  $|\mathcal{T}|$ , and how well it fits the training data,  $R(\mathcal{T})$ . In this sense,  $R_\alpha(\mathcal{T})$  can be viewed as a penalized objective function similar to what's used in *regularized regression*; see, for example, Hastie et al. [2009, Chap. 3] or Boehmke and Greenwell [2020, Chap. 6]. When  $\alpha = 0$ , no penalty is incurred, resulting in the most complex tree  $\mathcal{T}_0$ . On the other extreme, we can always find a large enough value of  $\alpha$  that results in a decision tree with no splits (i.e., the root node). Choosing the right value of  $\alpha$  is important and can be done using cross-validation or other methods; a specific cross-validation approach is covered in [Section 2.5.2](#).

Breiman et al. [1984, Chap. 10] showed that for each  $\alpha$ , there exists a unique smallest subtree, denoted  $\mathcal{T}_\alpha$ , that minimizes  $R_\alpha(\mathcal{T})$ . This result is important because it guarantees that no two equally sized subtrees of  $\mathcal{T}_0$  will have the same value of  $R_\alpha(\mathcal{T})$ . To obtain  $\mathcal{T}_\alpha$ , start pruning  $\mathcal{T}_0$  by successively collapsing the internal node that produces the smallest per-node increase to  $R(\mathcal{T})$ , and continue until reaching the root node. This process results in a (finite) sequence of nested subtrees (see [Figure 2.14](#) on page 73 for an example) that contains  $\mathcal{T}_\alpha$ ; for details, see Breiman et al. [1984, Chap. 10] or Ripley [1996, Sec. 7.2].

To illustrate, take  $\mathcal{T}_0$  to be the left tree in [Figure 2.12](#), which has a total of 154 splits. The corresponding tree diagram is displayed in the top left of [Figure 2.14](#). The rest of the tree diagrams in [Figure 2.14](#) correspond to the last 15 trees in the pruning sequence (minus the root node), ending with a decision stump. The optimal subtree,  $\mathcal{T}_\alpha$ , which has a total of 20 splits (or 21 terminal nodes), was found using 10-fold cross-validation and is highlighted in green.

For comparison, I compared how each subtree performed on an independent test set of 500 new observations. For each subtree in the pruned sequence, the prediction error on the test set, measured as  $1 - R^2$ , where  $R^2$  is the squared Pearson correlation between the observed and fitted values, was computed. Both the test and cross-validation errors are displayed in [Figure 2.15](#). Here, the results are similar, but the test error suggests a slightly simpler tree with only 18 splits.

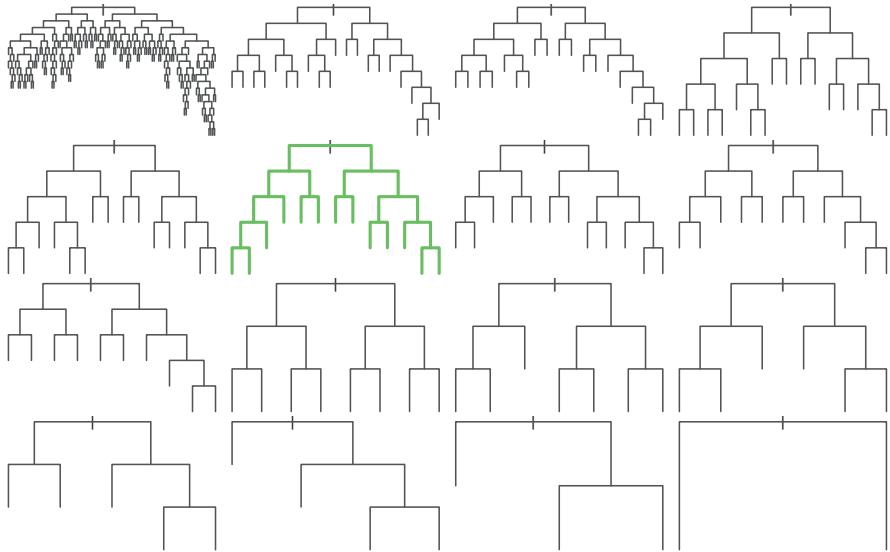


FIGURE 2.14: Nested subtrees for the sine wave example. The optimal subtree, chosen via 10-fold cross-validation, is highlighted in green.

So how is the sequence of  $\alpha$  values determined? For any internal node  $A$ , we can find  $\alpha$  using

$$\alpha = \frac{R(A) - R(\mathcal{T}_A)}{|\mathcal{T}_A| - 1},$$

where  $\mathcal{T}_A$  is the subtree rooted at node  $A$ . To start pruning, we need to find the first threshold value  $\alpha_1$ , which is just the smallest  $\alpha$  value among the  $|\mathcal{T}| - 1$  internal nodes of the tree  $\mathcal{T}$ . Once  $\alpha_1$  is obtained, we prune the tree by collapsing one of the  $|\mathcal{T}| - 1$  internal nodes and making it a terminal node whenever

$$\alpha_1 \geq \frac{R(A) - R(\mathcal{T}_A)}{|\mathcal{T}_A| - 1}.$$

This results in the optimal subtree,  $\mathcal{T}_{\alpha_1}$ , associated with  $\alpha = \alpha_1$ . Starting with  $\mathcal{T}_{\alpha_1}$ , we then continue this process by finding  $\alpha_2$  in the same way we found  $\alpha_1$  for the full tree  $\mathcal{T}$ . The process is continued until reaching the root node. It might sound confusing, but we'll walk through the calculations using the mushroom example in the next section.

The **rpart** package, which is used extensively throughout this chapter, employs a slightly friendlier, and rescaled, version of the cost-complexity parameter  $\alpha$ , which they denote as  $cp$ . Specifically, **rpart** uses

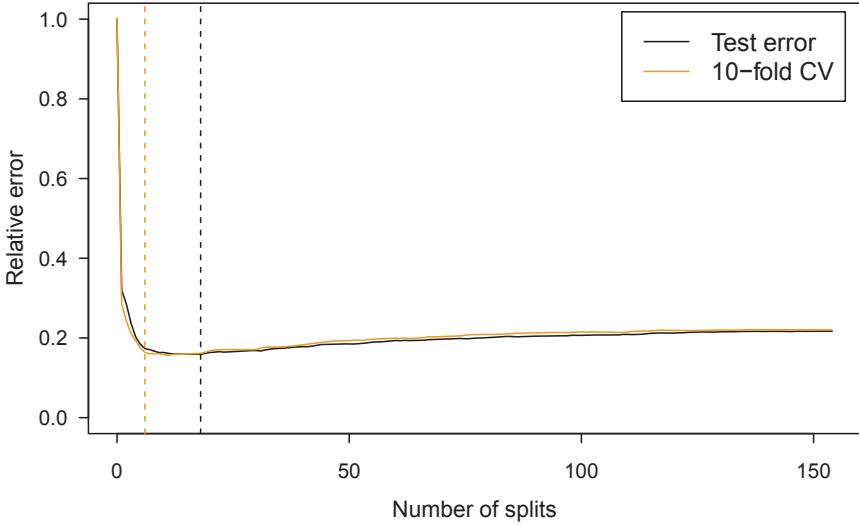


FIGURE 2.15: Relative error based on the test set (black curve) and 10-fold cross-validation (yellow curve) vs. the number of splits for the sine wave example. The vertical yellow line shows the optimal number of splits based on 10-fold cross-validation, while the vertical black line shows the optimal number of splits based on the independent test set.

$$R_{cp}(\mathcal{T}) \equiv R(\mathcal{T}) + cp \times |\mathcal{T}| \times R(\mathcal{T}_1),$$

where  $\mathcal{T}_1$  is the tree with zero splits (i.e., the root node). Compared to  $\alpha$ ,  $cp$  is unitless, and a value of  $cp = 1$  will always result in a tree with zero splits. The complexity parameter,  $cp$ , can also be used as a stopping rule during tree construction. In many open source implementations of CART, whenever  $cp > 0$ , any split that does not decrease the overall lack of fit by a factor of  $cp$  is not attempted. In a regression tree, for instance, this means that the overall  $R^2$  must increase by  $cp$  at each step for a split to occur. The main idea is to reduce computation time by avoiding potentially unworthy splits. However, this runs the risk of not finding potentially much better splits further down the tree.

### 2.5.1.1 Example: mushroom edibility

Let's drive the main ideas home by calculating a few  $\alpha$  values to prune a simple tree for the mushroom edibility data. Consider again a simple decision tree for the mushroom edibility data which is displayed in Figure 2.16. This is a simple tree with only three splits, but we'll use it to illustrate how

pruning works and how the sequence of  $\alpha$  values is computed. For clarity, the number of observations in each class is displayed within each node, and the node numbers appear at the top of each node. For example, node 8 contains 4208 edible mushrooms and 24 poisonous ones. The assigned classification, or majority class, is printed above the class frequencies in each node. This tree was also built using the observed class priors and equal misclassification costs; hence,  $R(\mathcal{T})$  is just the proportion of misclassifications in the learning sample:  $24/8124 \approx 0.003$ .

Let  $A_i$ ,  $i \in \{1, 2, 3, 4, 5, 8, 9\}$  denote the seven nodes of the tree in Figure 2.16; in **rpart**, the left and right child nodes for any node numbered  $x$  are always numbered  $2x$  and  $2x+1$ , respectively (the root node always corresponds to  $x = 1$ ). We can compute the risk of any terminal node using  $R(A_i) = N_{j,A}/N_A$ . For example, nodes  $A_5 - A_7$  all have a risk of zero (since they are pure nodes).

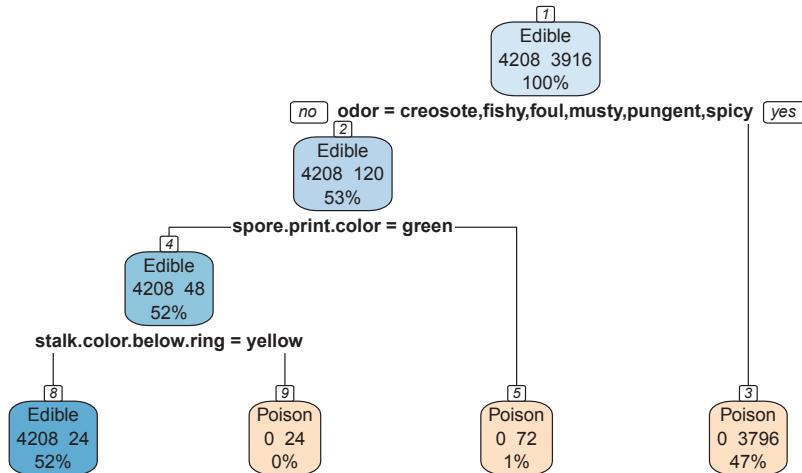


FIGURE 2.16: Classification tree with three splits for the mushroom edibility data. The overall risk of the tree is  $24/8124 \approx 0.003$ .

To find  $\alpha_1$ , we need to first compute  $\alpha$  for each of the  $|\mathcal{T}_0| - 1 = 3$  internal nodes of the tree, and find which one is the smallest; use the tree diagram in Figure 2.16 to follow along. The  $\alpha$  values for the three internal nodes are computed as follows:

$$\alpha_{A_1} = (3916/8124 - 24/8124) / (4 - 1) \approx 0.160$$

$$\alpha_{A_2} = (120/8124 - 24/8124) / (3 - 1) \approx 0.006 .$$

$$\alpha_{A_4} = (48/8124 - 24/8124) / (2 - 1) \approx 0.003$$

Since  $\alpha_{A_4}$  is the smallest, we collapse node  $A_4$ , resulting in the next optimal subtree in the sequence,  $\mathcal{T}_{\alpha_1}$ , which is displayed in the left side of Figure 2.17. The cost-complexity of this tree is  $R_{\alpha_1}(\mathcal{T}_{\alpha_1}) = 0.015$ . To find  $\alpha_2$ , we start with  $\mathcal{T}_{\alpha_1}$  and repeat the process by first finding the smallest  $\alpha$  value associated with the  $|\mathcal{T}_{\alpha_1}| - 1 = 2$  internal nodes of  $\mathcal{T}_{\alpha_1}$ . These are given by

$$\begin{aligned}\alpha_{A_1} &= (3916/8124 - 48/8124) / (3 - 1) \approx 0.238 \\ \alpha_{A_2} &= (120/8124 - 48/8124) / (2 - 1) \approx 0.009\end{aligned},$$

making  $\alpha_2 = 0.009$ . We would then prune the current subtree,  $\mathcal{T}_{\alpha_2}$ , by collapsing  $A_2$  into a terminal node, resulting in the decision stump displayed in the right side of Figure 2.17. This makes only one possibility for  $\alpha_3 = (3916/8124 - 120/8124) / (2 - 1) \approx 0.467$ , which results in the root node after pruning the decision stump,  $\mathcal{T}_{\alpha_3}$ . In the end, we have the following sequence of  $\alpha$  values:  $(\alpha_1 = 0.003, \alpha_2 = 0.009, \alpha_3 = 0.467)$ . In practice, we would use cross-validation, or some other validation procedure, to select a reasonable value of the complexity parameter  $\alpha$  from this sequence. The next two sections discuss choosing  $\alpha$  using  $k$ -fold cross-validation.

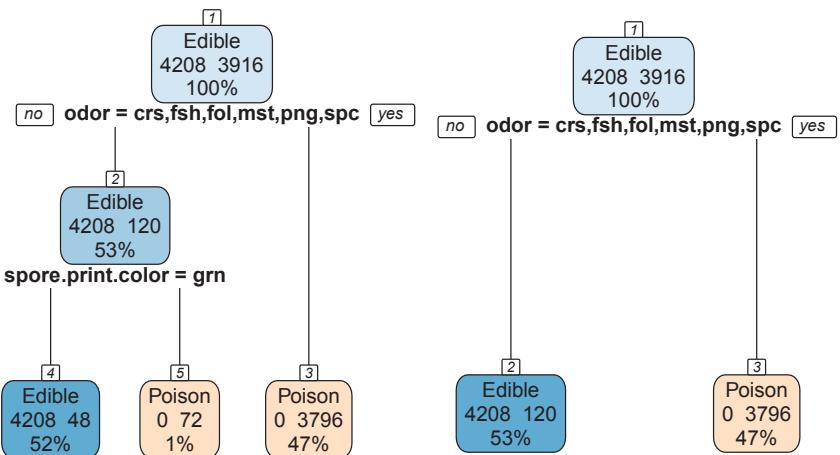


FIGURE 2.17: Optimal subtrees in the sequence with minimum cost-complexity. Since the original tree contains only three splits, there are only two possible subtrees, not counting the tree with zero splits. Here the category names have been truncated to three letters to fit more compactly in the display.

### 2.5.2 Cross-validation

Once the sequence  $\alpha_1, \alpha_2, \dots, \alpha_{k-1}$  has been found, we still need to estimate the overall risk/quality of the corresponding sequence of nested subtrees,  $R_{\alpha_i}(\mathcal{T})$ , for  $i = 1, 2, \dots, k-1$ . Breiman et al. [1984, Chap. 11] suggested picking  $\alpha$  using a separate validation set or  $k$ -fold cross-validation. The latter is more computational, but tends to be preferred since it makes use of all available data, and both tend to lead to similar results. The procedure described in Algorithm 2.1 below follows the implementation in the **rpart** package in R (see the “Introduction to Rpart” vignette):

---

**Algorithm 2.1**  $K$ -fold cross-validation for cost-complexity pruning.

---

- 1) Fit the full model to the learning sample to obtain  $\alpha_1, \alpha_2, \dots, \alpha_{k-1}$ .
- 2) Define  $\beta_i$  according to

$$\beta_i = \begin{cases} 0 & i = 1 \\ \sqrt{\alpha_{i-1}\alpha_i} & i = 2, 3, \dots, m-1 \\ \infty & i = m \end{cases}.$$

Since any value of  $\alpha$  in the interval  $(\alpha_i, \alpha_{i+1}]$  results in the same subtree, we instead consider the sequence of  $\beta_i$ 's, which represent typical values within each range using the geometric midpoint.

- 3) Divide the data into  $k$  groups (or folds),  $D_1, D_2, \dots, D_k$ , with approximately  $k/N$  observations in each ( $N$  being the number of rows in the learning sample). For  $i = 1, 2, \dots, k$ , do the following:
    - a) Fit the full model to the learning sample, but omit the subset  $D_i$ , and find the sequence of optimal subtrees  $\mathcal{T}_{\beta_1}, \mathcal{T}_{\beta_2}, \dots, \mathcal{T}_{\beta_k}$ .
    - b) Compute the prediction error from each tree on the validation set  $D_i$ .
  - 4) For each subtree, aggregate the results by averaging the  $k$  out-of-sample prediction errors.
  - 5) Return  $\mathcal{T}_\beta$  from the initial sequence of trees based on the full learning sample, where  $\beta$  corresponds to the  $\beta_i$  associated with the smallest prediction error in step 4).
-

### 2.5.2.1 The 1-SE rule

When choosing  $\alpha$  with  $k$ -fold cross-validation, Breiman et al. [1984, Sec. 3.4.3] recommend using the *1-SE rule*, and argue that it is useful in screening out irrelevant features. The 1-SE rule suggests using the most parsimonious tree (i.e., the one with fewest splits) whose cross-validation error is no more than one standard error above the cross-validation error of the best model. This of course requires an estimate of the standard error during cross-validation. A heuristic estimate of the standard error can be found in Breiman et al. [1984, pp. 306–309] or Zhang and Singer [2010, pp. 42–43], but the formula isn’t pretty! Applying cost-complexity pruning using cross-validation, with or without the 1-SE rule, would almost surely remove all of the nonsensical splits seen in [Figure 2.11](#). (In fact, this was the case after applying 10-fold cross-validation using the 1-SE rule.)

---

## 2.6 Hyperparameters and tuning

There are essentially three hyperparameters associated with CART-like decision trees:

- 1) the maximum depth or number of splits;
- 2) the maximum size of any terminal node;
- 3) the cost-complexity parameter  $cp$ .

Different software will have different names for these parameters and different default values. Arguably,  $cp$  is the most flexible and important tuning parameter in CART, and a good strategy is to relax the maximum depth and size of the terminal nodes as much as possible, and use cost-complexity pruning to find an optimal subtree using  $k$ -fold cross-validation, or some other validation procedure. In some cases, [Chapter 7](#), for example, trees are intentionally grown to maximal or near maximal depth (in some cases, leaving only a single observation in each terminal node).

---

## 2.7 Missing data and surrogate splits

One of the best features of CART is the flexibility with which missing values can be handled. More traditional statistical models, like linear or logistic regression, will often discard any observations with missing values. CART,

through the use of *surrogate splits*, can utilize all observations that have non-missing response values and at least one non-missing value for the predictors. Surrogate splits are essentially splits using other available features with non-missing values. The basic idea, which is fully described in Breiman et al. [1984, Sec. 5.3], is to estimate (or *impute*) the missing data point using the other available features.

Consider the decision stump in Figure 2.18, which corresponds to the optimal tree for the Swiss banknote data when using all available features.

What if we wanted to classify a new observation which had a missing value for `diagonal`? The surrogate approach finds *surrogate variables* for the missing splitter by building decision stumps, one for each of the other features (in this case, `length`, `left`, `right`, `bottom`, and `top`), to predict the binary response, denoted below by  $y^*$ , formed by the original split:

$$y^* = \begin{cases} 0 & \text{if } \text{diagonal} \geq 140.65 \\ 1 & \text{if } \text{diagonal} < 140.65 \end{cases}.$$

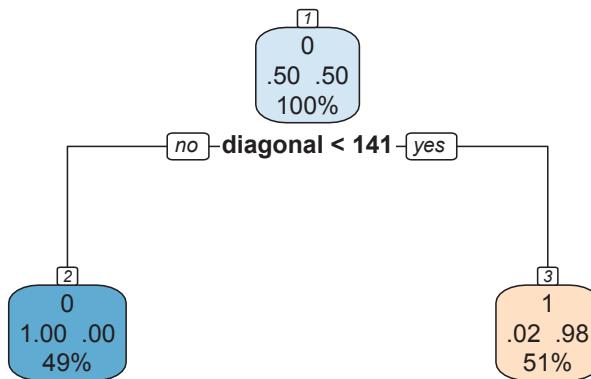


FIGURE 2.18: Decision stump for the Swiss banknote example.

For each feature, the optimal split is chosen using the procedure described in Section 2.2.1. (Note that when looking for surrogates, we do not bother to incorporate priors or losses since none are defined for  $y^*$ .) In addition to the optimal split for each feature, we also consider the *majority rule*, which just uses the majority class. Once the surrogates have been determined, they're ranked in terms of misclassification error, and any surrogate that does worse

than the majority class is discarded. Some implementations, like R's **rpart** package, further require surrogate splits to send at least two observations to each of the left and right child nodes.

Returning to the Swiss banknote example, let's find the surrogate splits for the primary split on **diagonal** depicted in [Figure 2.18](#). We can find the surrogate splits using the same splitting process as before, albeit with our new target variable  $y^*$ :

```
bn2 <- treemisc::banknote # load Swiss banknote data
bn2$y <- ifelse(bn2$diagonal >= 140.65, 1, 0) # new target
bn2$diagonal <- NULL # remove column
features <- c("length", "left", "right", "bottom", "top")
res <- sapply(features, FUN = function(feature) {
  find_best_split(bn2, x = feature, y = "y", n = nrow(bn2))
})
rownames(res) <- c("cutpoint", "gain")
res[, order(res[["gain"]], decreasing = TRUE)]
#>      bottom    right    top    left   length
#> cutpoint  9.550 129.850 10.950 130.050 215.1500
#> gain       0.343   0.169   0.157   0.137   0.0344
```

In this case, the ranked surrogate splits—in descending order of importance—are **bottom**, **right**, **top**, **left**, and **length**; the corresponding split points for each are also shown in the output (we'll verify these results in [Section 2.9](#) using real tree software). If we were to use the decision stump in [Figure 2.18](#) to classify a new banknote with a missing value for **diagonal**, the tree would use the next best surrogate split instead (in this case, whether or not **bottom**  $\geq 9.550$ ).

For each surrogate, we could also compute its *agreement* and *adjusted agreement* with the primary split, which are used in **rpart**'s definition of *variable importance* ([Section 2.8](#)). The agreement between a primary split and its surrogate is just the proportion of observations they send in the same direction. The adjusted agreement adjusts this proportion by subtracting the number of observations sent one way or another using the majority rule. An example is given in [Section 2.9](#).

### 2.7.1 Other missing value strategies

Aside from being able to handle missing predictor values directly, classification trees can be extremely useful in examining patterns of missing data [Harrell, 2015, [Sec. 3.2](#)]. For example, CART can be used to describe observations that tend to have missing values (a description problem). This can be done by growing a classification tree using a target variable that's just a binary

indicator for whether or not a variable of interest is missing; see Harrell [2015, pp. 302–304] for an example using real data in R.

It can also be informative to construct missing value indicators for each predictor under consideration. Imagine, for example, that you work for a bank and that part of your job is to help determine who should be denied for a loan and who should not. A missing credit score on a particular loan application might be an obvious red flag, and indicative of somebody with a bad credit history, hence, an important indicator in determining whether or not to approve them for a loan. A similar strategy for categorical variables is to treat missing values as an actual category. As noted in van Buuren [2018, Sec. 1.3.7], the missing value indicator method may have its uses in particular situations but fails as a generic method to handle missing data (e.g., it does not allow for missing data in the response and can lead to biased regression estimates across a wide range of scenarios).

*Imputation*—filling in missing values with a reasonable guess—is another common strategy, and trees make great candidates for imputation models (e.g., they’re fully nonparametric and naturally support both classification and regression).

Using CART for the purpose of missing value imputation has been suggested by several authors; see van Buuren [2018, Sec. 3.5] for details and several references. A generally useful approach is to use CART to generate *multiple imputations* [van Buuren, 2018, Sec. 3.5] via the *bootstrap* method<sup>j</sup> (see Davison and Hinkley [1997] for an overview of different bootstrap methods); multiple imputation is now widely accepted as one of the best general methods for dealing with incomplete data [van Buuren, 2018, Sec. 2.1.2].

The basic steps are outlined in Algorithm 2.2; also see `?mice::cart` for details on its implementation in the **mice** package [van Buuren and Groothuis-Oudshoorn, 2021]. Here, it is assumed that the response  $y$  corresponds to the predictor with incomplete observations (i.e., contains missing values) and that the predictors correspond to the original predictors with complete information (i.e., no missing values).

As described in Doove et al. [2014] and van Buuren [2018, Sec. 3.5], this process can be repeated  $m$  times using the bootstrap to produce  $m$  imputed data sets. As noted in van Buuren [2018, Sec. 3.5], Algorithm 2.2 is a form of *predictive mean matching* [van Buuren, 2018, Sec. 3.4], where the “predictive mean” is instead calculated by CART, as opposed to a regression model. An example using CART for multiple imputation is provided in Section 7.9.3.

But what if you’re using a decision tree as the model, and not just as a means for imputation: should you rely on surrogate splits or a different strategy,

---

<sup>j</sup>Unless stated otherwise, a bootstrap sample refers to a random sample of size  $N$  with replacement from a set of  $N$  observations; hence, some of the original observations will be sampled more than once and some not at all.

---

**Algorithm 2.2** Tree-based missing value imputation.

- 
- 1) fit a decision tree (e.g., CART) to the complete observations;
  - 2) find the terminal assigned to each observation with a missing  $y$  value;
  - 3) for each missing  $y$  value, randomly draw an observed response value from the terminal node to which it's assigned (i.e., the complete response values from the learning sample that summarize the node) to use for the imputed value.
- 

like imputation? Feelders [2000] suggests that imputation (especially multiple imputation), if done properly, tends to outperform trees based on surrogate splits. However, one should still consider whether or not the potential for improved performance outweighs the additional effort required in specifying an appropriate imputation scheme. Feelders further notes that with "...moderate amounts of missing data (say 10% or less) one can avoid generating imputations and just use surrogate splits."

---

## 2.8 Variable importance

In practice, it may be useful, or even necessary, to reduce the number of features in a model. One way to accomplish this is to rank them in some order of importance and use a subset of the top features. Loh [2012] showed that using a subset of only the important variables can lead to increased prediction accuracy. Reducing the number of features can also decrease model training time and increase interpretability. However, lack of a proper definition of "importance" has led to many variable importance measures being proposed; see Greenwell and Boehmke [2020] for some discussion and further references.

Decision trees probably offer the most natural model-specific approach to quantifying the importance of predictors. In a binary decision tree, at each node  $A$ , a single predictor is used to partition the data into two homogeneous groups. The chosen predictor is the one that maximizes (2.3). The relative importance of predictor  $x$  is the sum of the squared improvements over all internal nodes of the tree for which  $x$  was chosen as the primary splitter; see Breiman et al. [1984] for details. This idea also extends to regression trees and ensembles of decision trees, such as those discussed in [Chapters 5–8](#).

When surrogate splits are enabled, they can be accounted for in the quan-

tification of variable importance. In particular, a variable may appear in the tree more than once, either as a primary or surrogate splitter. The variable importance measure for a feature is the sum of the gains associated with each split for which it was the primary variable, plus the gains (adjusted for agreement) associated with each split for which it was a surrogate. The notation is a bit involved, but the interested reader is pointed to Loh and Zhou [2021, Sec. 3].

Including surrogate information can help improve interpretation when you have strongly correlated or redundant features. For instance, imagine two features  $x_1$  and  $x_2$  that are essentially redundant. If we only counted gains where each variable was a primary splitter, these two features would likely split the importance, with neither showing up as important as they should. Comparing the variable importance rankings with and without competing splitters (i.e., non-primary splitters) can also be informative. Variables that appear to be important, but rarely split nodes, are probably highly correlated with the primary splitters and contain very similar information.

The relative variable importance standardizes the raw importance values for ease of interpretation. The relative importance is typically defined as the percent improvement with respect to the most important predictor, and is often reported in statistical software. The relative importance of the most important feature is always 100%. So, if  $x_3$  is the most important feature, and the relative importance of another feature, say  $x_5$ , is 83%, you can say that  $x_5$  is roughly 83% as important as  $x_3$ .

It is well known, however, that CART-like variable importance scores are biased; see Loh and Zhou [2021] for a thorough (and more recent) review. According to Loh and Zhou, a variable importance procedure is said to be unbiased if all predictors have the same mean importance score when they are independent of the response. Solutions to CART's variable importance bias, which really stems from CART's split selection bias (Section 2.4.2), are discussed in several places throughout this book; see, for example, the discussion in Section 7.5.1.

---

## 2.9 Software and examples

Packages **rpart** and **tree** [Ripley, 2021] provide modern implementations of the CART algorithm in R, although **rpart** is recommended over **tree**, and so we won't be discussing the latter. The name **rpart** comes from the acronym for (binary) Recursive PARTitioning. Beyond simple classification and regression trees, **rpart** can also be used to model Poisson counts (e.g., the number of occurrences of some event per unit of time), and censored outcomes. Note

that **rpart** is extendable<sup>k</sup> and several R packages on CRAN extend **rpart** in various ways. For example, **rpartScore** [Galimberti et al., 2012] can be used to build classification trees for ordinal responses within the same CART-like framework, and **rpart.LAD** [Dlugosz, 2020] can be used to fit regression trees based on *least absolute deviation* [Breiman et al., 1984, Sec. 8.11]. The **treemisc** package provides some utility functions to support **rpart**, for example, to implement pruning based on the 1-SE rule (Section 2.5.2.1). The R package **treevalues** [Neufeld, 2022] can be used to construct confidence intervals and *p*-values for the mean response within a node or the difference in mean response between two nodes in a CART-like regression tree (built using the package **rpart**); see [Neufeld et al., 2021] for details.

CART-like decision trees are implemented in many other open source languages as well. Scikit-learn’s **sklearn.tree** module offers extensive decision tree functionality, but doesn’t support categorical features, unless they’ve been numerically re-encoded. The **DecisionTree.jl** package<sup>l</sup> for Julia provides an implementation of CART and random forest (Chapter 7), but is rather limited in terms of features, especially when compared to R and Python’s tree libraries. Decision trees are also implemented in Spark MLlib [Meng et al., 2016], Spark’s open-source distributed machine learning library.<sup>m</sup>

The following examples illustrate the basic use of **rpart** for building decision trees. We’ll confirm the results we computed manually in previous sections as well as construct decision trees for new data sets.

An excellent case study using decision trees in R to identify email spam is provided in Nolan and Lang [2015, Chapter 3].

### 2.9.1 Example: Swiss banknotes

In Section 2.2.2, I restricted my attention to just two predictors, **top** and **bottom**, and walked through the steps of constructing a two-split tree by hand (i.e., a tree with three terminal nodes). Here, I’ll use the **rpart** package to reconstruct the same tree and to confirm my previous split calculations.

By default, **rpart** uses the Gini splitting rule, equal misclassification costs, and the observed class priors<sup>n</sup> when building a classification tree; hence, we do not need to set any additional arguments (we’ll do that in the next section).

---

<sup>k</sup>As described in the “User Written Split Functions” vignette; see `vignette("usercode", package = "rpart")` for details.

<sup>l</sup><https://github.com/bensadeghi/DecisionTree.jl>.

<sup>m</sup>Spark has various interfaces, including R and Python; an example using R will be given in Section 7.9.5.

<sup>n</sup>Note that the balanced nature of these data is not very realistic, unless roughly half the Swiss banknotes truly are counterfeit. However, without any additional information about the true class priors, there’s not much that can be done here.

However, for ease of interpretation, I'll re-encode the outcome  $y$  from 0/1 to Genuine/Counterfeit<sup>o</sup>:

```
library(rpart)

# Load the Swiss banknote data and re-encode the response
bn <- banknote
bn$y <- ifelse(bn$y == 0, "Genuine", "Counterfeit")

# Fit a CART-like tree using top and bottom as the only features
(bn.tree <- rpart(y ~ top + bottom, data = bn))

#> n= 200
#>
#> node), split, n, loss, yval, (yprob)
#>      * denotes terminal node
#>
#> 1) root 200 100 Counterfeit (0.5000 0.5000)
#>    2) bottom>=9.55 88    2 Counterfeit (0.9773 0.0227) *
#>    3) bottom< 9.55 112   14 Genuine (0.1250 0.8750)
#>      6) top>=11.1 17    4 Counterfeit (0.7647 0.2353) *
#>      7) top< 11.1 95    1 Genuine (0.0105 0.9895) *
```

Note that this is the same tree that was displayed in [Figure 2.2](#) (p. 43). The output from printing an "`rpart`" object can seem intimidating at first, especially for large trees, so let's take a closer look. The output is split into three sections. The first section gives  $N$ , the number of rows in the learning sample (or root node). The middle section, starting with `node`), indicates the format of the tree structure that follows. The last section, starting at 1), provides a brief summary of the tree structure. All the nodes of the tree are numbered, with 1) indicating the root node and lines ending with a \* indicating the terminal nodes. The topology of the tree is conveyed through indented lines; for example, nodes 2) and 3) are nested within 1); the left and right child nodes for any node numbered  $x$  are always numbered  $2x$  and  $2x + 1$ , respectively.

For each node we can also see the split that was used, the number of observations it captured, the deviance or loss (in this case, the number of observations misclassified in that node), the fitted value (in this case, the classification given to observations in that node), and the proportion of each class in the node. Take node 2), for example. This is a terminal node, the left child of node 1), and contains 88 of the  $N = 200$  observations (two of which are genuine banknotes). Any observation landing in node 2) will be classified as counterfeit with a predicted probability of 0.977.

---

<sup>o</sup>I could leave the response numerically encoded as 0/1, but then I would need to tell `rpart` to treat this as a classification problem by setting `method = "class"` in the call to `rpart()`.

If you want even more verbose output, with details about each split, you can use the **summary()** method:

```
summary(bn.tree) # print more verbose tree summary

##> Call:
##> rpart(formula = y ~ top + bottom, data = bn)
##> n= 200
##>
##>      CP nsplit rel error xerror   xstd
##> 1 0.84      0     1.00   1.14 0.0700
##> 2 0.09      1     0.16   0.19 0.0415
##> 3 0.01      2     0.07   0.12 0.0336
##>
##> Variable importance
##> bottom    top
##>    66     34
##>
##> Node number 1: 200 observations,      complexity param=0.84
##> predicted class=Counterfeit  expected loss=0.5  P(node) =1
##>   class counts:   100   100
##>   probabilities: 0.500 0.500
##>   left son=2 (88 obs) right son=3 (112 obs)
##> Primary splits:
##>   bottom < 9.55 to the right, improve=71.6, (0 missing)
##>   top    < 11   to the right, improve=30.7, (0 missing)
##> Surrogate splits:
##>   top < 11   to the right, agree=0.685, adj=0.284, (0 split)
##>
##> Node number 2: 88 observations
##> predicted class=Counterfeit  expected loss=0.0227  P(node) =0.44
##>   class counts:    86     2
##>   probabilities: 0.977 0.023
##>
##> Node number 3: 112 observations,      complexity param=0.09
##> predicted class=Genuine    expected loss=0.125  P(node) =0.56
##>   class counts:    14     98
##>   probabilities: 0.125 0.875
##>   left son=6 (17 obs) right son=7 (95 obs)
##> Primary splits:
##>   top    < 11.1 to the right, improve=16.40, (0 missing)
##>   bottom < 9.25 to the right, improve= 2.42, (0 missing)
##>
##> Node number 6: 17 observations
##> predicted class=Counterfeit  expected loss=0.235  P(node) =0.085
##>   class counts:    13     4
##>   probabilities: 0.765 0.235
##>
##> Node number 7: 95 observations
##> predicted class=Genuine    expected loss=0.0105  P(node) =0.475
```

```
#>     class counts:    1    94
#>     probabilities: 0.011 0.989
```

Here, we can see each primary splitter, along with its corresponding split point and gain (i.e., a measure of the quality of the split). For example, using `bottom < 9.55` yielded the greatest improvement and was selected as the first primary split. The reported improvement (`improve=71.59091`) is  $N \times \Delta\mathcal{I}(s, A)$ , hence why it differs from the output of our previously defined `splits()` function, which just uses  $\Delta\mathcal{I}(s, A)$ ; but you can check the math:  $71.59091/200 = 0.358$ , which is the same value we obtained by hand back in [Section 2.2.2](#). Woot!

Before continuing, let's refit the tree using all available features:

```
summary(rpart(y ~ ., data = bn, method = "class"))

#> Call:
#> rpart(formula = y ~ ., data = bn, method = "class")
#> n= 200
#>
#>      CP nsplit rel error xerror   xstd
#> 1 0.98      0     1.00   1.12 0.0702
#> 2 0.01      1     0.02   0.03 0.0172
#>
#> Variable importance
#> diagonal    bottom    right     top     left   length
#>      28       22      15      14      14       6
#>
#> Node number 1: 200 observations,      complexity param=0.98
#>   predicted class=Counterfeit  expected loss=0.5  P(node) =1
#>   class counts:   100    100
#>   probabilities: 0.500 0.500
#>   left son=2 (102 obs) right son=3 (98 obs)
#> Primary splits:
#>   diagonal < 141  to the left,  improve=96.1, (0 missing)
#>   bottom   < 9.55 to the right, improve=71.6, (0 missing)
#>   right    < 130  to the right, improve=34.3, (0 missing)
#>   top      < 11   to the right, improve=30.7, (0 missing)
#>   left     < 130  to the right, improve=27.8, (0 missing)
#> Surrogate splits:
#>   bottom < 9.25 to the right, agree=0.910, adj=0.816, (0 split)
#>   right  < 130  to the right, agree=0.785, adj=0.561, (0 split)
#>   top    < 11   to the right, agree=0.765, adj=0.520, (0 split)
#>   left   < 130  to the right, agree=0.760, adj=0.510, (0 split)
#>   length < 215  to the left,  agree=0.620, adj=0.224, (0 split)
#>
#> Node number 2: 102 observations
#>   predicted class=Counterfeit  expected loss=0.0196  P(node) =0.51
#>   class counts:   100      2
```

```
#>   probabilities: 0.980 0.020
#>
#> Node number 3: 98 observations
#>   predicted class=Genuine      expected loss=0  P(node) =0.49
#>   class counts:    0    98
#>   probabilities: 0.000 1.000
```

Using all the predictors results in the same decision stump that was displayed in [Figure 2.18](#). As it turns out, the best tree uses a single split on the length of the diagonal (in mm) and only misclassifies two of the genuine banknotes in the learning sample. In addition to the chosen splitter, `diagonal`, we also see a description of the competing splits (four by default) and surrogate splits (five by default); note that these match the surrogate splits I found manually back in [Section 2.7](#). For example, if I didn't include `diagonal` as a potential feature, then `bottom` would've been selected as the primary splitter because it gave the next best reduction to weighted impurity (`improve=71.59091`).

While the `rpart` package provides `plot()` and `text()` methods for plotting and labeling tree diagrams, respectively, the resulting figures are not as polished as those produced by other packages; for example, `rpart.plot` [Milborrow, 2021b] and `partykit` [Hothorn and Zeileis, 2021]. All the tree diagrams in this chapter were constructed using a simple wrapper function around `rpart.plot()` called `tree_diagram()`, which is part of `treemisc`; see `?rpart.plot::rpart.plot` and `?treemisc::tree_diagram` for details. For example, the tree diagram from [Figure 2.2](#) (p. 43) can be constructed using:

```
treemisc::tree_diagram(bn.tree)
```

[Figure 2.19](#) shows a tree diagram depicting the primary split (left) as well as the second best surrogate split (right). In the printout from `summary()`, we also see the computed agreement and adjusted agreement for each surrogate. From [Figure 2.19](#), we can see that the surrogate sends  $(66 + 91) / 200 \approx 0.785$  of the observations in the same direction as the primary split (agreement). The majority rule gets 102 correct, giving an adjusted agreement of  $(66 + 91 - 102) / (200 - 102) \approx 0.561$ .

### 2.9.2 Example: mushroom edibility

In this section, we'll use `rpart` to fit a classification tree to the mushroom data, and explore a bit more of the output and fitting process. Recall from [Section 1.4.4](#), that the overall objective is to find a simple rule of thumb (if possible) for avoiding potentially poisonous mushrooms. For now, I'll stick with `rpart`'s defaults (e.g., the splitting rule is the Gini index), but set com-

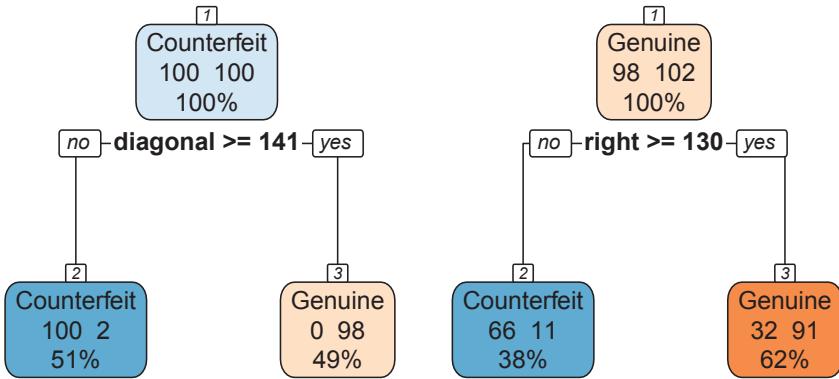


FIGURE 2.19: Decision stump for the Swiss banknote example (left) and one of the surrogate splits (right).

plexity parameter,  $cp$ , to zero ( $cp = 0$ ) for a more complex tree.<sup>P</sup> Although the tree construction itself is not random, the internal cross-validation results are, so I'll also set the random number seed before calling `rpart()`:

```
mushroom <- treemisc::mushroom

# Fit a default tree with zero penalty on tree size
set.seed(1054) # for reproducibility
(mushroom.tree <- rpart(Edibility ~ ., data = mushroom, cp = 0))

##> n= 8124
##>
##> node), split, n, loss, yval, (yprob)
##>      * denotes terminal node
##>
##> 1) root 8124 3920 Edible (0.51797 0.48203)
##>    2) odor=almond,anise,none 4328  120 Edible (0.97227 0.02773)
##>      4) spore.print.color=black,brown,buff,chocolate,orange,purple...
##>        8) stalk.color.below.ring=brown,gray,orange,pink,red,white ...
##>        16) stalk.color.below.ring=gray,orange,pink,red,white 4152...
##>          32) habitat=grasses,meadows,paths,urban,waste,woods 3952...
##>          33) habitat=leaves 200     8 Edible (0.96000 0.04000)
##>            66) cap.surface=smooth 192     0 Edible (1.00000 0.0000...
##>            67) cap.surface=grooves,scaly 8     0 Poison (0.00000 1...
```

<sup>P</sup>The default setting in `rpart` is  $cp = 0.01$ .

```
#> 17) stalk.color.below.ring=brown 80    16 Edible (0.80000 0...
#> 34) stalk.root=bulbous 64      0 Edible (1.00000 0.00000) *
#> 35) stalk.root=missing 16      0 Poison (0.00000 1.00000) *
#> 9) stalk.color.below.ring=yellow 24      0 Poison (0.00000 1.0...
#> 5) spore.print.color=green 72      0 Poison (0.00000 1.00000) *
#> 3) odor=creosote,fishy,foul,musty,pungent,spicy 3796      0 Poiso...
```

This is a complex tree with many splits, so let's use `treemisc`'s `tree_diagram()` function to plot it (see Figure 2.20).

```
tree_diagram(mushroom.tree) # Figure 2.20
```

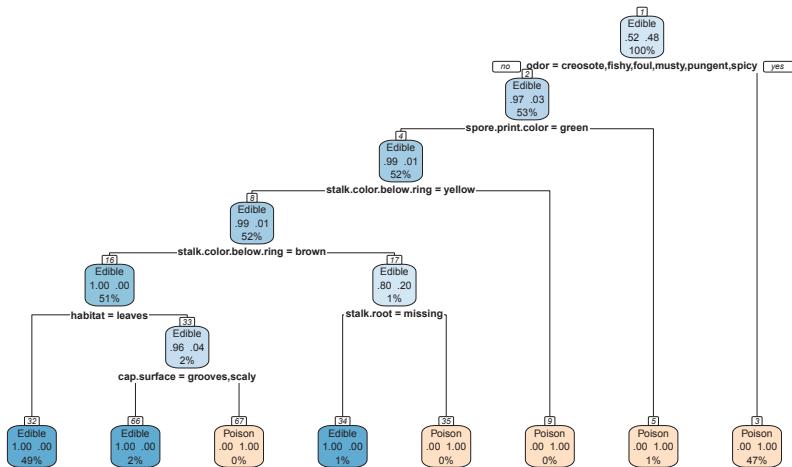


FIGURE 2.20: Decision tree diagram for classifying the edibility of mushrooms.

Setting `cp = 0` won't necessarily result in the most complex or saturated tree. This is because `rpart()` sets a number of additional parameters by default, many of which help control the maximum size of the tree; these are printed below for the current `mushroom.tree` object. For instance, `minsplit`, which defaults to 20, controls the number of observations that must exist in a node before a split can be attempted.

```
unlist(mushroom.tree$control)

#>      minsplit      minbucket          cp
#>        20            7            0
#> maxcompete  maxsurrogate usesurrogate
#>        4            5            2
#> surrogatestyle      maxdepth       xval
```

```
#>          0          30          10
```

You can change any of these parameters via `rpart()`'s `control` argument, or by passing them directly to `rpart()` via the ... (pronounced dot-dot-dot) argument.<sup>q</sup> For example, the two calls to `rpart()` below are equivalent. Each one fits a classification tree but changes the default complexity parameter from 0.01 to 0 (`cp = 0`) and the number of internal cross-validations from ten to five (`xval = 5`); see `?rpart::rpart.control` for further details about all the configurable parameters.

```
ctrl <- rpart.control(cp = 0, xval = 5) # can also be a names list
tree <- rpart(Edibility ~ ., data = mushroom, control = ctrl)
tree <- rpart(Edibility ~ ., data = mushroom, cp = 0, xval = 5)
```

Another useful option in `rpart()` is the `parms` argument, which controls how nodes are split in the tree<sup>r</sup>; it must be a named list whenever supplied. Below we print the `tree$parms` component, which in this case is a list containing the class priors, loss matrix, and impurity function used in constructing the tree.

```
mushroom.tree$parms

#> $prior
#>     1      2
#> 0.518 0.482
#>
#> $loss
#>     [,1] [,2]
#> [1,]    0    1
#> [2,]    1    0
#>
#> $split
#> [1] 1
```

The `$prior` component defaults to the class frequencies in the root node, which can easily be verified:

```
proportions(table(mushroom$Edibility)) # observed class proportions

#>
#> Edible Poison
#> 0.518 0.482
```

The loss matrix, given by component `$loss`, defaults to equal losses for false positives and false negatives (the off diagonals); there's no loss associated with a correct classification (i.e., the diagonal entries are always zero).

---

<sup>q</sup>In R, functions can have a special ... argument that allows them to take any number of additional arguments; see Wickham [2019, Sec. 6.6] for details.

<sup>r</sup>The `parms` argument only applies to response variables that are categorical (classification trees), counts (Poisson regression), or censored (survival analysis)

The `$split` component displays either a 1 (`split = "gini"`) or 2 (`split = "information"`)<sup>s</sup> (partial matching is allowed). All of these can be changed from their respective defaults by passing a named list to the `parms` argument in the call to `rpart()`. For example, to use the entropy splitting rule<sup>t</sup>, run the following:

```
parms <- list("split" = "information") # use cross-entropy split rule
rpart(Edibility ~ ., data = mushroom, parms = parms)
```

Specifying a loss matrix in `rpart` isn't well-documented, unfortunately. For binary outcomes, the matrix has the following structure:

$$L = \begin{bmatrix} TP & FP \\ FN & TN \end{bmatrix},$$

where rows represent the observed classes and columns represent the assigned classes. Here, TP, FP, FN, and TN stand for *true positive*, *false positive*, *false negative*, and *true negative*, respectively; for example, a false negative is the case in which the tree misclassifies a 1 as a 0. The order of the rows/columns correspond to the same order as the categories when sorted alphabetically or numerically.

Since there is no cost for correct classification, we take  $TP = TN = 0$ . Setting  $FP = FN = c$ , for some constant  $c$  (i.e., treat FPs and FNs equally), will always result in the same splits (although, the internal statistics used in selecting the splits will be scaled differently). When misclassification costs are not equal, specify the appropriate values in the loss matrix. For example, the following tree would treat false negatives (i.e., misclassifying poisonous mushrooms as edible) as five times more costly than false positives (i.e., misclassifying edible mushrooms as poisonous). We could also obtain the same tree by computing the altered priors based on this loss matrix and supplying them via the `parms` argument, but this is left as an exercise to the reader.

```
levels(mushroom$Edibility) # inspect order of levels
(loss <- matrix(c(0, 5, 1, 0), nrow = 2)) # loss matrix
rpart(Edibility ~ ., data = mushroom, parms = list("loss" = loss))
```

The variable importance scores (Section 2.8) are contained in the `$variable.importance` component of the `mushroom.tree` object; they're also printed at the top of the output from `summary()`, but rescaled to sum to 100.

---

<sup>s</sup>In `rpart`, setting `split = "information"` corresponds to using the cross-entropy split rule discussed in Section 2.2.1.

<sup>t</sup>Users can also supply their own custom splitting rules. The steps for doing so are well documented in `rpart`'s vignette on “User Written Split Functions”: `utils::vignette("usercode", package = "rpart")`.

```
mushroom.tree$variable.importance

#>                 odor          spore.print.color
#>      3823.407           2834.187
#>      gill.color stalk.surface.above.ring
#>      2322.460           2035.816
#>      stalk.surface.below.ring          ring.type
#>      2030.555           2026.526
#>      stalk.color.below.ring        stalk.root
#>      53.933             25.600
#>      stalk.color.above.ring       veil.color
#>      17.546              16.315
#>      cap.surface         cap.color
#>      15.360              14.032
#>      habitat            cap.shape
#>      13.409              3.840
#>      gill.attachment
#>      0.585
```

In many cases, predictors that weren't used in the tree will have a non-zero importance score. The reason is that surrogate splits are also incorporated into the calculation. In particular, a variable may effectively appear in the tree more than once, either as a primary or surrogate splitter. The variable importance measure for a particular feature is the sum of the gains associated with each split for which it was the primary variable, plus the gains (adjusted for agreement) associated with each split for which it was a surrogate. You can turn off surrogates by setting `maxsurrogate = 0` in `rpart.control()`.

How does  $k$ -fold cross-validation (Section 2.5.2) in `rpart` work? The `rpart()` function does internal 10-fold cross-validation by default. According to `rpart`'s documentation, 10-fold cross-validation is a reasonable default, and has been shown to be very reliable for screening out "pure noise" features. The number of folds ( $k$ ) can be changed, however, using the `xval` argument in `rpart.control()`.

You can visualize the cross-validation results of an "`rpart`" object using `plotcp()`, as illustrated in Figure 2.21 for the `mushroom.tree` object. A good rule of thumb in choosing `cp` for pruning is to use the leftmost value for which the average cross-validation score lies below the horizontal line; this coincides with the 1-SE rule discussed in Section 2.5.2.1. The columns labeled "`xerror`" and "`xstd`" provide the cross-validated risk and its corresponding standard error, respectively (Section 2.5).

```
plotcp(mushroom.tree, upper = "splits", las = 1) # Figure 2.21
mushroom.tree$cptable # print cross-validation results

#>      CP nsplit rel error    xerror    xstd
#> 1 0.96936      0   1.00000 1.000000 0.011501
#> 2 0.01839      1   0.03064 0.030644 0.002777
```

```
#> 3 0.00613      2  0.01226 0.012257 0.001764
#> 4 0.00204      3  0.00613 0.006129 0.001249
#> 5 0.00102      5  0.00204 0.002043 0.000722
#> 6 0.00000      7  0.00000 0.000511 0.000361
```

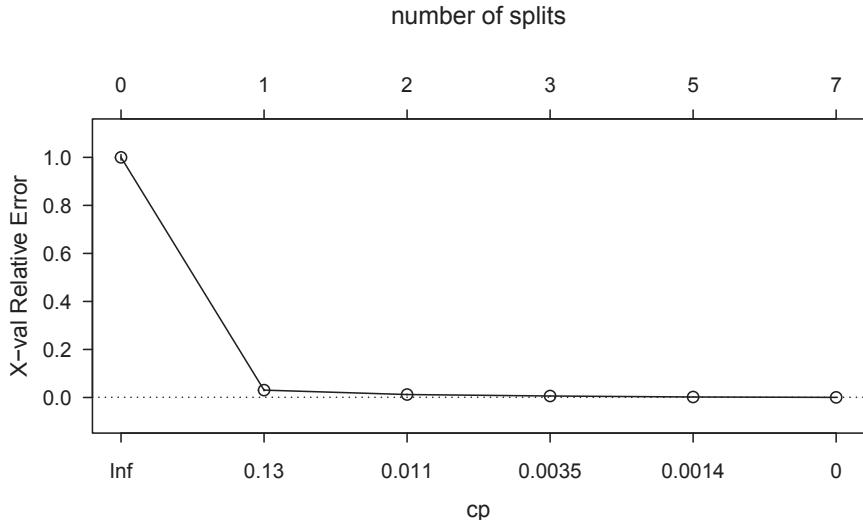


FIGURE 2.21: Cross-validation results from the fitted `mushroom.tree` object. A good choice of  $cp$  for pruning is often the leftmost value for which the mean lies below the horizontal line; this corresponds to the optimum value based on the 1-SE rule.

Don't be confused by the fact that the  $cp$  values between `printcp()` (and hence the `$cptable` component of an "rpart" object) and `plotcp()` don't match. The latter just plots the geometric means of the CP column listed in `printcp()` (these relate to the  $\beta_i$  values used in the  $k$ -fold cross-validation procedure described in Section 2.5). Any  $cp$  value between two consecutive rows will produce the same tree. For instance, any  $cp$  value between 0.002 and 0.001 will produce a tree with five splits. Also, these correspond to a scaled version of the complexity values  $\alpha_i$  from Section 2.5. Note that `rpart` scales the CP column, as well as the error columns, by a factor inversely proportional to the risk of the root node, so that the associated training error ("rel error") for the root node is always one (i.e., the first row in the table); which in this case is  $1 / (3916/8124) \approx 2.075$ . Dividing through by this scaling factor should return the raw  $\alpha_i$  values; the first three correspond to the values I computed by hand back in Section 2.5:

```
mushroom.tree$cptable[1L:3L, "CP"] / (8124 / 3916)
```

```
#>      1      2      3
```

```
#> 0.46726 0.00886 0.00295
```

Consequently, setting `cp = 1` will always result in a tree with no splits. The default, `cp = 0.01`, has been shown to be useful at “pre-pruning” the trees in a way such that the cross-validation step results in only the removal of 1–2 layers, although it can also occasionally over-prune. In practice, it seems best to set `cp = 0`, or some other number smaller than the default, and use the cross-validation results to choose an optimal subtree.

Using the 1-SE rule would suggest a tree with 5, or possibly 7, splits. However, since our main objective is to construct a simple rule-of-thumb for classifying the edibility of mushrooms, it seems like the simpler model with only a single split (i.e., a decision stump) will suffice; it only misclassifies 3% of the poisonous mushrooms as edible. To prune an `rpart` tree, use the `prune()` function with a specified value of the complexity parameter:

```
tree_diagram(prune(mushroom.tree, cp = 0.1)) # Figure 2.22
```

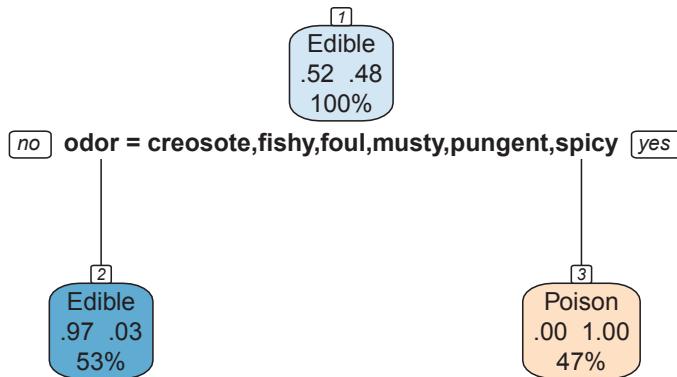


FIGURE 2.22: Decision tree diagram for classifying the edibility of mushrooms; in this case, the result is a decision stump.

The tree diagram displayed in Figure 2.22 provides us with a handy rule of thumb for classifying mushrooms as either edible or poisonous. If the mushroom smells fishy, foul, musty, pungent, spicy, or like creosote, it’s likely poisonous. In other words, if it smells bad, don’t eat it!

### 2.9.3 Example: predicting home prices

In this section, I'll use **rpart** to build a regression to the Ames housing data (Section 1.4.7). I'll also show how to easily prune an **rpart** tree using the 1-SE rule via **treemisc**'s **prune\_se()** function. The code chunk below loads in the data before splitting it into train/test sets using a 70/30 split:

```
ames <- as.data.frame(AmesHousing::make_ames())
ames$Sale_Price <- ames$Sale_Price / 1000 # rescale response
set.seed(2101) # for reproducibility
trn.id <- sample.int(nrow(ames), size = floor(0.7 * nrow(ames)))
ames.trn <- ames[trn.id, ] # training data/learning sample
ames.tst <- ames[-trn.id, ] # test data
```

Next, I'll intentionally fit an overgrown regression tree by setting **cp = 0**; in a regression tree, **rpart** will not attempt a split unless it increases the overall  $R^2$  by **cp**, so setting **cp = 0** will cause the tree to continue splitting until some other stopping criterion is met, such as minimum node size (in **rpart**, the default minimum number of observations that must exist in a node in order for a split to be attempted is 20). I'll also compare the RMSE between the train and test sets:

```
library(rpart)
library(treemisc) # for prune_se() function

# Fit a regression tree with no penalty on complexity
set.seed(1547) # for reproducibility
ames.tree <- rpart(Sale_Price ~ ., data = ames.trn, cp = 0)

rmse <- function(pred, obs) { # computes RMSE
  sqrt(mean((pred - obs) ^ 2))
}

# Compute train RMSE
rmse(predict(ames.tree, newdata = ames.trn), obs = ames.trn$Sale_Price)
#> [1] 23.4

# Compute test RMSE
rmse(predict(ames.tree, newdata = ames.tst), obs = ames.tst$Sale_Price)
#> [1] 31.4
```

The tree is likely overfitting, as indicated by the relatively large discrepancy between the train and test RMSE. Let's see if pruning the tree can help. The **prune\_se()** function from **treemisc** can be used to prune **rpart** trees using the 1-SE rule, as illustrated below:

```
ames.tree.1se <- prune_se(ames.tree, se = 1) # prune using 1-SE rule

# Train RMSE on pruned tree
```

```
rmse(predict(ames.tree.1se, newdata = ames.trn),
      obs = ames.trn$Sale_Price)

#> [1] 29.5

# Test RMSE on pruned tree
rmse(predict(ames.tree.1se, newdata = ames.tst),
      obs = ames.tst$Sale_Price)

#> [1] 34.1
```

A smaller discrepancy, but the pruned tree is slightly less accurate than the unpruned tree on the test set. So did pruning really help here? It depends on how you look at it. Both trees are displayed in Figure 2.23 without text or labels. The unpruned tree has 169 splits while pruning with the 1-SE rule and cross-validation resulted in a subtree with only 33 splits—a much more parsimonious tree

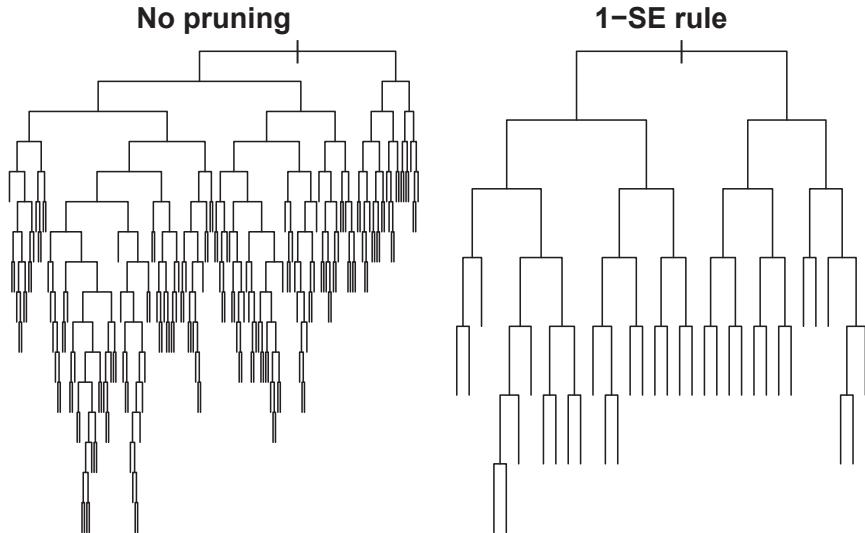


FIGURE 2.23: Regression trees for the Ames housing example. Left: unpruned regression tree. Right: pruned regression tree using 10-fold cross-validation and the 1-SE rule.

Even with pruning, we still ended up with a subtree that is too complex to easily interpret. In situations like this, it can be helpful to use different post hoc interpretability techniques to help the end user interpret the model in a way more understandable for humans. For instance, it can be quite informative to look at a plot of variable importance scores, like the Cleveland dot plot displayed in Figure ??; here, the importance scores are scaled to sum to 1 (see the code chunk below). From the results, we can see that the overall quality

of the home (Overall\_Qual), neighborhood (Neighborhood), and basement quality (Bsmt\_Qual) were some of the key features used to partition the data into groups of homes with similar sale prices.

```
vi <- sort(ames.tree.1se$variable.importance, decreasing = TRUE)
vi <- vi / sum(vi) # scale to sum to 1
dotchart(vi[1:10], xlab = "Variable importance", pch = 19)
```

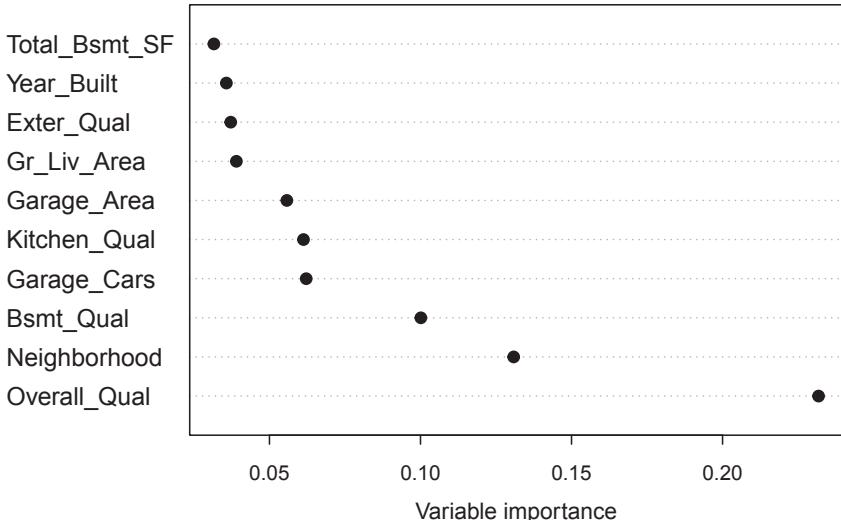


FIGURE 2.24: Variable importance plot for the top ten predictors in the pruned decision tree for the Ames housing data.

While variable importance plots can be useful, they don't tell us anything about the nature of the relationship between each feature and the predicted outcome. For instance, how does the above grade square footage (Gr\_Liv\_Area) impact the predicted sale price on average? This is precisely what *partial dependence plots* (PDPs) can tell us; see [Section 6.2.1](#). In a nutshell, PDPs are low-dimensional graphical renderings of the prediction function so that the relationship between the outcome and predictors of interest can be more easily understood. PDPs, along with other interpretability techniques, are discussed in more detail in [Chapter 6](#). For now, I'll just introduce the **pdp** package [Greenwell, 2021b], and show how it can be used to help visualize the relationship between above grade square footage and the predicted sale price:

```
library(ggplot2)
library(pdp)

# Compute partial dependence of predicted Sale_Price on Gr_Liv_Area
pd <- partial(ames.tree.1se, pred.var = "Gr_Liv_Area")
```

```
autoplot(pd, rug = TRUE, train = ames.trn) + # Figure 2.25
  ylab("Partial dependence")
```

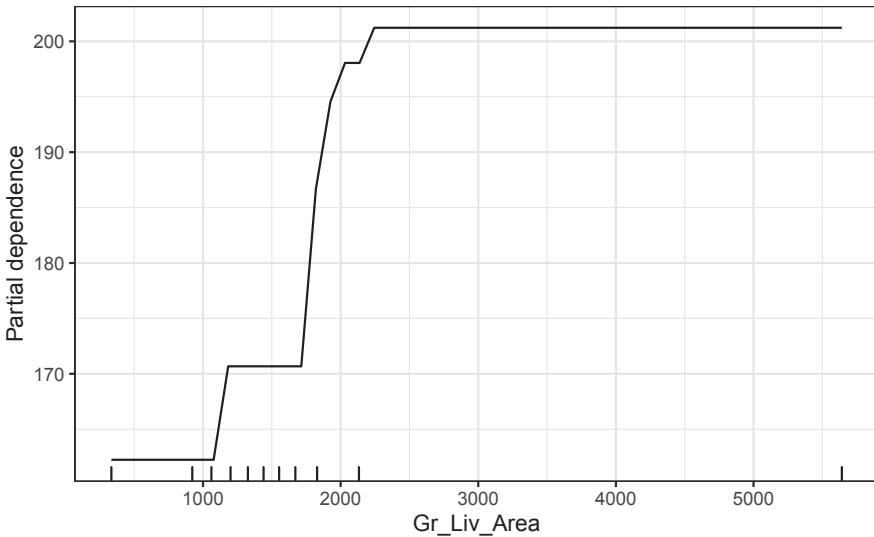


FIGURE 2.25: Partial dependence of `Sale_Price` on `Gr_Liv_Area` from the pruned regression tree. There's evidence of a monotonic increasing relationship between above grade square footage (`Gr_Liv_Area`) and the predicted sale price, while accounting for the average effect of all the other predictors.

Note that the *y*-axis is on the same scale as the response, and in this case, represents the averaged predicted sale price across the entire learning sample for a range of values of `Gr_Liv_Area`. The rug display (or 1-dimensional plot) on the *x*-axis shows the distribution of `Gr_Liv_Area` in the training data, with a tick mark at the min/max and each *decile*. As you would expect, larger size homes are associated with higher average predicted sales. Full details on the `pdp` package are given in Greenwell [2017].

Decision trees, especially smaller ones, can be rather self-explanatory. However, it is often the case that a usefully discriminating tree is too large to interpret by inspection. Variable importance scores, PDPs, and other interpretability techniques, can be used to help understand any tree, regardless of size or complexity; these techniques are even more critical for understanding the output from more complex models, like the tree-based ensembles discussed in Chapters 5–8.

## 2.9.4 Example: employee attrition

In this example, I'll revisit the employee attrition data ([Section 1.4.6](#)) and build a classification tree using **rpart** with altered priors to help understand drivers of employee attrition and confirm my previous calculations from [Section 2.2.4.1](#).

[Figure 2.6](#) showed two classification trees for the employee attrition data, one using the default priors and one with altered priors based on a specific loss matrix with unequal misclassification costs. In **rpart**, you can specify the loss matrix, priors, or both—it's quite flexible!

The next code chunk fits three depth-two classification trees to the employee attrition data. The first tree (`tree1`) assumes equal misclassification costs and uses the default (i.e., observed) class priors. The other two trees use different, but equivalent, approaches: `tree2` uses the previously defined loss matrix from [Section 2.2.4.1](#), while `tree3` uses the associated altered priors I computed by hand back in [Section 2.2.4.1](#). Although the internal statistics used in constructing each tree differ slightly, both trees are equivalent in terms of splits and will make the same classifications. The resulting tree diagrams are displayed in [Figure 2.26](#).

```
data(attrition, package = "modeldata")

# Fit classification trees with default priors and costs
set.seed(904) # for reproducibility
tree1 <- rpart(Attrition ~ OverTime + MonthlyIncome, data = attrition,
               maxdepth = 2, cp = 0)

# Specify unequal misclassification costs
loss <- matrix(c(0, 8, 1, 0), nrow = 2)
tree2 <- rpart(Attrition ~ OverTime + MonthlyIncome, data = attrition,
               maxdepth = 2, cp = 0, parms = list("loss" = loss))

# Equivalent approach using altered priors
tree3 <- rpart(Attrition ~ OverTime + MonthlyIncome, data = attrition,
               maxdepth = 2, cp = 0,
               parms = list("prior" = c(1 - 0.6059444, 0.6059444)))

# Display trees side by side (Figure 2.26)
par(mfrow = c(1, 3))
tree_diagram(tree1) # default costs and priors
tree_diagram(tree2) # unequal costs
tree_diagram(tree3) # altered priors
```

The subtle difference between `tree2` and `tree3` is that the within-node class proportions for `tree2` are not adjusted for cost/loss; hence, the predicted class probabilities will not match between the two trees. In essence, the tree based on the loss matrix (`tree2`) makes classifications using a predicted probability

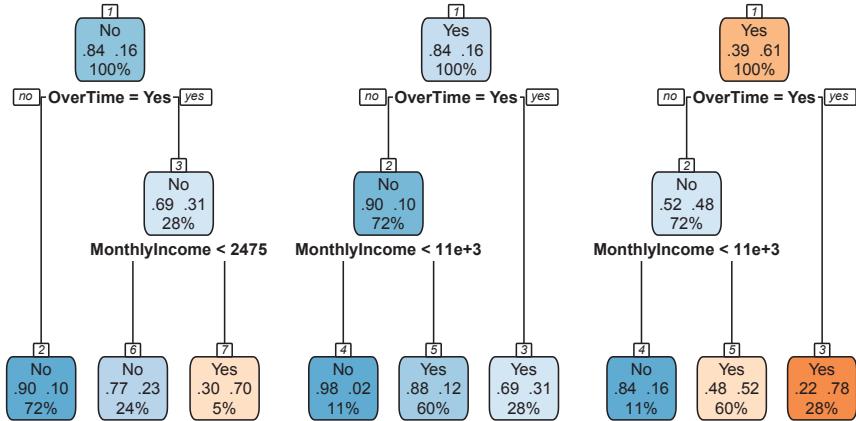


FIGURE 2.26: Decision trees fit to the employee attrition data set. Left: default priors and equal costs. Middle: Unequal costs specified via a loss matrix. Right: altered priors equivalent to the costs associated with the middle tree.

threshold different from  $0.5^u$  (i.e., classification is no longer based on a majority vote). On the other hand, in the altered priors tree (`tree3`), the expected class proportions are adjusted so that classification is still done via a majority vote (i.e., if  $A$  is the far right terminal node, then any observation in  $A$  would be classified as `Yes` since  $p_{yes}(A) = 0.779 > 0.5$ .

To summarize this particular example, specifying a loss matrix produced a tree and adjusted the probability threshold used for classification, whereas using altered priors produced a tree and adjusted the predicted probabilities to keep the threshold at 0.5; but both are equivalent in terms of the classifications they'll assign to new observations.

Note that the order of the rows and columns of the loss matrix, and the order of the priors, is the same as the sorted order for the response categories. For instance, in the matrix below, the first row/column corresponds to class `No` because it appears first in alphabetical order:

```
levels(attrition$Attrition) # can be changed with relevel()
#> [1] "No"   "Yes"
matrix(c(0, 8, 1, 0), nrow = 2)
```

<sup>u</sup>In general, the default probability threshold for classification is  $1/J$ , where  $J$  is the number of classes.

```
#>      [,1] [,2]
#> [1,]     0   1
#> [2,]     8   0
```

Although I restricted each tree to a max depth of two, the tree on the far left side of [Figure 2.26](#) actually coincides with the optimal tree I would've obtained using 10-fold cross-validation and the 1-SE rule (assuming unequal misclassification costs, of course), and shows that having to work overtime, as well as having a lower monthly income, was associated with the highest predicted probability of attrition ( $p = 0.70$ )

To further illustrate pruning using the 1-SE rule in **rpart**, let's look at a tree based on altered priors (**tree3**) with the full set of features. Below, I refit the same altered priors tree using all available features to maximum depth (i.e., intentionally overgrow the tree); the cross-validation results and pruned tree diagram are displayed in [Figure 2.27](#) (p. 109). The top plot in [Figure 2.27](#) shows the cost-complexity pruning results as a function of the number of splits (top axis). If I were to prune using the 1-SE rule, I would select the tree corresponding to the point farthest to the left that's below the horizontal line (the horizontal line corresponds to 1-SE above the minimum error). In this case, we would end up with a tree containing just four splits, as seen in the bottom plot in [Figure 2.27](#).

```
library(rpart)

# Saturated tree with altered priors using all predictors
att.cart <-
  rpart(Attrition ~ ., data = attrition, cp = 0, minsplit = 2,
        parms = list("prior" = c(1 - 0.6059444, 0.6059444)))

# Plot pruning results (Figure 2.27)
par(mfrow = c(2, 1))
plotcp(att.cart, upper = "splits")
(att.cart.1se <- prune_se(att.cart, se = 1))

#> n= 1470
#>
#> node), split, n, loss, yval, (yprob)
#>       * denotes terminal node
#>
#> 1) root 1470 579.0 Yes (0.394 0.606)
#> 2) OverTime>No 1054 413.0 No (0.518 0.482)
#>    4) TotalWorkingYears>=2.5 966 304.0 No (0.577 0.423)
#>      8) StockOptionLevel>=0.5 568 117.0 No (0.684 0.316) *
#>      9) StockOptionLevel< 0.5 398 163.0 Yes (0.465 0.535)
#>        18) JobRole=Healthcare_Representative,Human_Resources,Mana...
#>        19) JobRole=Laboratory_Technician,Research_Scientist,Sales...
#>    5) TotalWorkingYears< 2.5 88  27.7 Yes (0.203 0.797) *
```

```
#>   3) OverTime=Yes 416 136.0 Yes (0.221 0.779) *
tree_diagram(att.cart.1se, tweak = 0.8)
```

## 2.9.5 Example: letter image recognition

The goal of this example is to build an image classifier using a simple decision tree that incorporates additional information about the true class priors for a multiclass outcome with  $J = 26$  classes. The letter image recognition data, which are available in R via the **mlbench** package [Leisch and Dimitriadou, 2021]<sup>v</sup>, contains 20,000 observations on 17 variables. Each observation corresponds to a distorted black-and-white rectangular pixel image of a capital letter from the English alphabet (i.e., A–Z). A total of 16 ordered features was derived from each image (e.g., statistical moments and edge counts) which were then scaled to be integers in the range 0–15. The objective is to identify each letter using the 16 ordered features. Frey and Slate [1991] first analyzed the data using *Holland-style adaptive classifiers*, and reported an accuracy of just over 80%.

To start, I'll load the data and split it into train/test sets using a 70/30 split:

```
library(treemisc) # for prune_se()

data(LetterRecognition, package = "mlbench")
lr <- LetterRecognition # shorter name for brevity
set.seed(1051) # for reproducibility
trn.ids <- sample(nrow(lr), size = 14000, replace = FALSE)
lr.trn <- lr[trn.ids, ] # training data
lr.tst <- lr[-trn.ids, ] # test data
```

Next, I'll use **rpart()** to fit a classification tree that's been pruned using the 1-SE rule with 10-fold cross-validation, and see how accurate it is on the test sample:

```
set.seed(1703) # for reproducibility
lr.cart <- rpart(lettr ~ ., data = lr.trn, cp = 0, xval = 10)
lr.cart <- prune_se(lr.cart, se = 1) # prune using 1-SE rule

# Compute accuracy on test set
pred <- predict(lr.cart, newdata = lr.tst, type = "class")
sum(diag(table(pred, lr.tst$lettr))) / length(pred)

#> [1] 0.813
```

---

<sup>v</sup>The data are also available from the UCI Machine Learning Repository: <https://archive.ics.uci.edu/ml/datasets/Letter+Recognition>.

Nice, an overall accuracy of 81%—similar to the best accuracy reported in Frey and Slate [1991]. But don’t be fooled, the data have been artificially balanced:

```
table(lr$lettr)

#>
#>   A   B   C   D   E   F   G   H   I   J   K   L   M
#> 789 766 736 805 768 775 773 734 755 747 739 761 792
#>   N   O   P   Q   R   S   T   U   V   W   X   Y   Z
#> 783 753 803 783 758 748 796 813 764 752 787 786 734
```

As noted in Matloff [2017, Sec. 5.8], this is unrealistic since some letters tend to appear much more frequently than others in English text. For example, assuming balanced classes, `rpart()` uses a prior probability for the letter “A” of  $\pi_A = 1/26 \approx 0.0384$ , when in fact  $\pi_A$  is closer to 0.0855.<sup>w</sup>

A proper analysis should take these frequencies into account, which is easy enough to do with classification trees. Fortunately, the correct letter frequencies are conveniently available in the `regtools` package [Matloff, 2019]. Below, I’ll refit the model including the updated (and more realistic) class priors. I’ll then sample the test data so that the resulting class frequencies more accurately reflect the true prior probabilities of each letter:

```
data(ltrfreqs, package = "regtools")

# Compute correct class priors
prior <- ltrfreqs$percent
prior <- prior / sum(prior) # class priors should sum to 1
names(prior) <- ltrfreqs$ltr
prior <- prior[order(ltrfreqs$ltr)]

# Refit tree using correct priors
set.seed(1718) # for reproducibility
lr.cart.priors <- rpart(lettr ~ ., data = lr.trn, cp = 0,
                        parms = list(prior = prior))
lr.cart.priors <- prune_se(lr.cart.priors, se = 1)

# Sample test set to reflect correct class frequencies
ltrfreqs2 <- ltrfreqs
names(ltrfreqs2) <- c("lettr", "prior")
ltrfreqs2$prior <- ltrfreqs2$prior / sum(ltrfreqs2$prior)
temp <- merge(lr.tst, ltrfreqs2) # merge the two data sets
set.seed(1107) # for reproducibility
lr.tst2 <- temp[sample(nrow(temp), replace = TRUE,
                      prob = temp$prior), ]
```

Finally, let’s compare the two CART fits on the modified test set that reflects the more correct class frequencies:

---

<sup>w</sup>Based on the English letter frequencies reported at <http://practicalcryptography.com>.

```
pred2 <- predict(lr.cart, newdata = lr.tst2, type = "class")
pred3 <- predict(lr.cart.priors, newdata = lr.tst2, type = "class")
sum(diag(table(pred2, lr.tst2$lettr))) / length(pred2)

#> [1] 0.803

sum(diag(table(pred3, lr.tst2$lettr))) / length(pred3)

#> [1] 0.858
```

While the error of the original model based on equal priors decreased (albeit, not by much), the tree incorporating true prior information did much better. Woot!

---

## 2.10 Discussion

CART is one of the best off-the-shelf machine learning algorithms in existence, but it's not without its drawbacks. In closing, let's summarize many of the advantages and disadvantages of the CART algorithm for decision tree induction. While the emphasis here is on CART, much of this discussion equally applies to other decision tree algorithms as well (e.g., C4.5/C5.0).

### 2.10.1 Advantages of CART

**Small trees are easy to interpret.** Decision trees are often hailed as being simple and interpretable, relative to more complex algorithms. However, this is really only true for small trees with relatively few splits, like the one from Figure 2.22 (p. 95).

**Trees scale well to large  $N$ .** Individual decision trees scale incredibly well to large data sets, especially if most of the features are ordered, or categorical with relatively few categories. Even in the extreme cases, various shortcuts and approximations can be used to reduce computational burden (see, for example, Section 2.4).

**The leaves form a natural clustering of the data.** All observations that cohabit a terminal node necessarily satisfied all the same conditions when traversing the tree; in this sense, the records within a terminal node should be similar with respect to the feature values and can be considered *nearest neighbors*. We'll revisit this idea in Section 7.6.

**Trees can handle data of all types.** Trees can naturally handle data of mixed types, and categorical features do not necessarily have to be numerically

re-encoded, like in linear regression or neural networks. Trees are also invariant to monotone transformations of the predictors; that is, they only care about the rank order of the values of each ordered feature. For example, there's no need to apply logarithmic or square root transformations to any of the features like you might in a linear model.

**Automatic variable selection.** CART selects variables and splits one step at a time (“...like a carpenter that makes stairs”); hence the quote at the beginning of the chapter. If a variable cannot meaningfully partition the data into homogeneous subgroups, it will not likely be selected as a primary splitter. If it does, it'll likely get snipped off during the pruning phase.

**Trees can naturally handle missing data.** As discussed in [Section 2.7](#), CART can naturally avoid many of the problems caused by missing data by using surrogate splits (i.e., back up splitters that can be used whenever missing values are encountered during prediction).

**Trees are completely nonparametric.** CART is fully nonparametric. It does not require any distributional assumptions, and the user does not have to specify any parametric form for the model, like in linear regression. It can also automatically handle nonlinear relationships (although it tends to be quite biased since it uses step function to approximate potentially smooth surfaces) and interactions.

### 2.10.2 Disadvantages of CART

**Large trees are difficult to interpret.** Large tree diagrams, like the ones in [Figure 2.23](#) (p. 97), can be difficult to interpret and are probably not very useful to the end user. Fortunately, various interpretability techniques, like variable importance plots and PDPs, can help alleviate this problem. Such techniques are the topic of [Chapter 6](#).

**CART's splitting algorithm is quite greedy.** CART makes splits that are locally optimal. That is, the algorithm does not look through all possible tree structures to globally optimize some performance metric; that would be unfeasible, even for a small number of features. Instead, the algorithm recursively partitions the data by looking for the next best split at each stage. This is analogous to the difference between *forward-stepwise selection* and *best-subset selection*. Greedy algorithms use a more constrained search and tend to have lower variance but often pay the price in bias. [Chapters 5–8](#) discuss several strategies for breaking the bias-variance-tradeoff by combining many decision trees together.

**Splits lower on the tree are less accurate.** Data is essentially taken away after each split, making splits further down the tree less accurate (and noisy) compared to splits near the root node. This is part of the reason why binary

splits are used in the first place. While some decision tree algorithms allow multiway splits (e.g., CHAID and C4.5/C5.0), this is not a good strategy in general as the data would be fragmented too quickly, and the search for locally optimal splits becomes more challenging.

**Trees contain complex interactions.** CART finds splits in a sequential manner, and all splits in the tree depend on any that came before it. Once a final tree structure has been identified, the resulting prediction equation can be written using a linear model in *indicator functions*. For example, the prediction equation for the tree diagram in [Figure 2.9](#) (p. 63) can be written as follows:

$$\widehat{\text{Ozone}} = 75.41 \times I(\text{Temp} \geq 82.50) + 55.60 \times I(\text{Temp} < 82.50 \& \text{Wind} < 7.15) \\ + 22.33 \times I(\text{Temp} < 82.50 \& \text{Wind} \geq 7.15),$$

where  $I(\cdot)$  is the *indicator function* that evaluates to one whenever its argument is true, and zero otherwise. The right-hand side can be re-written more generally as  $f(\text{Temp}) + f(\text{Temp}, \text{Wind})$ , where the second term explicitly models an interaction effect between `Temp` and `Wind`. As you can imagine, a more complex tree with a larger number of splits easily leads to a model with high-order interaction effects. The presence of high-order interaction effects can make interpreting the main effects (i.e., the effect of individual predictors) more challenging.

**Biased variable selection.** As briefly discussed in [Section 2.4.2](#), CART's split selection strategy is biased towards features with many potential split points, such as categorical predictors with high cardinality. More contemporary decision tree algorithms, like those discussed in [Chapters 3–4](#), are unbiased in this sense.

**Trees are essentially step functions.** Trees can have a hard time adapting to smooth and/or linear response surfaces. Recall the twonorm problem from [Section 1.1.2.5](#), where the optimal decision boundary is linear. I fit an pruned `rpart` tree to the same sample using 10-fold cross-validation and the 1-SE rule; the resulting decision boundary (along with the optimal Bayes rule) is displayed in [Figure 2.28](#) (p. 110). Of course, I could increase the number of splits resulting in smaller steps, but in practice this often leads to overfitting and poor generalizability. This lack of smoothness causes more problems in the regression setting.

**Trees are noisy.** A common criticism of decision trees is that they are considered *unstable predictors*; this was also noted in the original CART monograph; see Breiman et al. [1984, Section 5.5.2]. By unstable, I mean high variance or, in other words, the tree structure (and therefore predictions) can vary, often wildly, from one sample to the next. For example, at any node in a particular

tree, there may be several competing splits that result in nearly the same decrease in node impurity and different samples may lead to different choices among these similar performing split contenders.

To illustrate, let's look at six independent samples of size  $N = 3,220$  from the email spam data described in [Section 1.4.5](#) ( $\approx 70\%$  training sample), and fit a CART-like tree to each using a maximum of four splits. The results are displayed in [Figure 2.29](#). Note the difference in structure and split variables across the six trees.

Fortunately, tree stability isn't always as problematic as it sounds (or looks). According to Zhang and Singer [2010, p. 57], "...the real cause for concern [in practice] regarding tree stability is the psychological effect of the appearance of a tree." Even though the structure of the tree can vary from sample to sample, Breiman et al. [1984, pp. 156–159] argued that competitive trees, while differing in appearance, can give fairly stable and consistent predictions. Strategies for improving the stability and performance of decision trees are discussed in [Chapters 5–8](#).

Instability is not a feature specific to trees, though. For example, traditional model selection techniques in linear regression—like forward selection, backward elimination, and hybrid variations thereof—all suffer from the same problem. However, averaging can improve the accuracy of unstable predictors, like overgrown decision trees, through variance reduction [Breiman, 1996a]; more on this in [Chapter 5](#).

---

## 2.11 Recommended reading

First and foremost, I highly recommend reading the original CART monograph [Breiman et al., 1984]. For a more approachable and thorough discussion of CART, I'd recommend Berk [2008, [Chap. 3](#)] (note that there's now a second edition of this book). I also recommend reading the vignettes accompanying the **rpart** package; R users can launch these from an active R session, as mentioned throughout this chapter, but they're also available from **rpart**'s CRAN landing page: <https://cran.r-project.org/package=rpart>. Scikit-learn's **sklearn.tree** module documentation is also pretty solid: <https://scikit-learn.org/stable/modules/tree.html>. There's also a fantastic talk by Dan Steinberg about CART, called "Data Science Tricks With the Single Decision Tree," that can be found on YouTube:

[https://www.youtube.com/watch?v=JVbU\\_tS6zKo&feature=youtu.be](https://www.youtube.com/watch?v=JVbU_tS6zKo&feature=youtu.be).

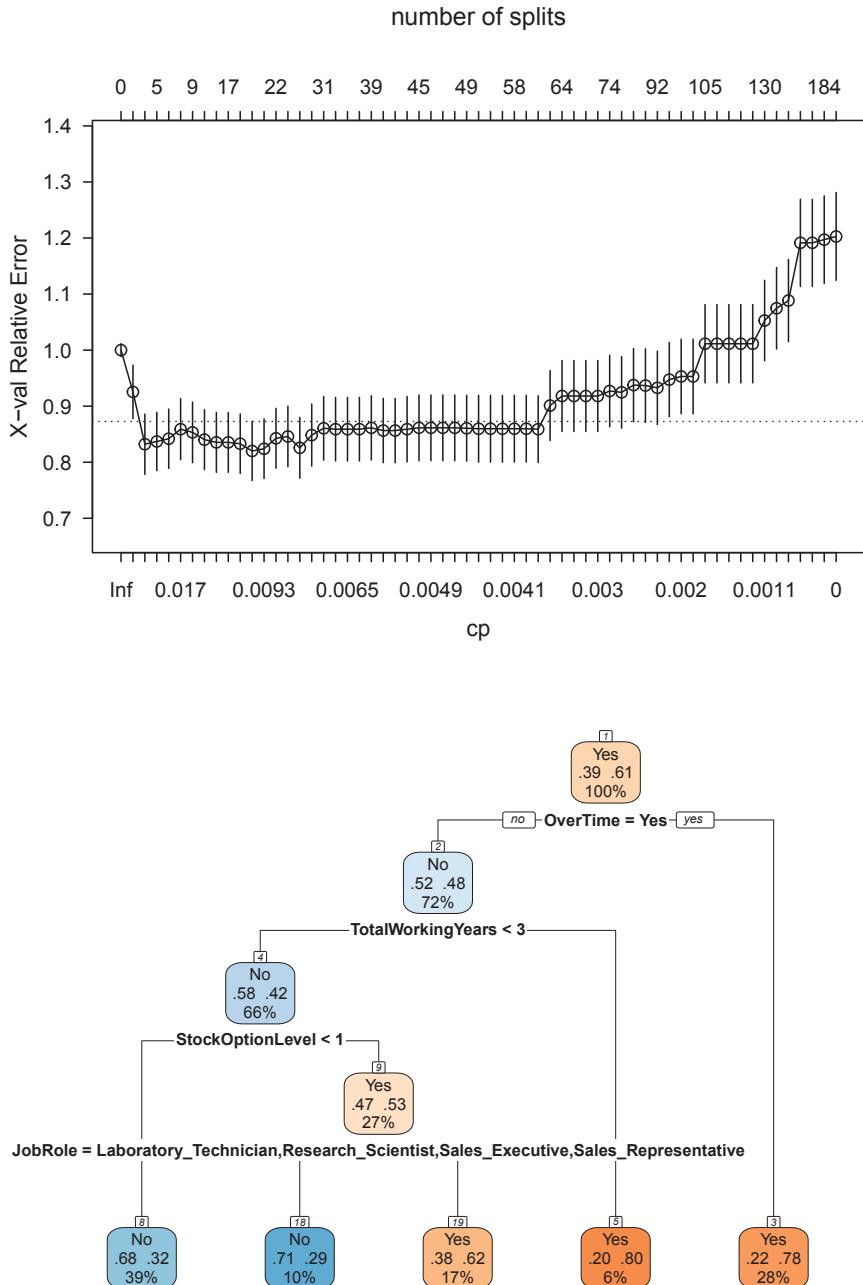


FIGURE 2.27: Employee attrition decision tree with altered priors. Top: 10-fold cross-validation results. Bottom: pruned tree using the 1-SE rule (which corresponds to the left-most point in the 10-fold cross-validation results that lies beneath the horizontal dotted line).

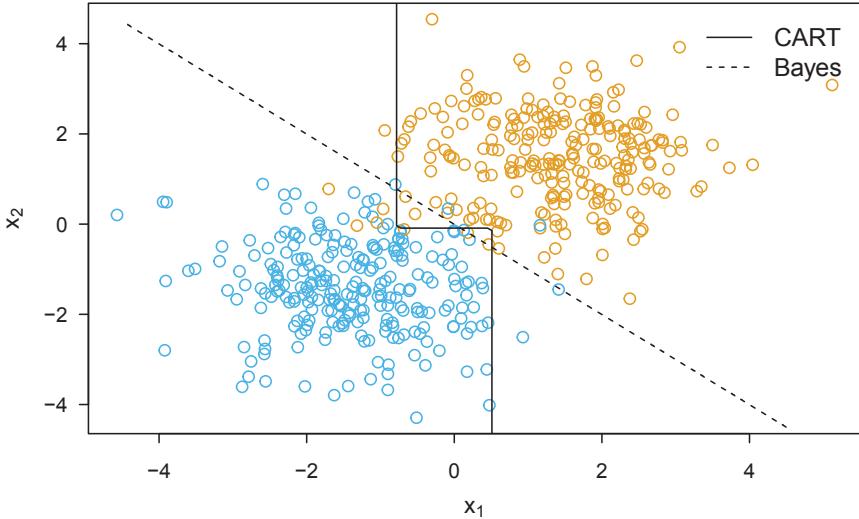


FIGURE 2.28: Decision boundaries for the twonorm benchmark example. In this case, the Bayes decision boundary is linear (dashed line). Also shown is the decision boundary from the pruned classification tree (solid line). The axis-oriented nature of decision tree splits makes it difficult to adapt to linear or smooth decision surfaces.

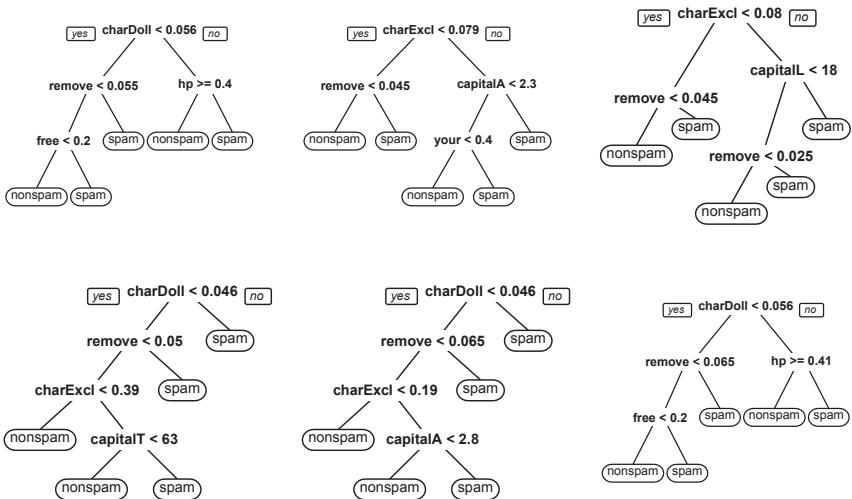


FIGURE 2.29: CART-like trees applied to six independent samples from the email spam data; for plotting purposes, each tree is restricted to just four splits.

# 3

---

## Conditional inference trees

---

The trees that are slow to grow bear the best fruit.

Moliere

---

The abundance of different tree algorithms makes it rather difficult to keep up with current developments. While CART-like decision trees ([Chapter 2](#)) are still popular for classification and regression problems, there have been numerous developments and improvements over exhaustive search procedures, like CART, especially from the year 2000 onward. In this chapter, I'll discuss one of the more important developments: unbiased recursive partitioning via *conditional inference*.

---

### 3.1 Introduction

As discussed in [Section 2.4.2](#), CART has a bias towards selecting variables with many potential split points. These issues also extend to C4.5 and C5.0 (which are discussed in the online complements), or any other recursive partitioning procedure that performs an exhaustive search over all possible splits to maximize some measure of node impurity.

To illustrate, consider a regression scenario where none of the predictors have any association to the response. Following Loh [2014] (see his rejoinder to the discussions), I'll look at 5,000 simulated data sets, each of which consisted of 100 observations on the following eight variables:

- $y$ : a  $N(0, 1)$  random variable (the response variable);
- $ch2$ : a  $\chi^2(2)$  random variable;

- `m2`: a random factor with 2 equiprobable categories;
- `m4`: a random factor with 4 equiprobable categories;
- `m10`: a random factor with 10 equiprobable categories;
- `m20`: a random factor with 20 equiprobable categories;
- `nor`: an  $N(0, 1)$  random variable;
- `uni`: a  $\mathcal{U}(0, 1)$  random variable.

Keep in mind that none of the seven predictors have any relationship to the response. For each simulated data set, two types of regression trees were fit, each with a single split (i.e., a decision stump):

- a CART-like decision tree, as discussed in [Chapter 2](#);
- a *conditional inference tree* (CTree), as will be discussed later in this chapter (see [Section 3.4](#)).

[Figure 3.1](#) shows the frequency with which each variable was selected as the primary splitter in each scenario. Notice how CART selects `m20`, the categorical predictor with 20 equiprobable categories, more than 80% of the time, and `m10`, the categorical predictor with 10 equiprobable categories, roughly 10% of the time! If CART’s split variable selection strategy was unbiased, we would expect each feature to be selected roughly  $1/7 \approx 14.29\%$  of the time. CTree, on the other hand, which uses an unbiased split variable selection procedure, selects each feature with a roughly equal frequency close to  $1/7$ . While this doesn’t necessarily imply that CART is less accurate, it does make interpretation more difficult (e.g., variable importance plots will be biased).

### 3.2 Early attempts at unbiased recursive partitioning

The basic idea behind unbiased recursive partitioning is to separate the split search into two sequential steps: 1) selecting the primary split variable, then 2) selecting an optimal split point. Typically, the primary splitter is selected first by comparing appropriate statistical tests (e.g., a chi-square test if both  $X$  and  $Y$  are nominal, or a correlation-type test if  $X$  and  $Y$  are both univariate continuous). Once a splitting variable has been identified, the optimal split point can be found using any number of approaches, including further statistical tests, or the impurity measures discussed in [Section 2.2.1](#).

The idea of using statistical tests for split variable selection in recursive partitioning is not new. In fact, the CTree algorithm discussed later in this chapter

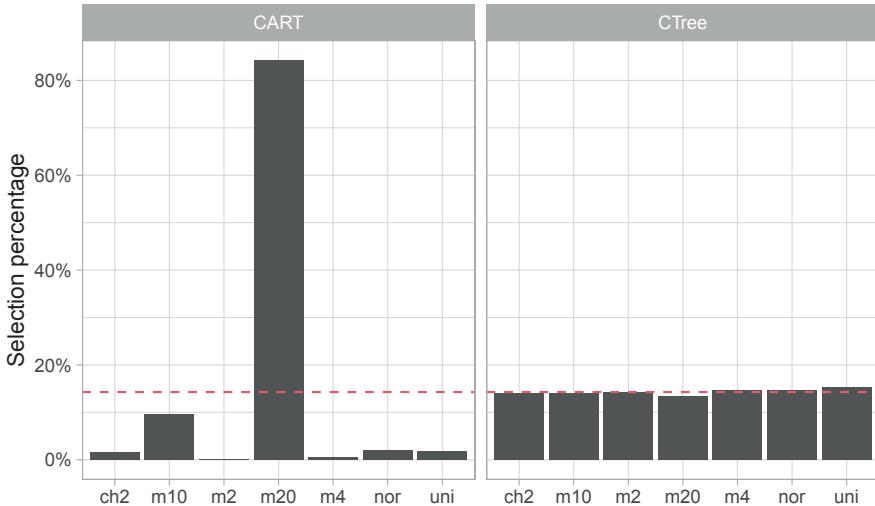


FIGURE 3.1: Split variable selection frequencies for two regression tree procedures (CART and CTree); the  $y$ -axis shows the proportion of times each feature—all of which are unrelated to the response—was selected to split the root node in 5,000 Monte Carlo simulations. A horizontal dashed red line is given at 1/7, the frequency corresponding to unbiased split variable selection.

was inspired by a number of approaches that came before it, like CHAID (Section 1.2.1). While these algorithms help to reduce variable selection bias compared to exhaustive search techniques like CART, most of them only apply to special circumstances. CHAID, for example, requires both the features and the response to be categorical, although, CHAID was eventually extended to handle ordered outcomes. CHAID can be used with ordered features if they’re binned into categories; this induces a bias and defeats the purpose of unbiased split variable selection in the first place. CTree, on the other hand, provides a unified framework for unbiased recursive partitioning that’s applicable to data measured on arbitrary scales (e.g., continuous, nominal categorical, censored, ordinal, and multivariate data). Before we introduce the details of CTree, it will be helpful to have a basic understanding of the conditional inference framework it relies on.

The following section involves a bit of mathematical detail and notation, mostly around linear algebra, and can be skipped by the uninterested reader.

---

### 3.3 A quick digression into conditional inference

This section offers a quick detour into the essentials of a general framework for conditional inference procedures commonly known as *permutation tests*; our discussion follows closely with Hothorn et al. [2006b]<sup>a</sup>, which is based on the theoretical results derived by Strasser and Weber [1999] for a special class of *linear statistics*. For a more traditional take on (nonparametric) permutation tests, see, for example, Davison and Hinkley [1997, Sec. 4.3].

Suppose  $\{(\mathbf{X}_i, \mathbf{Y}_i)\}_{i=1}^N$  are a random sample from some population of interest, where  $\mathbf{X}$  and  $\mathbf{Y}$  are from sample spaces  $\mathcal{X}$  and  $\mathcal{Y}$ , respectively, which may be multivariate (hence the **bold** notation) and measured at arbitrary scales (e.g., ordinal, nominal, continuous, etc.). Our primary interest is in testing the null hypothesis of independence between  $\mathbf{X}$  and  $\mathbf{Y}$ , namely,

$$H_0 : D(\mathbf{Y} | \mathbf{X}) = D(\mathbf{Y}),$$

where  $D()$  is the distribution function for  $\mathbf{Y}$ .

There are a number of ways to carry out such a test. For example, if  $\mathbf{X}$  and  $\mathbf{Y}$  are both  $N \times 1$  continuous variables, then a simple correlation test based on the Pearson correlation coefficient, or Spearman's  $\rho$ , can be used. If  $\mathbf{X}$  and  $\mathbf{Y}$  are both  $N \times 1$  nominal categorical variables, then a simple chi-square test is appropriate. And so on.

It would be rather convenient to have a standard approach to testing  $H_0$ , regardless of the shape or scale of  $\mathbf{X}$  and  $\mathbf{Y}$ . To that end, Strasser and Weber [1999] suggest using a linear statistic of the general form

$$\mathbf{T} = \text{vec} \left( \sum_{i=1}^N g(\mathbf{X}_i) h(\mathbf{Y}_i)^\top \right) \in \mathbb{R}^{pq \times 1}, \quad (3.1)$$

where  $\text{vec}()$  is the *matrix vectorization operator* that I'll explain in the examples that follow,  $g : \mathcal{X} \rightarrow \mathbb{R}^{p \times 1}$  is a transformation function specifying a possible transformation of the  $\mathbf{X}$  values, and  $h : \mathcal{Y} \rightarrow \mathbb{R}^{q \times 1}$  is called the *influence function* and specifies a possible transformation to the  $\mathbf{Y}$  values. Appropriate choices for  $g()$  and  $h()$  allow us to perform general tests of independence, including:

---

<sup>a</sup>An updated version of this paper is freely available in the “A Lego System for Conditional Inference” vignette accompanying the R package **coin** [Hothorn et al., 2021c]; to view, use `utils::vignette("LegoCondInf", package = "coin")` at the R console.

- correlation tests, similar to the Pearson and Spearman's rank correlation tests;
- two-sample tests, similar to the  $t$ -test and Wilcoxon rank-sum test;
- more general  $K$ -sample tests, similar to the one-way ANOVA  $F$ -test and Kruskal-Wallis test;
- independence tests for contingency tables of arbitrary dimension;
- and many more.

Typically,  $g()$  and  $h()$  are chosen to be the identity function (e.g.,  $g(\mathbf{X}_i) = \mathbf{X}_i$ ).

Any test of  $H_0$  based on (3.1) requires knowledge of the sampling distribution of  $\mathbf{T}$ , which is rarely (if ever) known in practice. However, under the null hypothesis ( $H_0$ ), we can dispose of this dependency by fixing the predictor values, and *conditioning* on all possible permutations  $S$  of the response values—whence the term conditional inference. This principle leads to test procedures known as permutation tests.

Let  $\mu_h = E(h(\mathbf{Y}_i) | S) = \sum_{i=1}^N h(\mathbf{Y}_i) / N$  be the conditional expectation for the influence function with corresponding  $q \times q$  covariance matrix

$$\begin{aligned}\Sigma_h &= V(h(\mathbf{Y}_i) | S) \\ &= \sum_{i=1}^N [h(\mathbf{Y}_i) - E(h(\mathbf{Y}_i) | S)] [h(\mathbf{Y}_i) - E(h(\mathbf{Y}_i) | S)]^\top.\end{aligned}$$

Strasser and Weber [1999] derived the conditional mean ( $\mu$ ) and variance ( $\sigma^2$ ) of  $\mathbf{T}|S$ , which are:

$$\begin{aligned}\mu &= E(\mathbf{T}|S) = \text{vec} \left( \left( \sum_{i=1}^N g(\mathbf{X}_i) \right) \mu_h^\top \right) \in \mathbb{R}^{pq \times 1}, \\ \sigma^2 &= V(\mathbf{T}|S) = \frac{N}{N-1} V_h \otimes \left( \sum_{i=1}^N g(\mathbf{X}_i) \otimes g(\mathbf{X}_i)^\top \right) \\ &\quad - \frac{1}{N-1} V_h \otimes \left( \sum_{i=1}^N g(\mathbf{X}_i) \right) \otimes \left( \sum_{i=1}^N g(\mathbf{X}_i) \right)^\top \in \mathbb{R}^{pq \times pq},\end{aligned}\tag{3.2}$$

where  $\otimes$  denotes the *Kronecker product*. While the equations listed in (3.1)–(3.2) might seem very complex, they simplify quite a bit in many standard situations—specific examples are given in [Sections 3.3.1–3.3.2](#).

The next step is to construct a *test statistic* for testing  $H_0$ . In order to do this, we can standardize the (possibly multivariate) linear statistic using  $\mu$  and  $\Sigma$ . Let  $c()$  be a function that maps  $\mathbf{T} \in \mathbb{R}^{pq \times 1}$  to the real line (i.e., a scalar or single number). Hothorn et al. [2006b] suggest using a quadratic form or the maximum of the absolute values of the standardized linear statistic:

$$\begin{aligned} c_q &= c_q(\mathbf{T}, \mu, \Sigma) = (\mathbf{T} - \mu) \Sigma^+ (\mathbf{T} - \mu)^\top, \\ c_m &= c_m(\mathbf{T}, \mu, \Sigma) = \max \left| \frac{\mathbf{T} - \mu}{\text{diag}(\Sigma)^{1/2}} \right|, \end{aligned} \quad (3.3)$$

where  $\Sigma^+$  denotes the *Moore-Penrose inverse* of  $\Sigma$ .

In order to construct  $p$ -values for testing  $H_0$ , we need to know the sampling distribution of  $c_q$  and  $c_m$ . Resampling approaches can be used to approximate the sampling distribution to any desired accuracy by evaluating the test statistic under all possible permutations  $S$ . This is often not feasible in practice as there are  $N!$  possible permutations to consider. However, approximations based on a random sample of permutations can be used to great effect; see Davison and Hinkley [1997, Sec. 4.3] for details on traditional permutation tests. For certain special cases, the exact sampling distribution of some test statistics is obtainable for small to moderate sample sizes [Hothorn et al., 2006b].

For larger sample sizes, a normal approximation can be used, regardless of the choice for  $g()$  and  $h()$ . In particular, Strasser and Weber [1999] showed that the conditional distribution of  $\mathbf{T}$  tends to a multivariate normal distribution with mean  $\mu$  and covariance  $\Sigma$  as  $N \rightarrow \infty$ . Consequently, the proposed test statistics,  $c_q$  and  $c_m$ , also have asymptotic approximations. For example,  $c_q$  has an asymptotic  $\chi^2(df)$  distribution with degrees of freedom ( $df$ ) given by the rank of  $\Sigma$ ; if  $p = q = 1$ , then asymptotically (i.e., as  $N \rightarrow \infty$ ),  $c_q \sim \chi^2(1)$  and  $c_m \sim N(0, 1)$ . With the conditional distribution of the test statistic in hand—whether asymptotic, approximate, or exact—we can compute a  $p$ -value ( $p$ ) for testing  $H_0$ , where we would reject  $H_0$  whenever  $p \leq \alpha$ , where  $\alpha$  is some prespecified threshold.

Akin to a “one size fits all” approach to statistical tests of independence, this is a powerful idea that opens the door to a wide range of possibilities. For example, if the outcome is censored, then  $h()$  can be chosen to map the response values to *log-rank scores*, which can be obtained in R using the `logrank_trafo()` function from package **coin**; see, for example, Hothorn et al. [2006b] or the “A Lego System for Conditional Inference” vignette from package **coin** mentioned earlier.

Enough with the mathematical jujitsu; let’s illustrate the main ideas with some concrete examples.

### 3.3.1 Example: $X$ and $Y$ are both univariate continuous

The simplest case occurs when  $X$  and  $Y$  are both univariate continuous variables; in the univariate case, we drop the bold notation and just write  $X$  and  $Y$ . In this case,  $p = q = 1$  and  $T = \sum_{i=1}^N g(X_i) h(Y_i)$ . If we take  $g()$  and  $h()$  to be the identity function (e.g.,  $g(X_i) = X_i$ ), then

$$\begin{aligned} T &= \sum_{i=1}^N X_i Y_i, \\ \mu &= \left( \sum_{i=1}^N X_i \right) \bar{Y}, \\ \sigma^2 &= S_Y^2 \sum_{i=1}^N X_i^2 - S_Y^2 \left( \sum_{i=1}^N X_i \right)^2 / N, \end{aligned}$$

where  $\bar{Y} = \sum_{i=1}^N Y_i / N$  and  $S_Y^2 = \sum_{i=1}^N (Y_i - \bar{Y})^2 / (N - 1)$  are the sample mean and variance of  $Y$ , respectively. Since  $T$  is univariate, the standardized test statistics (3.3) are  $c_m = |(T - \mu) / \sqrt{\Sigma}|$  and  $c_q = c_m^2$ ; hence, it makes no difference which test statistic we use in this case, as the results will be identical.

Let's revisit the New York air quality data set (Section 1.4.2) to demonstrate the required computations in R. Let  $X = \text{Temp}$  and  $Y = \text{Ozone}$  be the variables of interest. To test the null hypothesis of general independence between  $X$  and  $Y$  at the  $\alpha = 0.05$  level, I'll use the quadratic test statistic ( $c_q$ ), and compute a  $p$ -value for the test using an asymptotic  $\chi^2(1)$  approximation. I'll also choose  $g()$  and  $h()$  to be the identity function. The first line below removes any rows with a missing response value, which I'll be using later.

```
aq <- airquality[!is.na(airquality$Ozone), ]
N <- nrow(aq) # sample size
gX <- aq$Temp # g(X)
gY <- aq$Ozone # h(Y)
Tstat <- sum(gX * gY) # linear statistic
mu <- sum(gX) * mean(gY)
Sigma <- var(gY) * sum(gX ^ 2) - var(gY) * sum(gX) ^ 2 / N

# Quadratic test statistic (1.3)
(cq <- ((Tstat - mu) / sqrt(Sigma)) ^ 2)

#> [1] 56.1

1 - pchisq(cq, df = 1) # p-value
#> [1] 6.94e-14
```

Here we would reject the null hypothesis at the  $\alpha = 0.05$  level ( $p < 0.001$ ) and conclude that there is some degree of association between `Temp` and `Ozone`.

For comparison, we can use the `independence_test()` function from package `coin`, which provides a flexible implementation of the conditional inference procedures described in Hothorn et al. [2006b]; see `?coin::independence_test` for details. This is demonstrated in the code snippet below:

```
library(coin)
independence_test(Ozone ~ Temp, data = aq, teststat = "quadratic")

#>
#> Asymptotic General Independence Test
#>
#> data: Ozone by Temp
#> chi-squared = 56, df = 1, p-value = 7e-14
```

Happily, we obtain the exact same results. Note that  $c_m$  and  $c_q$  will only differ when the linear statistic (3.1) is multivariate.

### 3.3.2 Example: $X$ and $Y$ are both nominal categorical

When  $X$  and  $Y$  are both categorical,  $T$  is essentially the vectorized (i.e., flattened) contingency table between  $X$  and  $Y$ ; in this section, I'll continue to drop the bold notation. Assume  $X$  and  $Y$  have  $q$  and  $p$  unique categories, respectively. A contingency table between  $X$  and  $Y$  is nothing more than a  $q \times p$  table containing the observed frequencies of each  $qp$  pair of categories from  $X$  and  $Y$ .<sup>b</sup> Recall the appearance of the matrix vectorization operator, `vec()`, in formulas (3.1)–(3.2). The `vec()` operator turns an  $m \times n$  matrix into an  $mn \times 1$  column vector. So a vectorized  $q \times p$  contingency table is just a  $qp \times 1$  column vector containing the individual frequencies, where the vectorization happens columnwise. For example, if  $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$  is a  $2 \times 2$  matrix (or contingency table), then  $\text{vec}(A) = (a, c, b, d)^\top$ , where  $\top$  represents the transpose operator.

Let's return to the mushroom edibility example (Section 1.4.4). Let  $X = \text{odor}$  and  $Y = \text{Edibility}$ ; both are nominal categorical with nine and two categories, respectively. A contingency table cross-classifying each variable is given below using the `xtabs()` function (see `?stats::xtabs` for details). The vectorized contingency table is also constructed by calling `as.vector()`:

```
mushroom <- treemisc::mushroom
(ctab <- xtabs(~ odor + Edibility, data = mushroom))
```

---

<sup>b</sup>Here we assume that the  $q$  categories of  $X$  define the rows of the contingency table, but in general, it does not matter.

```
#>          Edibility
#> odor      Edible Poison
#> almond     400      0
#> anise      400      0
#> creosote    0      192
#> fishy       0      576
#> foul        0     2160
#> musty       0       36
#> none        3408     120
#> pungent     0      256
#> spicy       0      576

(Tstat <- as.vector(ctab)) # multivariate linear statistic

#> [1] 400 400 0 0 0 0 3408 0 0 0
#> [11] 0 192 576 2160 36 120 256 576
```

Typically, when  $X$  and  $Y$  are categorical, the transformation function,  $g()$ , and influence function,  $h()$ , map each category to a vector of dummy encoded values. For example, using the order of categories from the above contingency table (`ctab`),

$$g(X_i) = (0, 0, 0, 0, 1, 0, 0, 0, 0)^\top$$

and

$$h(Y_i) = (0, 1)^\top.$$

would indicate a foul smelling, poisonous mushroom. In essence,  $g(X) = \mathbf{X}$  and  $h(Y) = \mathbf{Y}$  are the associated one-hot encoded model matrices for  $X$  and  $Y$ , respectively.

In the special case where  $X$  and/or  $Y$  are binary, the identity transformation will lead to the same results as long as they're encoded as 0/1. For example, if  $X$  and  $Y$  are both binary 0/1 encoded variables, then the formulas in [Section 3.3.1](#) will lead to identical results when  $g()$  and  $h()$  are both the identity function.

The Kronecker product, denoted  $\otimes$ , also appears in (3.2). The Kronecker product between two matrices can be computed in R using the `kronecker()` function or the `%x%` operator. The Moore-Penrose inverse, denoted  $\Sigma^+$  in (3.2), can be computed using the `ginv()` function from the recommended **MASS** package [Ripley, 2022].

In the next code chunk, we compute the conditional expectation and variance using (3.2). Note that the formulas simplify greatly if we work directly with the one-hot encoded matrices  $g(X) = \mathbf{X}$  and  $h(Y) = \mathbf{Y}$ ; in R, these can be

obtained using the `model.matrix()` function.<sup>c</sup> For example, the Kronecker product  $\sum_{i=1}^N g(\mathbf{X}_i) \otimes g(\mathbf{X}_i)^\top$  in (3.2) reduces to  $\mathbf{X}^\top \mathbf{X}$ —which, in this case, is a  $p \times p$  diagonal matrix, whose  $j$ -th diagonal entry is equal to the frequency of the  $j$ -th category of  $X$  (or the sum of the  $j$ -th column of  $\mathbf{X}$ ). Also,  $\sum_{i=1}^N g(\mathbf{X}_i)$  in (3.2) is just a  $p \times 1$  column vector, whose  $j$ -th entry is equal to the frequency of the  $j$ -th category of  $X$ . Note that we use R’s built-in `qr()` function to compute the rank of  $\Sigma$ , which is used for the degrees of freedom of the asymptotic chi-square distribution of  $c_q$ ; in this example,  $df = \text{rank}(\Sigma) = 8$ . (Due to rounding, the printed output displays a  $p$ -value of zero.)

```
gX <- model.matrix(~ odor - 1, data = mushroom) # g(X)
hY <- model.matrix(~ Edibility - 1, data = mushroom) # h(Y)
mu <- as.vector(colSums(gX) %*% t(colMeans(hY)))
Sigma <- var(hY) %x% (t(gX) %*% gX) -
  var(hY) %x% (colSums(gX) %x% t(colSums(gX))) / nrow(hY)

# Quadratic test statistic (1.3)
(cq <- t(Tstat - mu) %*% MASS::ginv(Sigma) %*% (Tstat - mu))

#>      [,1]
#> [1,] 7659

1 - pchisq(cq, df = qr(Sigma)$rank) # p-value

#>      [,1]
#> [1,]    0
```

Again, we can compare the results with the output from `coin`’s `independence_test()` function. Once again, the results are equivalent.

```
independence_test(Edibility ~ odor, data = mushroom,
                  teststat = "quadratic")

#>
#> Asymptotic General Independence Test
#>
#> data: Edibility by
#>   odor (almond, anise, creosote, fishy, foul, musty, none, pungent...
#> chi-squared = 7659, df = 8, p-value <2e-16
```

### 3.3.3 Which test statistic should you use?

In the previous examples, we used the quadratic form of the test statistic in (3.3) and its asymptotic chi-square distribution, but what about the maxi-

---

<sup>c</sup>Here, I use `model.matrix(~ variable.name - 1)` to suppress the intercept—a column of all ones—and ensure that each category of `variable.name` gets dummy encoded.

mally selected test statistic ( $c_m$ ) in (3.3)? When  $\mathbf{X}$  and  $\mathbf{Y}$  are both univariate continuous, or binary variables encoded as 0/1, then the choice between  $c_q$  and  $c_m$  makes no difference, since  $c_q = c_m^2$  in this case. However, if  $\mathbf{X}$  and/or  $\mathbf{Y}$  are multivariate (e.g., when  $\mathbf{X}$  and/or  $\mathbf{Y}$  are multi-level categorical variables), then the two statistics can lead to different, although usually similar results. Some guidance on when one test statistic may be more useful than the other is given in Hothorn et al. [2006b]. For example, if  $\mathbf{X}$  and  $\mathbf{Y}$  are both categorical, then working with  $c_m$  and the standardized contingency table can be useful in gaining insight into the association structure between  $\mathbf{X}$  and  $\mathbf{Y}$ . For the general test of independence, it often doesn't matter which form of the test statistic you use. As we'll see in the next few sections, the CTree algorithm often defaults to using the quadratic test statistic (i.e.,  $c_q$ ) from (3.3).

---

## 3.4 Conditional inference trees

Conditional inference trees provide a unified framework for unbiased recursive partitioning based on conditional inference, the same idea behind permutation tests, and is general enough to handle data of many different types (e.g., continuous, categorical, ordinal, censored, multivariate, and more). CTree uses a general two-stage process for recursive partitioning based on null significance tests, which is described in Algorithm 3.1 below.

The next two sections dive a bit deeper into steps 1)–2), respectively.

### 3.4.1 Selecting the splitting variable

Step 1) of Algorithm 3.1 is to decide whether there is any (statistically significant) association between the response and any of the  $m$  predictors. This is accomplished via  $m$  partial tests of hypothesis  $H_0^j : D(\mathbf{Y}|X_j) = D(\mathbf{Y})$ , for  $j = 1, 2, \dots, m$ . The test statistic used for assessing the association between each feature and the response depends on the scale on which both are measured (e.g., multivariate, censored, continuous, ordinal, or nominal categorical). For example, if  $X$  and  $Y$  are both univariate continuous, then a correlation-based test can be carried out. If  $X$  is categorical and  $Y$  is continuous, then a  $K$ -sample test—like an ANOVA  $F$ -test or Kruskal-Wallis test—can be used. And so on.

In practice, the predictors will often be measured on different scales; hence, different test statistics need to be employed. Consequently, the test statistics associated with each test cannot be directly compared without biasing the selection of the splitting variable. Fortunately, using  $p$ -values provides a

---

**Algorithm 3.1** Unbiased recursive partitioning via conditional inference.

---

- 1) Individually test the null hypothesis of independence between each of the  $m$  features  $X_1, X_2, \dots, X_m$  and the response  $Y$  using the conditional inference approach outlined in [Section 3.3](#). If none of these hypotheses can be rejected at a prespecified  $\alpha$  level, then stop the procedure (i.e., no further splits occur). Otherwise, select the predictor  $X_j$  with the “strongest association” to  $Y$ , as measured by the corresponding multiplicity adjusted  $p$ -values (e.g., *Bonferroni corrected p-values*).
  - 2) Use  $X_j$ , the partitioning variable selected in step 1), to partition the data into two disjoint subsets (or child nodes),  $A_L$  and  $A_R$ . For each possible split  $S$ , a standardized test statistic (3.3) is computed, and the partition associated with the largest test statistic is used to partition the data into two child nodes.
  - 3) Repeat steps 1)–2) in a recursive fashion on the resulting child nodes until the global hypothesis in step 1) cannot be rejected at a prespecified  $\alpha$  level.
- 

standard scale by which to compare the strength of association between each feature and the response, and results in an unbiased method for selecting split variables, regardless of the scale on which each variable is measured [Hothorn et al., 2006c].

In particular, if we have  $m$  features,  $X_1, X_2, \dots, X_m$ , then we construct  $m$  general tests of independence  $H_0^j : D(\mathbf{Y}|X_j) = D(\mathbf{Y})$ , for  $j = 1, 2, \dots, m$ , using the conditional inference framework briefly discussed in [Section 3.3](#). The features themselves can be measured on different scales; hence, we compare the  $m$  tests on the basis of their  $p$ -values. Since this involves multiple hypothesis tests, the  $p$ -values need to be adjusted to keep the overall *family-wise error rate*  $\leq \alpha$ .<sup>d</sup> The simplest approach is to use a *Bonferroni adjustment*; that is, multiply each  $p$ -value by the total number of tests:  $p_j^* = m \times p_j$ , where  $m$  is the number of features being considered and  $p_j$  is the  $p$ -value from the  $j$ -th test of independence  $H_0^j$ . (R has a built-in function for adjusting  $p$ -values using a number of different approaches; see `?stats::p.adjust` for details.) The predictor associated with the test having the smallest adjusted  $p$ -value meeting a pre-selected threshold ( $\alpha$ ) is chosen as the splitter.

---

<sup>d</sup>For an overview of the problems associated with multiple tests of hypothesis, see Shaffer [1995] and Wright [1992]—the latter discusses the use of adjusted  $p$ -values.

### 3.4.1.1 Example: New York air quality measurements

To illustrate, let's write a simple function, called `gi.test()`, that uses the conditional inference procedure described in Section 3.3 to test the null hypothesis of general independence between two variables  $X$  and  $Y$ .<sup>e</sup> To keep it simple, this function applies only to univariate continuous variables, and computes an approximate  $p$ -value assuming an asymptotic  $\chi^2(1)$  distribution (see Section 3.3.1 for details)—although, it would not be too difficult to modify `gi.test()` to return approximate  $p$ -values using the permutation distribution instead. The arguments `g` and `h` allow for suitable transformations of the variables `x` and `y`, respectively; for example, if the relationship between  $X$  and  $Y$  is monotonic, but not necessarily linear, or if we suspect outliers, then we might consider converting  $X$  and/or  $Y$  to ranks (e.g., `g = rank`)—converting both  $X$  and  $Y$  to ranks is similar in spirit to conducting a correlation test based on Spearman's  $\rho$ . Both arguments default to R's built-in `identity()` function, which has no effect on the given values.

```
gi.test <- function(x, y, g = identity, h = identity) {
  xy <- na.omit(cbind(x, y)) # only retain complete cases
  gx <- g(xy[, 1L]) # transformation function applied to x
  hy <- h(xy[, 2L]) # influence function applied to y
  lin <- sum(gx * hy) # linear statistic
  mu <- sum(gx) * mean(hy) # conditional expectation
  sigma <- var(hy) * sum(gx ^ 2) - # conditional covariance
    var(hy) * sum(gx) ^ 2 / length(hy)
  c.quad <- ((lin - mu) / sqrt(sigma)) ^ 2 # quadratic test statistic
  pval <- 1 - pchisq(c.quad, df = 1) # p-value
  c("chisq" = c.quad, "pval" = pval) # return results
}
```

Continuing with the New York air quality example, let's see which variable, if any, is selected to split the root node. Following convention, I'll use  $\alpha = 0.05$  as the set threshold for failing to reject the global null hypothesis in step 1) of Algorithm 3.1.

The following code chunk applies the previously defined `gi.test()` function to test the null hypothesis of general independence between each of the five features and the response—if you skipped Section 3.3, then you can think of this as a simple test of association that defaults to using a test statistic whose asymptotic distribution (i.e., the approximate distribution for sufficiently large  $N$ ) is  $\chi^2(1)$ . Note that the `p.adjust()` function mentioned earlier is used to adjust the resulting  $p$ -values to account for multiple tests using a simple Bonferroni adjustment:

---

<sup>e</sup>We could also use the much more flexible `independence_test()` function from package `coin`, but writing your own function can help solidify your basic understanding of how the procedure actually works.

```

xnames <- setdiff(names(aq), "Ozone") # feature names
set.seed(1938) # for reproducibility
res <- sapply(xnames, FUN = function(x) { # test each feature
  gi.test(airquality[[x]], y = airquality[["Ozone"]])
})
t(res) # print transpose of results (nicer printing)

#>          chisq      pval
#> Solar.R 13.3476 2.59e-04
#> Wind     41.6137 1.11e-10
#> Temp     56.0863 6.94e-14
#> Month    3.1127 7.77e-02
#> Day      0.0201 8.87e-01

# Bonferroni adjusted p-values (same as 5 * pval in this case)
p.adjust(res["pval", ], method = "bonferroni")

#> Solar.R      Wind      Temp      Month      Day
#> 1.29e-03 5.56e-10 3.47e-13 3.88e-01 1.00e+00

```

In this example, the predictor associated with the smallest adjusted  $p$ -value is `Temp`, and since  $p_{\text{Temp}}^* \approx 3.469 \times 10^{-13} < \alpha = 0.05$ , `Temp` is the first variable that will be used to partition the data. The next step is to determine the optimal split point of `Temp` to use when partitioning the data (step 2) of Algorithm 3.1), which will be discussed in [Section 3.4.2](#).

### 3.4.1.2 Example: Swiss banknotes

Let's try a binary classification problem as well. If you followed [Section 3.3](#) and paid close attention, then you might have figured out that our `gi.test()` function should also work for 0/1 encoded binary variables.

Using the Swiss banknote data ([Section 1.4.1](#)), let's see which, if any, of the available features can be used to effectively partition the root node—recall that all the features are numeric and that the binary response (`y`) is already coded as 0 (for genuine banknotes) and 1 (for counterfeit banknotes):

```

bn <- treemisc::banknote # start with the root node
xnames <- setdiff(names(bn), "y") # feature names
res <- sapply(xnames, FUN = function(x) { # test each feature
  gi.test(bn[[x]], y = bn[["y"]])
})
t(res) # print transpose of results (nicer printing)

#>          chisq      pval
#> length     7.52 6.11e-03
#> left       48.89 2.71e-12
#> right      68.51 1.11e-16
#> bottom     118.61 0.00e+00
#> top        72.22 0.00e+00

```

```
#> diagonal 160.90 0.00e+00
# Bonferroni adjusted p-values (same as 6 * pval in this case)
p.adjust(res["pval", ], method = "bonferroni")
#> length      left      right     bottom      top diagonal
#> 3.67e-02 1.62e-11 6.66e-16 0.00e+00 0.00e+00 0.00e+00
```

Using  $\alpha = 0.05$ , we would select `diagonal` as the primary splitter (since `bottom`, `top`, and `diagonal` are essentially tied in terms of minimum adjusted  $p$ -value, we can just select the one with the max  $\chi^2$  statistic).

We can double check our computations by comparing the results to those produced by `coin`'s `independence_test()` function, which are given below; spoiler alert, the results are a match. The results are a match, nice! The `independence_test()` function is far more general than my `gi.test()` function, and can handle univariate or multivariate variables measured at arbitrary scales (e.g., censored response value, categorical variables with more than two categories, etc.).

```
res <- sapply(xnames, FUN = function(x) {
  it <- independence_test(bn[["y"]] ~ bn[[x]], teststat = "quadratic")
  c("chisq" = statistic(it), "pval" = pvalue(it))
})
t(res) # print transpose of results (nicer printing)

#>             chisq      pval
#> length      7.52 6.11e-03
#> left        48.89 2.71e-12
#> right       68.51 1.11e-16
#> bottom      118.61 0.00e+00
#> top         72.22 0.00e+00
#> diagonal    160.90 0.00e+00

# Bonferroni adjusted p-values (same as 6 * pval in this case)
p.adjust(res["pval", ], method = "bonferroni")
#> length      left      right     bottom      top diagonal
#> 3.67e-02 1.62e-11 6.66e-16 0.00e+00 0.00e+00 0.00e+00
```

Hopefully, by this point, you have a basic understanding of how CTree selects the splitting variable in step 1) of Algorithm 3.1. Let's now turn our attention to finding the optimal split condition for the selected splitter.

### 3.4.2 Finding the optimal split point

Once a splitting variable has been selected, the next step is to find the optimal split point. CTree uses binary splits like those discussed for CART in [Chapter 2](#); in particular, continuous and ordinal variables produce binary splits of the form  $x \leq c$  vs.  $x > c$ , where  $c$  is in the domain of  $x$ , and categorical

variables produce binary splits of the form  $x \in S$  vs.  $x \notin S$ , where  $S$  is a subset of the unique categories of  $x$ .

Finding the optimal split point can be done using any number of strategies, including those discussed in [Chapter 2](#) (e.g., maximizing reduction in node impurity). However, the choice of impurity function depends on the scale of the response (e.g., the Gini index for classification and sum of squares for regression). For this reason, CTree uses the same conditional inference framework for selecting the optimal split point as it does for selecting the optimal splitting variable. Instead of using  $p$ -values, however, the optimal split point is chosen using the individual test statistics (since we don't have to worry about different scales).

Note that for continuous predictors, CTree chooses a cut point from the observed predictor values. This is in contrast to CART, which uses the midpoints of the observed values; see, for example, Breiman et al. [1984, p. 30]. Other tree algorithms have different methods for selecting the split point values for ordered features (e.g., C4.5, which is discussed in the online complements), but this detail rarely matters in practice.

Every binary partition induces a two-sample test between the response values in each group (e.g.,  $\{\mathbf{Y}_i | \mathbf{X}_{ji} \in S\}$  and  $\{\mathbf{Y}_i | \mathbf{X}_{ji} \notin S\}$ ). The conditional inference framework discussed in [Section 3.3](#) is employed again at this step, and a test statistic (3.3) is computed for each possible split. The split associated with the largest test statistic is used to partition the data, before returning to step 1) of Algorithm 3.1.

### 3.4.2.1 Example: New York air quality measurements

Continuing with the New York air quality example, let's find the optimal split point for `Temp`, the feature selected previously in step 1) of Algorithm 3.1 (p. 122), to partition the root node. The code chunk below computes the test statistics for testing  $H_0 : D(\text{Ozone}|\text{Temp} \leq c) = D(\text{Ozone})$ , for each unique value  $c$  of `Temp`; the results are plotted in [Figure 3.2](#).

```
set.seed(912) # for reproducibility
xvals <- sort(unique(aq$Temp)) # potential cut points
splits <- matrix(0, nrow = length(xvals), ncol = 2)
colnames(splits) <- c("cutoff", "chisq")
for (i in seq_along(xvals)) {
  x <- ifelse(aq$Temp <= xvals[i], 0, 1) # binary indicator
  y <- aq$Ozone
  # Ignore pathological splits or splits that are too small
  if (length(table(x)) < 2 || any(table(x) < 7)) {
    res <- NA
  } else {
    res <- gi.test(x, y)["chisq"]
  }
  splits[i,] <- res
}
```

```

}
splits[i, ] <- c(xvals[i], res) # store cutpoint and test statistic
}
splits <- na.omit(splits)
splits[which.max(splits[, "chisq"]), ]

#> cutoff chisq
#> 82.0 55.3

# Plot the test statistic for each cutoff (Figure 3.2)
plot(splits, type = "b", pch = 19, col = 2, las = 1,
      xlab = "Temperature split value (degrees Fahrenheit)",
      ylab = "Test statistic")
abline(v = 82, lty = "dashed")

```

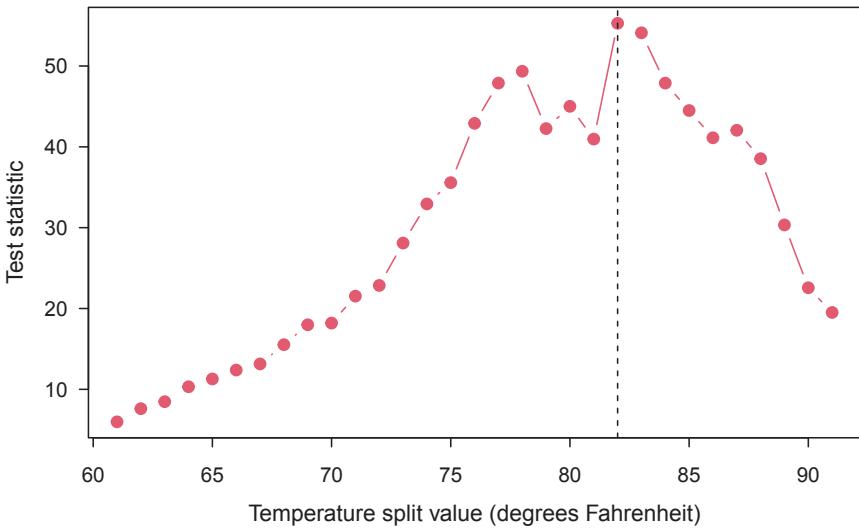


FIGURE 3.2: Test statistics from `gi.test()` comparing the two groups of `Ozone` values for every binary partition using `Temp`. A dashed line shows the optimal split point  $c = 82$ .

From the results, we see that the maximum of all the test statistics is  $c_q = 53.282$  and is associated with the split point  $c = 82$ , giving us our first split in the tree (i.e.,  $\text{Temp} \leq 82$ ).

Following Algorithm 3.1, we would continue splitting each of the resulting child nodes until the global null hypothesis in step 1) of Algorithm 3.1 cannot be rejected at the specified  $\alpha$  level. For example, applying the previous code to the resulting left child node ( $\text{Temp} \leq 82$ ) would result in a further partition using  $\text{Wind} \leq 6.9$ . I'll confirm these calculations using specific CTree software in [Section 3.5.1](#).

### 3.4.3 Pruning

Unlike CART and other exhaustive search procedures, CTree uses statistical stopping criteria (e.g., Bonferroni-adjusted  $p$ -values) to determine tree size and does not require pruning—although, pruning can still be beneficial in certain circumstances (see [Section 3.4.5](#)). That’s not to say that CTree doesn’t overfit. As we’ll see in [Sections 3.4.5](#) and [3.5](#), the threshold  $\alpha$  has a direct impact on the size and therefore complexity of the tree and is often treated as a tuning parameter.

### 3.4.4 Missing values

Similar to CART, CTree can use the idea of surrogates to handle missing values, but it is not the default in current implementations of CTree. By default, observations which can’t be classified to a child node because of missing values are either 1) randomly assigned according to the node distribution (as in the newer **partykit** package [Hothorn and Zeileis, 2021]), or 2) go with the majority (as in the older **party** package [Hothorn et al., 2021b]).

Observations with missing values in predictor  $X$  are simply ignored when computing the associated test statistics during step 1) of Algorithm 3.1 (p. 122). Similarly, missing values associated with the splitting variable are also ignored when computing the test statistics in step 2). Once a split has been found, surrogates can be constructed using an approach similar to the one described in [Section 2.7](#) for CART, in particular, creating a binary decision stump using the binary split in question as the response and trying to find the best splits associated with it using Algorithm 3.1 (p. 122).

### 3.4.5 Choice of $\alpha$ , $g()$ , and $h()$

From an inferential standpoint,  $\alpha$  is the prespecified nominal level of the general independence tests used for feature selection in step 1) of Algorithm 3.1, which controls the probability of type I error. Recall that the type I error (or a *false positive*) occurs when we reject a true null hypothesis—or in our case, when we conclude that  $\mathbf{X}$  and  $\mathbf{Y}$  are not independent, when in fact they are (i.e., potentially using an irrelevant splitting variable). Obviously, we want the probability of a type I error to be low, which is why we fix  $\alpha$  to a small number (e.g.,  $\alpha = 0.05$ ).

Although  $\alpha$  controls the probability of falsely rejecting  $H_0$  in each node, we still need to consider the *statistical power* of each test; that is, the probability of rejecting  $H_0$  when it is false. In the context of recursive partitioning, power essentially dictates the chance of selecting a relevant predictor at each node. As noted in Hothorn et al. [2006c], the general tests of independence

used in CTree will only have high power for certain directions of deviation from independence and depends on the choice of  $g()$  and  $h()$ . A useful guide for selecting  $g()$  and  $h()$  can be found in Table 4 of **coin**'s “Implementing a Class of Permutation Tests: The **coin** Package” vignette; to view, use `vignette("Implementation", package = "coin")` at the R console.

In the presence of outliers, the general test of independence discussed in [Section 3.3](#) would be more powerful at a given sample size and  $\alpha$  if  $g()$  and  $h()$  converted  $\mathbf{X}$  and  $\mathbf{Y}$  to ranks (because ranks are more robust to outlying observations).

To illustrate, let's run a quick Monte Carlo experiment. Suppose  $\mathbf{X} = X$  has a standard normal distribution and that  $\mathbf{Y} = Y$  is equal to  $X$  with a tad bit of noise:  $Y = X + \epsilon$ , where  $\epsilon \sim N(0, \sigma = 0.1)$ . [Figure 3.3](#) shows a scatterplot for two random samples of size  $N = 100$  generated from  $X$  and  $Y$ . The left panel in [Figure 3.3](#) shows a clear association between  $X$  and  $Y$ . The right panel shows a scatterplot of the same sample, but with three of the observations replaced by outliers. Even with the outliers, there is still a clear relationship between  $X$  and  $Y$ .

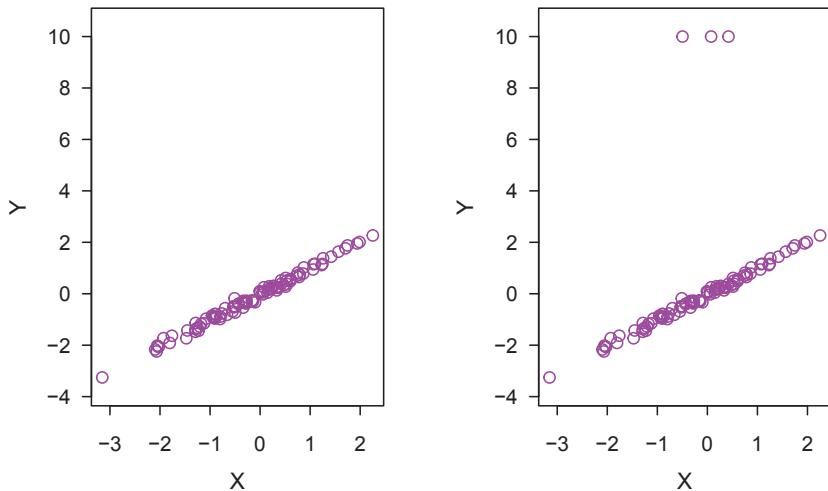


FIGURE 3.3: Scatterplots of two linearly related variables. Left: original sample. Right: original sample with three observations replaced by outliers.

The code chunk below applies the `gi.test()` function, using both the identity and rank transformations, to 100 random samples from  $X$  and  $Y$  for sample sizes ranging from 10–100; note that each sample includes three outlying  $Y$  values. For each sample size, the approximate power of each test is computed as the proportion of times out of 100 that the null hypothesis was rejected

at the  $\alpha = 0.05$  level. The results are plotted in [Figure 3.4](#). Clearly, using ranks provides a more powerful test across the range of sample sizes in this example.

```
set.seed(2142) # for reproducibility
N <- seq(from = 5, to = 100, by = 5) # range of sample sizes
res <- sapply(N, FUN = function(n) {
  pvals <- replicate(1000, expr = { # simulate 1,000 p-values
    x <- rnorm(n, mean = 0) # from each test
    y <- x + rnorm(length(x), mean = 0, sd = 0.1)
    y[1:3] <- 10 # insert outliers
    test1 <- gi.test(x, y) # no transformations
    test2 <- gi.test(x, y, g = rank, h = rank) # convert to ranks
    c(test1["pval"], test2["pval"])) # extract p-values
  })
  apply(pvals, MARGIN = 1, FUN = function(x) mean(x < 0.05))
})

# Plot the results (Figure 3.4)
plot(N, res[2L, ], xlab = "Sample size", ylab = "Power", type = "l",
      ylim = c(0, 1), las = 1)
lines(N, res[1L, ], col = 2, lty = 2)
legend("bottomright",
       legend = c("Rank transformation", "No transformation"),
       lty = c(1, 2), col = c(1, 2), inset = 0.01,
       box.col = "transparent")
```

Remember, in recursive partitioning, the sample size decreases as you proceed further down the tree. Consequently, the tests become less powerful the further down the tree they are. Similar to CART-like decision trees, the more accurate splits tend to occur at the top.

Hothorn et al. [2006c] suggest that increasing  $\alpha$  can help assure that any type of dependence structure is detected. However, increasing  $\alpha$  will result in more splits and therefore a more complex tree. To avoid overfitting in this situation, pruning can be applied in a variety of ways, for example, collapsing terminal nodes that don't meet a second threshold  $\alpha' < \alpha$ .

Similar to the cost-complexity parameter ( $cp$ ) in CART ([Section 2.5.1](#)), you can also think of  $\alpha$  as a tuning parameter controlling the overall complexity of the tree (i.e., the number of terminal nodes)—with smaller values of  $\alpha$  leading to shallower trees. Hence,  $\alpha$  can be optimized using cross-validation or similar techniques, in which case, the former definition related to the type I error rate no longer applies.

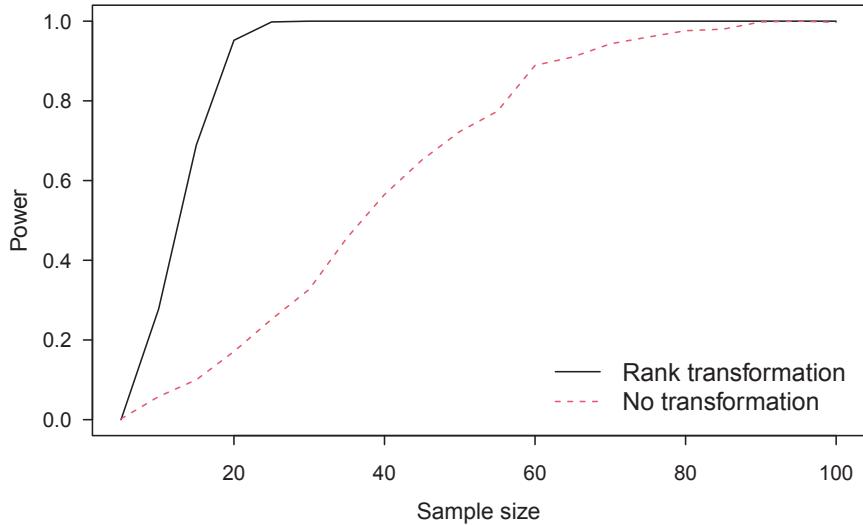


FIGURE 3.4: Power vs. sample size for the general test of independence between two univariate continuous variables,  $X$  and  $Y$ , using the conditional inference procedure outlined in [Section 3.3](#). The solid black curve corresponds to using ranks, whereas the dashed red curve corresponds to the identity (i.e., no transformation).

### 3.4.6 Fitted values and predictions

For univariate regression and classification trees, fitted values and predictions are obtained in the same manner as they are in CART. For example, the within-terminal node class proportions can be used for class probability estimates. For regression, the within-terminal node sample mean of the response values can be used as the fitted values or for predicting new observations.

However, CTree is quite flexible and can handle many other situations beyond simple classification and regression. With censored outcomes, for example, each terminal node can be summarized using the Kaplan-Meier estimator; see [Section 3.5.3](#). The median survival time can be used for making predictions.

### 3.4.7 Imbalanced classes

Unlike CART, CTree cannot explicitly take into account prior class probabilities, nor can it account for unequal misclassification costs. However, it can assign increased weight to specific observations (for brevity, I omitted the case

weights from the formulas in [Section 3.3](#), so see the references listed there for full details). For instance, we can assign higher weights to observations associated with higher misclassification costs. There is a drawback to this approach, however. Increasing the case weights essentially corresponds to increasing the sample size  $N$  in the statistical tests used in Algorithm 3.1, which will result in smaller  $p$ -values. Consequently, the resulting tree can be much larger since more splits will be significant. Decreasing  $\alpha$  and/or employing the tuning strategy discussed in [Section 3.4.5](#) can help, but it can be a difficult balancing act.

### 3.4.8 Variable importance

In contrast to the impurity-based variable importance measure used in CART ([Section 2.8](#)), CTree uses a *permutation*-based framework similar to the procedure outlined in Algorithm 6.1 (p. 205). This is not the same “permutation” as in “permutation test,” but rather, in how the individual feature columns are randomly permuted, one at a time, before recording the difference in prediction performance (as compared to the baseline performance when none of the feature columns are permuted).

---

## 3.5 Software and examples

Conditional inference trees are available in the **party** package via the `ctree()` function.<sup>f</sup> However, the **partykit** package contains an improved reimplementation of `ctree()` and is recommended for fitting individual conditional inference trees in general. The `ctree()` function in **partykit** is much more modular and written almost entirely in R; this flexibility seems to come at a price, however, as `partykit::ctree()` can be much slower than `party::ctree()` (the latter of which is implemented in C). It’s also worth noting that **partykit** is quite extensible and allows you to coerce tree models from different sources into a standard format that **partykit** can work with (e.g., importing trees saved in the *predictive model markup language* (PMML) format [Data Mining Group, 2014]). A good example of this is the R package **C5.0** [Kuhn and Quinlan, 2021], which provides an interface to C5.0 (C5.0, which evolved out of C4.5, is discussed in the online complements).

The R package **boot** [Canty and Ripley, 2021] can be used to carry out general permutation tests, as well as more general bootstrap procedures; for permutation tests, see the example code in Davison and Hinkley [1997, [Sec. 4.3](#)].

---

<sup>f</sup>The package name is apparently a play on the words “**partition** **y**”.

The **coin** package implements the conditional inference procedures briefly discussed in [Section 3.3](#); in short, **coin** provides a common framework for general tests of independence, including: two-sample tests,  $K$ -sample tests, and correlation-based tests, for continuous, nominal, ordered, and multivariate data.

If you’re not an R user, then you may be out of luck, as I’m unaware of any non-R implementations of CTree—all the more reason to be open to more than one opensource language!

Although there are many differences between `partykit::ctree()` and `party::ctree()`, both will produce the same tree almost every time under the default settings. For more information on the differences between the two, see [Section 7.4](#) of **partykit**’s “ctree: Conditional Inference Trees” vignette, which can be viewed in R by calling `utils::vignette("ctree", package = "partykit")`.

The following examples illustrate the basic use of **party** and **partykit** for unbiased recursive partitioning via conditional inference. I’ll confirm the results I computed manually in previous sections, as well as construct conditional inference trees for new data sets, including both regression and survival examples.

### 3.5.1 Example: New York air quality measurements

In earlier sections, we used our own `gi.test()` function to split the root node of the `airquality` data set. Using conditional inference, we found that the first best split occurred with `Temp <= 82`. Now, let’s use **partykit** to apply Algorithm 3.1, and recursively split the `airquality` data set until we can no longer reject the null hypothesis of general independence between `Ozone` and any of the five numeric features at the  $\alpha = 0.05$  level.

You can use `ctree_control()` to specify a number of parameters governing the CTree algorithm; in the code chunk below, we stick with the defaults; see `?partykit::ctree_control` for a description of all the parameters that can be set. In this chapter, we used the quadratic form of the test statistic  $c_q$  for steps 1)–2) of Algorithm 3.1, which is the default in both **party** and **partykit** (`teststat = "quad"`). We also stick with the default Bonferroni adjusted  $p$ -values. To specify  $\alpha$ , set either the `alpha` or `mincriterion` arguments in `ctree_control()`, where the value of `mincriterion` corresponds to  $1 - \alpha$  (only the `mincriterion` argument is available in **party**); both packages use a default significance level of  $\alpha = 0.05$ . In **party**’s implementation of `ctree()`, the transformation functions `g()` and `h()` can be specified via the `xtrafo` and `ytrafo` arguments, respectively; in **partykit**’s implementation, only `ytrafo` is available.

Next, I call `ctree()` to recursively partition the data and plot the resulting tree diagram using `partykit`'s built-in `plot()` method (see Figure 3.5):

```
library(partykit)

# Fit a default CTree using Bonferroni adjusted p-values
aq <- airquality[!is.na(airquality$Ozone), ]
(aq.cit <- ctree(Ozone ~ ., data = aq))

#>
#> Model formula:
#> Ozone ~ Solar.R + Wind + Temp + Month + Day
#>
#> Fitted party:
#> [1] root
#> |   [2] Temp <= 82
#> |   |   [3] Wind <= 6.9: 56 (n = 10, err = 21946)
#> |   |   [4] Wind > 6.9
#> |   |   |   [5] Temp <= 77: 18 (n = 48, err = 3956)
#> |   |   |   [6] Temp > 77: 31 (n = 21, err = 4621)
#> |   |   [7] Temp > 82
#> |   |   |   [8] Wind <= 10.3: 82 (n = 30, err = 15119)
#> |   |   |   [9] Wind > 10.3: 49 (n = 7, err = 1183)
#>
#> Number of inner nodes:    4
#> Number of terminal nodes: 5

plot(aq.cit) # Figure 3.5
```

Note that I again removed the rows with missing response values. The fitted tree contains four splits (i.e., five terminal nodes) on only two predictors: `Temp` and `Wind`. The `plot()` method for `ctree()` objects is quite flexible, and I encourage you to read the documentation in `?partykit::plot`. By default, the terminal nodes are summarized using an appropriate plot that depends on the scale of the response variable—in this case, boxplots. The  $p$ -values from step 1) of Algorithm 3.1 are printed in each node, along with the selected splitting variable and the node number:

In `partykit`, we can print the test statistics and adjusted  $p$ -values associated with any node using the `sctest()` function from package `strucchange` [Zeileis et al., 2019], which is illustrated below; the 1 specifies the node of interest, which, according to the printed output and tree diagram, corresponds to the root node.. These correspond to the tests carried out in step 1) of Algorithm 3.1. The results are a match to our earlier computations using `gi.test()` and `p.adjust()`, woot! As far as I'm aware, you cannot currently obtain the test statistics from step 2) in `partykit`, although this is possible in `party`'s implementation of `ctree()`, which I'll demonstrate next.

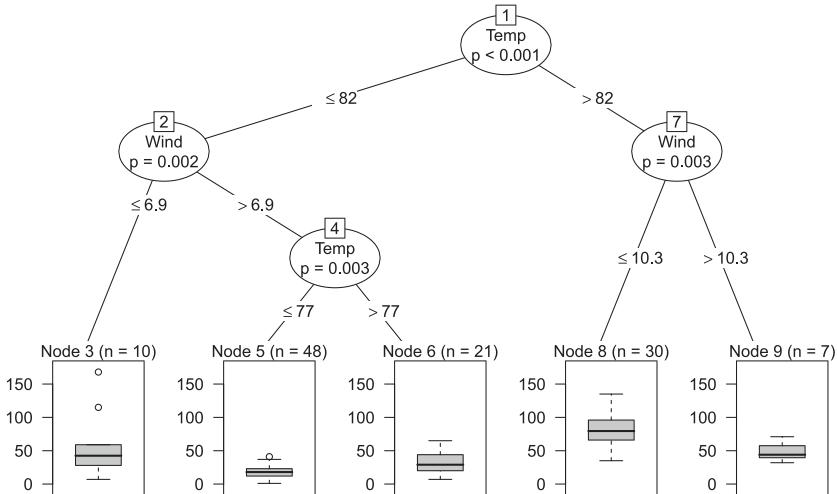


FIGURE 3.5: A default CTree fit to the New York air quality measurements data set.

```
strucchange::sctest(aq.cit, 1)
#>          Solar.R      Wind      Temp Month     Day
#> statistic 13.34761 4.16e+01 5.61e+01 3.113 0.0201
#> p.value    0.00129 5.56e-10 3.47e-13 0.333 1.0000
```

When fitting conditional inference trees with **party**, the **nodes()** function can be used to extract a list of nodes of a tree; the **where** argument specifies the node ID (i.e., the node numbers used to label the nodes in the associated tree diagram). Below, I'll refit the same tree using **party::ctree()**, extract the split associated with the root node, and plot the corresponding test statistics comparing the different cut points of the split variable (in this case, **Temp**). Note that **party::ctree()** only uses the maximally selected statistic ( $c_m$ ) for step 2) of Algorithm 3.1<sup>g</sup>, but recall that in the univariate case,  $c_m^2 = c_q$ , so I'll square them and compare them to the results I plotted earlier in Figure 3.2 (p. 127). As they should, the results from **party::ctree()**, which are displayed in Figure 3.6, match with what I obtained earlier using my **gi.test()** function.

```
aq.cit2 <- party::ctree(Ozone ~ ., data = aq) # refit the same tree
root <- party::nodes(aq.cit2, where = 1)[[1L]] # extract root node
split.stats <- root$psplit$splitstatistic # split statistics
cutpoints <- aq[[root$psplit$variableName]][split.stats > 0]
cq <- split.stats[split.stats > 0] ^ 2
```

<sup>g</sup>In contrast, **partykit** lets you choose which test statistic to use in step 2) of Algorithm 3.1, and defaults to the quadratic form  $c_q$  we used earlier in **gi.test()**.

```
# Plot split statistics (Figure 3.6; compare to Figure 3.2)
plot(cutpoints[order(cutpoints)], cq[order(cutpoints)], col = 4,
      pch = 19, type = "b", las = 1,
      xlab = "Temperature split value (degrees Fahrenheit)",
      ylab = "Test statistic")
abline(v = root$psplit$splitpoint, lty = "dashed")
```

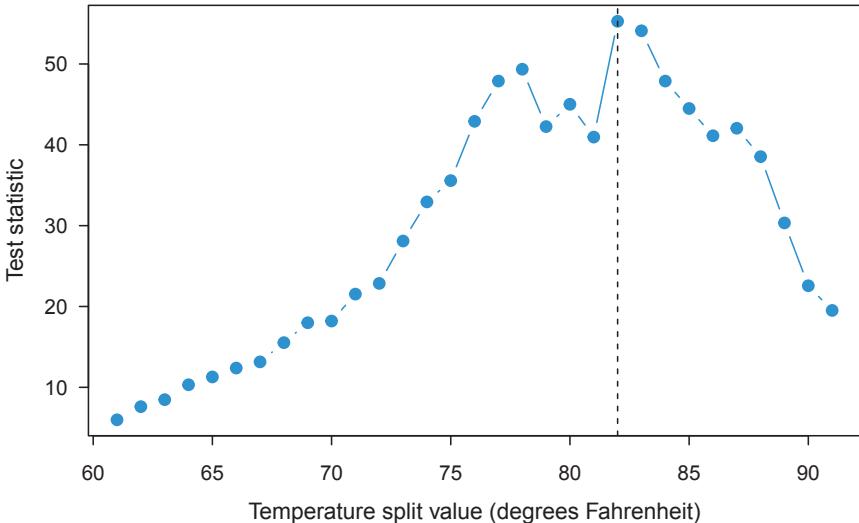


FIGURE 3.6: Test statistics from `party::ctree()` comparing the two groups of `Ozone` values for every binary partition using `Temp`. A dashed line shows the optimal split point  $c = 82$ . Compare these results to those from [Figure 3.2](#).

You can coerce decision trees produced by various other implementations into "party" objects using the `partykit::as.party()` function. This means, for example, we can fit a decision tree using `rpart`, and visualize it using `partykit`'s `plot()` method.

To illustrate, the next code chunk fits an `rpart` tree to the same `aq` data, coerces it to a "party" object, and plots the associated tree diagram. Here, I'll set the complexity parameter  $c_p$  to zero (i.e., no penalty on the size of the tree) and use the default 10-fold cross-validation along with the 1-SE rule to prune the tree ([Section 2.5.2.1](#)). In this example, CART produced a decision stump (i.e., a tree with only a single split).

```
set.seed(1525) # for reproducibility
aq.cart <- rpart::rpart(Ozone ~ ., data = aq, cp = 0)
aq.cart.pruned <- treemisc::prune_se(aq.cart, se = 1) # 1-SE rule
plot(partykit::as.party(aq.cart.pruned))
```

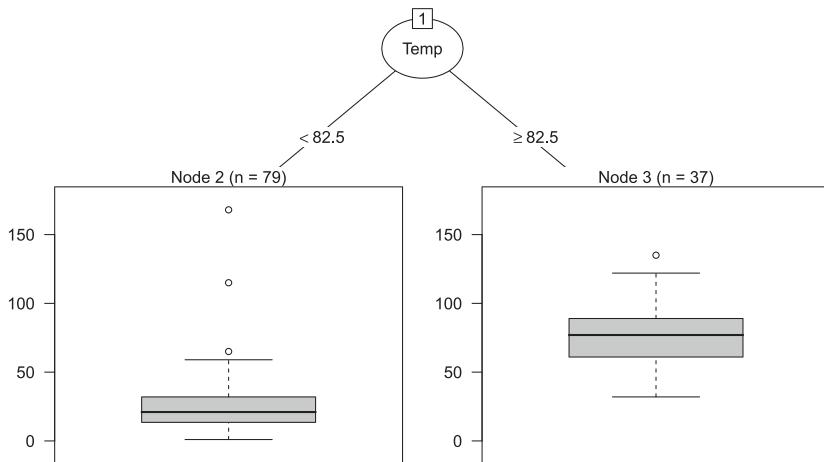


FIGURE 3.7: CART-like decision tree fit to the New York air quality measurements data set. The tree was pruned according to the 1-SE rule discussed in [Section 2.5.2.1](#).

### 3.5.2 Example: wine quality ratings

Ordinal outcomes are common in scientific research and everyday practice. In medical and epidemiological studies, for example, the ordinal response often represents the levels of a standard measurement scale such as severity of pain (e.g., none < mild < moderate < severe) [Harrell, 2015, p. 311].

In this example, I'll take up the wine quality data introduced in [Section 1.4.8](#). The goal is to model the quality of red wines based solely on physicochemical tests, like acidity, and interpret the results. The target variable, the quality of the wine, is an ordinal variable from 0–10 derived from wine tasting by human experts. Note that in this data set, however, we only have observed ratings in the range 3–9. See `?treemisc::wine` for additional background and column information.

As mentioned in [Section 1.4.8](#), this is a well-known data set and is often used in statistical learning tutorials. However, many individuals treat this as a classification problem by arbitrarily dichotomizing the wine quality score and treating it as a binary classification problem. This is poor statistical practice and results in loss of information. Such “dichotomania” [Senn, 2005] is unfortunately also prevalent in medical and epidemiological research.

With CTree, we can easily model the outcome as an ordinal variable. To start, I'll load the data from **treemisc** and coerce the (integer) response to an *ordered factor*; see `?as.ordered` for details:

```
wine <- treemisc::wine
reds <- wine[wine$type == "red", ] # reds only
rm(wine) # remove from global environment
reds$type <- NULL # remove column
reds$quality <- as.ordered(reds$quality) # coerce to ordinal
head(reds$quality) # print first few quality scores

#> [1] 5 5 5 6 5 5
#> Levels: 3 < 4 < 5 < 6 < 7 < 8
```

Next, I'll fit a (default) conditional inference tree via **partykit**. Unfortunately, the tree diagram is too large to print neatly on this page, so I'll show a printout of the fitted tree instead:

```
(reds.cit <- ctree(quality ~ ., data = reds))

#>
#> Model formula:
#> quality ~ fixed.acidity + volatile.acidity + citric.acid + residua...
#>     chlorides + free.sulfur.dioxide + total.sulfur.dioxide +
#>     density + pH + sulphates + alcohol
#>
#> Fitted party:
#> [1] root
#> |   [2] alcohol <= 10.5
#> |   |   [3] volatile.acidity <= 0.3
#> |   |   |   [4] sulphates <= 0.7: 5 (n = 27, err = 48%)
#> |   |   |   [5] sulphates > 0.7: 6 (n = 58, err = 41%)
#> |   |   |   [6] volatile.acidity > 0.3
#> |   |   |   |   [7] volatile.acidity <= 0.7
#> |   |   |   |   |   [8] alcohol <= 9.8
#> |   |   |   |   |   |   [9] total.sulfur.dioxide <= 39: 5 (n = 171, er...
#> |   |   |   |   |   |   [10] total.sulfur.dioxide > 39
#> |   |   |   |   |   |   |   [11] pH <= 3.4: 5 (n = 205, err = 22%)
#> |   |   |   |   |   |   |   [12] pH > 3.4: 5 (n = 53, err = 42%)
#> |   |   |   |   |   |   |   [13] alcohol > 9.8: 6 (n = 228, err = 54%)
#> |   |   |   |   |   |   |   [14] volatile.acidity > 0.7
#> |   |   |   |   |   |   |   [15] fixed.acidity <= 8.5: 5 (n = 172, err = 26%)
#> |   |   |   |   |   |   |   [16] fixed.acidity > 8.5: 5 (n = 69, err = 35%)
#> |   |   [17] alcohol > 10.5
#> |   |   |   [18] volatile.acidity <= 0.9
#> |   |   |   |   [19] sulphates <= 0.6
#> |   |   |   |   |   [20] volatile.acidity <= 0.3: 6 (n = 33, err = 45%)
#> |   |   |   |   |   [21] volatile.acidity > 0.3: 6 (n = 207, err = 45%)
#> |   |   |   |   |   [22] sulphates > 0.6
#> |   |   |   |   |   |   [23] alcohol <= 11.5
#> |   |   |   |   |   |   |   [24] total.sulfur.dioxide <= 49
#> |   |   |   |   |   |   |   |   [25] volatile.acidity <= 0.4: 7 (n = 72, e...
```

```
#> |   |   |   |   | [26] volatile.acidity > 0.4: 6 (n = 80, err...
#> |   |   |   |   | [27] total.sulfur.dioxide > 49: 6 (n = 55, err...
#> |   |   |   | [28] alcohol > 11.5: 7 (n = 142, err = 48%)
#> |   |   | [29] volatile.acidity > 0.9: 5 (n = 27, err = 59%)
#>
#> Number of inner nodes: 14
#> Number of terminal nodes: 15
```

To see how well the model performs (on the learning sample), we can cross-classify the observed quality ratings with the fitted values (i.e., the prediction from the learning sample):

```
p <- predict(reds.cit, newdata = reds) # fitted values
table(predicted = p, observed = reds$quality) # contingency table

#>          observed
#> predicted  3   4   5   6   7   8
#>       3   0   0   0   0   0   0
#>       4   0   0   0   0   0   0
#>       5   9  33  483  194   5   0
#>       6   1  20  191  361  85   3
#>       7   0   0   7  83  109  15
#>       8   0   0   0   0   0   0
```

For example, of all the red wines with a rating quality score of 7, 5 were predicted to have a quality rating of 5, 85 were predicted to have a quality rating of 6, and the rest (109) were predicted to have a quality rating of 7.

So which variables seem to be the most predictive of the wine quality rating? At first glance, alcohol by volume (`alcohol`) and volatile acidity (`volatile.acidity`) seem to be important predictors, as they appear at the top of the tree and are used multiple times to partition the data. We can quantify this in CTree using `partykit`'s `varimp()` function. This function computes importance using a permutation-based approach akin to the procedure discussed in [Section 6.1.1](#). For now, just think of the returned importance scores as an estimate of the decrease in performance as a result of removing the effect of the predictor in question. By default, performance is measured by the *negative log-likelihood*<sup>h</sup>.

```
set.seed(2023) # for reproducibility
(vi <- varimp(reds.cit, nperm = 100)) # variable importance scores
dotchart(vi, pch = 19, xlab = "Variable importance") # Figure 3.8

#>           alcohol      volatile.acidity
#>           0.5465          0.3537
#>           sulphates total.sulfur.dioxide
```

---

<sup>h</sup>For ordinal outcomes in CTree, the log-likelihood is defined as  $\sum_{i=1}^N \log(p_i)/N$ , where  $p_i$  is the proportion of observations in the same node as case  $i$  sharing the same class.

```
#>          0.1852      0.0395
#>      pH      fixed.acidity
#>  0.0235      0.0188
```

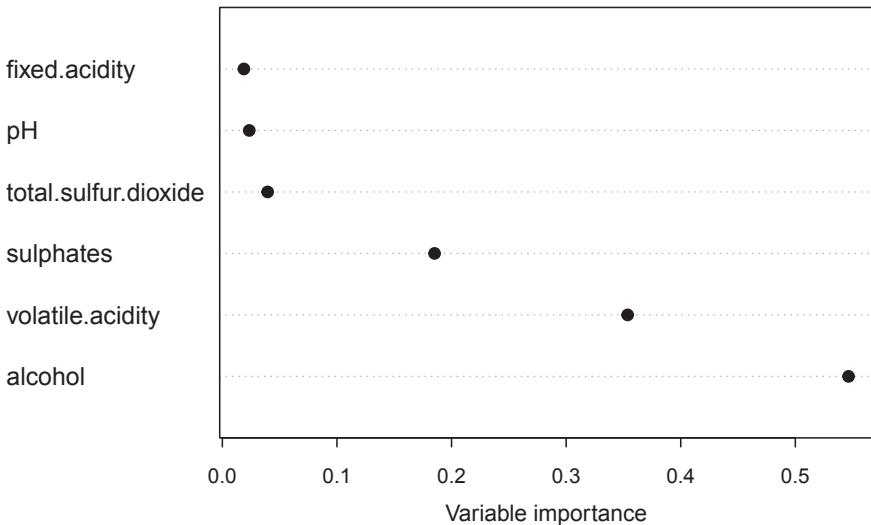


FIGURE 3.8: Variable importance plot for the red wine quality conditional inference tree.

As we suspected from looking at the tree's output, alcohol by volume and volatile acidity, followed by sulphate, are the most important predictors of wine quality rating in this model. The partial dependence (Section 6.2.1) of wine quality rating against each of the top three predictors is given in Figure 3.9<sup>i</sup>; note that the  $y$ -axis is interpreted on the same scale as the response. Here we can see the functional effect of each predictor. For example, alcohol has a monotonic increasing relationship with the predicted quality score. This makes sense and is probably why I never buy any red wine that's less than 14% alcohol by volume. Do the effects of the other two predictors make sense to you?

### 3.5.3 Example: Mayo Clinic liver transplant data

In this example, I'll revisit the PBC data described in Section 1.4.9. A tree-based analysis of the data was briefly discussed in Ahn and Loh [1994]. Below we load the **survival** package and prepare the data:

---

<sup>i</sup>As always, the code to reproduce this plot is available on the book website.

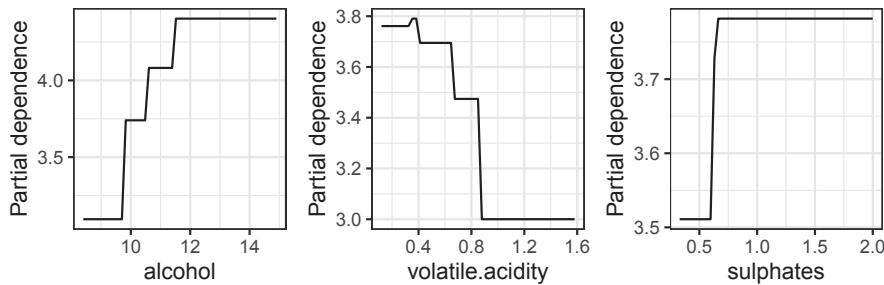


FIGURE 3.9: Partial dependence of wine quality rating on alcohol by volume (left), volatile acidity (middle), and level of potassium sulphate (right).

```
library(survival)

# Prep the data a bit
pb2 <- pb[!is.na(pb$trt), ] # use randomized subjects
pb2$id <- NULL # remove ID column
# Consider transplant patients to be censored at day of transplant
pb2$status <- ifelse(pb2$status == 2, 1, 0)
facs <- c("sex", "spiders", "hepato", "ascites", "trt", "edema")
for (fac in facs) { # coerce to factor
  pb2[[fac]] <- as.factor(pb2[[fac]])
}
```

As briefly discussed in [Section 1.4.9](#), survival (or reliability) analysis is concerned with the distribution of lifetimes, typically of humans, animals, or components of a machine. The response variable in survival analysis is *time-to-event*, for example, time until death or readmission (e.g., for subjects in a randomized clinical trial comparing two drugs) or time until failure of some component in a machine (e.g., a running motor).

Kaplan-Meier estimates of the survival function for both the control and treatment (i.e., drug) group were given in [Figure 1.11](#). As with all survival curves, the chance of survival decreases with time. Often we don't care as much about the overall survival curve but rather how it varies between groups. In this example, our goal is to see if the additional covariates can be used to usefully discriminate groups with different survival distributions.

An overview on tree-structured survival models is provided in Loh [2014]. The approach taken in CTree is rather straightforward. The follow-up times (**time**) and event indicator (**status**) are combined into a single numeric response that is treated as univariate continuous; hence, the techniques discussed earlier apply directly. In CTree, right-censored data are converted to log-rank scores using the **logrank\_trafo()** function from package **coin**; that is, the

influence function  $h()$  converts the right-censored survival times to log-rank scores.

To illustrate, let's test the null hypothesis of general independence between the predictor `bili` (serum bilirubin (mg/dl)) and the log-rank scores of the response:

```
library(coin)

independence_test(Surv(time, status) ~ bili, data = pbc2,
                  teststat = "quadratic")

#>
#> Asymptotic General Independence Test
#>
#> data: Surv(time, status) by bili
#> chi-squared = 77, df = 1, p-value <2e-16

# Our `gi.test()` function from earlier should also work
lr.scores <- coin::logrank_trafo(Surv(pbc2$time, pbc2$status))
gi.test(pbc2$bili, y = lr.scores)

#> chisq  pval
#> 77.5   0.0
```

Using  $\alpha = 0.05$ , we would reject the null hypothesis ( $p < 0.001$ ) and conclude that the level of serum bilirubin is associated with survival rate. But this doesn't tell us much beyond that. Do subjects with higher levels of serum bilirubin tend to survive longer? To answer questions like this, we can use CTree to recursively partition the data using conditional inference-based tests of independence between each feature and the log-rank scores:

```
(pbc2.cit <- partykit::ctree(Surv(time, status) ~ ., data = pbc2))

#>
#> Model formula:
#> Surv(time, status) ~ trt + age + sex + ascites + hepato + spiders +
#>     edema + bili + chol + albumin + copper + alk.phos + ast +
#>     trig + platelet + protime + stage
#>
#> Fitted party:
#> [1] root
#> | [2] bili <= 1.9
#> | | [3] edema in 0
#> | | | [4] stage <= 2: Inf (n = 61)
#> | | | [5] stage > 2: 4191 (n = 104)
#> | | | [6] edema in 0.5, 1: Inf (n = 16)
#> | | [7] bili > 1.9
#> | | | [8] protime <= 11.2
#> | | | | [9] age <= 44.5
#> | | | | | [10] bili <= 5.6: 3839 (n = 29)
#> | | | | | [11] bili > 5.6: 1080 (n = 7)
```

```
#> |   |   [12] age > 44.5: 1487 (n = 45)
#> |   |   [13] protime > 11.2
#> |   |   [14] albumin <= 3.6: 597 (n = 43)
#> |   |   [15] albumin > 3.6: 2540 (n = 7)
#>
#> Number of inner nodes:    7
#> Number of terminal nodes: 8
```

Notice how treatment group (`drug`) was not selected as a splitting variable at any node. This is not surprising since Fleming and Harrington [1991, p. 2] concluded that there was no practically significant difference between the survival times of those taking the placebo and those taking the drug.

We can also display the tree diagram using the `plot()` method; the results are displayed in [Figure 3.10](#). For censored outcomes, the Kaplan-Meier estimate of the survival curve is displayed in each node. The tree diagram in [Figure 3.10](#) makes it clear that subjects with higher serum bilirubin levels tended to have shorter survival times. What other conclusions can you draw from the tree diagram?

```
plot(pbc2.cit) # Figure 3.10
```

---

## 3.6 Final thoughts

CTree is one of the more important developments in recursive partitioning in the last two decades; other important developments are discussed in [Chapter 4](#), as well as the online complements to this book. In summary, compared to CART, CTree:

- uses adjusted statistical tests to separately determine the split variable and split point at each node (CART just uses an exhaustive search);
- provides unbiased split variable selection;
- does not require pruning (or much tuning);
- can naturally take into account the nature of the data—for example, when the variables are of arbitrary type (e.g., multivariate, ordered, right-censored, etc.).

If CTree is competitive with CART in terms of accuracy, doesn't require post pruning, and provides unbiased split variable selection, then why is CART still so popular? As pointed out in the rejoinder to Loh [2014], “This seems to tie in with a third bad effect: Many authors who propose or apply tree algorithms either are not aware of—or choose to ignore—similar work in that area. It

happens that even recent papers do not refer to work carried out from 2000 onward, therefore ignoring more than a decade of active development that may be highly relevant.” Another important factor is software availability. Many tree algorithms do not have easy to use opensource implementations. For example, of the 99 tree algorithms considered by Rusch and Zeileis (see their discussion at the end of Loh [2014]), roughly one-third had free opensource implementations available (including CART and CTree). CART-like decision trees are also broadly implemented across a variety of opensource platforms (see [Section 2.9](#)). CTree, on the other hand, is only available in R—as far as I’m aware.

Should you be concerned about biased variable selection when using CART-like decision trees? Certainly. However, as pointed out in Loh’s rejoinder in Loh [2014], “...selection bias may not cause serious harm if a tree model is used for prediction but not interpretation, in some situations.” While biased variable selection can lead to more spurious splits on irrelevant features, if the sample size is large and there are not too many such variables, pruning with cross-validation is often effective at removing them.

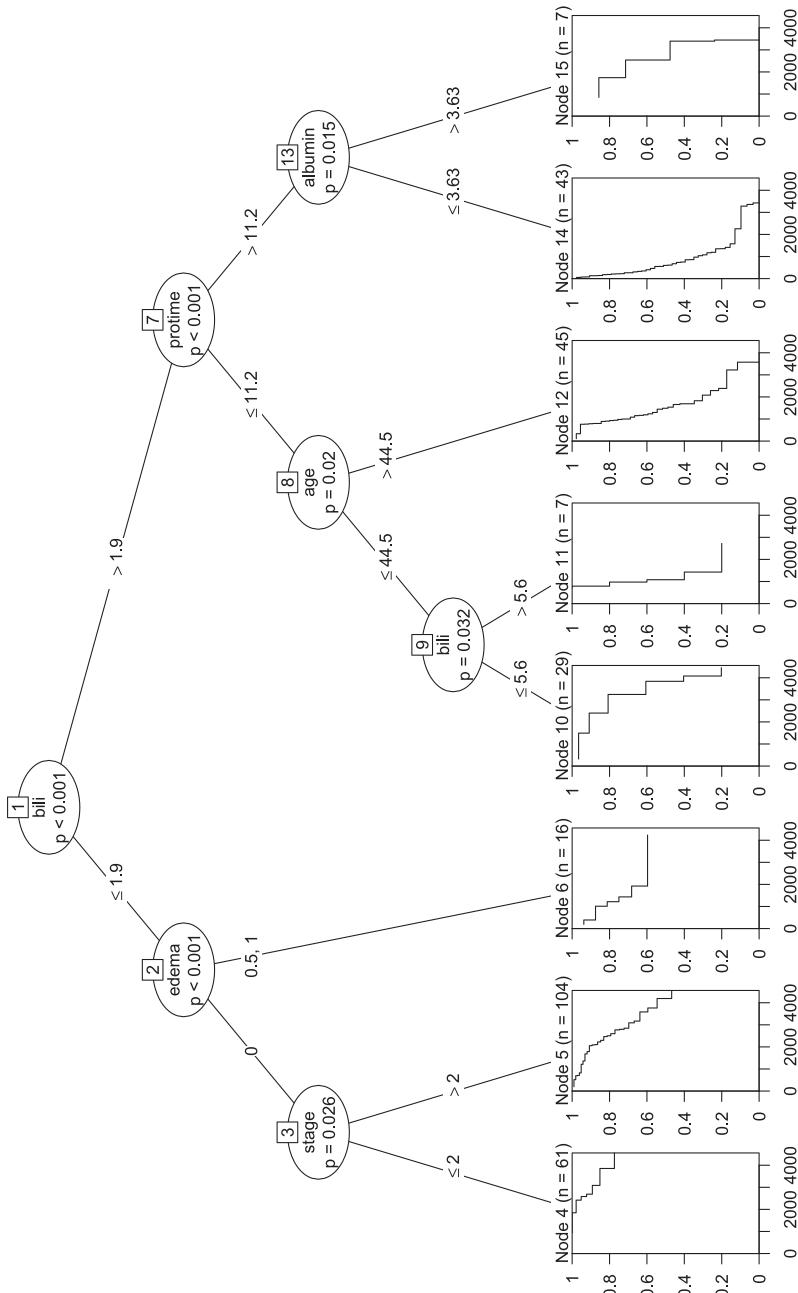


FIGURE 3.10: Conditional inference tree fit to a subset of the Mayo Clinic PBC data. The terminal nodes are summarized using Kaplan-Meier estimates of the survival function. The tree diagram highlights potential risk factors associated with different survival distributions.



Taylor & Francis  
Taylor & Francis Group  
<http://taylorandfrancis.com>

# 4

---

## *The hitchhiker's GUIDE to modern decision trees*

---

Of all the trees we could have hit, we had to get one that hits back.

J.K. Rowling

Harry Potter and the Chamber of Secrets

---

This chapter introduces a powerful, yet seemingly lesser known, decision tree algorithm for *generalized, unbiased, interaction detection, and estimation* (GUIDE), which was introduced in Loh [2002, 2009]. Unfortunately, GUIDE seems much less adopted among practitioners and researchers when compared to other algorithms with easy-to-use open source implementations, like CART (e.g., **rpart**) and CTree (e.g., **partykit**).

Like CTree, GUIDE is based on statistical tests of hypothesis striving to achieve unbiased split variable selection. In particular, GUIDE was specifically designed to solve three problems that can adversely affect the interpretability of decision trees:

- 1) split variable selection bias;
- 2) insensitivity to local interactions;
- 3) overly complicated tree structures (like the Ames housing regression trees from [Figure 2.23](#) on page 97).

---

## 4.1 Introduction

GUIDE evolved from earlier tree growing procedures, starting with the *fast and accurate classification tree* (FACT) algorithm [Loh and Vanichsetakul, 1988]. FACT was novel at the time for its use of *linear discriminant analysis* (LDA) to find splits based on linear combinations of two predictors. FACT only applies to classification problems, and any node based on linear splits is partitioned into as many child nodes as there are class labels (i.e., FACT can use multiway splits). Split variable selection for continuous variables is based on comparing ANOVA-based  $F$ -statistics; LDA is applied to the variable with the largest  $F$ -statistic to find the optimal split point (i.e.,  $x \leq c$  vs.  $x > c$ , where  $c$  is in the domain of  $x$ ) to partition the data. Categorical predictors are converted to ordinal variables by using LDA to project their dummy encoded vectors onto the *largest discriminant coordinate* (also called the *canonical variate* in *canonical analysis*); the final splits are expressed back in the form  $x \in S$ , where  $S$  is a subset of the categories of  $x$ . Since FACT depends on ANOVA  $F$ -tests for split variable selection, it is only unbiased if all the predictors are ordered (i.e., it is biased towards nominal categorical variables) [Loh, 2014].

The *quick, unbiased, and efficient statistical tree* (QUEST) procedure [Loh and Shih, 1997] improves upon the bias in FACT by using chi-square tests for categorical variables (i.e., by forming contingency tables between each categorical variable and the response). Like CART, QUEST only permits binary splits. Let  $J_t$  be the number of response categories in any particular node  $t$ . Whenever  $J_t > 2$ , QUEST produces binary splits by merging the  $J_t$  classes into two *super classes* before applying the  $F$ - and chi-square tests to select a splitting variable. The optimal split point for ordered predictors is found using either an exhaustive search (like in [Chapter 2](#)) or *quadratic discriminant analysis* (QDA). For categorical splitting variables, the optimal split point is found in the same way after converting the dummy encoded vectors to the largest discriminant coordinate as in FACT.

Kim and Loh [2001] introduced the *classification rule with unbiased interaction selection and estimation* (CRUISE) algorithm, as a successor to QUEST. In contrast to QUEST, however, CRUISE allows multiway splits depending on the number of response categories in a particular node. Also, while QUEST uses  $F$ -tests for ordered variables and chi-square tests for nominal, CRUISE uses chi-square tests for both after discretizing the ordered variables. For split variable selection, CRUISE uses a two-step procedure involving testing both *main effects* and *two-way interactions* at each node.

CRUISE was later succeeded by GUIDE [Loh, 2009]<sup>a</sup>, which improves upon both QUEST and CRUISE by retaining their strengths and fixing their main weaknesses. One of the drawbacks of CRUISE is that the number of interaction tests greatly outnumbers the number of main effect tests; for example, with  $k$  features, there are  $k$  main effect tests and  $k(k - 1)$  pairwise interaction tests. Since most of the  $p$ -values come from the interaction tests, CRUISE is biased towards selecting split variables with potentially weak main effects, relative to the other predictors. GUIDE, on the other hand, restricts the number of interaction tests to only those predictors whose main effects are significant based on Bonferroni-adjusted  $p$ -values.

The next two sections cover many of the details associated with GUIDE for standard regression and classification, respectively; but note that GUIDE has been extended to handle several other situations as well (e.g., censored outcomes and longitudinal data). In general, GUIDE uses a two-step procedure when selecting the splitting variables. Consequently, GUIDE involves many more steps compared to CART (Chapter 2) and CTree (Chapter 3). The individual steps themselves are not complicated (most of them involve transformations of continuous features and what to do when interaction tests are significant), but for brevity, I'm only going to cover the nitty-gritty details, while pointing to useful references along the way.

Note that the official GUIDE software—which is freely available, but not open source—has evolved quite a lot over the years; the GUIDE program is discussed briefly in Section 4.9. Consequently, some of the fine details on the various GUIDE algorithms may have changed since their original publications. Any important updates are likely to be found in the revision history for the official GUIDE software:

<http://pages.stat.wisc.edu/~loh/treeprogs/guide/history.txt>.

If you're interested in going deeper on GUIDE for regression and classification, I encourage you to read Loh [2002] (the official reference to GUIDE for regression), Loh [2009] (the official reference to GUIDE for classification), Loh [2011] (an updated overview with some comparisons to other tree algorithms), and Loh [2012] (variable selection and importance). The current GUIDE software manual is also useful and can be obtained from the GUIDE website: <http://pages.stat.wisc.edu/~loh/guide.html>.

---

<sup>a</sup>GUIDE had already been introduced for regression problems in Loh [2002].

## 4.2 A GUIDE for regression

Whether building a classification tree or a regression tree, GUIDE uses chi-square tests throughout. This is convenient since chi-square tests are quick to compute, and can detect a large variety of patterns (e.g., curvature and interaction effects). Consequently, the response and any ordered features must be converted to nominal categorical before attempting to split a node<sup>b</sup>. Under the hood, GUIDE treats regression problems like a classification problem. In particular, at any node in the tree, GUIDE fits a model (e.g., a constant or a linear regression model) to the available data and computes the residuals. The sign of the residuals (i.e., positive/negative residuals are mapped to  $+1/-1$ ) is used as a binary response to partition the node. In contrast to CART, GUIDE can fit either constant or non-constant fits in the terminal nodes. Both strategies are discussed in the sections that follow.

### 4.2.1 Piecewise constant models

Using piecewise constant models is equivalent to a standard regression tree, where a constant (e.g., the average response value) is used to summarize each terminal node. Let  $N_t$  be the number of observations in an arbitrary node  $t$ , and let  $y_{t,1}, y_{t,2}, \dots, y_{t,N_t}$  be the available response values in  $t$ . The residuals for node  $t$  are nothing more than the difference between the observed response values and their sample mean:  $r_{t,j} = y_{t,j} - \sum_{i=1}^{N_t} y_{t,i}/N_t$ . The residuals can then be dichotomized at zero using their sign to create a new binary response variable in node  $t$ :

$$y'_{t,j} = \begin{cases} 1 & \text{if } r_{t,j} > 0 \\ -1 & \text{if } r_{t,j} < 0 \end{cases}.$$

Similarly, at any node  $t$ , ordered features (e.g., an ordered categorical or continuous feature) are also converted to non-ordered (i.e., nominal) categoricals by discretizing them into either three or four intervals, depending on the sample size in node  $t$  ( $N_t$ ).

Like CTree, GUIDE uses statistical tests (chi-square tests in particular) to select the splitting variable. However, unlike CTree, GUIDE employs a two-stage approach that tests for both main effects (called *curvature tests*) and two-way interaction effects between all pairs of features. The details are quite

<sup>b</sup>I mentioned the dangers of “dichotomania” in [Section 3.5.2](#), but keep in mind that discretizing the predictors is only used here for split variable selection, and full predictor information is used in selecting the split point and making predictions.

involved, but the basic steps (skipping the two-way interaction tests) are outlined in Algorithm 4.1 (151); GUIDE's interaction tests are discussed briefly in [Section 4.2.2](#).

---

**Algorithm 4.1** Simplified version of the original GUIDE algorithm for regression. Note that some of the details may have changed as the official software has continued to evolve over the years.

---

- 1) Start with  $t$  being the root node.
  - 2) Obtain the signed residuals from a constant fit to the data (e.g., the mean response).
  - 3) Convert ordered predictors (e.g., continuous features) to categorical by discretizing them into four intervals based on the sample quartiles (or *tertiles* if  $N_t < 60$ ).
  - 4) Using the  $N_t$  observations in  $t$ , perform a chi-square test of independence between each feature and the signed residuals; the dichotomized residuals form the two rows of the corresponding contingency table. (Call these the main effect, or curvature, tests.) Let  $x^*$  be the feature associated with the smallest Bonferroni-adjusted  $p$ -value from the curvature tests.
  - 5) Use an exhaustive search to find the best split on  $x^*$  yielding the greatest reduction in node SSE (see [Section 2.3](#)). By default, GUIDE uses univariate splits similar to CART and CTree. In particular, if  $x^*$  is unordered, splits are of the form  $x^* \in S$ , where  $S$  is a subset of the categories of  $x^*$ . If  $x^*$  is ordered, splits have the form  $x^* \leq c$ , where  $c$  is a midpoint in the observed range of  $x^*$ ; for speed, GUIDE will optionally use the within node sample median of  $x^*$  for the cutoff  $c$ .
  - 6) Recursively apply steps 2)–5) on all the resulting child nodes until all nodes are pure or suitable stopping criteria are met (e.g., the maximum number of allowable splits is reached).
  - 7) Similar to CART, prune the resulting tree using cost-complexity pruning (see [Section 2.5](#) for details).
- 

A few comments regarding step 4) of Algorithm 4.1 are in order. First, any rows or columns with zero margin totals are removed. Second, to avoid difficulties in computing very small  $p$ -values and to account for the fact that the degrees of freedom are not fixed across the chi-square tests, GUIDE sometimes uses a modification of the *Wilson-Hilferty transformation* [Wilson and Hilferty, 1931] to ensure all the test statistics approximately correspond to a

chi-square distribution with a single degree of freedom.<sup>c</sup> In particular, let  $x$  be an observed value from a chi-square distribution with  $\nu$  degrees of freedom ( $\chi_\nu^2$ ). If we define

$$w_1 = \left( \sqrt{2x} - \sqrt{2\nu - 1} + 1 \right)^2 / 2,$$

$$w_2 = \max \left\{ 0, \left( 7/9 + \sqrt{\nu} \left[ \sqrt[3]{x/\nu} = 1 + 2/9\nu \right] \right)^3 \right\},$$

$$w = \begin{cases} w_2 & \text{if } x < \nu + 10\sqrt{2\nu} \\ (w_1 + w_2) / 2 & \text{if } x \geq \nu + 10\sqrt{2\nu} \text{ and } w_2 < x \\ w_1 & \text{otherwise} \end{cases}$$

then it follows that  $Pr(\chi_\nu^2 > x) \approx Pr(\chi_1^2 > w)$ ; this transformation is implemented in the `wilson_hilferty()` function in package `treemisc` (for details, see `?treemisc::wilson_hilferty`). Finally, for brevity, the tests for two-way interactions that GUIDE carries out by default are omitted; see [Section 4.2.2](#) for details.

Although GUIDE uses chi-square tests throughout (which requires discretizing ordered features into 3–4 groups), Loh [2002] provided a simulation study which gave empirical evidence that GUIDE's split variable selection procedure is indeed unbiased, relative to exhaustive search procedures like CART.

#### 4.2.1.1 Example: New York air quality measurements

To illustrate, let's return to the New York air quality example introduced in [Section 1.4.2](#). Below is a simple function, called `guide.chisq.test()`, for carrying out steps 2)–4) of Algorithm 4.1. For brevity, and since the degrees of freedom are the same for each test, it omits the modified Wilson-Hilferty transformation discussed previously:

```
guide.chisq.test <- function(x, y) {
  y <- as.factor(sign(y - mean(y))) # discretize response
  if (is.numeric(x)) { # discretize numeric features
    bins <- quantile(x, probs = c(0.25, 0.5, 0.75), na.rm = TRUE)
    bins <- c(-Inf, bins, Inf)
    x <- as.factor(findInterval(x, vec = bins)) # quartiles
  }
  tab <- table(y, x) # form contingency table
  if (any(row.sums <- rowSums(tab) == 0)) { # check rows
    tab <- tab[-which(row.sums == 0), ] # omit zero margin totals
  }
}
```

---

<sup>c</sup>CTree ([Chapter 3](#)) avoids the small  $p$ -value problem internally by working with  $p$ -values on the log scale.

```

if (any(col.sums <- colSums(tab) == 0)) { # check columns
  tab <- tab[, -which(col.sums == 0)] # omit zero margin totals
}
chisq.test(tab)$p.value # p-value from chi-squared test
}

```

Next, I omit any rows with missing response values and compute the Bonferroni-adjusted  $p$ -values from step 2) of Algorithm 4.1 for each feature:

```

aq <- airquality[!is.na(airquality$Ozone), ]
pvals <- sapply(setdiff(names(aq), "Ozone"), FUN = function(x) {
  guide.chisq.test(aq[[x]], y = aq[["Ozone"]])
})
p.adjust(pvals, method = "bonferroni") # Bonferroni adjusted p-values

#> Solar.R      Wind      Temp     Month      Day
#> 2.23e-03 1.40e-06 2.50e-14 2.83e-06 5.88e-01

```

As we previously found with CART and CTree, `Temp` is selected to split the root node (as it has the smallest adjusted  $p$ -value). We previously found  $\text{Temp} = 82.5$  to be the optimal split point using an exhaustive search in [Section 2.3.1](#).

#### 4.2.2 Interaction tests

Exhaustive search procedures, like CART, can be insensitive to local interactions; according to Loh [2002], splits that are sensitive to two-way interaction effects can produce shorter trees. GUIDE circumvents this issue by explicitly testing for two-way interactions between the response and each pair of features. The basic idea is to partition the feature space between a pair of predictors to form the columns of a new table, then apply the same chi-square test outlined in step 4) of Algorithm 4.1. If there are  $k$  predictors in total, then  $k(k + 1)/2$  chi-square tests are employed each time a split variable is selected to partition the data (when including two-way interaction effects, that is).

To illustrate, suppose we want to test for an interaction between  $(x_i, x_j)$  and  $y$ . If both  $x_i$  and  $x_j$  are ordered, we divide the  $(x_i, x_j)$  into four quadrants by splitting the range of each feature into two halves using the sample median. From this, a  $2 \times 4$  contingency table can be formed, where the rows still represent the discretized residuals, and a chi-square test can be applied. If  $x_i$  and  $x_j$  are both nominal categorical variables, with  $c_i$  and  $c_j$  unique categories, respectively, then we form the  $2 \times c_i c_j$  contingency table, where the columns are based on all the possible pairs of categories between  $x_i$  and  $x_j$ . Finally, if  $x_i$  is ordered and  $x_j$  is nominal (with  $c_j$  unique categories), then we split the range of  $x_i$  into two halves using the sample median and form a  $2 \times 2c_j$  contingency

table, where the columns correspond to all possible pairs of values between the binned  $x_i$  values and  $x_j$ . In any of the above cases, rows or columns with zeros in the margin are omitted before applying the chi-square test.

When including interaction tests, we need to modify how the split variable  $x^*$  is selected in step 4) of Algorithm 4.1. If the smallest  $p$ -value is from a curvature test, then select the associated predictor to split the node. If the smallest  $p$ -value comes from an interaction test, then the choice of splitting variable depends upon whether both features are ordered or not. If  $x_i$  and  $x_j$  are both ordered, the node is split using the sample mean for each variable (e.g.,  $x_i \leq \sum_{i=1}^{N_t} x_i / N_t$ ). For each of the two splits, a constant (e.g., the mean response) is fit to the resulting nodes. The split yielding the greatest reduction in SSE (p. 58) is selected to split the node. On the other hand, if either  $x_i$  or  $x_j$  is nominal, select the variable with the smallest  $p$ -value from the associated curvature tests. For details, see Algorithm 2 in Loh [2002].

Using a split variable selected from an interaction test does not guarantee that the interacting variable will be used to split one of the child nodes. While it may be intuitive to force this behavior to highlight the specific interaction in the tree, Loh [2002] argues that letting variables compete at each individual split can lead to shorter trees.

#### 4.2.3 Non-constant fits

In contrast to CART and CTree, GUIDE is not restricted to fitting a constant in each node. This generality is due to the fact that the residuals are used to select the splitting variable. Hence, any model that produces residuals can be used to construct the tree. The GUIDE software ([Section 4.9](#)) allows a wide range of regression models to be used in each node: simple linear regression, Poisson regression, regression for censored outcomes, and more. The benefit to fitting non-constant models in each node is the potential reduction in tree size and increase to predictive accuracy. Of course, the same idea can be applied to exhaustive search procedures like CART, but this can be too computationally expensive. By abandoning a fully-exhaustive search criteria, GUIDE can afford to fit a richer class of models in the nodes, while substantially reducing split variable selection bias—a win-win, so to speak.

Fitting non-constant models in the nodes of the tree means that the predictors can potentially serve more than one role during tree construction. In particular, predictors can compete for splits and/or serve as a regressor in terminal node fits. For simplicity, GUIDE only allows ordered features to serve as regressors, unless the categorical variables are dummy encoded and treated as numeric [Loh, 2002, p. 371].

Borrowing the same terminology in Loh [2002], there are four basic roles a predictor can serve:

- $n$ -variable: a numeric feature used to fit regression models and to split nodes;
- $f$ -variable: a numeric feature used to fit regression models but not split nodes;
- $s$ -variable: a numeric feature used to split nodes but not fit regression models;
- $c$ -variable: a categorical feature used to split nodes but not fit regression models.

Therefore, numeric features can fill one of three roles (e.g., an  $n$ -,  $f$ -, or  $s$ -variable), while categorical features can only be used to split nodes. This gives a great deal of flexibility when fitting regression models in the nodes. For example, we can fit a quadratic model in  $x_j$  in each of the nodes by specifying  $x_j^2$  as an  $f$ -variable, so it's not used to split any nodes.

When employing non-constant fits, Algorithm 4.1 requires a few simple modifications, but I'll defer to Loh [2002, Algorithms 3–4] for details.

#### 4.2.3.1 Example: predicting home prices

To illustrate, let's return to the Ames housing data (Section 1.4.7). Recall that I initially split the data into train/test sets using a 70/30 split; since I'm not plotting anything, I did not bother to rescale the response in this example. Using the GUIDE software (Section 4.9), I built a default regression tree with *stepwise linear regression* models in each node.<sup>d</sup> All variables were allowed to compete for splits, and all numeric features were allowed to compete as predictors in the stepwise procedure applied to each node. The tree was pruned using 10-fold cross-validation along with the 1-SE rule. The resulting tree diagram is displayed in Figure 4.1—the inner caption is part of the output from GUIDE and explains the tree diagram.

The 1-SE pruned GUIDE-based tree for the Ames housing data, using non-constant fits, is substantially smaller than the 1-SE pruned CART tree from Figure 2.23 (p. 97); it is also far more accurate, with a test set RMSE of \\$28,870.78.<sup>e</sup> For further comparison, CTree, using  $\alpha = 0.05$ , resulted in a tree with 75 terminal nodes and a test RMSE of \\$35,331.88.

GUIDE will also output a text file containing the variable importance scores (Section 4.7), estimated regression equation for each terminal node, and more.

---

<sup>d</sup>Since linear models are being used to summarize the terminal nodes, it would be wise to consider log-transforming the response first, or use a similar transformation, since it is quite right skewed, but for comparison to tree fits from previous chapters, I elected not to in this example.

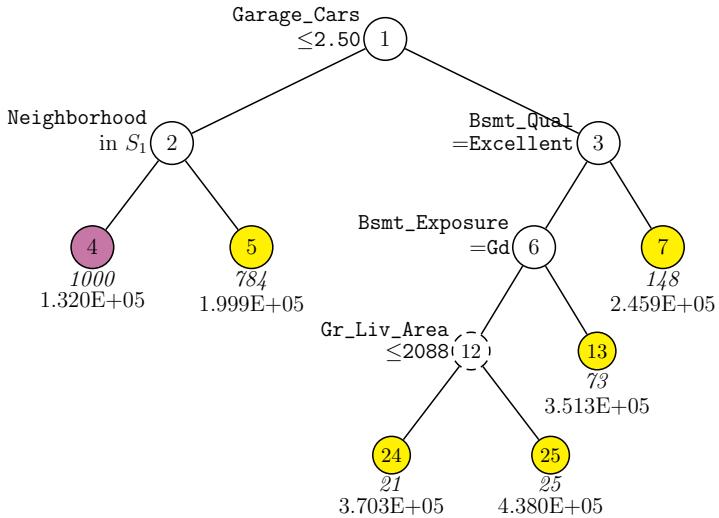
<sup>e</sup>While smaller in size, one could argue that the pruned GUIDE tree is no less interpretable, since the terminal nodes are summarized using regression fits in different subsets of the predictors.

For example, the sale price of any home with a garage capacity for three or more cars, excellent basement quality, good basement exposure, and an above ground living area of less than 2,088 sq. ft. would be estimated according to the following equation:

$$\widehat{\text{Sale\_Price}} = -149,100.00 + 283.30 \text{First\_Flr\_SF}, \quad (4.1)$$

where `First_Flr_SF` is the square footage of the first floor. This corresponds to terminal node 24 in [Figure 4.1](#).

The output file from GUIDE also reported that the tree in [Figure 4.1](#) explains roughly 87.89% of the variance in `Sale_Price` on the training data (i.e.,  $R^2 = 0.8789$ ).



GUIDE v.38.0 0.25-SE piecewise linear least-squares regression tree with stepwise variable selection for predicting `Sale_Price`. Tree constructed with 2051 observations. Maximum number of split levels is 12 and minimum node sample size is 20. At each split, an observation goes to the left branch if and only if the condition is satisfied. Set  $S_1 = \{\text{Briardale, Brookside, Edwards, Iowa_DOT\_and\_Rail\_Road, Landmark, Meadow\_Village, Mitchell, North\_Ames, Northpark\_Villa, Old\_Town, Sawyer, South\_and\_West\_of\_Iowa\_State\_University}\}$ . Circles with dashed lines are nodes with no significant split variables. Sample size (in *italics*) and mean of `Sale_Price` printed below nodes. Terminal nodes with means above and below value of  $1.802\text{E}+05$  at root node are colored yellow and purple respectively. Second best split variable at root node is `Neighborhood`.

**FIGURE 4.1:** Example tree diagram produced by GUIDE for the Ames housing example. Stepwise linear regression models were fit in each node. The autogenerated caption produced by the GUIDE software is also included.

#### 4.2.3.2 Bootstrap bias correction

It is difficult to achieve unbiased split variable selection in regression trees that employ regression models in the nodes because the predictors can be used for splitting, referred to as a “split” variable, or as a regressor in the models, where it’s referred to as a “fit” variable [Loh, 2014].

When non-constant models are fit to the nodes, the split variable selection procedure in GUIDE is heavily biased towards the  $c$ - and  $s$ -variables. This is because the  $n$ -variables, which are used for both splitting and as regressors, are uncorrelated with the resulting residuals (a property of *ordinary least squares*).

Loh [2002, Algorithm 5] proposed a bootstrap calibration procedure to help correct the split variable selection bias in this situation. I’ll omit the details, but the basic idea is to shrink the  $p$ -values associated with the chi-square tests for the  $n$ -variables.

---

### 4.3 A GUIDE for classification

GUIDE for classification is not that different from its regression counterpart. Instead of residuals, the categorical outcome is used directly in the chi-square tests for split variable selection. Also, once a splitting variable, say  $x^*$ , is selected, the optimal split point is found using an exhaustive search, similar to CART’s approach based on a weighted sum of Gini impurities (see [Section 2.2.1](#)).

#### 4.3.1 Linear/oblique splits

Although orthogonal (or binary) splits are more interpretable, Loh [2009] makes a compelling case for splits based on linear combinations of predictors (which are referred to as either *linear splits* or *oblique splits*, since they are no longer orthogonal to the feature axes). An oblique split on two continuous features,  $x_i$  and  $x_j$ , takes the form  $ax_i + bx_j \leq c$ , where  $a$ ,  $b$ , and  $c$  are constants determined from the data; see Loh [2009, Sec. 3] for details.

Using orthogonal splits can result in smaller trees and greater predictive accuracy. GUIDE only allows linear splits for classification problems<sup>f</sup> and is restricted to two variables,  $x_i$  and  $x_j$  (say), only when an interaction test between  $x_i$  and  $x_j$  is not significant using another Bonferroni correction. The form of the linear split is chosen using LDA; see Loh [2009, Procedure 3.1] for details. In the official GUIDE software, oblique splits can be given higher or lower priority than orthogonal splits (see [Section 4.9](#)). Loh [2009] also mentions that while oblique splits are more powerful than orthogonal splits, it is not necessary to apply them to split each node, which he illustrates with an example on classifying fish species.

Even when linear splits are allowed, Loh [2009] showed that the GUIDE procedure for classification is still practically unbiased in terms of split variable selection.

#### 4.3.1.1 Example: classifying the Palmer penguins

To appreciate the benefits of using linear splits, let's look at an example with the Palmer penguins data. The data, which are available in the R package **palmerpenguins** [Horst et al., 2020], contain the size measurements (flipper length, body mass, and bill dimensions) for three species of adult foraging penguins near Palmer Station, Antarctica. For this example, I'll try to classify the three species using just two measurements: bill length in mm (`bill_length_mm`) and bill depth in mm (`bill_depth_mm`); see [Figure 4.2](#).

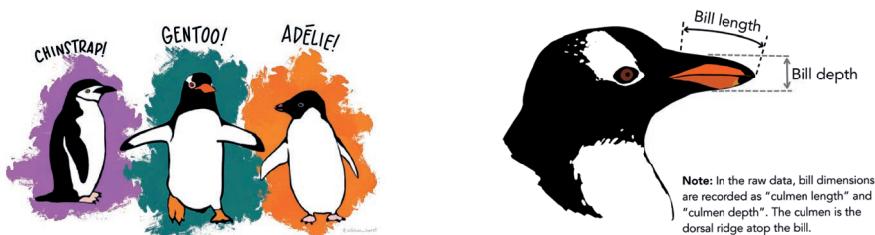


FIGURE 4.2: Artwork by Allison Horst. Source: <https://github.com/allisonhorst/palmerpenguins/>.

[Figure 4.3](#) shows a scatterplot of the bill length vs. bill depth for the three species of penguins. While there does seem to be a good deal of separation between the three species using `bill_depth_mm` and `bill_length_mm`, it will

<sup>f</sup>Breiman et al. [1984, p. 248] argue that splits on linear combinations of predictors are less effective in regression problems compared to classification, apparently because linear combination splits tend to produce rectangular-like regions when partitioning the feature space in regression, similar to the more common, but easier to obtain, orthogonal splits.

be challenging for a classification tree that uses splits that are orthogonal to the  $x$ - and  $y$ -axes (e.g., CART and CTree). If the data come from a multivariate normal distribution with a common covariance matrix across the three species, then LDA would give the optimal linear decision boundary (if the covariance matrices differ between the classes, then QDA would be optimal). If we cannot make those assumptions, then a tree-based approach using oblique splits is a good alternative.

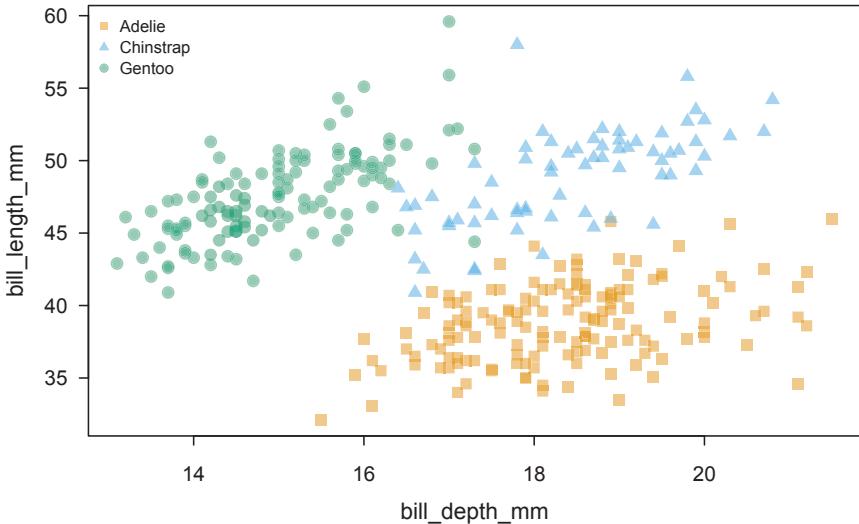


FIGURE 4.3: Scatterplot of bill depth (mm) vs. bill length (mm) for the three species of Palmer penguins.

To illustrate, consider the plots in [Figure 4.4](#), which show the decision boundaries from a GUIDE decision tree with linear splits (top left), LDA (top right), CART (middle left), CTree (middle right), a *random forest* (bottom left), *gradient boosted tree ensemble* (bottom right); the latter two are special types of tree-based ensembles and are discussed in [Chapters 7–8](#). Both GUIDE and CART were pruned using 10-fold cross-validation with the 1-SE rule (for CTree, I used the default  $\alpha = 0.05$ ). Notice the similarity (and simplicity) of the linear decision boundaries produced by GUIDE and LDA; these models are likely to generalize better to new data from the same population. Furthermore, GUIDE only misclassified 8 observations, while LDA, CART, and CTree misclassified 13, 21, and 15 observations, respectively. Compared to CART and CTree, the tree-based ensembles (bottom row) are a bit more flexible and able to adapt to linear decision boundaries, but in this case, they're not as smooth or simple to explain as the LDA or GUIDE decision boundaries.

The associated tree diagram for the fitted GUIDE tree with linear splits is shown in [Figure 4.5](#). The GUIDE tree using linear splits is simpler compared

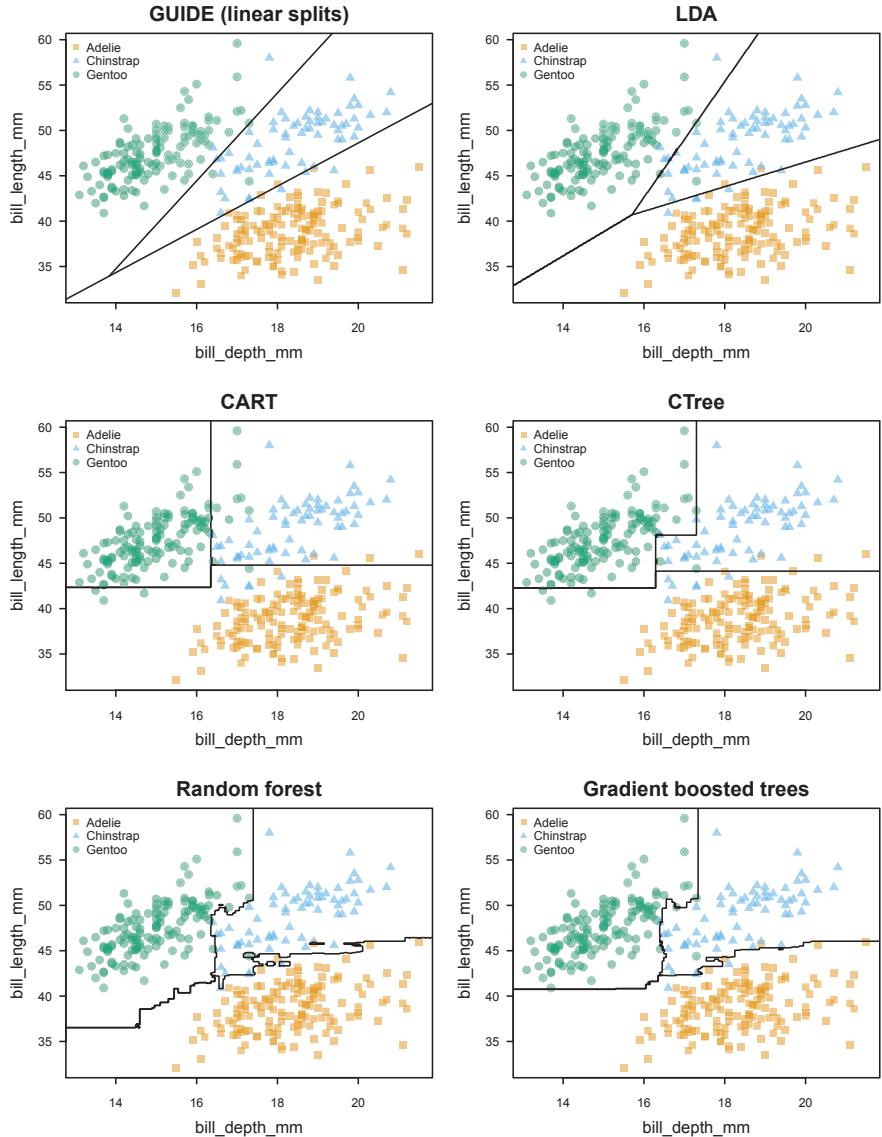
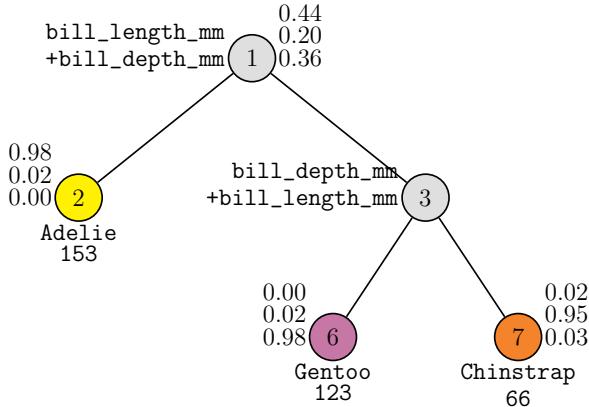


FIGURE 4.4: Decision boundaries from several classification models applied to the Palmer penguins data. Top left: a single GUIDE decision tree with linear splits. Top right: linear discriminant analysis. Middle left: a single CART tree. Middle right: a single CTree. Bottom left: a random forest of 1000 CART-like trees. Bottom right: gradient boosted CART-like trees.

to the associated CART and CTree trees (not shown); the former uses two splits, while the latter two require three and seven splits, respectively.



GUIDE v.38.0 1.00-SE classification tree for predicting `species` using linear split priority, estimated priors and unit misclassification costs. Tree constructed with 342 observations. Maximum number of split levels is 10 and minimum node sample size is 3. At each split, an observation goes to the left branch if and only if the condition is satisfied. Intermediate nodes in lightgray indicate linear splits. Predicted classes and sample sizes printed below terminal nodes; class sample proportions for `species` = `Adelie`, `Chinstrap`, and `Gentoo`, respectively, beside nodes.

FIGURE 4.5: Example tree diagram (with description) produced by GUIDE for the Palmer penguins data using linear splits. The autogenerated caption produced by the GUIDE software is also included.

### 4.3.2 Priors and misclassification costs

Like CART, GUIDE can incorporate different priors and unequal misclassification costs. See [Section 2.2.4](#) and Loh [2009] for details.

### 4.3.3 Non-constant fits

Similar to GUIDE for regression, we can increase flexibility by fitting non-constant models in the nodes of the classification tree. For increased accuracy, or a greater reduction in the size of the tree, Loh [2009] suggests fitting either a *kernel density estimate* or a *k*-nearest neighbor model to each node. Owing to the flexibility of these models, however, GUIDE dispenses with linear splits when allowing non-constant fits. When there are weak main effects, but strong two-way interaction effects, classification trees constructed with non-constant fits can achieve substantial gains in accuracy and tree compactness. Although

they require more computation than classification trees with constant fits, Loh [2009] empirically shows that their prediction accuracy is often relatively high.

#### 4.3.3.1 Kernel-based and $k$ -nearest neighbor fits

Kernel-based fits essentially select the class with the highest kernel density estimate in each node. For example, if the selected split variable  $x$  is ordered, then for each class in node  $t$ , a kernel density estimate  $\hat{f}(x)$  is computed according to

$$\hat{f}(x) = \frac{1}{N_t h} \sum_{i=1}^{N_t} \phi\left(\frac{x - x_i}{h}\right),$$

where  $\phi(\cdot)$  denotes the density function of a standard normal distribution, and

$$h = \begin{cases} 2.5 \min(s, 0.7413r) N_t^{-1/5} & \text{if } r > 0 \\ 2.5s N_t^{-1/5} & \text{otherwise} \end{cases},$$

is the *bandwidth*; here,  $s$  and  $r$  denote the sample standard deviation and *interquartile range* (IQR) of the observed  $x$  values. Some motivation for using this bandwidth, which is more than twice as large as the usual bandwidth recommended for density estimation, is given in Ghosh et al. [2006].

A similar idea is used when applying  $k$ -nearest neighbor fits to each node. In particular,  $k$  is given by

$$k = \max(3, \lceil \log(N_t) \rceil),$$

where  $\lceil x \rceil$  just means round  $x$  up to the nearest integer. The minimum number of neighbors ( $k$ ) is three to avoid too much trouble in dealing with ties.

## 4.4 Pruning

The GUIDE algorithm continues to recursively perform splits until some stopping criteria are met (e.g., the minimum number of observations required for splitting has been reached in each node). Like CART, this will often lead to overfitting and an overly complex tree with too many splits. To circumvent

the issue, GUIDE adopts the same cost-complexity pruning strategy used in CART (revisit Section 2.5 for the details). Loh [2002] showed via simulation that pruning can reduce the variable selection bias in exhaustive search procedures like CART, provided there are no interaction effects.

---

## 4.5 Missing values

GUIDE solves the problem of missing values by assigning missing categorical values to a new “missing” category. The same happens to ordered variables during the construction of the chi-square tests; their missing values are then mapped back to  $-\infty$  for split point selection. Hence, the split  $x^* \leq c$  will always send missing values to the left child node. If informative enough, missing values can be isolated completely by choosing  $c = -\infty$ . See Loh et al. [2020] for more details and comparisons with other approaches to handling missing values.

---

## 4.6 Fitted values and predictions

For constant fits, the fitted values and predictions are obtained in the same way as CART and CTree. For example, if  $y$  is continuous, then the average within-node response is used. For classification, the within node class proportions can serve as predicted class probabilities. For non-constant fits, the fitted values and predictions for new observations depend on the model fit to each node (e.g., polynomial regression or  $k$ -nearest neighbor). The latter tend to produce shorter and more accurate trees, but are more computationally intensive and less interpretable.

---

## 4.7 Variable importance

Variable importance scores in GUIDE are based on the sum of the weighted one- $df$  chi-square test statistics across the nodes of the tree, where the weights are given by the square root of the sample size of the corresponding nodes. In

particular, let  $q_t(x)$  be the one- $df$  chi-square statistic associated with predictor  $x$  at node  $t$ . The variable importance score for  $x$  is

$$\text{VI}(x) = \sum_t \sqrt{N_t} q_t(x),$$

where  $N_t$  is the sample size in node  $t$ ; see Loh [2012] for details. Later, Loh et al. [2015] suggest using  $N_t$  rather than its square root to weight the chi-square statistics, which increases the probability that the feature used to split the root node has the largest importance score. This approach is approximately unbiased unless there is a mix of both ordered and nominal categorical variables. Loh and Zhou [2021] provide an improved version for regression that ensures unbiasedness.

While there are a number of approaches to computing variable importance (especially from decision trees), few include thresholds for identifying the irrelevant (or pure noise) features. In GUIDE, any feature  $x$  with a variable importance score less than the 0.95-quantile of the approximate null distribution of  $\text{VI}(x)$  is considered unimportant. For example, running GUIDE in variable importance mode (with default settings, which caps the total number of splits at four) on the Ames housing data flagged 64 of the 80 features as being highly important, 4 as being less important, and 12 as being unimportant. The results are displayed in [Figure 4.6](#) (p. 174).

As previously mentioned, the GUIDE software continues to evolve and the details mentioned above may not correspond exactly with what's currently implemented. Fortunately, Loh and Zhou [2021] give a relatively recent account of GUIDE's approach to variable importance, and provide a thorough comparison against other tree-based approaches to computing variable importance, including CART, CTree, and many of the tree-based ensembles discussed later in this book.

## 4.8 Ensembles

Although I'll defer the discussion of ensembles until [Chapter 5](#), it's worth noting that GUIDE supports two types of GUIDE-based tree ensembles: *bagging* ([Section 5.1](#)) and random forest ([Chapter 7](#)).

## 4.9 Software and examples

GUIDE, along with its predecessors, is not open source and exists as a command line program. Compiled binaries for QUEST, CRUISE, and GUIDE are freely available from <http://pages.stat.wisc.edu/~loh/guide.html> and are compatible with most major operating systems. If you're comfortable with the terminal, GUIDE is straightforward to install and use; see the available manual for installation specifics and example usage. Although it's a terminal application, GUIDE will optionally generate R code that can be used for scoring after a tree has been fit. This makes it easy to run simulations and the like after the tree has been built. My only criticism of GUIDE is that it's not currently available via easy-to-use open source code (like R or Python); if it were, it'd probably be much more widely adopted by practitioners.

As discussed in the previous section, GUIDE is a command line program that requires input from the user. For this reason, I'll limit this section to a single example; the GUIDE software manual [Loh, 2020] offers plenty of additional examples. Note that GUIDE can optionally generate R code to reproduce predictions from the fitted tree model, which can be useful for simulations and deployment. Further, I used GUIDE v38.0 for all the examples in this chapter.

### 4.9.1 Example: credit card default

The credit card default data [Yeh and hui Lien, 2009], available from the UCI Machine Learning Repository at

<https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients>,

contains demographic and payment information about credit card customers in Taiwan in the year 2005. The data set contains 30,000 observations on the following 23 variables:

- **default**: A binary indicator of whether or not the customer defaulted on their payment (**yes** or **no**).
- **limit\_bal**: Amount of credit (NT dollar) given to the customer.
- **sex**: The customer's gender (**male** or **female**).
- **education**: The customer's level of education (**graduate school**, **university**, **high school**, or **other**).
- **marriage**: The customer's marital status (**married**, **single**, or **other**).

- **age:** The customer's age in years.
- **pay\_0, pay\_2-pay\_6:** History of past payment; **pay\_0** = the repayment status in September, 2005; **pay\_2** = the repayment status in August, 2005; ...; **pay\_6** = the repayment status in April, 2005. The measurement scale for the repayment status is: -1 = pay duly; 1 = payment delay for one month; 2 = payment delay for two months; ...; 8 = payment delay for eight months; 9 = payment delay for nine months and above.
- **bill\_amt1-bill\_amt6:** Amount of bill statement (NT dollar). **bill\_amt1** = amount of bill statement in September, 2005; **bill\_amt2** = amount of bill statement in August, 2005; ...; **bill\_amt6** = amount of bill statement in April, 2005.
- **pay\_amt1-pay\_amt6:** Amount of previous payment (NT dollar). **pay\_amt1** = amount paid in September, 2005; **pay\_amt1** = amount paid in August, 2005; ...; **pay\_amt6** = amount paid in April, 2005.

The goal is to build a model to predict the probability of a customer defaulting on their credit card payment. For that I'll build a GUIDE classification tree. The R code I used to download and clean the data is shown below. To start, I download the data into a temporary file before reading in the resulting XLS file (i.e., Microsoft Excel spreadsheet) using the **readxl** package [Wickham and Bryan, 2019] and printing a compact summary of the data using **str()**:

```
# Download and read in the credit default data from the UCI ML repo
tf <- tempfile(fileext = ".xls")
url <- paste0("https://archive.ics.uci.edu/ml/", # sigh, long URLs...
              "machine-learning-databases/",
              "00350/default%20of%20credit%20card%20clients.xls")
download.file(url, destfile = tf)
credit <- as.data.frame(readxl::read_xls(tf, skip = 1))

# Clean up column names a bit
names(credit) <- tolower(names(credit))
names(credit)[names(credit) == "default payment next month"] <-
  "default"

str(credit) # compactly display structure of the data frame

#> 'data.frame': 30000 obs. of  25 variables:
#> $ id      : num  1 2 3 4 5 6 7 8 9 10 ...
#> $ limit_bal: num  20000 120000 90000 50000 50000 500...
#> $ sex     : num  2 2 2 2 1 1 1 2 2 1 ...
#> $ education: num  2 2 2 2 2 1 1 2 3 3 ...
#> $ marriage : num  1 2 2 1 1 2 2 2 1 2 ...
#> $ age      : num  24 26 34 37 57 37 29 23 28 35 ...
#> $ pay_0    : num  2 -1 0 0 -1 0 0 0 0 -2 ...
#> $ pay_2    : num  2 2 0 0 0 0 0 -1 0 -2 ...
#> $ pay_3    : num  -1 0 0 0 -1 0 0 -1 2 -2 ...
```

```
#> $ pay_4      : num  -1 0 0 0 0 0 0 0 0 0 -2 ...
#> $ pay_5      : num  -2 0 0 0 0 0 0 0 0 -1 ...
#> $ pay_6      : num  -2 2 0 0 0 0 0 -1 0 -1 ...
#> $ bill_amt1: num  3913 2682 29239 46990 8617 ...
#> $ bill_amt2: num  3102 1725 14027 48233 5670 ...
#> $ bill_amt3: num  689 2682 13559 49291 35835 ...
#> $ bill_amt4: num  0 3272 14331 28314 20940 ...
#> $ bill_amt5: num  0 3455 14948 28959 19146 ...
#> $ bill_amt6: num  0 3261 15549 29547 19131 ...
#> $ pay_amt1  : num  0 0 1518 2000 2000 ...
#> $ pay_amt2  : num  689 1000 1500 2019 36681 ...
#> $ pay_amt3  : num  0 1000 1000 1200 10000 657 38000 0...
#> $ pay_amt4  : num  0 1000 1000 1100 9000 ...
#> $ pay_amt5  : num  0 0 1000 1069 689 ...
#> $ pay_amt6  : num  0 2000 5000 1000 679 ...
#> $ default   : num  1 1 0 0 0 0 0 0 0 0 ...
```

Note that the categorical variables have been numerically re-encoded. The next code chunk removes the column ID and cleans up some of the categorical features by re-encoding them from numeric back to the actual categories based on the provided column descriptions:

```
# Remove ID column
credit$id <- NULL

# Clean up categorical features
credit$sex <- ifelse(credit$sex == 1, yes = "male", no = "female")
credit$education <- ifelse(
  test = credit$education == 1,
  yes = "graduate school",
  no = ifelse(
    test = credit$education == 2,
    yes = "university",
    no = ifelse(
      test = credit$education == 3,
      yes = "high school",
      no = "other"
    )
  )
)
credit$marriage <- ifelse(
  test = credit$marriage == 1,
  yes = "married",
  no = ifelse (
    test = credit$marriage == 2,
    yes = "single",
    no = "other"
  )
)
```

```

credit$default <- ifelse(credit$default == 1, yes = "yes", no = "no")

# Coerce character columns to factors
for (i in seq_len(ncol(credit))) {
  if (is.character(credit[[i]])) {
    credit[[i]] <- as.factor(credit[[i]])
  }
}

```

Finally, I'll split the data into train/test sets using a 70/30 split, leaving 21,000 observations for training and 9,000 for estimating the generalization performance:

```

set.seed(1342) # for reproducibility
trn.ids <- sample(nrow(credit), size = 0.7 * nrow(credit),
                  replace = FALSE)
credit.trn <- credit[trn.ids, ]
credit.tst <- credit[-trn.ids, ]

```

The GUIDE program requires two special text files before it can be called:

- the data input file;
- a description file.

See the GUIDE reference manual [Loh, 2020] for full details. The data input file is essentially just a text file containing the training data in a format that can be consumed by GUIDE. The description file provides some basic metadata, like the missing value flag and variable roles. These files can be a pain to generate, especially for data sets with lots of columns, so I included a little helper function in `treemisc` to help generate them; see `?treemisc::guide_setup` for argument details.

Below is the code I used to generate the data input and description files for the credit card default example. By default, numeric columns are used both for splitting the nodes and for fitting the node regression models for non-constant fits, and categorical variables are used for splitting only. In my setup, I have a `/guide-v38.0/credit` directory containing the GUIDE executable and where the generated files will be written to:

```

treemisc::guide_setup(credit.trn, path = "guide-v38.0/credit",
                      dv = "default", file.name = "credit",
                      verbose = TRUE)

#> Writing data file to guide-v38.0/credit/credit.txt...
#> Writing description file to guide-v38.0/credit/credit_desc.txt...

```

This resulted in the creation of two files in `/guide-v38.0/credit` (my directory for this example):

- `credit.txt` (the training data input file in the format required by GUIDE);
- `credit_desc.txt` (the description file).

Below are the contents of the generated `credit_desc.txt` file. The first line gives the name of the training data file; if the file is not in the current working directory, its full path must be given with quotes (e.g., "`some/path/to/credit.txt`"). The second line specifies the missing value code (if it contains non-alphanumeric characters, then it too must be quoted). The remaining lines specify the column number, name, and role for each variable in the data input file. As you can imagine, creating this file for a data set with lots of variables can be tedious, hence the reason for writing a helper function.

```
credit.txt
NA
2
1 limit_bal n
2 sex c
3 education c
4 marriage c
5 age n
6 pay_0 n
7 pay_2 n
8 pay_3 n
9 pay_4 n
10 pay_5 n
11 pay_6 n
12 bill_amt1 n
13 bill_amt2 n
14 bill_amt3 n
15 bill_amt4 n
16 bill_amt5 n
17 bill_amt6 n
18 pay_amt1 n
19 pay_amt2 n
20 pay_amt3 n
21 pay_amt4 n
22 pay_amt5 n
23 pay_amt6 n
24 default d
```

With the `credit.txt` and `credit_desc.txt` files in hand (and in the appropriate directories required by GUIDE), we can spin up a terminal and call the GUIDE program. I'll omit the details since it's OS-specific, but the GUIDE reference manual will take you through each step. Once the program is called, GUIDE will ask the user for several inputs (e.g., whether to build a classification or regression tree, whether to use constant or non-constant fits, number of folds

to use for cross-validation, etc.). In the end, GUIDE generates a special input text file to be consumed by the software.

Below are the contents of the input file for the credit card default example, called `credit_in.txt`, highlighting all the options I selected (I basically requested a default classification tree that's been pruned using the 1-SE rule with 10-fold cross-validation, but you can see several of the available options in the output).

```

GUIDE      (do not edit this file unless you know what you are doing)
38.0      (version of GUIDE that generated this file)
1         (1=model fitting, 2=importance or DIF scoring, 3=data con...
"credit_out.txt"  (name of output file)
1         (1=one tree, 2=ensemble)
1         (1=classification, 2=regression, 3=propensity score group...
1         (1=simple model, 2=nearest-neighbor, 3=kernel)
2         (0=linear 1st, 1=univariate 1st, 2=skip linear, 3=skip li...
1         (0=tree with fixed no. of nodes, 1=prune by CV, 2=by test...
"credit_desc.txt"  (name of data description file)
10        (number of cross-validations)
1         (1=mean-based CV tree, 2=median-based CV tree)
1.000    (SE number for pruning)
1         (1=estimated priors, 2=equal priors, 3=other priors)
1         (1=unit misclassification costs, 2=other)
2         (1=split point from quantiles, 2=use exhaustive search)
1         (1=default max. number of split levels, 2=specify no. in ...
1         (1=default min. node size, 2=specify min. value in next l...
2         (0=no LaTeX code, 1=tree without node numbers, 2=tree wit...
"credit.tex" (latex file name)
1         (1=color terminal nodes, 2=no colors)
2         (0=#errors, 1=sample sizes, 2=sample proportions, 3=poste...
3         (1=no storage, 2=store fit and split variables, 3=store s...
"credit_splits.txt" (split variable file name)
2         (1=do not save fitted values and node IDs, 2=save in a file)
"credit_fitted.txt" (file name for fitted values and node IDs)
2         (1=do not write R function, 2=write R function)
"credit_pred.R" (R code file)
1         (rank of top variable to split root node)

```

Now, all you have to do is feed this input file back into the GUIDE program (again, see the official manual for details). Once the modeling process is complete, you'll end up with several files depending on the options you specified during the initial setup. A portion of the output file produced by GUIDE for my input file is shown below; the corresponding tree diagram is displayed in [Figure 4.7](#).

```

Node 1: Intermediate node
A case goes into Node 2 if pay_0 <= 1.5000000
pay_0 mean = -0.15095238E-01

```

```

Class      Number   Posterior
no         16354   0.7788E+00
yes        4646    0.2212E+00
Number of training cases misclassified = 4646
Predicted class is no
-----
Node 2: Terminal node
Class      Number   Posterior
no         15686   0.8331E+00
yes        3143    0.1669E+00
Number of training cases misclassified = 3143
Predicted class is no
-----
Node 3: Terminal node
Class      Number   Posterior
no         668     0.3077E+00
yes        1503    0.6923E+00
Number of training cases misclassified = 668
Predicted class is yes
-----
Classification matrix for training sample:
Predicted      True class
class           no       yes
no             15686    3143
yes            668     1503
Total          16354    4646

Number of cases used for tree construction: 21000
Number misclassified: 3811
Resubstitution estimate of mean misclassification cost: 0.18147619

```

The train and test set accuracies for this tree are 81.85% and 82.21%, respectively (I had to use the R function produced by GUIDE to compute the test accuracy). Despite the reasonably high accuracy, we have a big problem! If you didn't first notice when initially exploring the data on your own, then hopefully you see it now...the model is biased towards predicting `yes` since the data are imbalanced (and naturally so, since we'd hope that most people are not defaulting on their credit card payments).

The original goal was to build a model to predict the probability of defaulting (`default = "yes"`), but the train and test accuracy within that specific class are 32.35% and 33.87%, respectively. By default, GUIDE (and many other algorithms) treat the misclassification for both types of error (i.e., predicting a yes as a no and vice versa) as equal. Fortunately, like CART, GUIDE can incorporate a matrix of misclassification costs into the tree construction (see [Section 2.2.4](#)).

Suppose, for whatever reason, we considered misclassifying a yes as a no (i.e., predicting that someone will not default on their next payment, when in fact they did) to be five times more costly than predicting a no as a yes. In GUIDE, it's as easy as setting up a loss matrix text file. For this example, I created a file called `credit_loss.txt` (in the same `/guide-v38.0/credit` directory as before) with the following two lines:

```
0 5
1 0
```

This corresponds to the following loss (or misclassification cost) matrix

$$\mathbf{L} = \begin{matrix} & \text{No} & \text{Yes} \\ \text{No} & 0 & 5 \\ \text{Yes} & 1 & 0 \end{matrix},$$

where  $L_{i,j}$  denotes the cost of classifying an observation as class  $i$  when it really belongs to class  $j$ . Note that GUIDE sorts the class values in alphabetical order (i.e., "no" then "yes"). Re-running the previous program, but specifying the cost matrix file when prompted, leads to the much more useful tree structure shown in [Figure 4.8](#). Although the overall test accuracy dropped from 82.21% to 60.00%, the accuracy within the class of interest (the *true positive rate* or *sensitivity*) increased from 7.49% to 17.79%—a significant improvement. I'll leave it to the reader to explore further with linear splits and non-constant fits to see if the results can be improved further.

---

## 4.10 Final thoughts

This chapter introduced the GUIDE algorithm for building classification and regression trees. GUIDE was developed to solve three problems often encountered with exhaustive search procedures (like CART):

1. split variable selection bias;
2. insensitivity to local interactions;
3. overly complex tree structures.

Like CTree, GUIDE solves the first problem by decoupling the search for split variables from the split point selection using statistical tests; in contrast to CTree, GUIDE exclusively uses one-*df* chi-square tests throughout. In selecting the splitting variable, GUIDE also looks at two-way interaction effects that can potentially mask the importance of a split when only main effects are considered. Moreover, GUIDE can often produce smaller and more accurate tree

structures by allowing splits in linear combinations of (two) predictors, and fitting more complex (i.e., non-constant) models in the nodes (e.g.,  $k$ -nearest neighbor or polynomial regression). Although this can lead to much shorter tree structures, the trees themselves are not necessarily simpler to interpret (e.g., is the GUIDE-based regression tree for the Ames housing example based on stepwise regression fits actually that simple to interpret?). Fortunately, the post-hoc interpretation procedures outlined in Chapter 6 are model-agnostic, and allow us to easily interpret various aspects of any supervised learning model (including GUIDE-based decision trees with complex terminal node summaries).

GUIDE is not just for simple classification and regression problems, and can be used in all sorts practical situations, including:

- quantile regression;
- Poisson regression (i.e., for modeling count data and rates);
- multivariate outcomes;
- censored outcomes;
- longitudinal data [Loh and Zheng, 2013] (e.g., when multiple subjects are continually measured over time);
- propensity score grouping;
- variable reduction;
- subgroup identification [Loh et al., 2015];
- and more.

For more, visit Wei-Yin Loh's website at <http://pages.stat.wisc.edu/~loh/guide.html>. An example of the effectiveness of *riluzole*, a drug approved for the treatment of ALS (Amyotrophic Lateral Sclerosis) by the US FDA, can be found in Loh and Zhou [2020]<sup>g</sup>.

---

<sup>g</sup>If you have trouble accessing any of Loh's papers, many of them are freely available on his website at <http://pages.stat.wisc.edu/~loh/guide.html>.

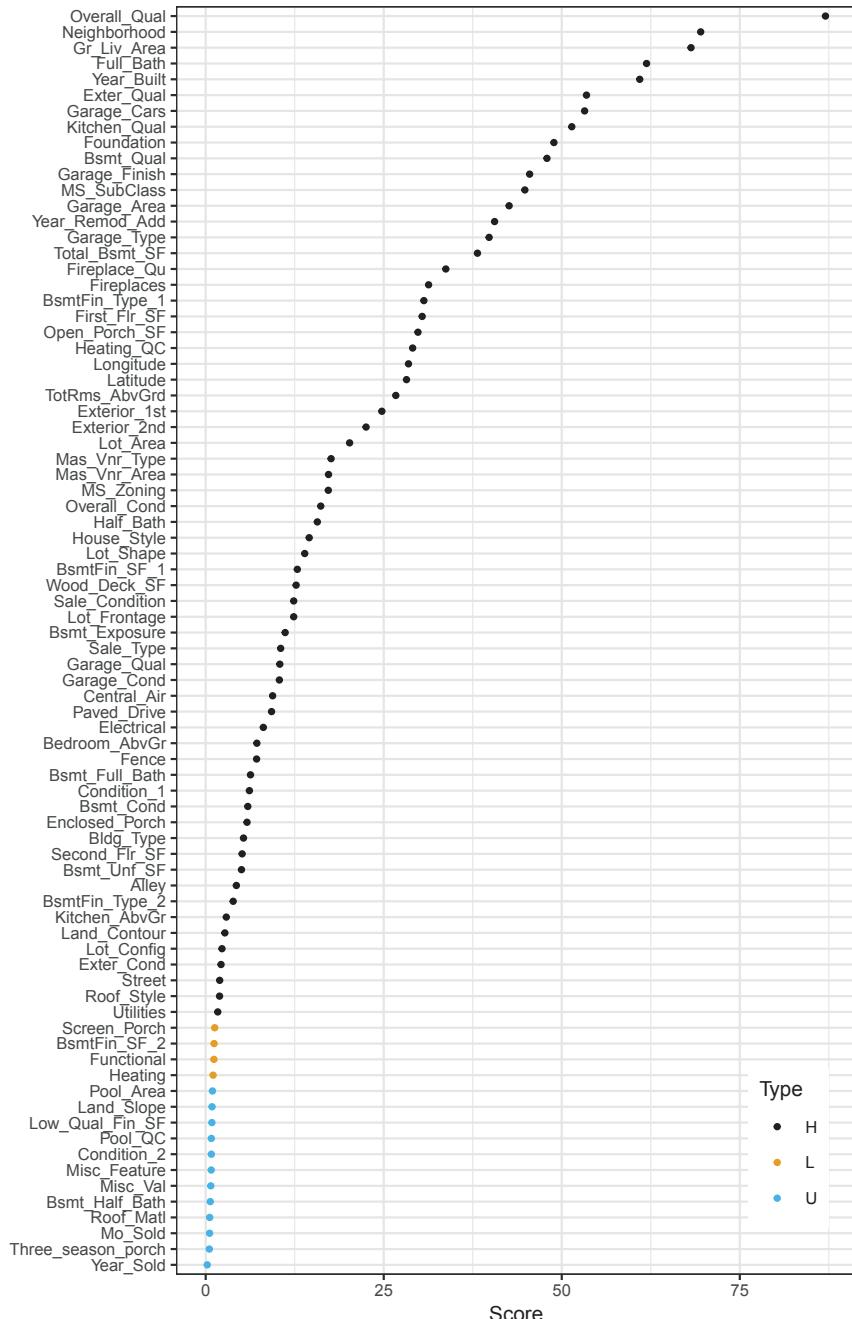
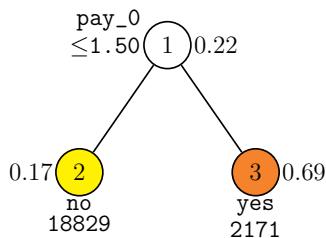
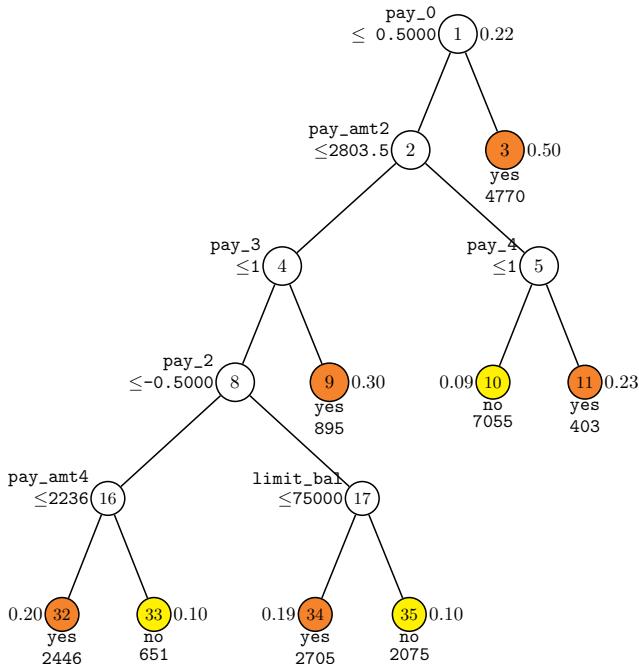


FIGURE 4.6: GUIDE-based variable importance scores for the Ames housing example. GUIDE distinguished between highly important (H), less important (L), and unimportant (U).



GUIDE v.38.0 1.00-SE classification tree for predicting `default` using estimated priors and unit misclassification costs. Tree constructed with 21000 observations. Maximum number of split levels is 30 and minimum node sample size is 210. At each split, an observation goes to the left branch if and only if the condition is satisfied. Predicted classes and sample sizes printed below terminal nodes; class sample proportion for `default = yes` beside nodes. Second best split variable at root node is `pay_2`.

FIGURE 4.7: GUIDE-based classification tree for the credit card default example. The autogenerated caption produced by the GUIDE software is also included.



GUIDE v.38.0 1.00-SE classification tree for predicting `default` using estimated priors and specified misclassification costs. Tree constructed with 21000 observations. Maximum number of split levels is 30 and minimum node sample size is 210. At each split, an observation goes to the left branch if and only if the condition is satisfied. Predicted classes and sample sizes printed below terminal nodes; class sample proportion for `default = yes` beside nodes. Second best split variable at root node is `pay_2`.

FIGURE 4.8: GUIDE-based classification tree with unequal misclassification costs for the credit card default example. The autogenerated caption produced by the GUIDE software is also included.

## Part II

# Tree-based ensembles



Taylor & Francis  
Taylor & Francis Group  
<http://taylorandfrancis.com>

# 5

---

## *Ensemble algorithms*

---

You know me, I think there ought to be a big old tree right there.  
And let's give him a friend. Everybody needs a friend.

Bob Ross

---

This chapter serves as a basic introduction to *ensembles*; specifically, ensembles of decision trees, although the ensemble methods discussed in this chapter are general algorithms that can also be applied to non-tree-based methods. The idea of ensemble modeling is to combine many models together in an attempt to increase overall prediction accuracy. As we'll see in this chapter, how the individual models are created and combined differs between the various ensembling techniques.

Have you ever watched the show *Who Wants to be a Millionaire?* If not, the game is very simple. A contestant is asked a series of multiple choice questions (each with four choices) of increasing difficulty, with a top prize of \$1,000,000. The contestant is allowed to use three "lifelines," a form of assistance to help the contestant with difficult questions. Over the years, a number of different lifelines were made available (e.g., the contestant could choose to randomly eliminate two of the incorrect answers). One of the most useful lifelines (IMO) involved polling the audience. If the contestant chose this lifeline, each audience member was able to use a device to cast their vote as to what they thought was the correct answer. The proportion of votes for each multiple choice answer was displayed to the contestant who could then choose to go with the popular vote or not. This lifeline was notorious for its accuracy; according to some sources, Regis Philbin (one of the hosts) once stated that the audience is right 95% of the time!

So why was polling the audience so accurate? As it turns out, this phenomenon has been observed for centuries and is often referred to as *the wisdom of the crowd*. In particular, it is often the case that the aggregated answers from a

large, diverse group of individuals is as accurate, if not more accurate, than the answer from any one individual from the group. For an interesting example, try looking up the phrase "Francis Galton Ox weight guessing" in a search engine. Another neat example is to ask a large number of individuals to guess how many jelly beans are in a jar, after you've eaten a handful, of course. If you look at the individual guesses, you'll likely notice that they vary all over the place. The average guess, however, tends to be closer than most of the individual guesses.

In a way, ensembles use the same idea to help improve the predictions (i.e., guesses) of an individual model and are among the most powerful supervised learning algorithms in existence. While there are many different types of ensembles, they tend to share the same basic structure:

$$f_B(\mathbf{x}) = \beta_0 + \sum_{b=1}^B \beta_b f_b(\mathbf{x}), \quad (5.1)$$

where  $B$  is the size of the ensemble, and each member of the ensemble  $f_b(\mathbf{x})$  (also called a *base learner*) is a different function of the input variables derived from the training data.

In this chapter, our interests lie primarily in using decision trees for the base learners—typically, CART-like decision trees ([Chapter 2](#)), but any tree algorithm will work. As discussed in Hastie et al. [2009, Section 10.2], many supervised learning algorithms (not just ensembles) can be seen as some form of additive expansion like (5.1). A single decision tree is one such example of an additive expansion. For a single tree,  $f_b(\mathbf{x}) = f_b(\mathbf{x}; \theta_b)$ , where  $\theta_b$  collectively represents the splits and split points leading to the  $b$ -th terminal node region, whose prediction is given by  $\beta_b$  (i.e., the terminal node mean response for ordinary regression trees). Other examples include *single-hidden-layer neural networks* and MARS [Friedman, 1991], among others.

There exist many different flavors of ensembles, and they all differ in the following ways:

- the choice of the base learners  $f_b(\mathbf{x})$  (although, in this book, the base learners will always be some form of decision tree);
- how the base learners are derived from the training data;
- the method for obtaining the estimated coefficients (or weights)  $\{\beta_b\}_{b=1}^B$ .

The ensemble algorithms discussed in this book fall into two broad categories, to be discussed over the next two sections: *bagging* ([Section 5.1](#)), short for **bootstrapping aggregating**, and *boosting* ([Section 5.2](#)). First, I'll discuss bagging, one of the simplest approaches to constructing an ensemble.

---

## 5.1 Bootstrap aggregating (bagging)

Bagging [Breiman, 1996a] is a *meta-algorithm* based on aggregating the results from multiple bootstrap samples. In the context of machine learning, this means aggregating the predictions from different base learners derived from independent bootstrap samples. When applied to unstable learners that are adaptive to the data, like overgrown/unpruned decision trees, the aggregated predictions can often be more accurate than the individual predictions from a single base learner trained on the original learning sample.

While bagging is a general algorithm, and can be applied to any type of base learner, it is most often successfully applied to decision trees, in particular, trees that have been fully grown to near-maximal depth without any pruning. As we learned in [Chapter 2](#), unpruned decision trees are considered unstable learners and have high variance (i.e., the predictions will vary quite a bit from sample to sample), which often results in overfitting and poor generalization performance. However, through averaging, bagging can often stabilize and reduce variance while maintaining low bias, which can result in improved performance.

We'll illustrate the effect of bagging through a simple example. Consider a training sample of size  $N = 500$  generated from the following sine wave with Gaussian noise:

$$Y = \sin(X) + \epsilon,$$

where  $X \sim \mathcal{U}(0, 2\pi)$  and  $\epsilon \sim \mathcal{N}(0, \sigma = 0.3)$ . [Figure 5.1](#) (left) shows the prediction surface from a single (overfit) decision tree grown to near full depth.<sup>a</sup> In contrast, [Figure 5.1](#) (right) shows a bagged ensemble of  $B = 1000$  such trees whose predictions have been averaged together; here, each tree was induced from a different bootstrap sample of the original data points. Clearly the individual tree is too complex (i.e., low bias and high variance) and will not generalize well to new samples, but averaging many such trees together resulted in a smoother, more stable prediction. The MSE from an independent test set of 10,000 observations was 0.173 for the single tree and 0.1 for the bagged tree ensemble; the optimal MSE for this example is  $\sigma^2 = 0.3^2 = 0.09$ .

The general steps for bagging classification and regression trees are outlined in Algorithm 5.1. To help further illustrate, a simple schematic of the process for building a bagged tree ensemble with four trees is given in [Figure 5.2](#). Note that bagged tree ensembles can be extended beyond simple classification and regression trees. For example, it is also possible to bag *survival trees* [Hothorn

---

<sup>a</sup>In this example, each tree was fit using `rpart()` with `minsplit = 2` and `cp = 0`.

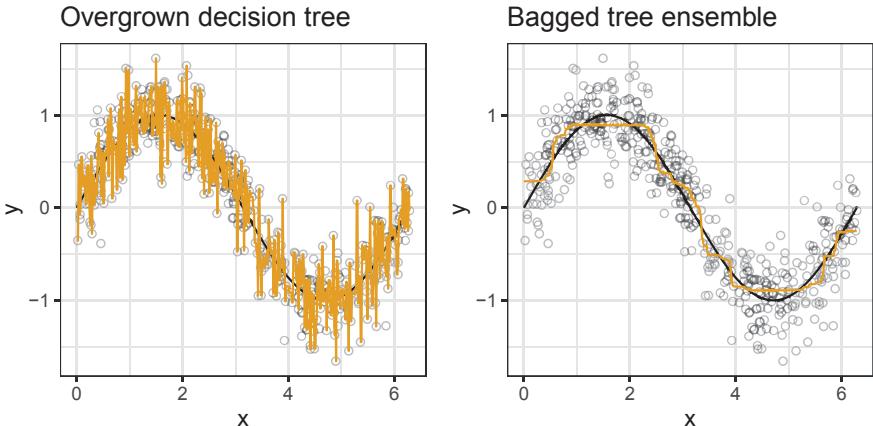


FIGURE 5.1: Simulated sine wave example ( $N = 500$ ). Left: a single (overgrown) regression tree. Right: a bagged ensemble of  $B = 1000$  overgrown regression trees whose predictions have been averaged together; here each tree was induced from a different bootstrap sample of the original data points. The individual tree is too complex (i.e., low bias and high variance) but averaging many such trees together results in a more stable prediction and smoother fit.

et al., 2004]. An improved bagging strategy specific to decision trees, called *random forest*, is the topic of [Chapter 7](#).

The (optional) OOB data in step 2) (b) of Algorithm 5.1 will be discussed in [Section 7.3](#), so just ignore it for now. The recommended default value for  $n_{min}$  (the minimum size of any terminal node in a tree) depends on the type of response variable:

$$n_{min} = \begin{cases} 1, & \text{for classification} \\ 5, & \text{for regression} \end{cases}.$$

Step 4) of Algorithm 5.1 mentions classification via voting. By voting, I mean that each tree makes a classification (i.e., casts a vote), and the ensemble classification is obtained by a majority vote (or plurality in the multiclass setting), with ties<sup>b</sup> typically handled at random. Class probability estimates for categorical outcomes can also be obtained from bagged tree ensembles and a discussion of different approaches is deferred to [Section 7.2.1](#); often, the class proportions from each tree are just averaged across all the trees in the bagged ensemble (similar to regression).

<sup>b</sup>To avoid issues with ties (i.e., an equal number of votes for each class), use an odd number of trees or base learners.

---

**Algorithm 5.1** Bagging for classification and regression trees.

---

- 1) Start with a training sample,  $\mathbf{d}_{trn} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ , and specify integers  $n_{min}$  (the minimum node size of a particular tree), and  $B$  (the number of trees in the ensemble).
  - 2) For  $b$  in  $1, 2, \dots, B$ :
    - a) Select a bootstrap sample  $\mathbf{d}_{trn}^*$  of size  $N$  from  $\mathbf{d}_{trn}$ .
    - b) **Optional:** Keep track of which observations from  $\mathbf{d}_{trn}$  were not selected to be in  $\mathbf{d}_{trn}^*$ ; these are called the *out-of-bag* (OOB) observations.
    - c) Fit a decision tree  $\mathcal{T}_b$  to  $\mathbf{d}_{trn}^*$  by recursively splitting each terminal node until the minimum node size ( $n_{min}$ ) is reached.
  - 3) Return the ensemble of trees:  $\{\mathcal{T}_b\}_{b=1}^B$ .
  - 4) To obtain the bagged prediction for a new case  $\mathbf{x}$ , denoted  $\hat{f}_B(\mathbf{x})$ , pass the observation down each tree—which will result in  $B$  separate predictions (one from each tree)—and aggregate as follows:
    - Classification:  $\hat{f}_B(\mathbf{x}) = \text{vote}\{\mathcal{T}_b(\mathbf{x})\}_{b=1}^B$ , where  $\mathcal{T}_b(\mathbf{x})$  is the predicted class label for  $\mathbf{x}$  from the  $b$ -th tree in the ensemble (in other words, let each tree vote on the classification for  $\mathbf{x}$  and take a majority/plurality vote at the end).
    - Regression:  $\hat{f}_B(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^B \mathcal{T}_b(\mathbf{x})$  (in other words, we just average the predictions for case  $\mathbf{x}$  across all the trees in the ensemble).
- 

Bagging has the same structural form as (5.1) with  $\beta_0 = 0$  and  $\{\beta_b = 1/B\}_{b=1}^B$ , and where each tree is induced from an independent bootstrap sample of the original training data and grown to near maximal depth (as specified by  $n_{min}$ ).

An important aspect of how the trees are constructed in bagging is that they are induced from independent bootstrap samples, which makes the bagging procedure trivial to parallelize. See Boehmke and Greenwell [2020, Sec. 10.4] for details and an example using the Ames housing data (Section 1.4.7) in R using the wonderful **foreach** package [Revolution Analytics and Weston, 2020].

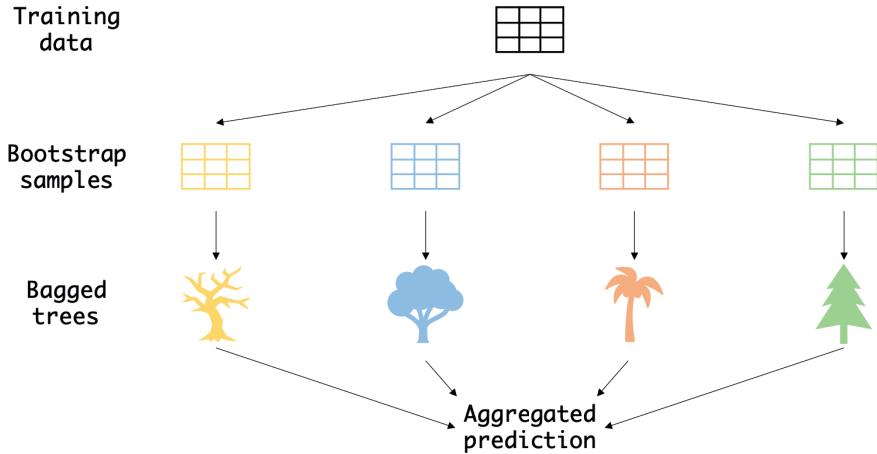


FIGURE 5.2: A simple schematic of the process for building a bagged tree ensemble with four trees.

### 5.1.1 When does bagging work?

In general, bagging helps to improve the accuracy of unstable procedures that are adaptive, nonlinear functions of the training data [Breiman, 1996a]. Let  $\hat{f}$  represent an individual model (e.g., a single decision tree) trained to the learning sample and  $\hat{f}_B$  represent a bagged ensemble thereof. Specifically, if small changes in  $d_{trn}$  lead to small changes in  $\hat{f}$ , then  $\hat{f}_B \approx \hat{f}$  and not much is gained in terms of improvement. On the other hand, if small changes in  $d_{trn}$  lead to large changes in  $\hat{f}$ , then  $\hat{f}_B$  will often be an improvement over  $\hat{f}$ . In the latter case, we often call  $\hat{f}$  an unstable model. Breiman [1996b] noted that algorithms like neural networks, CART, and subset selection in linear regression were unstable, while algorithms like  $k$ -nearest neighbor methods were stable; the MARSprocedure [Friedman, 1991], an extension of CART, can also be considered as unstable predictors and therefore potentially benefit from bagging).

Unpruned CART-like decision trees are particularly unstable predictors; that is, the tree structure will often vary heavily from one sample to the next. Hence, bagging is often most successful when applied to unpruned decision trees that have been grown to maximal (or near maximal) depth.

### 5.1.2 Bagging from scratch: classifying email spam

To illustrate, let's return to the email spam example first introduced in [Section 1.4.5](#). In the code snippet below, we load the data from the **kernlab**

package and split the observations into train/test sets using the same 70/30 split as before:

```
data(spam, package = "kernlab")
set.seed(852) # for reproducibility
id <- sample.int(nrow(spam), size = floor(0.7 * nrow(spam)))
spam.trn <- spam[id, ] # training data
spam.tst <- spam[-id, ] # test data
```

Rather than writing our own bagger function, I'll construct a bagged tree ensemble using a basic `for` loop that stores the individual trees in a list called `spam.bag`. Note that I turn off cross-validation (`xval = 0`) when calling `rpart()` to save on computing time. The code is shown below.

```
library(rpart)

B <- 500 # number of trees in ensemble
ctrl <- rpart.control(minsplit = 2, cp = 0, xval = 0)
N <- nrow(spam.trn) # number of training observations
spam.bag <- vector("list", length = B) # to store trees
set.seed(900) # for reproducibility
for (b in seq_len(B)) { # fit trees to independent bootstrap samples
  boot.id <- sample.int(N, size = N, replace = TRUE)
  boot.df <- spam.trn[boot.id, ] # bootstrap sample
  spam.bag[[b]] <- rpart(type ~ ., data = boot.df, control = ctrl)
}
```

Now that we have the individual trees, each of which was fit to a different bootstrap sample from the training data, we can obtain predictions and assess the performance of the ensemble using the test sample. To that end, I'll loop through each tree to obtain predictions on the test set (`spam.tst`), and store the results in an  $N \times B$  matrix, one column for each tree in the ensemble. I then compute the test error as a function of  $B$  by cumulatively aggregating the predictions from trees 1 through  $B$  by means of voting (e.g., if we are computing the bagged prediction using only the first three trees, the final prediction for each observation will simply be the the class with the most votes across the three trees).

To help with the computations, I'll write two small helper functions, `vote()` and `err()`, for carrying out the voting and computing the misclassification error, respectively:

```
vote <- function(x) names(which.max(table(x)))
err <- function(pred, obs) 1 - sum(diag(table(pred, obs))) /
  length(obs)
```

Next, I obtain the  $N \times B$  matrix of predictions and cumulatively aggregate them across all the trees in the ensemble to compute the test error as a function of the number of trees:

```

spam.bag.preds <- sapply(spam.bag, FUN = function(tree) {
  predict(tree, newdata = spam.tst, type = "class")
}) # N x B matrix of individual tree predictions

# Compute test error as a function of number of trees
spam.bag.err <- sapply(seq_len(B), FUN = function(b) {
  agg.pred <- apply(spam.bag.preds[, seq_len(b), drop = FALSE],
                     MARGIN = 1, FUN = vote) # aggregate trees 1:b
  err(agg.pred, obs = spam.tst$type) # compute test error
})
min(spam.bag.err) # minimum misclassification error

#> [1] 0.0485

```

The results are displayed in Figure 5.3. The error stabilizes after around 200 trees and achieves a minimum misclassification error rate of 4.85% (horizontal dashed line). For reference, a single tree (pruned using the 1-SE rule) achieved a test error of 9.99%. Averaging the predictions from several hundred overgrown trees cut the misclassification error by more than half!

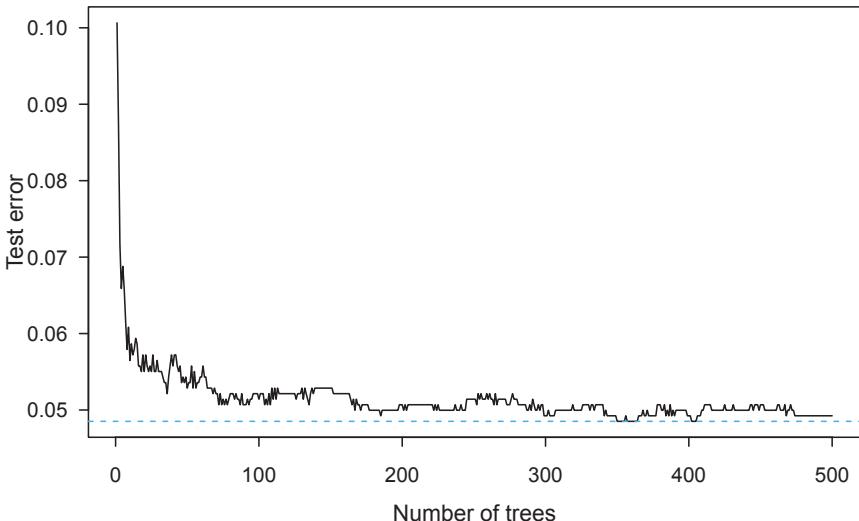


FIGURE 5.3: Test misclassification error for the email spam bagging example. The error stabilizes after around 200 trees and achieves a minimum misclassification error rate of 4.85% (horizontal dashed line).

While bagging was quite successful in the email spam example, sometimes bagging can make things worse. For a good discussion on how bagging can worsen bias and/or variance, see Berk [2008, Sec. 4.5.2–4.5.3].

### 5.1.3 Sampling without replacement

Inducing trees from independent learning sets that are bootstrap samples from the original training data imitates the process of building trees on independent samples of size  $N$  from the true underlying population of interest. While bagging traditionally utilizes bootstrap sampling (i.e., sampling with replacement) for training the individual base learners, it can sometimes be advantageous to use subsampling without replacement; Breiman [1999] referred to this as *pasting*. In particular, if  $N$  is “large enough,” then bagging using random subsamples of size  $N/2$  (i.e., sampling without replacement) can be an effective alternative to bagging based on the bootstrap [Friedman and Hall, 2007]. Strobl et al. [2007b] suggest using a subsample size of 0.632 times the original sample size  $N$ —because in bootstrap sampling about 63.2% of the original observations end up in any particular bootstrap sample.

This is quite fortunate since sampling half the data without replacement is much more efficient and can dramatically speed up the bagging process. Applying this to the email spam data from the previous section, which only required modifying one line of code in the previous example, resulted in a minimum test error of 5.21%, quite comparable to the previous results using the bootstrap but much faster to train.

Another reason why subsampling can sometimes improve the performance of bagging is through “de-correlation”. Recall that bagging can improve the performance of unstable learners through variance reduction. As discussed in more detail in [Section 7.2](#), correlation limits the variance-reducing effect of averaging. The problem here is that the trees in a bagged ensemble will often be correlated since they are all induced off of bootstrap samples from the same training set (i.e., they will share similar splits and structure, to some degree). Using subsamples of size  $N/2$  will help to de-correlate the trees which can further reduce variance, resulting in improved generalization performance. A more effective strategy to de-correlate trees in a bagged ensemble is discussed in [Section 7.2](#).

### 5.1.4 Hyperparameters and tuning

Bagged tree ensembles are convenient because they don’t require much tuning. That’s not to say that you can’t improve performance by tuning some of the tree parameters (e.g., tree depth). However, in contrast to gradient tree boosting ([Chapter 8](#)), increasing the number of trees ( $B$ ) does not necessarily lead to overfitting (see [Figure 5.4](#) on page 193), and isn’t really a tuning parameter—although, computation time increases with  $B$ , so it can be advantageous to monitor performance on a validation set to determine when performance has plateaued or reached a point of diminishing return.

### 5.1.5 Software

Bagging is rather straightforward to implement directly, as is seen in this chapter. Nonetheless, several R packages exist which can be used to implement Algorithm 5.1. The R package **ipred** [Peters and Hothorn, 2021], which stands for improved **predictors**, implements bagged decision trees using the **rpart** package and supports classification, regression, and survival problems. The R package **adabag** [Alfaro et al., 2018] also implements bagging via **rpart**, but only supports classification problems. In Python, bagging is implemented in scikit-learn’s **sklearn.ensemble** module and can be applied to any base estimator (i.e., not just decision trees).

In general, I think it’s more efficient to use random forest software to implement bagging, especially in R. This is because random forest software typically builds the entire ensemble using highly efficient and compiled code, whereas **ipred**, for example, utilizes existing tree software and builds the ensemble in a way similar to what we did in the email spam example. A regression example using random forest software to implement bagging is given in [Section 5.5](#). Random forest is the topic of [Chapter 7](#).

---

## 5.2 Boosting

Recall that bagging reduces variance by averaging several unstable learners together. Boosting, on the other hand, was originally devised for binary classification problems as a way to boost the performance of weak learners—a model that only does slightly better than the *majority vote classifier* (i.e., a model that always predicts the majority class in the learning sample). So how does boosting improve the performance of a weak learner? The basic idea is quite intuitive: fit models in a sequential manner, where each model in the sequence is fit on a resampled version of the training data that gives more weight to the observations that were previously misclassified, hence, “boosting” the performance of the previous models. Each successive model in a boosting sequence effectively homes in on the rows of data where the previous model had the largest errors.

Several different flavors of boosting exist, and the procedure has evolved quite a bit since its initial inception for binary classification. In the following section, I’ll discuss one of the earliest and most popular flavors of boosting for binary classification: *AdaBoost.M1* [Freund and Schapire, 1996]. It’s important to call out that AdaBoost.M1, along with its many variants, assumes that the base learners (in our case, binary classification trees) can incorporate case

weights. A more general (and flexible) boosting strategy will be covered in Chapter 8.

### 5.2.1 AdaBoost.M1 for binary outcomes

AdaBoost.M1—also referred to as *Discrete AdaBoost* in Friedman et al. [2000] due to the fact that the base learners each return a discrete class label—fits an additive model of the form

$$C(\mathbf{x}) = \text{sign} \left( \sum_{b=1}^B \alpha_b C_b(\mathbf{x}) \right),$$

where  $\{\alpha_b\}_{b=1}^B$  are coefficients that weight the contribution of each respective base learner  $C_b(\mathbf{x})$  and

$$\text{sign}(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ 0 & \text{otherwise} \end{cases}.$$

In essence, classifiers in the sequence with higher accuracy receive more weight and therefore have more influence on the final classification  $C(\mathbf{x})$ .

The details of AdaBoost.M1 are given in Algorithm 5.2. The crux of the idea is this: start with an initial classifier built from the training data using equal case weights  $\{w_i = 1/N\}_{i=1}^N$ , then increase  $w_i$  for those cases that have been most frequently misclassified. The process is continued a fixed number of times ( $B$ ).

Like bagging, boosting is a *meta-algorithm* that can be applied to any type of model, but it's often most successfully applied to shallow decision trees (i.e., decision trees with relatively few splits/terminal nodes). While bagging relies upon aggregating the results from several unstable learners, boosting tends to benefit from sequentially improving the performance of a weak learner (like a simple decision stump). In the next section, I'll code up Algorithm 5.2 and apply it to the email spam data for comparison with the previously obtained bagged tree ensemble.

While AdaBoost.M1 was one of the most accurate classifiers at the time<sup>c</sup>, the fact that it only produced a classification was a severe limitation. To that end, Friedman et al. [2000] generalized the AdaBoost.M1 algorithm so that the weak learners return a class probability estimate, as opposed to a discrete class

---

<sup>c</sup>In fact, shortly after its introduction, Leo Breiman referred to AdaBoost as the “...best off-the-shelf classifier in the world.”

---

**Algorithm 5.2** Vanilla AdaBoost.M1 algorithm for binary classification.

---

- 1) Initialize case weights  $\{w_i = 1/N\}_{i=1}^N$ .
  - 2) For  $b = 1, 2, \dots, B$ :
    - a) Fit a classifier  $C_b(\mathbf{x})$  to the training observations using case weights  $w_i$ .
    - b) Compute the weighted misclassification error
$$err_b = \frac{\sum_{i=1}^N w_i I(y_i \neq C_b(\mathbf{x}))}{\sum_{i=1}^N}$$
  - c) Compute  $\alpha_b = \log(1/err_b - 1)$ .
  - d) Update case weights:  $\{w_i \leftarrow \exp[\alpha_b I(y_i \neq C_b(\mathbf{x}))]\}_{i=1}^N$ .
- 3) Return the weighted majority vote:  $C(\mathbf{x}) = \text{sign}\left(\sum_{b=1}^B \alpha_b C_b(\mathbf{x})\right)$
- 

label; the contribution to the final classifier is half the logit-transform of this probability estimate. They refer to this procedure as *Real AdaBoost*. Other generalizations (e.g., to multi-class outcomes) also exist. In [Chapter 8](#), I'll discuss a much more flexible flavor of boosting, called *stochastic gradient tree boosting*, which can naturally handle general outcome types (e.g., continuous, binary, Poisson counts, censored, etc.).

### 5.2.2 Boosting from scratch: classifying email spam

To illustrate, let's apply AdaBoost.M1 (Algorithm 5.2) to the email spam data and show how it “boosts” the performance of an individual **rpart** tree; I'll continue with the same train/test splits from the previous example in [Section 5.1.2](#). For this example, I'll use  $B = 500$  depth-10 decision trees. Since AdaBoost.M1 requires  $y \in \{-1, +1\}$ , I'll re-code the response (**type**) so that **type = "spam"** corresponds to  $y = +1$ :

```
spam.trn$type <- ifelse(spam.trn$type == "spam", 1, -1)
spam.tst$type <- ifelse(spam.tst$type == "spam", 1, -1)
spam.xtrn <- subset(spam.trn, select = -type) # feature columns only
spam.xtst <- subset(spam.tst, select = -type) # feature columns only
```

Following the previous example on bagging, I'll use a simple **for** loop and **list()** to sequentially construct and store the fitted trees, respectively.

For AdaBoost.M1, we also have to collect and store the  $\{\alpha_b\}_{b=1}^B$  coefficients in order to make predictions later. Note that `predict.rpart()` returns a factor—in this case, with factor levels "`-1`" and "`1`"—which needs to be coerced to numeric before further processing; this is the purpose of the `fac2num()` helper function in the code below<sup>d</sup>:

```
library(rpart)

# Helper function to coerce factors to numeric
fac2num <- function(x) as.numeric(as.character(x))

# Apply AdaBoost.M1 algorithm
B <- 500 # number of trees in ensemble
ctrl <- rpart.control(maxdepth = 10, xval = 0)
N <- nrow(spam.trn) # number of training observations
w <- rep(1 / N, times = N) # initialize weights
spam.ada <- vector("list", length = B) # to store sequence of trees
alpha <- numeric(B) # to hold coefficients
for (i in seq_len(B)) { # for b = 1, 2, ..., B
  spam.ada[[i]] <- rpart(type ~ ., data = spam.trn, weights = w,
                         control = ctrl, method = "class")
  # Compute predictions and coerce factor output to +1/-1
  pred <- fac2num(predict(spam.ada[[i]], type = "class"))
  err <- sum(w * (pred != spam.trn$type)) / sum(w) # weighted error
  if (err == 0 | err == 1) { # to avoid log(0) and dividing by 0
    err <- (1 - err) * 1e-06 + err * 0.999999
  }
  alpha[i] <- log((1 / err) - 1) # coefficient from step 2) (c)
  w <- w * exp(alpha[i] * (pred != spam.trn$type)) # update weights
}
}
```

Next, I'll generate predictions for the test data (`spam.tst`) using the first  $b$  trees (where  $b$  will be varied over the range  $1, 2, \dots, B$ ) and compute the misclassification error for each; note that I'm using the same `err()` function defined in the previous example for bagging:

```
spam.ada.preds <- sapply(seq_len(B), FUN = function(i) {
  class.labels <- predict(spam.ada[[i]], newdata = spam.tst,
                          type = "class")
  alpha[i] * fac2num(class.labels)
}) # (N x B) matrix of un-aggregated predictions

# Compute test error as a function of number of trees
spam.ada.err <- sapply(seq_len(B), FUN = function(b) {
  agg.pred <- apply(spam.ada.preds[, seq_len(b)], drop = FALSE),
              MARGIN = 1, FUN = function(x) sign(sum(x)))
  err(agg.pred, obs = spam.tst$type)
})
```

---

<sup>d</sup>According to the R FAQ guide (<https://cran.r-project.org/doc/FAQ/>), a more efficient but harder to remember solution is to use `as.numeric(levels(x))[as.integer(x)]`.

```
}
min(spam.ada.err) # minimum misclassification error
#> [1] 0.0406
```

The results are plotted in [Figure 5.4](#), along with those from the previously obtained bagged tree ensembles (i.e., using sampling with/without replacement). The minimum test error from the AdaBoost.M1 ensemble is 0.041. Compare this to the bagged tree ensemble based on sampling with replacement, which achieved a minimum test error of 0.049. In this case, AdaBoost.M1 slightly outperforms bagging.

For comparison, let's see how a single depth-10 decision tree—the base learner for our AdaBoost.M1 ensemble—performs on the same data.

```
spam.tree.10 <- rpart(type ~ ., data = spam.trn,
                       maxdepth = 10, method = "class")
pred <- predict(spam.tree.10, newdata = spam.tst, type = "class")
pred <- as.numeric(as.character(pred)) # coerce to numeric
mean(pred != spam.tst$type)
#> [1] 0.12
```

Wow, boosting decreased the misclassification error of a single depth-10 tree by roughly 66.27%, nice!

### 5.2.3 Tuning

In contrast to bagging, the number of base learners is often a critical tuning parameter in boosting algorithms, as they can often overfit for large enough  $B$ . While [Figure 5.4](#) doesn't give any indication of overfitting, AdaBoost.M1 (and any boosting algorithm) can certainly overfit; an example of overfitting with AdaBoost is given in Hastie et al. [2009, p. 616; Figure 16.5]. The performance of a boosted tree ensemble can also be sensitive to the tree-specific parameters, such as the tree depth or maximum number of terminal nodes. Further refinements to AdaBoost, like the addition of *shrinkage* and subsampling, introduce other important tuning parameters. These are discussed in more detail in [Section 8.3](#).

### 5.2.4 Forward stagewise additive modeling and exponential loss

Aside from bagging, additive expansions like (5.1) are often fit by minimizing some *loss function*<sup>e</sup>, like *least squares* loss,

---

<sup>e</sup>A loss function measures the error in predicting  $f(\mathbf{x})$  instead of  $y$ .

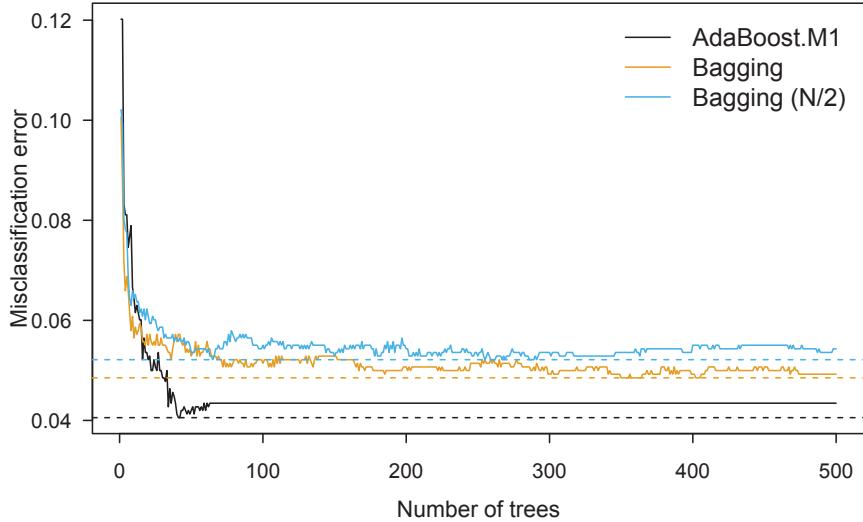


FIGURE 5.4: Misclassification error on the email spam test set from several different tree ensembles: 1) an AdaBoost.M1 classifier with depth-10 classification trees (black curve), 2) a bagged tree ensemble using max depth trees and sampling with replacement (yellow curve), and 3) a bagged tree ensemble using max depth trees and subsampling with replacement (blue curve). The horizontal dashed lines represent the minimum test error obtained by each ensemble.

$$\min_{\{\beta_b, \theta_b\}_{b=1}^B} \sum_{i=1}^N L \left( y_i, \sum_{b=1}^B \beta_b f_b(\mathbf{x}_i; \theta_b) \right).$$

For many combinations of loss functions and base learners, the solution can involve complicated and expensive numerical techniques. Fortunately, a simple approximation can often be used when it is more feasible to solve the optimization problem for a single base learner. This approximate solution is called *stagewise additive modeling*, the details of which are listed in Algorithm 5.3 below.

Friedman et al. [2000] show that AdaBoost.M1 (Algorithm 5.2) is equivalent to *forward stagewise additive modeling* using the exponential loss function

$$L(y, f(\mathbf{x})) = \exp(-yf(\mathbf{x})). \quad (5.2)$$

The product  $yf(\mathbf{x})$  is referred to as the “margin” and is analogous to the residual,  $y - f(\mathbf{x})$ , for continuous outcomes. Hence, AdaBoost.M1 can be

---

**Algorithm 5.3** Stagewise additive modeling

---

- 1) Initialize  $f_b(\mathbf{x}_i; \theta_b) = 0$  (a constant).
- 2) For  $b = 1, 2, \dots, B$ 
  - a) Optimize the loss for a single basis function; in particular, solve

$$(\beta_b, \theta_b) = \arg \min_{\beta_b, \theta_b} \sum_{i=1}^N L(y_i, \beta f_{b-1}(\mathbf{x}_i) + \beta f(\mathbf{x}_i; \theta_b)).$$

- b) Set  $f_b(x) = f_{b-1}(x) + \beta_b f(x)$ .
- 

derived in an equivalent way that conforms to exactly the same structure as (5.1).

The boosting procedure discussed in [Chapter 8](#) follows the forwards stagewise fitting approach more explicitly and includes AdaBoost as a special case, so more on this later. The R package **gbm** [Greenwell et al., 2021b] (originally by Greg Ridgeway) implements AdaBoost.M1 via this approach.

### 5.2.5 Software

Different flavors of AdaBoost (e.g., Discrete AdaBoost, Real AdaBoost, and Gentle AdaBoost) are available in the R package **ada** [Culp et al., 2016]; AdaBoost.M1 and AdaBoost.SAMME are also implemented in the R package **adabag**, as well as scikit-learn’s **sklearn.ensemble** module. While R’s implementations utilize CART-like decision trees for the base learner (using **rpart**), scikit-learn’s implementation allows you to boost any compatible scikit-learn model that supports case weights. As mentioned in the previous section, the R package **gbm** implements AdaBoost.M1 as a forward stagewise additive model; more on **gbm** in [Chapter 8](#).

A boosting strategy similar to AdaBoost is used to boost C5.0 decision trees and is available via the **C5.0** package.

### 5.3 Bagging or boosting: which should you use?

Among tree-based ensembles, it is generally regarded that boosting outperforms bagging (and its variants, like the random forest procedure discussed in [Chapter 7](#)). However, this is not always the case and, as discussed briefly in this chapter and in [Chapter 8](#), boosting tends to require more work up front in terms of tuning in order to see the gains, whereas bagged tree ensembles tend to perform well right out of the box with little to no tuning<sup>f</sup>; this is especially true for random forests. My opinion is summarized by the following relationship:

$$\text{Gradient boosted trees} \geq \text{Random forest} > \text{Bagged trees} > \text{Single tree}.$$

So, while boosted tree ensembles tend to outperform their bagged counterparts, I don't often find the performance increase to be worth the added complexity and time associated with the additional tuning. It's a trade off that we all must take into consideration for the problem at hand. It should also be noted that sometimes a single decision tree is the right tool for the job, and an ensemble thereof would be overkill; see, for example, [Section 7.9.1](#).

---

### 5.4 Variable importance

Recall from [Section 2.8](#) that the relative importance of predictor  $x$  is essentially the sum of the squared improvements over all internal nodes of the tree for which  $x$  was chosen as the partitioning variable. This idea also extends to ensembles of decision trees, such as bagged and boosted tree ensembles. In ensembles, the improvement score for each predictor is averaged across all the trees in the ensemble. Because of the stabilizing effect of averaging, the aggregated tree-based variable importance score is often more reliable in large ensembles; see Hastie et al. [2009, p. 368], although, as we'll see in [Chapter 7](#), split variable selection bias will also affect the variable importance scores often produced by tree-based ensembles using CART-like decision trees.

---

<sup>f</sup>By “well” here, I mean close to how well they would perform with optimal tuning; the default is usually “in the ballpark.”

## 5.5 Importance sampled learning ensembles

Tree-based ensembles (especially those discussed in Chapters 7–8) often do a good job in building a prediction model, but at the end of the day can involve a lot of trees which can limit their use in production since they can require more memory and take longer to score new data sets.

To help overcome these issues, Friedman and Popescu [2003] introduced the concept of *importance sampled learning ensembles* (ISLEs). Many of the tree-based ensembles discussed in this book—including bagged tree ensembles—are examples of ISLEs. The main point here is that ISLEs can sometimes benefit from post-processing via a technique called the LASSO, which stands for *least absolute shrinkage and selection operator* [Tibshirani, 1996]. Such post-processing can often maintain or, in some cases, improve the accuracy of the original ensemble while dramatically improving computational performance (e.g., lower memory requirements and faster training times). For full details, see Friedman and Popescu [2003], Hastie et al. [2009, Sec. 16.3.1], and Efron and Hastie [2016, 346–347].

The idea is to use the LASSO to select a subset of trees from a fitted ensemble and re-weight them, which can result in an ensemble with far fewer trees and (hopefully) comparable, if not better, accuracy. This is important to consider in real applications since tree ensembles can sometimes require many thousands of decision trees to reach peak performance, often resulting in a large model to maintain in memory and slower scoring times (aspects that are important to consider before deploying a model in a production process).

The LASSO-based post-processing procedure essentially involves fitting an  $L_1$ -penalized regression model of the form

$$\min_{\{\beta_b\}_{b=1}^B} \sum_{i=1}^N L \left[ y_i, \sum_{b=1}^B \hat{f}_b(\mathbf{x}_i) \beta_b \right] + \lambda \sum_{b=1}^B |\beta_b|,$$

where  $\hat{f}_b(\mathbf{x}_i)$  ( $b = 1, 2, \dots, B$ ) is the prediction(s) from the  $b$ -th tree for observation  $i$ ,  $\beta_b$  are fixed, but unknown coefficients to be estimated via the LASSO, and  $\lambda$  is the  $L_1$ -penalty to be applied.

The wonderful and efficient **glmnet** package [Friedman et al., 2021] for R can be used to fit the entire LASSO regularization path<sup>g</sup>; that is it efficiently computes the estimated model coefficients for an entire grid of relevant  $\lambda$  values.

---

<sup>g</sup>The **glmnet** package actually implements the entire *elastic net* regularization path for many types of generalized linear models. The LASSO is just a special case of the elastic net, which combines both the LASSO and ridge (i.e.,  $L_2$ ) penalties.

The optimal value of  $\lambda$  can be chosen via cross-validation or an independent test set.

Note that not all ensembles will perform well with post-processing. As discussed in Hastie et al. [2009, section 16.3.1], the individual trees should cover the space of predictors where needed and be sufficiently different from each other for the post-processor to be effective. Strategies for different tree ensembles are provided in Friedman and Popescu [2003] (e.g., using smaller subsamples when sampling without replacement, like 5–10%, and shallower trees for bagged tree ensembles). The next example shows how to apply this post-processing strategy to a bagged tree ensemble using the Ames housing data ([Section 1.4.7](#)).

### 5.5.1 Example: post-processing a bagged tree ensemble

To illustrate, let's return to the Ames housing example. Below, I'll load the data into R and apply the same 70/30 split from the previous example. Note that I continue to rescale `Sale_Price` by dividing by 1000; this is strictly for plotting purposes.

```
ames <- as.data.frame(AmesHousing::make_ames())
ames$Sale_Price <- ames$Sale_Price / 1000 # rescale response
set.seed(2101) # for reproducibility
id <- sample.int(nrow(ames), size = floor(0.7 * nrow(ames)))
ames.trn <- ames[id, ] # training data/learning sample
ames.tst <- ames[-id, ] # test data
ames.xtst <- subset(ames.tst, select = -Sale_Price) # features only
```

Next, I'll fit a bagged tree ensemble using the **randomForest** package [Breiman et al., 2018] (computational reasons for doing so are discussed in [Section 5.1.5](#)). Random forest, and its open source implementations, are not discussed until [Chapter 7](#). For now, just note that the **randomForest** package, among others, can be used to implement bagged tree ensembles by tweaking a special parameter, often referred to as  $m_{try}$  (to be discussed in [Section 7.2](#)), and setting this parameter equal to the number of total predictors will result in an ordinary bagged tree ensemble). This will be much more efficient than relying on the **ipred** package and will also allow us to obtain predictions from the individual trees, rather than just the aggregated predictions. Examples of post-processing an RF and boosted tree ensemble are given in [Sections 7.9.2](#) and [8.9.3](#), respectively.

Here, I'll fit two models, each containing  $B = 500$  trees:

- a standard bagged tree ensemble where each tree is fully grown to bootstrap samples of size  $N$  (`ames.bag`);

- a bagged tree ensemble consisting of shallow six-node trees, each of which is grown using only a 5% random sample of the training data without replacement (`ames.bag.6.5`).

Both models are trained in the code chunk below. Note that I recorded the training time of each fit using `system.time()`, which will provide some insight into the potential computational savings offered by this post-processing method<sup>h</sup>. Although substantially less accurate (see Figure 5.6), notice how much faster it is to train `ames.bag.6.5`:

```
library(randomForest)

# Fit a typical bagged tree ensemble
system.time({
  set.seed(942) # for reproducibility
  ames.bag <-
    randomForest(Sale_Price ~ ., data = ames.trn, mtry = 80,
                 ntree = 500, xtest = ames.xtst,
                 ytest = ames.tst$Sale_Price, keep.forest = TRUE)
})

#>   user  system elapsed
#> 120.407  0.788 123.070

# Print results
print(ames.bag)

#>
#> Call:
#>   randomForest(formula = Sale_Price ~ ., data = ames.trn, mtry = 80...
#>                 Type of random forest: regression
#>                           Number of trees: 500
#> No. of variables tried at each split: 80
#>
#>           Mean of squared residuals: 690
#>                           % Var explained: 89
#>                               Test set MSE: 628
#>                           % Var explained: 90.6

# Fit a bagged tree ensemble using six-node trees on 5% samples
system.time({
  set.seed(1021)
  ames.bag.6.5 <-
    randomForest(Sale_Price ~ ., data = ames.trn, mtry = 80,
                 ntree = 500, maxnodes = 6,
                 sampsize = floor(0.05 * nrow(ames.trn)),
                 replace = FALSE, keep.forest = TRUE,
                 xtest = ames.xtst, ytest = ames.tst$Sale_Price)
})
```

---

<sup>h</sup>Note that there are better ways to benchmark and time expressions in R; see, for example, the `microbenchmark` package [Mersmann, 2021].

```
#>    user  system elapsed
#> 0.395   0.003   0.402

# Print results
print(ames.bag.6.5)

#>
#> Call:
#> randomForest(formula = Sale_Price ~ ., data = ames.trn, mtry = 80...
#>                   Type of random forest: regression
#>                   Number of trees: 500
#> No. of variables tried at each split: 80
#>
#>           Mean of squared residuals: 1489
#>           % Var explained: 76.2
#>           Test set MSE: 1450
#>           % Var explained: 78.2

# Test set MSE as a function of the number of trees
mse.bag <- ames.bag$test$mse
mse.bag.6.5 <- ames.bag.6.5$test$mse
```

Next, I'll use **glmnet** to post-process each ensemble using the LASSO. The following steps are conveniently handled by **treemisc**'s **isle\_post()** function, which I'll use to post-process the **ames.bag.6.5** ensemble. But first, I think it's prudent to show the individual steps using the **ames.bag** ensemble.

To start, I'll compute the individual tree predictions for the train and test sets and store them in a matrix

```
preds.trn <- predict(ames.bag, newdata = ames.trn,
                      predict.all = TRUE)$individual
preds.tst <- predict(ames.bag, newdata = ames.tst,
                      predict.all = TRUE)$individual
```

Next, I'll use the **glmnet()** function to fit the entire regularization path using the training predictions from the  $B = 500$  individual trees:

```
library(glmnet)

# Fit the LASSO regularization path
lasso.ames.bag <- glmnet(
  x = preds.trn, # individual tree predictions are the predictors
  y = ames.trn$Sale_Price, # same response variable
  lower.limits = 0, # coefficients should be strictly positive
  standardize = FALSE, # no need to standardize
  family = "gaussian" # least squares regression
)
```

A few things to note about the above code chunk are in order. Since this is a regression problem, I set **family = "gaussian"** (for least squares) in

the call to `glmnet()`. Second, since the individual tree predictions are all on the same scale, there's no need to standardize the inputs (`standardize = FALSE`). Lastly, we could argue that the estimated coefficients (one for each tree) should be non-negative (`lower.limits = 0`).

[Figure 5.5](#) shows the regularization path for the estimated coefficients. In particular, the  $\lambda$  values (on the log scale) are plotted on the  $x$ -axis, and the  $y$ -axis corresponds to the estimated coefficient value (one curve per coefficient/tree). The top axis highlights the number of non-zero coefficients at each particular value of the penalty parameter  $\lambda$ :

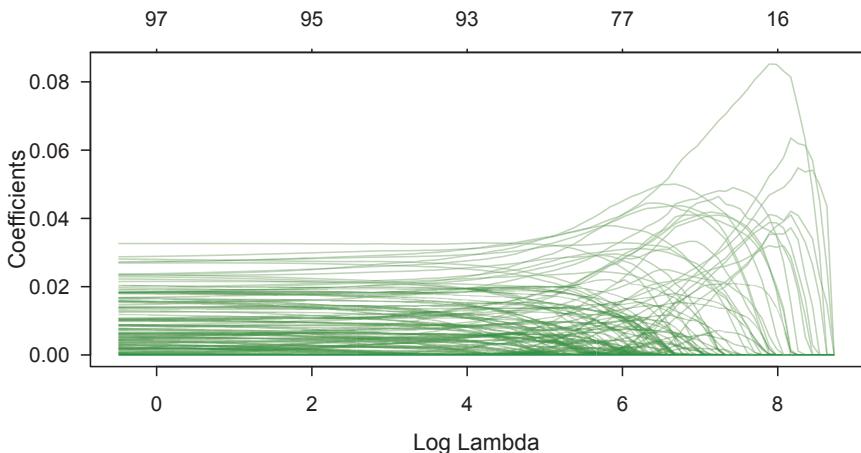


FIGURE 5.5: Profiles of LASSO coefficients for the `ames.bag.6.5` ensemble, as the regularization parameter  $\lambda$  is varied. The top axis indicates the number of non-zero coefficients at a particular value of  $\lambda$  (plotted on the log scale).

From here, we use cross-validation or an independent test set to choose a reasonable value of the penalty parameter  $\lambda$ . This can be done easily using `glmnet`'s `assess.glmnet()` function with the test set predictions using, as shown below (see `?glmnet::assess.glmnet` for details):

```
# Assess performance of fit using an independent test set
perf <- assess.glmnet(
  object = lasso.ames.bag, # fitted LASSO model
  newx = preds.tst, # test predictions from individual trees
  newy = ames.tst$Sale_Price, #same response variable (test set)
  family = "gaussian" # for MSE and MAE metrics
)
perf <- do.call(cbind, args = perf) # bind results into matrix
```

```

# List of results
ames.bag.post <- as.data.frame(cbind(
  "ntree" = lasso.ames.bag$df, perf,
  "lambda" = lasso.ames.bag$lambda)
)

# Sort in ascending order of number of trees
head(ames.bag.post <- ames.bag.post[order(ames.bag.post$ntree), ])

#>      ntree  mse   mae lambda
#> s0       0 6658 59.6   6164
#> s1       4 5672 55.0   5616
#> s2       5 4851 50.7   5117
#> s3       8 4163 46.8   4663
#> s4       9 3592 43.3   4248
#> s5      10 3114 40.1   3871

# Print results corresponding to smallest test MSE
ames.bag.post[which.min(ames.bag.post$mse), ]

#>      ntree  mse   mae lambda
#> s93     97 612 15.9   1.08

```

According to the test MSE, the optimal value of the penalty parameter  $\lambda$  is 1.077, which corresponds to 97 trees or non-zero coefficients in the LASSO model (an appreciable reduction from the original 500).

In the next code chunk, I'll follow the exact same process with the `ames.bag.6.5` ensemble, but using the `isle_post()` function instead:

```

library(treemisc)

# Post-process ames.bag.6.5 ensemble
preds.trn.6.5 <- predict(ames.bag.6.5, newdata = ames.trn,
                           predict.all = TRUE)$individual
preds.tst.6.5 <- predict(ames.bag.6.5, newdata = ames.tst,
                           predict.all = TRUE)$individual
ames.bag.6.5.post <-
  isle_post(preds.trn.6.5, y = ames.trn$Sale_Price,
            family = "gaussian", newX = preds.tst.6.5,
            newy = ames.tst$Sale_Price)

```

The overall results from each ensemble are shown in [Figure 5.6](#). Here, I show the MSE as a function of the number of trees from each model (or non-zero coefficients in the LASSO). In this example, the simpler `ames.bag.6.5` ensemble benefits substantially from post-processing and appears to perform on par with the ordinary bagged tree ensemble (`ames.bag`) in terms of MSE, while requiring only a small fraction of trees and being orders of magnitude faster to train! The original ensemble (`ames.bag`) did not see nearly as much improvement from post-processing.

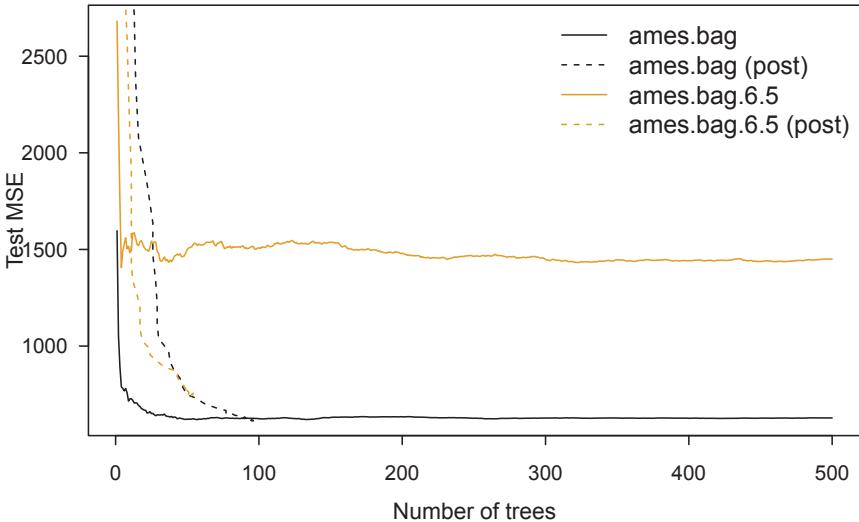


FIGURE 5.6: MSE for the test data from several bagged tree ensembles. The dashed lines correspond to the LASSO-based post-processed versions. Clearly, the `ames.bag.6.5` ensemble benefits the most from post-processing, performing nearly on par with the standard bagged tree ensemble (`ames.bag`).

## 5.6 Final thoughts

Loh [2014] compared the accuracy of single decision trees to tree ensembles using both real and simulated data sets. He found that, on average, the best single-tree algorithm was about 10% less accurate than that of a tree ensemble. Nonetheless, tree ensembles will not always outperform a simpler individual tree [Loh, 2009]. These points aside, tree ensembles are a powerful class of models that are highly competitive in terms of state-of-the-art prediction accuracy. Chapters 7–8 are devoted to two powerful tree ensemble techniques.

It is also worth pointing out that while tree-based ensembles often out perform carefully tuned individual trees (like CART, CTree, and GUIDE), they are less interpretable compared to a single decision tree; hence, they are often referred to as black box models. Fortunately, post-hoc procedures exist that can help us peek into the black box to understand the relationships uncovered by the model and explain their output to others. This is the topic of [Chapter 6](#).

# 6

---

## Peeking inside the “black box”: post-hoc interpretability

---

The lack of understanding does not hurt, as much as the lack of effort to understand does!

Wordions

---

This chapter is dedicated to select topics from the increasingly popular field of *interpretable machine learning* (IML), which easily deserves its own book-length treatment, and it has; see, for example, Molnar [2019] and Biecek and Burzykowski [2021] (both of which are freely available online). The methods covered in this chapter can be categorized into whether they help interpret a black box model at a global or local (e.g., individual row or prediction) level.

To be honest, I don’t really like the term “black box,” especially when we now have access to a rich ecosystem of interpretability tools. For example, linear regression models are often hailed as interpretable models. Sure, but this is really only true when the model has a simple form. Once you start including transformations and interaction effects—which are often required to boost accuracy and meet assumptions—the coefficients become much less interpretable.

Tree-based ensembles, especially the ones discussed in the next two chapters, can provide state-of-the-art performance, and are quite competitive with other popular supervised learning algorithms, especially on tabular data sets. Even when tree-based ensembles perform as advertised, there’s a price to be paid in terms of parsimony, as we lose the ability to summarize the model using a simple tree diagram. Luckily, there exist a number of post-hoc techniques that allow us to tease the same information out of an ensemble of trees that we would ordinarily be able to glean from looking at a simple tree diagram (e.g., which variables seem to be the most important, the effect of each predictor,

and potential interactions). Note that the techniques discussed in this chapter are *model-agnostic*, meaning they can be applied to any type of supervised learning algorithm, not just tree-based ensembles. For example, they can also be used to help interpret neural networks or a more complicated tree structure that uses linear splits or non-constant models in the terminal nodes.

The next three sections cover post-hoc methods to help comprehend various aspects of any fitted model:

- feature importance ([Section 6.1](#));
  - feature effects ([Section 6.2](#));
  - feature contributions ([Section 6.3](#)).
- 

## 6.1 Feature importance

For the purposes of this chapter, we can think of variable importance (VI) as the extent to which a feature has a “meaningful” impact on the predicted outcome. A more formal definition and treatment can be found in van der Laan [2006]. Given that point of view, a natural way to assess the impact of an arbitrary feature  $x_j$  is to remove it from the training data and examine the drop in performance that occurs after refitting the model without it. This procedure is referred to as *leave-one-covariate-out* (LOCO) importance; see Hooker et al. [2019] and the references therein.

Obviously, the LOCO importance method is computationally prohibitive for larger data sets and complex fitting procedures because it requires retraining the model once more for each dropped feature. In the next section, I’ll discuss an approximate approach based on reassessing performance after randomly permuting each feature (one at a time). This procedure is referred to as *permutation importance*.

### 6.1.1 Permutation importance

While some algorithms, like tree-based models, have a natural way of quantifying the importance of each predictor, it is useful to have a model-agnostic procedure that can be used for any type of supervised learning algorithm. This also makes it possible to directly compare the importance of features across different types of models. In this section, I’ll discuss a popular method for measuring the importance of predictors in any supervised learning model called permutation importance.

Permutation-based VI scores exist in various forms and was made popular in Breiman [2001] for random forests (Chapter 7), before being generalized and extended in Fisher et al. [2018]. A general permutation-based VI procedure is outlined in Algorithm 6.1 below. The idea is that if we randomly permute the values of an important feature in the training data, the training performance would degrade (since permuting the values of a feature effectively destroys any relationship between that feature and the target variable). This of course assumes that the model has been properly tuned (e.g., using cross-validation) and is not overfitting.

The permutation approach uses the difference between some baseline performance measure (e.g.,  $R^2$  or RMSE for regression, Brier score or log loss for probability estimation, and AUC for discrimination) and the same performance measure obtained after permuting the values of a particular feature in the training data (Note that the model is NOT refit to the training data after randomly permuting the values of a feature). It is also important to note that this method may not be appropriate when you have, for example, highly correlated features (since permuting one feature at a time may lead to unlikely or unrealistic data instances).

---

**Algorithm 6.1** General steps for constructing permutation-based VI scores for any type of supervised learning model.

---

Let  $x_1, x_2, \dots, x_p$  be the features of interest and let  $\mathcal{M}_{orig}$  be the baseline performance metric for the trained model; for brevity, I'll assume smaller is better (e.g., classification error or RMSE). The permutation-based importance scores can be computed as follows:

- 1) For  $i = 1, 2, \dots, p$ :
    - (a) Permute the values of feature  $x_i$  in the training data.
    - (b) Recompute the performance metric on the permuted data, denoted  $\mathcal{M}_{perm}$ .
    - (c) Record the difference from baseline using  $VI(x_i) = \mathcal{M}_{perm} - \mathcal{M}_{orig}$ .
  - 2) Return the VI scores  $VI(x_1), VI(x_2), \dots, VI(x_j)$ .
- 

Algorithm 6.1 can be improved or modified in a number of ways. For instance, the process can (and should) be repeated several times and the results averaged together. This helps to provide more stable VI scores, and also the opportunity to measure their variability. Rather than taking the difference in step (c), Molnar [2019, sec. 5.5.4] argues that using the ratio  $\mathcal{M}_{perm}/\mathcal{M}_{orig}$  makes the importance scores more comparable across different problems. It's also possible to assign importance scores to groups of features (e.g., by

permuting more than one feature at a time); this would be useful if features can be categorized into mutually exclusive groups, for instance, categorical features that have been one-hot encoded.

### 6.1.2 Software

The permutation approach to variable importance is implemented in several R packages, including: **vip** [Greenwell et al., 2021a], **iml** [Molnar and Schratz, 2020], **ingredients** [Biecek and Baniecki, 2021], and **mmpf** [Jones, 2018]. Further details, some comparisons, and an in-depth explanation of **vip** are provided in Greenwell and Boehmke [2020]. Starting with version 0.22.0, scikit-learn’s **inspection** module provides an implementation of permutation importance for any fitted model.

### 6.1.3 Example: predicting home prices

To illustrate the basic steps, let’s compute permutation importance scores for the Ames housing bagged tree ensemble (**ames.bag**) from [Section 5.5.1](#). I’ll start by writing a simple function to compute the RMSE, the performance metric of interest, and use it to obtain a baseline value for computing the permutation-based importance scores.

```
rmse <- function(predicted, actual, na.rm = TRUE) {
  sqrt(mean((predicted - actual) ^ 2, na.rm = na.rm))
}
(baseline.rmse <- rmse(predict(ames.bag, newdata = ames.trn),
                        actual = ames.trn$Sale_Price))

#> [1] 10.6
```

To get more stable VI scores, I’ll use 30 independent permutations for each predictor; since the permutations are done independently, Algorithm 6.1 can be trivially parallelized across repetitions or features. This is done using a nested **for** loop in the next code chunk:

```
nperm <- 30 # number of permutation to use per feature
xnames <- names(subset(ames.trn, select = -Sale_Price))
vi <- matrix(nrow = nperm, ncol = length(xnames))
colnames(vi) <- xnames
for (j in colnames(vi)) {
  for (i in seq_len(nrow(vi))) {
    temp <- ames.trn # temporary copy of training data
    temp[[j]] <- sample(temp[[j]]) # permute feature values
    pred <- predict(ames.bag, newdata = temp) # score permuted data
    permuted.rmse <- rmse(pred, actual = temp$Sale_Price) ^ 2
    vi[i, j] <- permuted.rmse - baseline.rmse # smaller is better
```

```

}

}

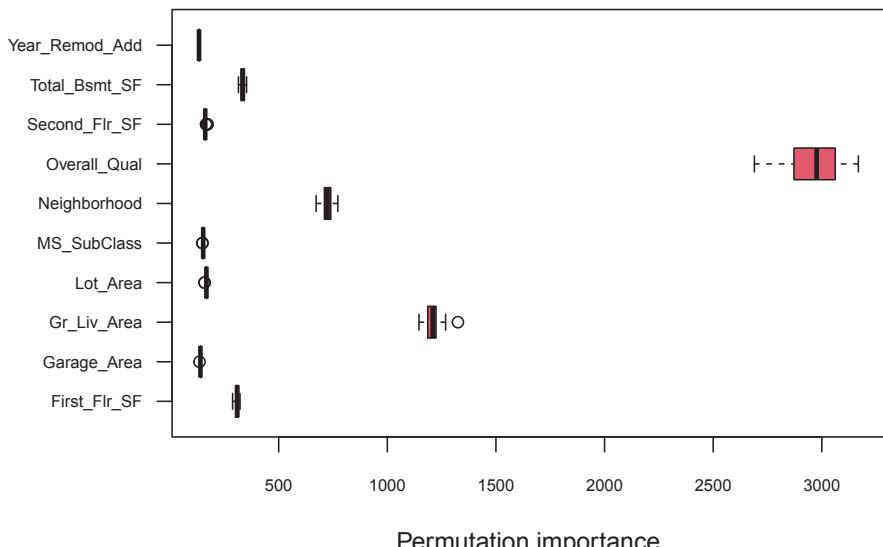
# Average VI scores across all permutations
head(vi.avg <- sort(colMeans(vi), decreasing = TRUE))

#> Overall_Qual    Gr_Liv_Area Neighborhood
#>      2959          1211        725
#> Total_Bsmt_SF  First_Flr_SF Lot_Area
#>      334           308         167

```

Note that the individual permutation importance scores are computed independently of each other, making it relatively straightforward to parallelize the whole procedure; in fact, many R implementations of Algorithm 6.1, like **vip** and **iml**, have options to do this in parallel using a number of different parallel backends.

A boxplot of the unaggregated permutation scores for the top ten features, as measured by the average across all 30 permutations, is displayed in [Figure 6.1](#). Here, you can see that the overall quality rating of the home and its above grade square footage are two of the most important predictors of sale price, followed by neighborhood. A simple dotchart of the average permutation scores would suffice, but fails to show the variability in the individual VI scores.



**FIGURE 6.1:** Permutation-based VI scores for the top ten features in the Ames housing bagged tree ensemble, as measured by the average across all 30 permutations.

Although permutation importance is most naturally computed on the training data, it may also be useful to do the shuffling and measure performance on new data. This is discussed in depth in Molnar [2019, sec. 5.2]. Next, I’ll discuss a general technique for interpreting the effect of an individual feature (main effect) or combination of features (interaction effect).

---

## 6.2 Feature effects

While determining predictor importance is a crucial task in any supervised learning problem, ranking variables is only part of the story, and once a subset of “important” features is identified it is often necessary to assess the relationship between them (or subset thereof) and the response. This can be done in many ways, but in practice it is often accomplished by constructing plots of *partial dependence* or *individual conditional expectation*; see Friedman [2001] and [Goldstein et al., 2015], respectively, for details.

### 6.2.1 Partial dependence

Partial dependence (PD) plots (or PDPs) help visualize the relationship between a subset of the features (typically 1–3) and the response while accounting for the average effect of the other predictors in the model. They are particularly effective with black box models like random forests, support vector machines, and neural networks.

Let  $\mathbf{x} = \{x_1, x_2, \dots, x_p\}$  represent the predictors in a model whose prediction function is  $\hat{f}(\mathbf{x})$ . If we partition  $\mathbf{x}$  into an interest set,  $\mathbf{z}_s$ , and its compliment,  $\mathbf{z}_c = \mathbf{x} \setminus \mathbf{z}_s$ , then the “partial dependence” of the response on  $\mathbf{z}_s$  is defined as

$$f_s(\mathbf{z}_s) = E_{\mathbf{z}_c} \left[ \hat{f}(\mathbf{z}_s, \mathbf{z}_c) \right] = \int \hat{f}(\mathbf{z}_s, \mathbf{z}_c) p_c(\mathbf{z}_c) d\mathbf{z}_c, \quad (6.1)$$

where  $p_c(\mathbf{z}_c)$  is the marginal probability density of  $\mathbf{z}_c$ :  $p_c(\mathbf{z}_c) = \int p(\mathbf{x}) d\mathbf{z}_c$ . Equation (6.1) can be estimated from a set of training data by

$$\bar{f}_s(\mathbf{z}_s) = \frac{1}{n} \sum_{i=1}^n \hat{f}(\mathbf{z}_s, \mathbf{z}_{i,c}), \quad (6.2)$$

where  $\{\mathbf{z}_{i,c}\}_{i=1}^N$  are the values of  $\mathbf{z}_c$  that occur in the training sample; that is, we average out the effects of all the other predictors in the model.

Mathematical gibberish aside, computing partial dependence (6.2) in practice is rather straightforward. To simplify, let  $\mathbf{z}_s = x_1$  be the predictor variable of interest with unique values  $\{x_{1i}\}_{i=1}^k$ . The partial dependence of the response on  $x_1$  can be constructed by following the basic steps outlined in 6.2.

---

**Algorithm 6.2** A simple algorithm for constructing the partial dependence of the response on a single predictor  $x_1$ .

---

- 1) For  $i \in \{1, 2, \dots, k\}$ :
    - (a) Copy the training data and replace the original values of  $x_1$  with the constant  $x_{1i}$ .
    - (b) Compute the vector of predicted values from the modified copy of the training data.
    - (c) Compute the average prediction to obtain  $\bar{f}_1(x_{1i})$ .
  - 2) Plot the pairs  $\{x_{1i}, \bar{f}_1(x_{1i})\}_{i=1}^k$ .
- 

Algorithm 6.2 can be quite computationally intensive since it involves  $k$  passes over the training records. Fortunately, the algorithm is trivial to parallelize. It can also be easily extended to larger subsets of two or more features as well. See Greenwell [2017] for additional details and examples.

### 6.2.1.1 Classification problems

Traditionally, for classification problems, partial dependence functions are on a scale similar to the logit; see, for example, Hastie et al. [2009, pp. 369—370]. Suppose the response is categorical with  $J$  levels; then for each class we compute

$$f_j(x) = \log[p_j(x)] - \frac{1}{J} \sum_{j=1}^J \log[p_j(x)], \quad j = 1, 2, \dots, J, \quad (6.3)$$

where  $p_j(x)$  is the predicted probability for the  $j$ -th class. Plotting  $f_j(x)$  helps us understand how the log-odds for the  $j$ -th class depends on different subsets of the predictor variables. Nonetheless, there's no reason partial dependence can't be displayed on the raw probability scale. The same goes for ICE plots (Section 6.2.3). A multiclass classification example of PD plots on the probability scale is given in Section 6.2.6.

### 6.2.2 Interaction effects

While partial dependence can be used to help visualize potential interaction effects, it is often desirable to know where to look in the first place. To that end, Friedman and Popescu [2008] proposed a model-agnostic method, called the  $H$ -statistic, for identifying predictors that are involved in interactions with other variables, the strength of those interactions, as well as the identities of the other variables with which they interact.

For example, to test for the presence of an interaction effect between predictors  $x_j$  and  $x_k$ , we can use the statistic

$$H_{jk}^2 = \sum_{i=1}^N [\bar{f}_{jk}(x_{ij}, x_{ik}) - \bar{f}_j(x_{ij}) - \bar{f}_k(x_{ik})]^2 / \sum_{i=1}^N \bar{f}_{jk}(x_{ij}, x_{ik}). \quad (6.4)$$

In essence, (6.4) measures the fraction of variance of  $\bar{f}_{jk}(x_j, x_k)$ —the joint partial dependence of  $y$  on  $x_j$  and  $x_k$ —not captured by  $f_j(x_j)$  and  $f_k(x_k)$  (the individual partial dependence of  $y$  on  $x_j$  and  $x_k$ , respectively) over the training data (or representative sample thereof). Note that  $H_{jk}^2 \geq 0$ , with zero indicating no interaction between  $x_j$  and  $x_k$ . To determine whether a single predictor,  $x_j$ , say, interacts with any other variables, a similar  $H$ -statistic can be computed. Unfortunately, these statistics are not widely implemented; the R **gbm** package [Greenwell et al., 2021b], probably has the most efficient implementation (see `?gbm:::interact.gbm` for details), but it's only available for GBMs (Chapter 8).

According to Friedman and Popescu [2008], only predictors with strong main effects (e.g., high relative importance) should be examined for potential interactions; the strongest interactions can then be further explored via two-way PD plots. Be warned, however, that collinearity among predictors can lead to spurious interactions that are not present in the target function.

A major drawback of the  $H$ -statistic (6.4) is that it requires computing both the individual and joint partial dependence functions, which can be expensive; the fast recursion method of Section 8.6.1 makes it feasible to compute the  $H$ -statistic for binary decision trees (and ensembles of shallow trees). A simpler approach, based on just the joint partial dependence function, is discussed in Greenwell et al. [2018].

### 6.2.3 Individual conditional expectations

PD plots can be misleading in the presence of strong interaction effects [Goldstein et al., 2015] (akin to interpreting a main effect in a linear model that's

also involved in an interaction term). To overcome this issue, Goldstein, Kapelner, Bleich, and Pitkin developed the concept of individual conditional expectation (ICE) plots. ICE plots display the estimated relationship between the response and a predictor of interest for each observation. Consequently, the PD plot for a predictor of interest can be obtained by averaging the corresponding ICE plots across all observations.

As described in [Goldstein et al., 2015], when the individual curves have a wide range of intercepts and consequently overlay each other, heterogeneity in the model can be difficult to discern. For that reason, Goldstein, Kapelner, Bleich, and Pitkin suggest centering the ICE plots to produce a centered ICE plot (or c-ICE plot for short). They also suggest other modifications, like derivative ICE plots (or d-ICE plots), to further explore the presence of interaction effects. Centered ICE plots are obtained after shifting the ICE curves up or down by subtracting off the first value from each curve, effectively pinching them together at the beginning.

#### 6.2.4 Software

PD plots and ICE plots (and many variants thereof) are implemented in several R packages. Historically, PD plots were only implemented in specific tree-based ensemble packages, like **randomForest** [Breiman et al., 2018] and **gbm**. However, they were made generally available in package **pdp**, which was soon followed by **iml** and **ingredients**, among others; these packages also support ICE plots; the R package **ICEbox** [Goldstein et al., 2017] provides the original implementation of ICE plots and several variants thereof, like c-ICE and d-ICE plots. PD plots and ICE plots were also made available in scikit-learn's **inspection** module, starting with versions 0.22.0 and 0.24.0, respectively.

#### 6.2.5 Example: predicting home prices

Using the Ames housing bagged tree ensemble, I'll show how to construct PD plots and ICE curves by hand and using the **pdp** package. To start, let's construct a PD plot for above grade square footage (**Gr\_Liv\_Area**), one of the top predictors according to permutation-based VI scores from [Figure 6.1](#) (p. 207).

The first step is to create a grid of points over which to construct the plot. For continuous variables, it is sufficient to use a fine enough grid of percentiles, as is done in the example below. Then, I simply loop through each grid point and 1) copy the training data, 2) replace all the values of **Gr\_Liv\_Area** in the copy with the current grid value, and 3) score the modified copy of the training data and average the predictions together. Lastly, I simply plot the grid points

against the averaged predictions obtained from the `for` loop. The results are displayed in Figure 6.2 and show a relatively monotonic increasing relationship between above grade square footage and predicted sale price.

```
x.grid <- quantile(ames.trn$Gr_Liv_Area, prob = 1:30 / 31)
pd <- numeric(length(x.grid))
for (i in seq_along(x.grid)) {
  temp <- ames.trn # temporary copy of data
  temp[["Gr_Liv_Area"]] <- x.grid[i]
  pd[i] <- mean(predict(ames.bag, newdata = temp))
}

# PD plot for above grade square footage (Figure 6.2)
plot(x.grid, pd, type = "l", xlab = "Above ground square footage",
      ylab = "Partial dependence", las = 1)
rug(x.grid) # add rug plot to x-axis
```

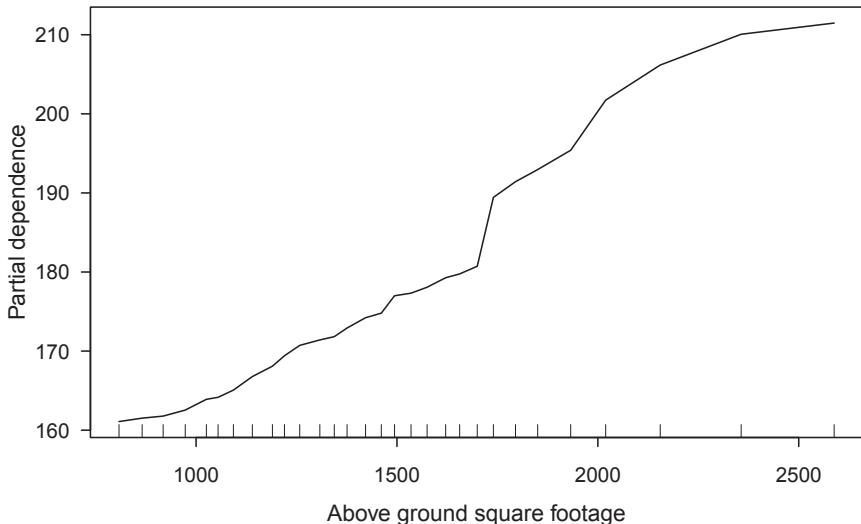


FIGURE 6.2: Partial dependence of sale price on above grade square footage for the bagged tree ensemble.

While looking at the partial dependence of the response on a single feature (i.e., main effect) is informative, it is often useful to look at the dependence on two or three predictors simultaneously<sup>a</sup> (i.e., interaction effects). Fortunately, it is rather straightforward to modify the above `for` loop to accommodate multiple predictors; however, this is a good opportunity to show an alternative

<sup>a</sup>Theoretically, we can look at any order of interaction effect of interest; however, the computational complexity usually prohibits going beyond just two- or three-way interactions.

method to computing partial dependence using simple *data wrangling* operations, which can be more efficient if you're working in SQL or Spark.

In essence, we can generate all of the modified training copies in a single stacked data frame using a *cross-join*, score it once, then aggregate the predictions into the partial dependence values. Below is an example using base R, but it should be rather straightforward to translate to **data.table** [Dowle and Srinivasan, 2021], **dplyr** [Wickham et al., 2021b], **SparkR** [Venkataraman et al., 2016], **sparklyr** [Luraschi et al., 2021], or any other language that can perform simple cross-joins—provided you can hold the resulting Cartesian product in memory! (An example of how to accomplish this in Spark is provided in [Section 7.9.5](#)).

To illustrate, I'll construct the partial dependence of sale price on both above grade and first floor square footage. I still need to construct a grid of points over which to construct the plot; here, I'll use a Cartesian product between the percentiles of each feature using `expand.grid()`.

```
x1.grid <- quantile(ames.trn$Gr_Liv_Area, prob = 1:30 / 31)
x2.grid <- quantile(ames.trn$First_Flr_SF, prob = 1:30 / 31)
df1 <- expand.grid("Gr_Liv_Area" = x1.grid,
                    "First_Flr_SF" = x2.grid) # Cartesian product
```

In the next step, I perform a cross-join between the grid of plotting values (`df1`) and the original training data with the plotting features removed (`df2`):<sup>b</sup>:

```
df2 <- subset(ames.trn, select = -c(Gr_Liv_Area, First_Flr_SF))
```

```
# Perform a cross-join between the two data sets
pd <- merge(df1, df2, all = TRUE) # Cartesian product
dim(pd) # print dimensions
```

```
#> [1] 1845900      81
```

Then, I simply score the data and aggregate by computing the average prediction within each grid point, as shown in the example below:

```
pd$yhat <- predict(ames.bag, newdata = pd) # might take a few minutes!
pd <- aggregate(yhat ~ Gr_Liv_Area + First_Flr_SF, data = pd,
                  FUN = mean)
```

The code snippet below constructs a *false color level plot* of the data with contour lines using the built-in **lattice** package; the results are displayed in [Figure 6.3](#). Here, you can see the joint effect of both features on the predicted sale price.

---

<sup>b</sup>BE CAREFUL as the resulting data set, which is a Cartesian product, can be quite large!

```
library(lattice)

# PD plot for above grade and first floor square footage
levelplot(yhat ~ Gr_Liv_Area * First_Flr_SF, data = pd,
           contour = TRUE, col = "white", scales = list(tck = c(1, 0)),
           col.regions = hcl.colors(100, palette = "viridis"))
```

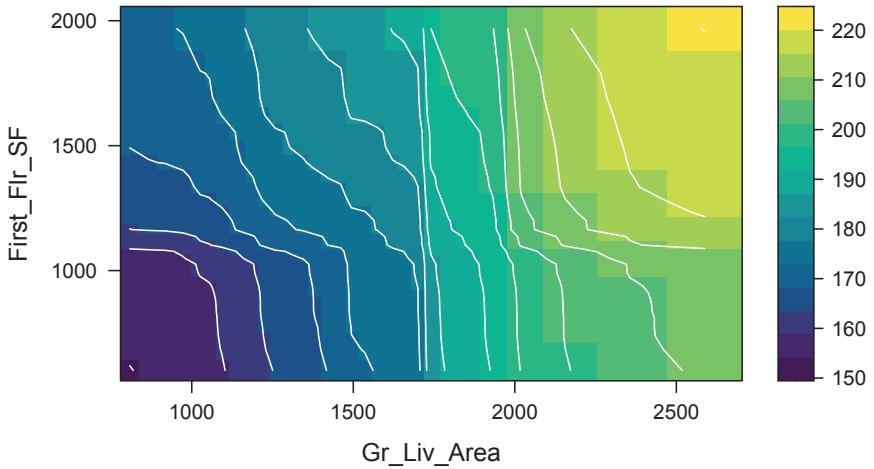


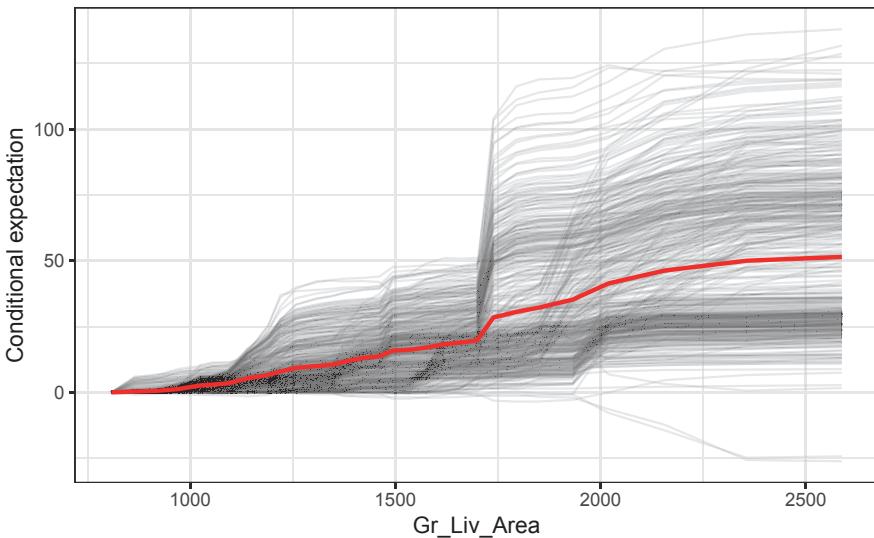
FIGURE 6.3: Partial dependence of sale price on above grade and first floor square footage for the bagged tree ensemble.

It is not wise to draw conclusions from PD plots (and ICE plots) in regions outside the area of the training data. Greenwell [2017] describes two ways to mitigate the risk of extrapolation in PD plots: rug displays, like the one I used in [Figure 6.2](#), and *convex hulls* (which can be used with bivariate displays, like in [Figure 6.3](#)).

Constructing ICE curves is just as easy; just skip the aggregation step and plot each of the individual curves. In the example below, I’ll use the **pdp** package to construct c-ICE curves showing the partial dependence of above grade square footage on sale price for each observation in the learning sample. There’s no need to construct a curve for each sample, especially when you have thousands (or more) data points; here, I’ll just plot a random sample of 500 curves. I’ll use the same percentiles to construct the plot as I did for the PD plot in [Figure 6.2](#) (p. 212) by invoking the **quantiles** and **probs** arguments in the call to **partial()**; note that **partial()**’s default is to use an evenly spaced grid of points across the range of predictor values.

The results are displayed in [Figure 6.4](#); the red line shows the average c-ICE value at each above grade square footage (i.e., the centered partial dependence). The heterogeneity in the c-ICE curves indicates a potential interaction effect between `Gr_Liv_Area` and at least one other feature. The c-ICE curves also indicate a relatively monotonic increasing relationship for the majority of houses in the training set, but you can see a few of the curves at the bottom deviate from this overall pattern.

```
ice <- partial(ames.bag, pred.var = "Gr_Liv_Area", ice = TRUE,
               center = TRUE, quantiles = TRUE, probs = 1:30 / 31)
set.seed(1123) # for reproducibility
samp <- sample.int(nrow(ames.trn), size = 500) # sample 500 homes
autoplot(ice[ice$yhat.id %in% samp, ], alpha = 0.1) +
  ylab("Conditional expectation")
```



**FIGURE 6.4:** A random sample of 500 c-ICE curves for above grade square footage using the Ames housing bagged tree ensemble. The curves indicate a relatively monotonic increasing relationship for the majority of houses in the sample. The average of the 500 c-ICE curves is shown in red.

### 6.2.6 Example: Edgar Anderson's iris data

For a classification example, I'll consider Edgar Anderson's iris data from the `datasets` package in R. The `iris` data frame contains the sepal length, sepal width, petal length, and petal width (in centimeters) for 50 flowers from each of three species of iris: setosa, versicolor, and virginica. Below, I fit a bagged tree ensemble to the data using the `randomForest` package:

```
library(randomForest)

# Fit a bagged tree ensemble
set.seed(1452) # for reproducibility
(iris.bag <- randomForest(Species ~ ., data = iris, mtry = 4))

#>
#> Call:
#>   randomForest(formula = Species ~ ., data = iris, mtry = 4)
#>   Type of random forest: classification
#>   Number of trees: 500
#>   No. of variables tried at each split: 4
#>
#>       OOB estimate of error rate: 4.67%
#> Confusion matrix:
#>
#>             setosa versicolor virginica class.error
#> setosa      50          0          0     0.00
#> versicolor    0         47          3     0.06
#> virginica     0          4         46     0.08
```

Next, I plot the partial dependence of `Species` on `Petal.Width` for each of the three classes using the `pdp` package. The code chunk below exploits a simple trick to computing partial dependence with `partial()` for several classes simultaneously. The results are displayed in Figure 6.5. Here, you can clearly see the average effect petal width has on the probability of belonging to each species.

```
library(pdp)
library(ggplot2)

# Prediction wrapper that returns average prediction for each class
pfun <- function(object, newdata) {
  colMeans(predict(object, newdata = newdata, type = "prob"))
}

# Partial dependence of probability for each class on petal width
p <- partial(iris.bag, pred.var = "Petal.Width", pred.fun = pfun)
ggplot(p, aes(Petal.Width, yhat, color = as.factor(yhat.id))) +
  geom_line() +
  theme(legend.title = element_blank(),
        legend.position = "top")
```

Note that without the aid of a user-supplied prediction function (via the `pred.fun` argument), `pdp`'s `partial()` function can only compute partial dependence in regards to a single class; see Greenwell [2017] for more details on the use of this package.

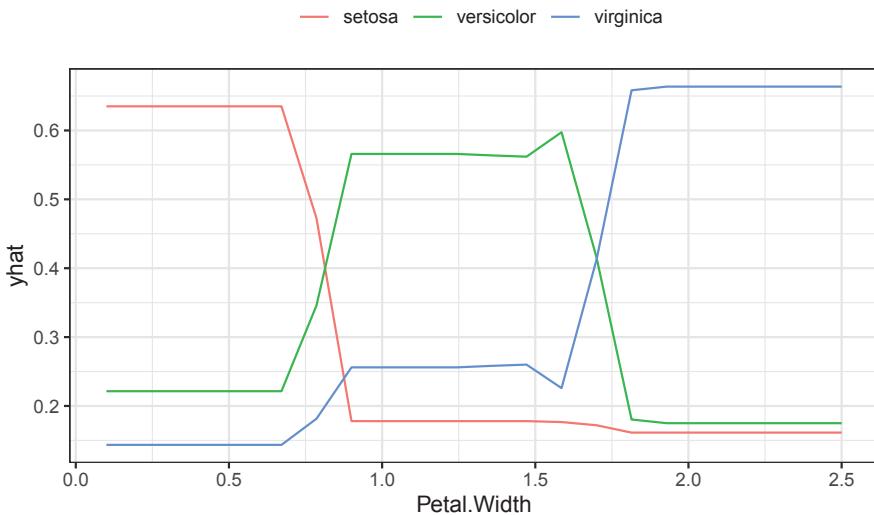


FIGURE 6.5: Partial dependence of species probability on petal width for each of the three iris species using a bagged tree ensemble.

## 6.3 Feature contributions

In general, a *local feature contribution* is the worth assigned to a feature's value that is proportional to the feature's share in the model's prediction for a particular observation. You can think of feature contributions as a directional variable importance at the individual prediction level. While there are a wide variety of feature contribution methodologies, the next section briefly covers one of the most popular methods in current use: *Shapley values* (or *Shapley explanations*). Shapley values necessarily involve a lot of mathematical notation, but I will try to avoid as much as possible while trying to convey the main concepts. For more details, start with Štrumbelj and Kononenko [2014] and Lundberg and Lee [2017].

### 6.3.1 Shapley values

The Shapley value [Shapley, 2016] is an idea from coalitional/cooperative game theory. In a coalitional game, assume there are  $p$  players that form a grand coalition ( $S$ ) worth a certain payout ( $\Delta_S$ ). Suppose it is also known how much any smaller coalition ( $Q \subseteq S$ ) (i.e., any subset of  $p$  players) is worth ( $\Delta_Q$ ). The goal is to distribute the total payout  $\Delta_S$  to the individual  $p$  players

in a “fair” way; that is, so that each player receives their “fair” share. The Shapley value is one such solution and the only one that uniquely satisfies a particular set of “fairness properties.”

Let  $v$  be a *characteristic function* that assigns a value to each subset of players; in particular,  $v : 2^p \rightarrow \mathbb{R}$ , where  $v(S) = \Delta_S$  and  $v(\emptyset) = 0$ , with  $\emptyset$  denoting the empty set (i.e., zero players). Let  $\phi_i(v)$  be the contribution (or portion of the total payout) attributed to player  $i$  in a particular game with total payout  $v(S) = \Delta_S$ . The Shapley value satisfies the following properties:

- efficiency:  $\sum_{i=1}^p \phi_i(v) = \Delta_S$ ;
- null player:  $\forall W \subseteq S \setminus \{i\} : \Delta_W = \Delta_{W \cup \{i\}} \implies \phi_i(v) = 0$ ;
- symmetry:  $\forall W \subseteq S \setminus \{i, j\} : \Delta_{W \cup \{i\}} = \Delta_{W \cup \{j\}} \implies \phi_i(v) = \phi_j(v)$ ;
- linearity: If  $v$  and  $w$  are functions describing two coalitional games, then  $\phi_i(v + w) = \phi_i(v) + \phi_i(w)$ .

The above properties can be interpreted as follows:

- the individual player contributions sum to the total payout, hence, are implicitly normalized;
- if a player does not contribute to the coalition, they receive a payout of zero;
- if two players have the same impact across all coalitions, they receive equal payout;
- the local contributions are additive across different games.

Shapley [2016] showed that the unique solution satisfying the above properties is given by

$$\phi_i(v) = \frac{1}{p!} \sum_{\mathcal{O} \in \pi(p)} [v(S^{\mathcal{O}} \cup i) - v(S^{\mathcal{O}})], \quad i = 1, 2, \dots, p, \quad (6.5)$$

where  $\mathcal{O}$  is a specific permutation of the player indices  $\{1, 2, \dots, p\}$ ,  $\pi(p)$  is the set of all such permutations of size  $p$ , and  $S^{\mathcal{O}}$  is the set of players joining the coalition before player  $i$ .

In other words, the Shapley value is the average marginal contribution of a player across all possible coalitions in a game. Another way to interpret (6.5) is as follows. Imagine the coalitions (subsets of players) being formed one player at a time (which can happen in different orders), with the  $i$ -th player demanding a fair contribution/payout of  $v(S^{\mathcal{O}} \cup i) - v(S^{\mathcal{O}})$ . The Shapley value for player  $i$  is given by the average of this contribution over all possible permutations in which the coalition can be formed.

A simple example may help clarify the main ideas. Suppose three friends (players)—Alex, Brad, and Brandon—decide to go out for drinks after work (the game). They shared a few pitchers of beer, but nobody paid attention to how much each person drank (collaborated). What’s a fair way to split the tab (total payout)?

Suppose we knew the following information, perhaps based on historical happy hours:

- if Alex drank alone, he’d only pay \$10;
- if Brad drank alone, he’d only pay \$20;
- if Brandon drank alone, he’d only pay \$10;
- if Alex and Brad drank together, they’d only pay \$25;
- if Alex and Brandon drank together, they’d only pay \$15;
- if Brad and Brandon drank together, they’d only pay \$13;
- if Alex, Brad, and Brandon drank together, they’d only pay \$30.

With only three players, we can enumerate all possible coalitions. In [Table 6.1](#), I list all possible permutations of the three players and list the marginal contribution of each. Take the first row, for example. In this particular permutation, we start with Alex. We know that if Alex drinks alone, he’d spend \$10, so his marginal contribution by entering first is \$10. Next, we assume Brad enters the coalition. We know that if Alex and Brad drank together, they’d pay a total of \$25, leaving \$15 left over for Brad’s marginal contribution. Similarly, if Brandon joins the party last, his marginal contribution would be only \$5 (the difference between \$30 and \$25). The Shapley value for each player is the average across all six possible permutations (these are the column averages reported in the last row). In this case, Brandon would get away with the smallest payout (i.e., have to pay the smallest portion of the total tab). The next time the bartender asks how you want to split the tab, whip out a pencil, and do the math!

### 6.3.2 Explaining predictions with Shapley values

From this section forward, let  $\{x_i\}_{i=1}^p$  represent the  $p$  feature values comprising  $\mathbf{x}^*$ , the observation whose prediction we want to try to explain. Štrumbelj and Kononenko [2014] suggested using the Shapley value (6.5) to help explain predictions from a supervised learning model. In the context of statistical and machine learning,

- a game is represented by the prediction task for a single observation  $\mathbf{x}^*$ ;

Permutation/order of players	Marginal contribution		
	Alex	Brad	Brandon
Alex, Brad, Brandon	\$10	\$15	\$5
Alex, Brandon, Brad	\$10	\$15	\$5
Brad, Alex, Brandon	\$5	\$20	\$5
Brad, Brandon, Alex	\$10	\$20	\$0
Brandon, Alex, Brad	\$5	\$15	\$10
Brandon, Brad, Alex	\$17	\$3	\$10
Shapley contribution:	\$9.50	\$14.67	\$5.83

TABLE 6.1: Marginal contribution for each permutation of the players/beer drinkers {Alex, Brad, Brandon} (i.e., the order in which they arrive). The Shapley contribution is the average marginal contribution across all permutations. (Notice how each row sums to the total bill of \$30.)

- the total payout/worth ( $\Delta_S$ ) for  $\mathbf{x}^*$  is the prediction for  $\mathbf{x}^*$  minus the average prediction for all training observations (the latter is referred to as the baseline and denoted  $\bar{f}$ ):  $\hat{f}(\mathbf{x}^*) - \bar{f}$ ;
- the players are the individual feature values of  $\mathbf{x}^*$  that collaborate to receive the payout  $\Delta_S$  (i.e., predict a certain value).

The second point, combined with the efficiency property stated in the previous section, implies that the  $p$  Shapley explanations (or feature contributions) for an observation of interest  $\mathbf{x}^*$ , denoted  $\{\phi_i(\mathbf{x}^*)\}_{i=1}^p$ , are inherently standardized since  $\sum_{j=1}^p \phi_j(\mathbf{x}^*) = \hat{f}(\mathbf{x}^*) - \bar{f}$ .

### 6.3.2.1 Tree SHAP

Several methods exist for estimating Shapley values in practice. The most common is arguably Tree SHAP [Lundberg et al., 2020], an efficient implementation of exact Shapley values for decision trees and ensembles thereof.

Tree SHAP is a fast and exact method to estimate Shapley values for tree-based models (including tree ensembles), under several different possible assumptions about feature dependence. The specifics of Tree SHAP are beyond the scope of this book, so I’ll defer to [Lundberg et al., 2020] for the details. It’s implemented in the Python **shap** module, and embedded in several tree-based modeling packages across several open source languages (like **xgboost** [Chen et al., 2021] and **lightgbm** [Shi et al., 2022]). While the details of Tree SHAP are beyond the scope of this book, we’ll see an example of it in action in [Section 8.9.4](#).

In the following section, I'll discuss a general way to estimate Shapley values for any supervised learning model using a simple *Monte Carlo* approach.

### 6.3.2.2 Monte Carlo-based Shapley explanations

Except in special circumstances, like Tree SHAP, computing the exact Shapley value is computationally infeasible in most applications. To that end, Štrumbelj and Kononenko [2014] suggest a Monte Carlo approximation, which I'll call Sample SHAP for short, that assumes independent features<sup>c</sup>. Their approach is described in Algorithm 6.3 below.

Here, a single estimate of the contribution of feature  $x_i$  to  $f(\mathbf{x}^*) - \bar{f}$  is nothing more than the difference between two predictions, where each prediction is based on a set of “Frankenstein instances”<sup>d</sup> that are constructed by swapping out values between the instance being explained ( $\mathbf{x}^*$ ) and an instance selected at random from the training data ( $\mathbf{w}^*$ ). To help stabilize the results, the procedure is repeated a large number, say,  $R$ , times, and the results averaged together:

---

**Algorithm 6.3** Approximating the  $i$ -th feature's contribution to  $\hat{f}(\mathbf{x}^*)$  for some instance with predictor values  $\mathbf{x}^* = (x_1, x_2, \dots, x_p)$ .

---

- 1) For  $j = 1, 2, \dots, R$ :
    - (a) Select a random permutation  $\mathcal{O}$  of the sequence  $1, 2, \dots, p$ .
    - (b) Select a random instance  $\mathbf{w}$  from the set of training observations  $\mathbf{X}$ .
    - (c) Construct two new instances as follows:
      - $\mathbf{b}_1 = \mathbf{x}^*$ , but all the features in  $\mathcal{O}$  that appear after feature  $x_i$  get their values swapped with the corresponding values in  $\mathbf{w}$ .
      - $\mathbf{b}_2 = \mathbf{x}^*$ , but feature  $x_i$ , as well as all the features in  $\mathcal{O}$  that appear after  $x_i$ , get their values swapped with the corresponding values in  $\mathbf{w}$ .
    - (d)  $\phi_{ij}(\mathbf{x}^*) = f(\mathbf{b}_1) - f(\mathbf{b}_2)$ .
  - 2)  $\phi_i(\mathbf{x}^*) = \frac{1}{R} \sum_{j=1}^R \phi_{ij}(\mathbf{x}^*)$ .
- 

<sup>c</sup>While Sample SHAP, along with many other common Shapley value procedures, assumes independent features, several arguments can be made in favor of this assumption; see, for example, Chen et al. [2020] and the references therein.

<sup>d</sup>The terminology used here takes inspiration from Molnar [2019, p. 231].

A simple R implementation of Algorithm 6.3 is given below. Here, `obj` is a fitted model with scoring function `f` (e.g., `predict()`), `nsim` is the number of Monte Carlo repetitions to perform, `feature` gives the name of the corresponding feature in `x` to be explained, and `X` is the training set of features.

```
sample.shap <- function(f, obj, R, x, feature, X) {
  phi <- numeric(R) # to store Shapley values
  N <- nrow(X) # sample size
  p <- ncol(X) # number of features
  b1 <- b2 <- x
  for (m in seq_len(R)) {
    w <- X[sample(N, size = 1), ]
    ord <- sample(names(w)) # random permutation of features
    swap <- ord[seq_len(which(ord == feature) - 1)]
    b1[swap] <- w[swap]
    b2[c(swap, feature)] <- w[c(swap, feature)]
    phi[m] <- f(obj, newdata = b1) - f(obj, newdata = b2)
  }
  mean(phi) # return approximate feature contribution
}
```

To illustrate, let’s continue with the Ames housing example (`ames.bag`). Below, I use the `sample.shap()` function to estimate the contribution of the value of `Gr_Liv_Area` to the prediction of the first observation in the learning sample (`ames.trn`):

```
X <- subset(ames.trn, select = -Sale_Price) # features only
set.seed(2207) # for reproducibility
sample.shap(predict, obj = ames.bag, R = 100, x = X[1, ],
            feature = "Gr_Liv_Area", X = X)

#> [1] -6.7
```

So, having  $\text{Gr\_Liv\_Area} = 1474$  helped push the predicted sale price down toward the baseline average; in this case, the baseline average is just the average predicted sale price across the entire training set:  $\bar{f} = \$181.53$  (don’t forget that I rescaled the response in this example).

If there are  $p$  features and  $m$  instances to be explained, this requires  $2 \times R \times p \times m$  predictions (or calls to the scoring function  $f$ ). In practice, this can be quite computationally demanding, especially since  $R$  needs to be large enough to produce good approximations to each  $\phi_i(\mathbf{x}^*)$ . How large does  $R$  need to be to produce accurate explanations? It depends on the variance of each feature in the observed training data, but typically  $R \in [30, 100]$  will suffice. The R package `fastshap` [Greenwell, 2021a] provides an optimized implementation of Algorithm 6.3 that only requires  $2mp$  calls to  $f$ ; see the package documentation for details.

Sample SHAP can be computationally prohibitive if you need to explain large data sets (optimized or not). Fortunately, you often only need to explain a

handful of predictions, the most extreme ones, for example. However, generating explanations for the entire training set, or a large enough sample thereof, can be useful for generating aggregated global model summaries. For example, Shapley-based dependence plots [Lundberg et al., 2020] show how a feature's value impacts the prediction of every observation in a data set of interest.

### 6.3.3 Software

Various flavors of Shapley values are starting to become widely available in R. Implementations of Sample SHAP, for example, are provided in **fastshap**, **iml**, and **iBreakDown** [Biecek et al., 2021]. Maksymiuk et al. [2021] discuss several others.

The **shap** module in Python is arguably one of the first and most well known implementations of Shapley values for statistical and machine learning. It offers several different flavors of Shapley explanations, including Tree SHAP, Kernel SHAP [Lundberg and Lee, 2017], Sample SHAP, and many more specific to different applications, like *deep learning*.

### 6.3.4 Example: predicting home prices

In the example below, I'll use **fastshap** to estimate feature contributions for the record in the test set with the highest predicted sale price using  $R = 100$  Monte Carlo repetitions. Note that **fastshap**'s `explain()` function includes an adjustment argument to ensure the efficiency property; see <https://github.com/bgreenwell/fastshap/issues/6> for details.

As with **pdp**, **fastshap** defines its own `autoplot()` method for automatically producing various **ggplot2**-based Shapley plots; IMO it's far better (and more flexible) to manually produce your own plots from the raw output. In the code chunk below, I use `explain()` and `autoplot()` to produce a bar plot of the feature contributions for the training observation with the highest predicted sale price<sup>e</sup>:

```
library(fastshap)
library(ggplot2)

# Find observation with highest predicted sale price
pred <- predict(ames.bag, newdata = ames.tst)
highest <- which.max(pred)
```

---

<sup>e</sup>An alternative way to visualize individual feature contributions using a *waterfall chart* is given in [Section 8.9.1](#).

```

pred[highest]

#> 433
#> 503

# fastshap needs to know how to compute predictions from your model
pfun <- function(object, newdata) predict(object, newdata = newdata)

# Need to supply feature columns only in fastshap::explain()
X <- subset(ames.trn, select = -Sale_Price) # feature columns only
newx <- ames.tst[highest, names(X)]

# Compute feature contributions for observation with highest prediction
set.seed(1434) # for reproducibility
ex <- explain(ames.bag, X = X, nsim = 100, newdata = newx,
               pred_wrapper = pfun, adjust = TRUE)
ex[1, 1:5] # peek at a few

#> # A tibble: 1 x 5
#>   MS_SubClass MS_Zoning Lot_Frontage Lot_Area Street
#>   <dbl>        <dbl>       <dbl>      <dbl>    <dbl>
#> 1     0.930     0.0275     0.472      3.80     0

autoplot(ex, type = "contribution", num_features = 10,
         feature_values = newx)

```

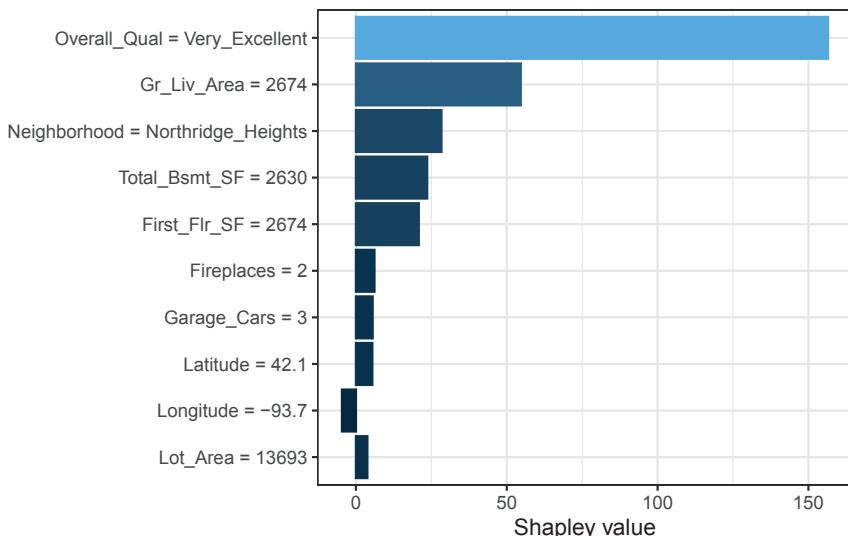
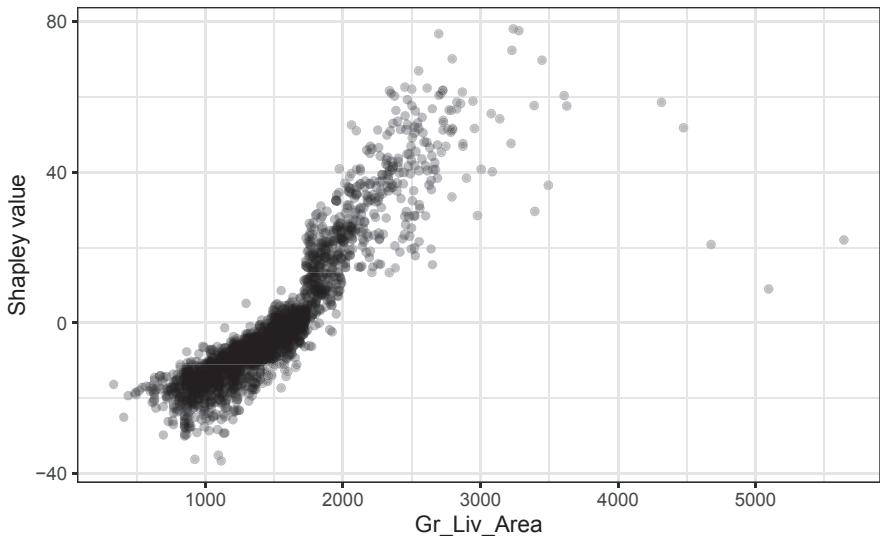


FIGURE 6.6: Top ten (Shapley-based) feature contributions to the highest training prediction from the Ames housing bagged tree ensemble.

Next, I'll construct a Shapley dependence plot for `Gr_Liv_Area` using `fastshap` with  $R = 50$  Monte Carlo repetitions. The results are displayed in [Figure 6.2](#). As with [Figures 6.2](#) and [6.4](#), the predicted sale price tends to increase with above grade square footage. As with the c-ICE curves in [Figure 6.4](#), the increasing dispersion in the plot indicates a potential interaction with at least one other feature. Coloring the Shapley dependence plot by the values of another feature can help visualize such an interaction, if you know what you're looking for.

```
ex <- explain(ames.bag, feature_names = "Gr_Liv_Area", X = X,
               nsim = 50, pred_wrapper = pfun)

# Shapley dependence plot
autoplot(ex, type = "dependence", X = X, alpha = 0.3)
```



**FIGURE 6.7:** Shapley dependence of above grade square footage on predicted sale price.

## 6.4 Drawbacks of existing methods

As discussed in Hooker et al. [2019], *permute-and-predict* methods—like PD plots, ICE plots, and permutation importance—can produce results that

are highly misleading.<sup>f</sup> For example, the standard approach to computing permutation-based VI scores involves independently permuting individual features. This implicitly makes the assumption that the observed features are statistically independent. In practice, however, features are often not independent which can lead to nonsensical VI scores. One way to mitigate this issue is to use the conditional approach described in Strobl et al. [2008b]; Hooker et al. [2019] provides additional alternatives, such as *permute-and-relearn importance*. Unfortunately, to the best of my knowledge, this approach is not yet available for general purposes. A similar modification can be applied to PD plots [Parr and Wilson, 2019].

I already mentioned that PD plots can be misleading in the presence of strong interaction effects. As discussed earlier, this can be mitigated by using ICE plots instead. Another alternative would be to use *accumulated local effect* (ALE) plots [Apley and Zhu, 2020]. Compared to PD plots, ALE plots have the advantage of being faster to compute and less affected by strong dependencies among the features. The downside, however, is that ALE plots are more complicated to implement. ALE plots are available in the **ALEPlot** [Apley, 2018] and **iml** packages in R.

Hooker [2007] also argues that feature importance (which concerns only main effects) can be misleading in high dimensional settings, especially when there are strong dependencies and interaction effects among the features, and suggests an approach based on a *generalized functional ANOVA decomposition*—though, to my knowledge, this approach is not widely implemented in open source software.

---

## 6.5 Final thoughts

IML is on the rise, and so is IML-related open source software. There are simply too many methods and useful packages to discuss in one chapter, so I only just covered a handful. If you’re looking for more, I’d recommend starting with the IML awesome list hosted by Patrick Hall at

<https://github.com/jphall1663/awesome-machine-learning-interpretability>.

---

<sup>f</sup>It’s been argued that approximate Shapley values share the same drawback; however, Janzing et al. [2019] makes a compelling case against those arguments.

A good resource for R users is Maksymiuk et al. [2021]. And of course, Molnar [2019] is a freely available resource, filled with intuitive explanations and links to relevant software in both R and Python. Molnar et al. [2021] is also worth reading, as they discuss a number of pitfalls to watch out for when using model-agnostic interpretation methods.



Taylor & Francis  
Taylor & Francis Group  
<http://taylorandfrancis.com>

# 7

---

## Random forests

---

In a forest of a hundred thousand trees, no two leaves are alike.  
And no two journeys along the same path are alike.

Paulo Coelho

---

---

### 7.1 Introduction

Random forests (RFs) are essentially bagged tree ensembles with an added twist, and they tend to provide similar accuracy to many state-of-the-art supervised learning algorithms on tabular data, while being relatively less difficult to tune. In other words, RFs tend to be competitive right out of the box. But be warned, RFs—like any statistical and machine learning algorithm—enjoy their fair share of disadvantages. As we’ll see in this chapter, RFs also include many bells and whistles that data scientists can leverage for non-prediction tasks, like detecting anomalies/outliers, imputing missing values, and so forth.

---

### 7.2 The random forest algorithm

Recall that a bagged tree ensemble (Section 5.1) consists of hundreds (sometimes thousands) of independently grown decision trees, where each tree is trained on a different bootstrap sample from the original training data. Each tree is intentionally grown deep (low bias), and variance is reduced by aver-

aging the predictions across all the trees in the ensemble. For classification, a plurality vote among the individual trees is used.

Unfortunately, correlation limits the variance-reducing effect of averaging. Take the following example for illustration. Suppose  $\{X_i\}_{i=1}^N \stackrel{iid}{\sim} (\mu, \sigma^2)$  is a random sample from some distribution with mean  $\mu$  and variance  $\sigma^2$ . Let  $\bar{X} = \sum_{i=1}^N X_i/N$  be the sample mean. If the observations are independent (as is the usual connotation of a random sample), then  $E(\bar{X}) = \mu$  and  $V(\bar{X}) = \sigma^2/N$ . In other words, the variance of the average is less than the variance of the sample elements. This of course assumes that the  $X_i$  are uncorrelated. If the pairwise correlation between any two observations is  $\rho = \rho(X_i, X_j)$  ( $i \neq j$ ), then

$$V(\bar{X}) = \rho\sigma^2 + \frac{1-\rho}{N}\sigma^2,$$

which converges to  $\rho\sigma^2$  as  $N \rightarrow \infty$ . In other words, regardless of sample size, correlation limits the variance-reducing effect of averaging. This is illustrated in Figure 7.1, where each boxplot is constructed from 30,000 sample means, each of which is based on a sample of size  $N = 30$  from a centered Gaussian distribution with specified pairwise correlation ( $x$ -axis); note the increasing variability in the sample means as we go from  $\rho = 0$  to  $\rho = 1$  (left to right).

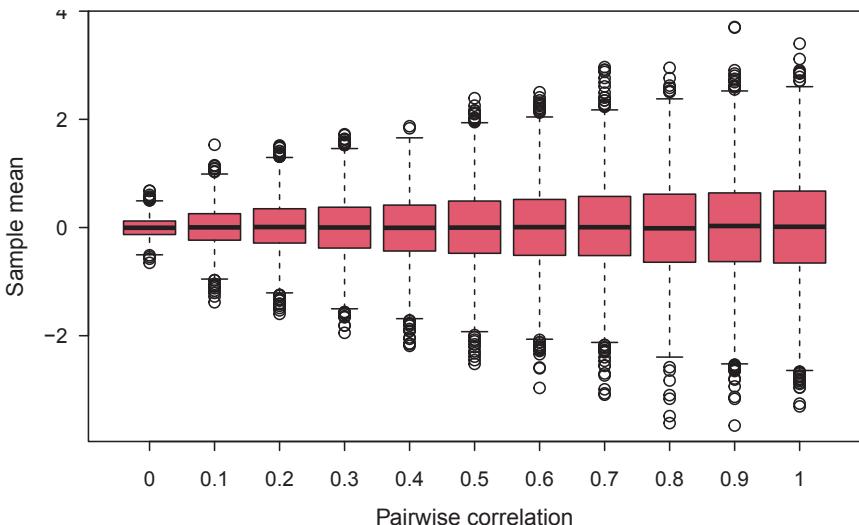


FIGURE 7.1: 100 simulated averages from samples of size  $N = 30$  with pairwise correlation increasing from zero to one.

Similarly, bagging a set of correlated predictors (i.e., models producing similar predictions) will only reduce the variance to a certain point. Since each tree is built using an independent bootstrap sample from the original training data, the trees in the bagged ensemble will be somewhat correlated. If we can reduce correlation between the trees, the trees will be more diverse and averaging can further improve the prediction and generalization performance of the ensemble.

**Figure 7.2** shows six bagged decision trees applied to the email spam training data; each tree was constrained to a max depth of three to help with the visualization. Since each tree was induced from an independent bootstrap sample from the same training set, the trees are naturally very similar to each other. Notice, for example, that the path to terminal node 15 (highlighted in green) is the same in four of the six trees, albeit the split points are slightly different. The performance of the combined ensemble might improve if we could make the trees more diverse.

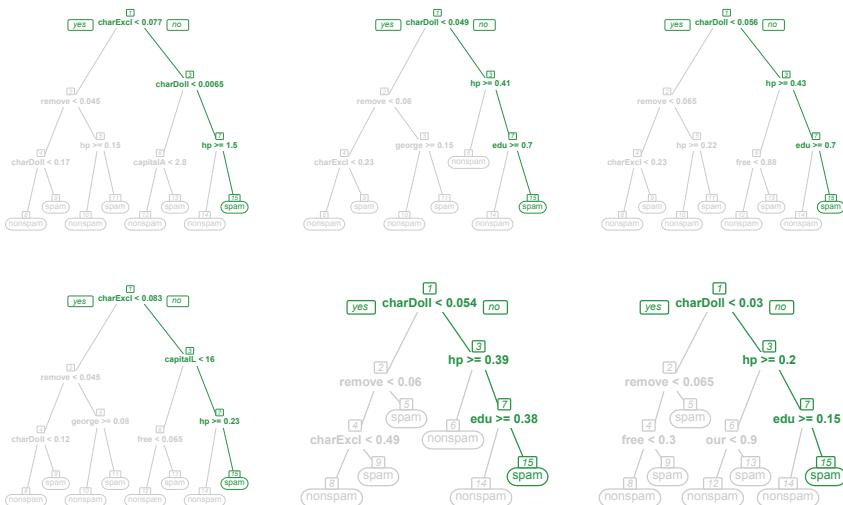


FIGURE 7.2: Six bagged decision trees applied to the email spam training data. The path to terminal node 15 is highlighted in each tree.

Luckily, Leo Breiman and Adele Cutler thought of a clever way to reduce correlation in a bagged tree ensemble; that is, make the trees more diverse. The idea is to limit the potential splitters at each node in a tree to a random subset of the available predictors, which will often result in a much more diverse ensemble of trees. In essence, bagging constructs a diverse tree ensemble by introducing randomness into the rows via sampling with replacement, while an RF further increases tree diversity by also introducing randomness into the columns via subsampling the features.

In other words, an RF is just a bagged tree ensemble with an additional layer of randomness produced by selecting a random subset of candidate splitters prior to partitioning the data at every node in every tree—*extremely randomized trees* ([Section 7.8.4](#)), take this randomization a step further in an attempt to reduce variance even more. Let  $m_{try} \leq p$  be the number of candidate splitters selected at random from the entire set of  $p$  features prior to each split in each tree. Setting  $m_{try} << p$  can often dramatically improve performance compared to bagged decision trees; note that setting  $m_{try} = p$  will result in an ordinary bagged tree ensemble.

That's it. That's essentially the difference between an ordinary bagged tree ensemble and an RF.

The general steps for constructing a traditional RF are given in Algorithm 7.1 (compare this to Algorithm 5.1).

The recommended default values for  $m_{try}$  (the number of features randomly sampled before each split) and  $n_{min}$  (the minimum size of any terminal node) depend on the type of outcome:

- For classification, the typical defaults are  $m_{try} = \lfloor \sqrt{p} \rfloor$ , where  $p$  is the number of features, and  $n_{min} = 1$ .
- For regression, the typical defaults are  $m_{try} = \lfloor p/3 \rfloor$  and  $n_{min} = 5$ .

These default values may vary slightly from one implementation to another. Note that  $\lfloor x \rfloor$  is just  $x$  rounded down to the nearest integer. As we'll see in [Section 7.4](#), generalization performance is typically most sensitive to the value of  $m_{try}$ , but the default is usually in the ballpark, and the tuning space for  $m_{try}$  is simple since  $m_{try} \in \{1, 2, \dots, p\}$ , where  $p$  is the total number of available features.

Traditionally, CART was used for the base learners in an RF, but any decision tree algorithm will work (e.g., GUIDE or CTree); I'll use CTree in [Section 7.2.3](#) to build an RF from scratch. Also, the traditional RF algorithm used the Gini splitting criterion (2.1) for classification and the SSE splitting criterion (2.6) for regression. However, many other splitting criteria can be used to great affect. For example, Ishwaran et al. [2008] proposed several splitting rules more appropriate for right-censored outcomes, including a log-rank splitting rule that splits nodes by maximization of the log-rank test statistic; see also Segal [1988], LeBlanc and Crowley [1992].

### 7.2.1 Voting and probability estimation

The voting scheme for classification outlined in step 4) of Algorithms 5.1 and 7.1 is called *hard voting*. In hard voting, each base learner casts a direct vote on the class label, and a majority vote (binary outcome) or plurality

---

**Algorithm 7.1** Traditional RF algorithm for classification and regression.

- 1) Start with a training sample,  $\mathbf{d}_{trn}$ , and specify integers,  $n_{min}$  (the minimum node size),  $B$  (the number of trees in the forest), and  $mtry \leq p$  (the number of predictors to select at random as candidate splitters prior to splitting the data at each node in each tree).
- 2) For  $b$  in  $1, 2, \dots, B$ :
  - (a) Select a bootstrap sample  $\mathbf{d}_{trn}^*$  of size  $N$  from the training data  $\mathbf{d}_{trn}$ .
  - (b) **Optional:** Keep track of which observations from the original training data were not selected to be in the bootstrap sample; these are called the out-of-bag (OOB) observations.
  - (c) Fit a decision tree  $\mathcal{T}_b$  to the bootstrap sample  $\mathbf{d}_{trn}^*$  according to the following rules:
    - (i) Before each attempted split, select a random sample of  $mtry$  features to use as candidate splitters.
    - (ii) Continue recursively splitting each terminal node until the minimum node size  $n_{min}$  is reached.
- 3) Return the “forest” of trees  $\{\mathcal{T}_b\}_{b=1}^B$ .
- 4) To obtain the RF prediction for a new case  $\mathbf{x}$ , pass the observation down each tree and aggregate as follows:
  - Classification:  $\hat{C}_B^{rf}(\mathbf{x}) = \text{vote} \left\{ \hat{C}_b(\mathbf{x}) \right\}_{b=1}^B$ , where  $\hat{C}_b(\mathbf{x})$  is the predicted class label for  $\mathbf{x}$  from the  $b$ -th tree in the forest (in other words, let each tree vote on the classification for  $\mathbf{x}$  and take the majority/plurality vote).
  - Regression:  $\hat{f}_B^{rf}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^B \hat{f}_b(\mathbf{x})$  (in other words, we just average the predictions for case  $\mathbf{x}$  across all the trees in the forest).

---

vote (multiclass outcome) is used to determine the overall classification of an observation.

With categorical outcomes, however, we often care more about the predicted probability of class membership, as opposed to directly predicting a class label. In an RF (or a bagged tree ensemble) there are two ways to obtain predicted probabilities:

- 1) Take the proportion of votes for each class over the entire forest.
- 2) Average the class probabilities from each tree in the forest. (In this case,  $n_{min}$  should be considered a tuning parameter; see, for example, Malley et al. [2012].)

The first approach can be problematic. For example, suppose the probability that  $\mathbf{x}$  belongs to class  $j$  is  $\Pr(Y = j|\mathbf{x}) = 0.91$ . If each tree correctly predicts class  $j$  for  $\mathbf{x}$ , then  $\widehat{\Pr}(\mathbf{x}) = 1$ , which is incorrect. If  $n_{min} = 1$ , the two approaches are equivalent and neither will produce consistent estimates of the true class probabilities (see, for example, Malley et al. [2012]). So which approach is better for probability estimation? Hastie et al. [2009, p. 283] argue that the second method tends to provide improved estimates of the class probabilities with lower variance, especially for small  $B$ .

Malley et al. [2012] make a similar argument for the binary case, but from a different perspective. In particular, they suggest treating the 0/1 outcome as numeric and fitting a regression forest using the standard MSE splitting criterion (an example of a so-called *probability machine*). It seems strange to use MSE on a 0/1 outcome, right? Not really. Recall from [Section 2.2.1](#) that the Gini index for binary outcomes is equivalent to using the MSE. Malley et al. recommend using a minimum node size equal to 10% of the number of training cases:  $n_{min} = \lfloor 0.1 \times N \rfloor$ . However, for probability estimation, it seems natural to treat  $n_{min}$  as a tuning parameter. Devroye et al. [1997, Chap. 21–22] provide some guidance on the choice of  $n_{min}$  for consistent probability estimation in decision trees.

The predicted probabilities can be converted to class predictions (i.e., by comparing each probability to some threshold), which gives us an alternative to hard voting called *soft voting*. In soft voting, we classify  $\mathbf{x}$  to the class with the largest averaged class probability. This approach to classification in RFs tends to be more accurate since predicted probabilities closer to zero or one are given more weight during the averaging step; hence, soft voting attaches more weight to votes with higher confidence (or smaller standard errors; [Section 7.7](#)).

### 7.2.1.1 Example: Mease model simulation

To illustrate the difference between a classification and regression forest for probability estimation with binary outcomes, I'll expand upon one of the simulation studies in Malley et al. [2012], the Mease example, in particular [Mease et al., 2007]. This is a two-dimensional circle problem with a binary outcome (0/1) and two independent features. The features are independent  $\mathcal{U}(0, 50)$  random variables (i.e., the points are generated at random in the square  $[0, 50]^2$ ). The probability function is defined as

$$p(\mathbf{x}) = \Pr(Y = 1|\mathbf{x}) = \begin{cases} 1, & r(\mathbf{x}) < 8 \\ \frac{28-r(\mathbf{x})}{20}, & 8 \leq r(\mathbf{x}) \leq 20 \\ 0, & r(\mathbf{x}) \geq 28 \end{cases} \quad (7.1)$$

where  $r(\mathbf{x})$  is the Euclidean distance from  $\mathbf{x} = (x_1, x_2)$  to the point  $(25, 25)$ . A sample of  $N = 1000$  observations from the Mease model is displayed in Figure 7.3; note that the observed 0/1 outcomes were generated according to the above probability rule  $p(\mathbf{x})$ . (As always, the code to reproduce the simulation is available on the companion website.)

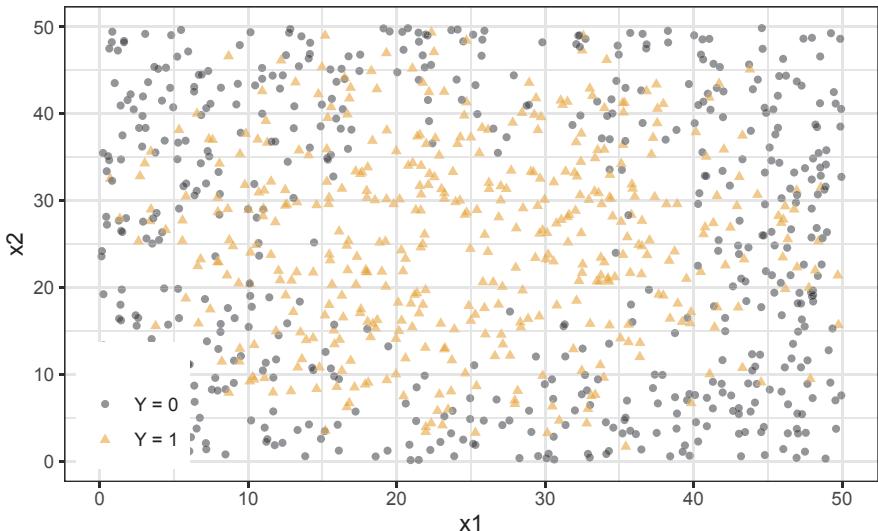


FIGURE 7.3: A sample of  $N = 1000$  observation from the Mease model.

Figures 7.4–7.5 display the results of the simulation. In Figure 7.4, the median predicted probability across all 250 simulations was computed and plotted vs. the true probability of class 1 membership; the dashed 45-degree line corresponds to perfect agreement. Here it is clear that the regression forest (i.e., treating the 0/1 outcome as continuous and building trees using the MSE splitting criterion) outperforms the classification forest (except when  $n_{min} = 1$ , in which case they are equivalent.) This is also evident from Figure 7.5, which shows the distribution of the MSE between the predicted class probabilities and the true probabilities for each case. In essence, for binary outcomes, regression forests produce consistent estimates of the true class probabilities.<sup>a</sup> This goes to show that  $m_{try}$  isn't the only important tuning parameter when

---

<sup>a</sup>By consistent, I mean that  $\widehat{\Pr}(Y = 1|\mathbf{x}) \rightarrow \Pr(Y = 1|\mathbf{x})$  as  $N \rightarrow \infty$ .

it comes to probability estimation, and you should make an effort to tune  $n_{min}$  as well.

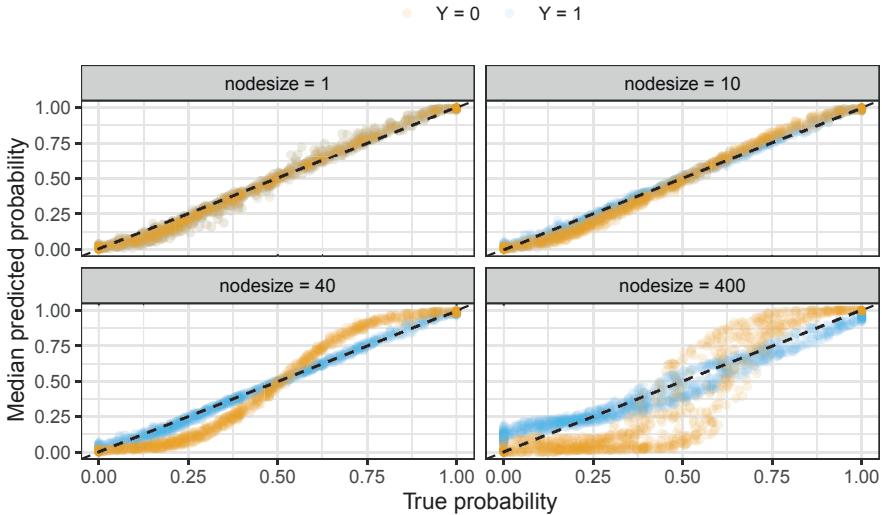


FIGURE 7.4: Class probability estimation using regression forests (yellow) and classification forests (black) in the Mease simulation. Starting from the top-left and moving clockwise, we have:  $n_{min} = 1$  (the typical default for classification forests),  $n_{min} = 10$  (the current default for probability forests in R’s `ranger` package [Wright et al., 2021]),  $n_{min} = 40$  (1% of the learning sample), and  $n_{min} = 400$  (10% of the learning sample), respectively. The dashed 45-degree line corresponds to perfect agreement.

### 7.2.2 Subsampling (without replacement)

While a traditional RF (Algorithm 7.1) uses bootstrap sampling (i.e., sample with replacement), it can be useful to subsample the training data without replacement, prior to constructing each tree. This was noted in Section 5.1.3 for bagged tree ensembles, and that discussion equally applies to RFs as well. Furthermore, as I’ll discuss in Section 7.5, subsampling with replacement can help eliminate certain bias in computing predictor importance, resulting in variable importance scores that can be used reliably for variable selection even in situations where the potential predictors vary in their scale of measurement or their number of categories. Most RF software includes the option to use subsampling with or without replacement.

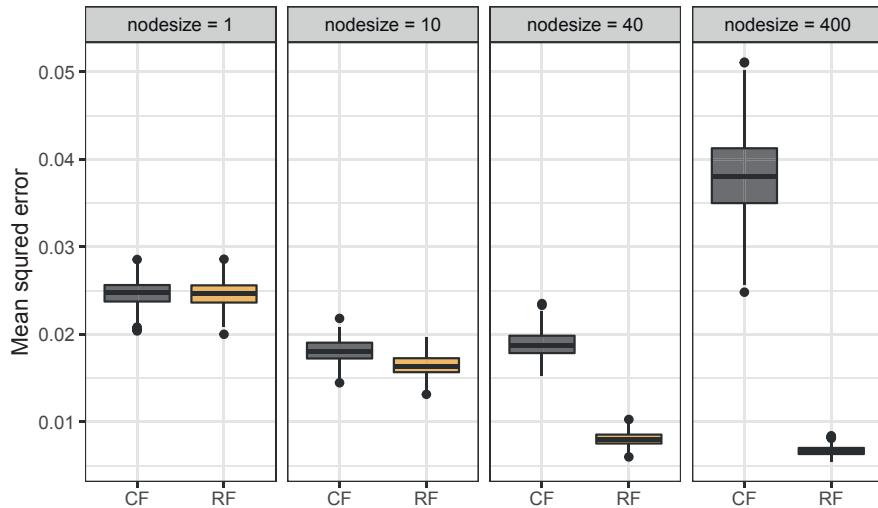


FIGURE 7.5: Mean squared errors (MSEs) from the Mease simulation. Here the MSE between the predicted probabilities and the true probabilities for each simulation are displayed using boxplots. Clearly, in this example, the regression forest (RF) with  $n_{min} > 1$  produces more accurate class probability estimates.

### 7.2.3 Random forest from scratch: predicting home prices

To help solidify the basic concepts of an RF, let's construct one from scratch.<sup>b</sup> To do that, we need a decision tree implementation that will allow us to randomly select a subset of features for consideration at each node in the tree. Such arguments are available in the `sklearn.tree` module in Python, as well as R's `party` and `partykit` packages—unfortunately, this option is not currently available in `rpart`. In this example, I'll go with `party`, since it's `ctree()` function is faster, albeit less flexible, than `partykit`'s implementation.

Below is the definition for a function called `crforest()`, which constructs a *conditional random forest* (CRF) [Hothorn et al., 2006a, Strobl et al., 2008a, 2007b], that is, an RF using conditional inference trees (Chapter 3) for the base learners.<sup>c</sup> The `oob` argument will come into play in Section 7.3, so just

<sup>b</sup>The code I'm about to show is for illustration purposes only. It will not be nearly as efficient as actual RF software, which is typically written in a compiled language, like C, C++, or Fortran.

<sup>c</sup>Note that `party` and `partykit` both contain a `cforest()` function for fitting CRFs.

ignore that part of the code for now. Note that the function returns a list of fitted CTrees that we can aggregate later for the purposes of prediction.

```
crforest <- function(X, y, mtry = NULL, B = 5, oob = TRUE) {
  min.node.size <- if (is.factor(y)) 1 else 5
  N <- nrow(X) # number of observations
  p <- ncol(X) # number of features
  train <- cbind(X, "y" = y) # training data frame
  fo <- as.formula(paste("y ~ ", paste(names(X), collapse = "+")))
  if (is.null(mtry)) { # use default definition
    mtry <- if (is.factor(y)) sqrt(p) else p / 3
    mtry <- floor(mtry) # round down to nearest integer
  }
  # CTree parameters; basically force the tree to have maximum depth
  ctrl <- party::ctree_control(mtry = mtry, minbucket = min.node.size,
                                 minsplit = 10, mincriterion = 0)
  forest <- vector("list", length = B) # to store each tree
  for (b in 1:B) { # fit trees to bootstrap samples
    boot.samp <- sample(1:N, size = N, replace = TRUE)
    forest[[b]] <- party::ctree(fo, data = train[boot.samp, ],
                                control = ctrl)
    if (isTRUE(oob)) { # store row indices for OOB data
      attr(forest[[b]], which = "oob") <-
        setdiff(1:N, unique(boot.samp))
    }
  }
  forest # return the "forest" (i.e., list) of trees
}
```

Let's test out the function on the Ames housing data, using the same 70/30 split from previous examples (Section 1.4.7). Here, I'll fit a default CRF (i.e.,  $m_{try} = \lfloor p/3 \rfloor$  and  $n_{min} = 5$ ) using our new `crforest()` function. (Be warned, this code may take a few minutes to run; the code on the book website includes an optional progress bar and the ability to run in parallel using the `foreach` package [Revolution Analytics and Weston, 2020].)

```
X <- subset(ames.trn, select = -Sale_Price) # feature columns
set.seed(1408) # for reproducibility
ames.crf <- crforest(X, y = ames.trn$Sale_Price, B = 300)
```

To obtain predictions from the fitted model, we can just loop through each tree, extract the predictions, and then average them together at the end. This can be done with a simple `for` loop, which is demonstrated in the code chunk below. Here, I obtain the averaged predictions from `ames.crf` on the test data and compute the test RMSE.

```
B <- length(ames.crf) # number of trees in forest
preds.tst <- matrix(nrow = nrow(ames.tst), ncol = B)
for (b in 1:B) { # store predictions from each tree in a matrix
  preds.tst[, b] <- predict(ames.crf[[b]], newdata = ames.tst)
```

```

}

pred.tst <- rowMeans(preds.tst) # average predictions across trees

# Root-mean-square error function
rmse <- function(pred, obs, na.rm = FALSE) {
  sqrt(mean((pred - obs) ^ 2, na.rm = na.rm))
}

# Root mean square error on test data
rmse(pred.tst, obs = ames.tst$Sale_Price)

#> [1] 24.3

```

Rather than reporting the test RMSE for the entire forest, we can compute it for each sub-forest of size  $b \leq B$  to see how it changes as the forest grows. We can do this using a simple `for` loop, as demonstrated in the code chunk below. (Note that I use `drop = FALSE` here so that the subset matrix of predictions doesn't lose its dimension when `b = 1`.)

```

rmse.tst <- numeric(B) # to store RMSEs
for (b in 1:B) {
  pred <- rowMeans(preds.tst[, 1:b, drop = FALSE], na.rm = TRUE)
  rmse.tst[b] <- rmse(pred, obs = ames.tst$Sale_Price, na.rm = TRUE)
}

```

The above test RMSEs are displayed in Figure 7.6 (black curve). For comparison, I also included the test error for a single CTree fit (horizontal dashed line). Here, the CRF clearly outperforms the single tree, and the test error stabilizes after about 50 trees. Next, I'll discuss an internal cross-validation strategy based on the OOB data.

## 7.3 Out-of-bag (OOB) data

One of the most useful by-products of an RF (or bagging in general, for that matter) is the so-called OOB data (see Step 2) (b) in Algorithm 7.1)<sup>d</sup>. Recall that an RF, or any bagged tree ensemble, is constructed by combining predictions from decision trees trained on different bootstrap samples.<sup>e</sup> Since bootstrapping involves sampling with replacement, each tree in the forest only uses a subset of the original learning sample; hence, for each tree in the forest,

<sup>d</sup>While the concept of OOB is usually discussed in the context of an RF, it equally applies to bagging and boosting when sampling is involved, regardless if the sampling is done with or without replacement.

<sup>e</sup>This discussion also applies to subsampling without replacement.

a portion of the original learning sample isn't used—these observations are referred to as out-of-bag (or OOB for short). The OOB data associated with a particular tree can be used to obtain an unbiased estimate of prediction error. The OOB errors can then be aggregated across all the trees in the forest to obtain an overall out-of-sample, albeit unstructured, estimate of the overall prediction performance of the forest.

Since bagging/bootstrapping involves sampling with replacement, the probability that a particular case is not selected in a particular bootstrap sample is

$$\Pr(\text{case } i \notin \text{bootstrap sample } b) = \left(1 - \frac{1}{N}\right)^N.$$

As  $N \rightarrow \infty$  it can be shown that  $\left(1 - \frac{1}{N}\right)^N \rightarrow e^{-1} \approx 0.368$ . In other words, on average, each bootstrap sample contains approximately  $1 - e^{-1} \approx 0.632$  of the original training records; the remaining  $e^{-1} \approx 0.368$  observations are OOB and can be used as an independent validation set for the corresponding tree. This is rather straightforward to observe without a mathematical derivation. The code below computes the proportion of non-OOB observations in  $B = 10000$  bootstrap samples of size  $N = 100$ , and averages the results together:

```
set.seed(1226) # for reproducibility
N <- 100 # sample size
obs <- 1:N # original observations
res <- replicate(10000, sample(obs, size = N, replace = TRUE))
inbag <- apply(res, MARGIN = 2, FUN = function(boot.sample) {
  mean(obs %in% boot.sample) # proportion in bootstrap sample
})
mean(inbag)

#> [1] 0.634
```

Let  $w_{b,i} = 1$  if observation  $i$  is OOB in the  $b$ -th tree and zero otherwise. Further, if we let  $B_i = \sum_{b=1}^B w_{b,i}$  be the number of trees in the forest for which observation  $i$  is OOB, then the OOB prediction for the  $i$ -th training observation is given by

$$\hat{y}_i^{OOB} = \frac{1}{B_i} \sum_{b:w_{b,i}=1} \hat{y}_i^b, i = 1, 2, \dots, N. \quad (7.2)$$

The OOB error estimate is just the error computed from these OOB predictions. (See [Hastie et al., 2009, Sec. 7.11] for a more general discussion on using the bootstrap to estimate prediction error and its apparent bias.)

To illustrate, I'm going to compute the OOB RMSE for the CRF I previously fit to the Ames housing data. There are numerous ways in which this can be

done programmatically given our setup; I chose the easy route. Recall that each tree in our `rfo` object contains an attribute called "oob" which stores the row numbers for the training records that were OOB for that particular tree. From these we can easily construct an  $N \times B$  matrix, where the  $(i, j)$ -th element is given by

$$\begin{cases} \hat{y}_i^b & \text{if } w_{b,i} = 1 \\ \text{NA} & \text{if } w_{b,i} = 0 \end{cases}.$$

The reason for using `NA`s in place of the predictions for the non-OOB observations will hopefully become apparent soon.

```
preds.oob <- matrix(nrow = nrow(ames.trn), ncol = B) # OOB predictions
for (b in 1:B) { # WARNING: Might take a minute or two!
  oob.rows <- attr(ames.crf[[b]], which = "oob") # OOB row IDs
  preds.oob[oob.rows, b] <-
    predict(ames.crf[[b]], newdata = ames.trn[oob.rows, ])
}
pred.oob <- rowMeans(preds.oob) # average OOB predictions across trees

# Peek at results
preds.oob[1:3, 1:6]

#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,] 184   NA   NA   NA   NA 163
#> [2,] NA   143   NA   NA   NA 154
#> [3,] NA   136  115   NA  136   NA
```

Peeking at the first few rows and columns you can see that the first training observation (which corresponds to the first row in the above matrix) was OOB in the first and sixth trees (since the rest of the columns are `NA`), whereas the second observation was OOB for trees two and six, so I obtained the corresponding OOB predictions for these. Next, I compute  $\hat{y}_i^{OOB}$  as in Equation (7.2) by computing the row means of our matrix `pred.oob`—setting `na.rm = TRUE` in the call to `rowMeans()` ensures that the `NA`s in the matrix aren't counted, so that the average is taken only over the OOB predictions (i.e., the correct denominator  $B_i$  will be used). Note that the OOB error is slightly larger than the test error I computed earlier; this is typical in many common settings, as noted in Janitza and Hornung [2018].

```
pred.oob <- rowMeans(preds.oob, na.rm = TRUE)
rmse(pred.oob, obs = ames.trn$Sale_Price, na.rm = TRUE)

#> [1] 26.6
```

Similar to what I did in the previous section, I can compute the OOB RMSE as a function of the number of trees in the forest. The results are displayed in Figure 7.6, along with the test RMSEs from the same forest (black curve)

and test error from a single CTree fit (horizontal blue line). Here, we can see that the OOB error is consistently higher than the test error, but both begin to stabilize at around 50 trees.

```
rmse.oob <- numeric(B) # to store RMSEs
for (b in 1:B) {
  pred <- rowMeans(preds.oob[, 1:b, drop = FALSE], na.rm = TRUE)
  rmse.oob[b] <- rmse(pred, obs = ames.trn$Sale_Price, na.rm = TRUE)
}
```

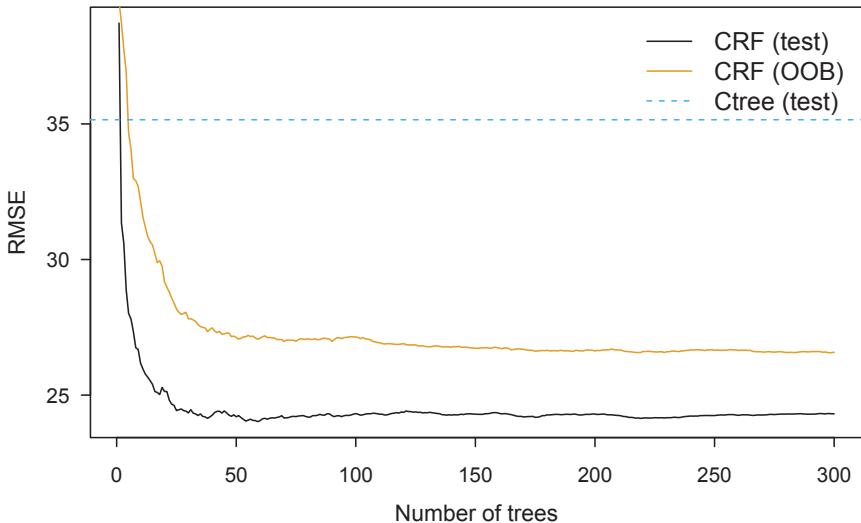


FIGURE 7.6: RMSEs for the Ames housing example: the CRF test RMSEs (black curve), the CRF OOB RMSEs (yellow curve), and the test RMSE from a single CTree fit (horizontal blue line).

As noted in Hastie et al. [2009], the OOB error estimate is almost identical to that obtained by  $N$ -fold cross-validation, where  $N$  is the number of rows in the learning sample; this is also referred to as *leave-one-out cross-validation* (LOOCV). Hence, algorithms that produce OOB data can be fit in one sequence, with cross-validation being performed along the way. The OOB error can be monitored during fitting and the training can stop once the OOB error has “stabilized”. In the Ames housing example (Figure 7.6) it can be seen that the test and OOB errors both stabilize after around 50 trees.

While the OOB error is computationally cheap, Janitza and Hornung [2018] observed that it tends to overestimate the true error in many practical situations, including

- when the class frequencies are reasonably balanced in classification settings;

- when the sample size  $N$  is large;
- when there is a large number of predictors;
- when there is correlation between the predictors;
- when the main effects are weak.

The positive bias of OOB error estimates was also noted in Bylander [2002]. In light of this, it seems reasonable to consider the OOB error estimate as an upper bound on the true error. Fortunately, Janitza and Hornung [2018] argue that the OOB error can be used effectively for hyperparameter tuning in RFs—which I’ll discuss in the next section—without substantially affecting performance. This is good news since  $k$ -fold cross-validation can be computationally expensive, especially when tuning more complex models, like tree-based ensembles.

---

## 7.4 Hyperparameters and tuning

RF is one of the most useful *off-the-shelf* statistical learning algorithms you can know. By off-the-shelf, I mean a procedure that can be used effectively without much tweaking or tuning. Don’t get me wrong, you can (and should try to) improve performance with a bit of tuning, but relative to other algorithms, the RFs often do reasonably well at their default settings. In contrast, *gradient tree boosting* (Chapter 8) can often outperform RFs, but typically require a lot more tuning.

The most important tuning parameter in an RF is  $m_{try}$ . But I’d argue that the typical defaults (i.e.,  $m_{try} = \lfloor \sqrt{p} \rfloor$  for classification and  $m_{try} = \lfloor p/3 \rfloor$  for regression) are quite good. For selecting  $m_{try}$ , a simple heuristic is to try the default, half of the default, and twice the default, and pick the best [Liaw and Wiener, 2002]. According to Liaw and Wiener [2002], the results generally do not change dramatically, and even setting  $m_{try} = 1$  can give very good performance for some data. Setting  $m_{try} < p$  also lessens the computational burden of split variable selection (e.g., CART’s exhaustive search for the best split), making RFs more computationally efficient than bagged tree ensembles, especially for larger data sets. On the other hand, if you only suspect a small fraction of the predictors to be “important,” then larger values of  $m_{try}$  may give better generalization performance.

The number of trees in the forest ( $B$ ) is arguably not a tuning parameter. You just need to make sure enough trees are aggregated for the error to stabilize (see, for example, Figure 7.6). However, it can be wasteful to fit more trees than necessary, especially when dealing with large data sets. For this reason,

some RF implementations have the option to "stop early" if the validation error stops improving; the R package **h2o** [LeDell et al., 2021] includes an RF implementation that supports early stopping with a wide variety of performance metrics. Such early stopping can be based on an independent test set, cross-validation, or the OOB error.

What about using the OOB error estimate or tuning? Although it's been argued that the OOB error tends to overestimate the true error in certain cases (see, for example, Mitchell [2011]), Janitza and Hornung [2018] noted that the overestimation seems to have little to no impact on tuning parameter selection, at least in their simulations. If ordinary cross-validation is too expensive, and you don't have access to separate validation and test sets, then using the OOB error is a certainly reasonable thing to do, and in many cases, more efficient.

To illustrate, I carried out a small simulation using the Friedman 1 benchmark problem introduced in [Section 1.4.3](#). For these data, there are 10 possible values for the  $m_{try}$  parameter. For each value, I generated 100 separate train and test sets of  $N = 1,000$  observations each. For each repetition, I computed the OOB and test MSE and plotted the results; the results are displayed in [Figure 7.7](#). In this example, you can see that the OOB error is quite in line with the test error, and both suggest an optimal value of  $m_{try}$  around 5 or 6 (the traditional default here, indicated by a dashed vertical line, is  $m_{try} = \lfloor 10/3 \rfloor = 3$ ).

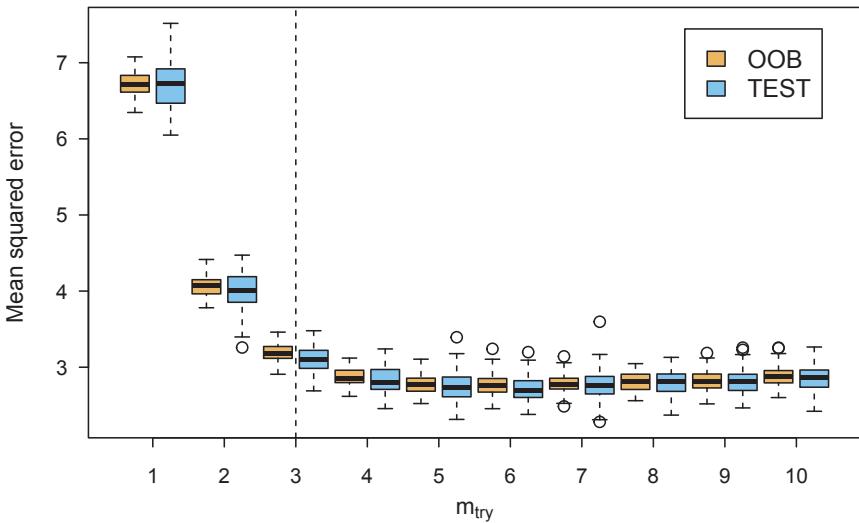


FIGURE 7.7: OOB and test error vs.  $m_{try}$  for the Friedman benchmark data using  $N = 2000$  with a 50/50 split. The dashed line indicates the standard default for regression; in this case,  $m_{try} = 3$ .

---

## 7.5 Variable importance

Breiman [2002] proposed two measures of variable importance for RFs:

- A measure based on the mean decrease in node impurity ([Section 7.5.1](#)), which I'll refer to as the impurity-based measure. Originally, only the Gini splitting criterion ([Section 2.2.1](#)) was used. This variable importance measure was discussed for general tree ensembles in [Section 5.4](#).
- A novel permutation-based measure ([Section 7.5.2](#)), which I'll refer to as the OOB-based permutation measure. This variable importance measure was discussed for general use in any supervised learning model in [Section 6.1.1](#).

While these variable importance measures were originally introduced for the classification case, they naturally extend to the regression case as well. Note that many other measures have also been defined. An up-to-date and thorough overview of quantifying predictor importance in different tree-based methods is given in Loh and Zhou [2021].

### 7.5.1 Impurity-based importance

As discussed in [Section 5.4](#), the importance of any predictor can be measured by aggregating the individual variable importance scores across all the trees in the forest. For an arbitrary feature  $x$ , we can use the total decrease in node impurities from splitting on  $x$  (e.g., as measured by MSE for regression or the Gini index for classification), averaged over all  $B$  trees in the forest:

$$\text{VI}(x) = \frac{1}{B} \sum_{b=1}^B \text{VI}_{T_b}(x), \quad (7.3)$$

where  $\text{VI}_{T_b}(x)$  is the relative importance of  $x$  in tree  $T_b$  ([Section 2.8](#)). Since averaging helps to stabilize variance,  $\text{VI}(x)$  tends to be more reliable than  $\text{VI}_{T_b}(x)$  [Hastie et al., 2009, p. 368].

The split variable selection bias inherent in CART-like decision trees also affects the impurity-based importance measure in their ensembles (7.3). The bias tends to result in higher variable importance scores for predictors with more potential split points (e.g., categorical variables with many categories). Several authors have proposed methods for eliminating the bias when the Gini index is used as the splitting criterion; see, for example, Sandri and Zuccolotto [2008] and the references therein. An interesting (and rather simple) approach

is provided in Sandri and Zuccolotto [2008], and later modified in Nembrini et al. [2018] for RFs.

The idea in Sandri and Zuccolotto [2008] is to realize that the impurity-based importance measure from a single CART-like decision tree can be expressed as the sum of two components:

$$\text{VI}_{T_b}(x) = \text{VI}_{T_b}^{true}(x) + \text{VI}_{T_b}^{bias}(x),$$

where  $\text{VI}_{T_b}^{true}(x_i)$  is the part attributable to informative splits and is related to the “true” importance of  $x_i$ , and  $\text{VI}_{T_b}^{bias}(x_i)$  is the part attributable to uninformative splits and is a source of bias. The algorithm they propose attempts to eliminate the bias in  $\text{VI}_{T_b}(x_i)$  by subtracting off an estimate of  $\text{VI}_{T_b}^{bias}(x_i)$ . This is done many times and the results averaged together. The basic steps are outlined in Algorithm 7.2 below.

---

**Algorithm 7.2** Bias-corrected Gini importance.

---

- 1) For  $r = 1, 2, \dots, R$ :
    - 1) Given the original  $N \times p$  matrix of predictor values  $\mathbf{X}$ , generate an  $N \times p$  matrix of *pseudo predictors*  $\mathbf{Z}_r$  using one of the following techniques:
      - Randomly permuting each column of the original predictor values  $\mathbf{X}$  (hence, the  $j$ -th column of  $\mathbf{Z}_r$  can be obtained by randomly shuffling the values in the  $j$ -th column of  $\mathbf{X}$ ).
      - Randomly permuting the rows of  $\mathbf{X}$ ; this has the advantage of maintaining any existing relationships between the original predictors.
    - 2) Apply the ensemble procedure (e.g., bagging, boosting, or RF) using  $\hat{\mathbf{X}}_r = (\mathbf{X}, \mathbf{Z}_r)$  as the set of available predictors (i.e., use both the original predictors, as well as the randomly generated pseudo predictors).
    - 3) Use Equation 7.3 to compute both  $\text{VI}(x_i)$  and  $\text{VI}(z_i)$ ; that is, compute the usual impurity-based variable importance measure for each predictor  $x_i$  and pseudo predictor  $z_i$ , for  $i = 1, 2, \dots, p$ .
  - 2) Compute the bias-adjusted impurity-based importance measure for each predictor  $x_i$  ( $i = 1, 2, \dots, p$ ) as  $\text{VI}^*(x_i) = R^{-1} \sum_{r=1}^R (\text{VI}(x_i) - \text{VI}(z_i))$ .
-

Algorithm 7.2 can be used to correct biased variable importance scores from a single CART-like tree or an ensemble thereof. Also, while the original algorithm was developed for the Gini-based importance measure, Sandri and Zuccolotto [2008] suggest it is also effective at eliminating bias for other impurity measures, like cross-entropy and SSE. One of the drawbacks of Algorithm 7.2., however, is that it effectively doubles the number of predictors to  $2p$  and requires multiple ( $R$ ) iterations. This can be computationally prohibitive for large data sets, especially for tree-based ensembles. Fortunately, Nembrini et al. [2018] proposed a similar technique specific to RFs that only requires a single replication. I'll omit the details, but the procedure is available in the **ranger** package for R (which has also been ported to Python and is available in the **skranger** package [Flynn, 2021]); an example is given in [Figure 7.8](#).

Even though our quick-and-dirty `crforest()` function in [Section 7.2.3](#) used bootstrap sampling, the actual CRF procedure described in Strobl et al. [2007b], and implemented in R packages **party** and **partykit**, defaults to growing trees on random subsamples of the training data without replacement (by default, the size of each sample is given by  $\lfloor 0.632N \rfloor$ ), as opposed to bootstrapping. Strobl et al. [2007b] showed that this effectively removes the bias in CRFs due to the presence of predictor variables that vary in their scale of measurement or their number of categories.

### 7.5.2 OOB-based permutation importance

RFs offer an additional (and unbiased) variable importance method; the approach is quite similar to the more general permutation approach discussed in [Section 6.1.1](#), but it's based on permuting observations in the OOB data instead. The idea is that if predictor  $x$  is important, then the OOB error will go up when  $x$  is perturbed in the OOB data. In particular, we start by computing the OOB error for each tree. Then, each predictor is randomly shuffled in the OOB data, and the OOB errors are computed again. The difference in the two errors is recorded for the OOB data, then averaged across all trees in the forest.

As with the more general permutation-based importance measure, these scores can be unreliable in certain situations; for example, when the predictor variables vary in their scale of measurement or their number of categories [Strobl et al., 2007a], or when the predictors are highly correlated [Strobl et al., 2008b]. Additionally, the corrected Gini-based importance discussed in Nembrini et al. [2018] has the advantage of being faster to compute and more memory efficient.

[Figure 7.8](#) shows the results from three difference RF variable importance measures on the simulation example from [Section 3.1](#); the simulation comparing the split variable selection bias between CART and CTree. Here, we

can see that the traditional Gini-based variable importance measure is biased towards the categorical variables, while the corrected Gini and permutation-based variable importance scores are relatively unbiased.

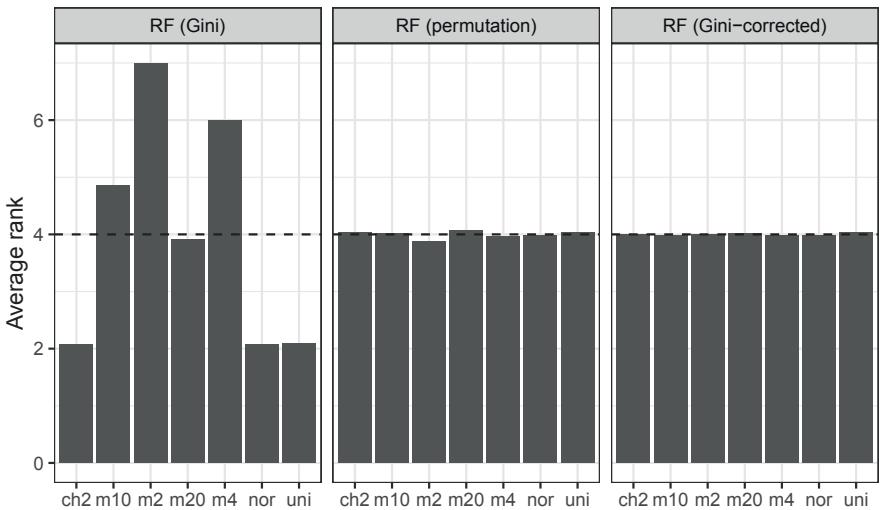


FIGURE 7.8: Average feature importance ranking for three RF-based variable importance measures. Left: the traditional Gini-based measure. Middle: the OOB-based permutation measure. Right: The corrected Gini-based measure.

The next two sections discuss more contemporary permutation schemes for RFs that deserve some consideration.

### 7.5.2.1 Holdout permutation importance

One drawback to computing variable importance in general is the lack of a natural cutoff that can be used to discriminate between “important” and “non-important” predictors. A number of approaches based on null hypothesis testing and *thresholding* have been developed for addressing this problem; see, for example, Altmann et al. [2010] and Loh and Zhou [2021, Sec. 6]. Janitza et al. [2018] argued that the null distribution of the OOB-based permutation measure is not necessarily symmetric; in particular, for irrelevant features. This makes the OOB-based permutation variable importance scores less suitable for selecting relevant features using a hypothesis-driven approach. Instead, Janitza et al. [2018] propose a method referred to as *holdout variable importance*, which has a symmetric null distribution for both relevant and irrelevant predictors. The idea is to split the data into two halves, grow an RF on one half, and use the leftover half to compute a permutation-based variable importance score. This method is available in the R package **ranger**.

### 7.5.2.2 Conditional permutation importance

A major drawback of permutation-based importance measures is the inherent assumption of independent features (e.g., the features are uncorrelated). For example, if  $x_1$  and  $x_2$  have a strong dependency, then it doesn't make sense to randomly permute  $x_1$  while holding  $x_2$  constant (or vice versa). To this end, [Strobl et al., 2008b] describe a *conditional permutation importance* measure that adjusts for correlations between predictor variables. In particular, the conditional permutation importance of each variable is computed by permuting within a grid defined by the covariates that are associated with the variable of interest. According to Strobl et al. [2008b], the resulting variable importance scores are conditional in the sense of coefficients in a regression model, but represent the effects of a variables in both main effects and interactions. When missing values are present in the predictors, the procedure described in Hapfelmeier et al. [2014] can be used to measure variable importance. While this idea applies in general to any type of RF, its implementations currently seem limited to the conditional inference trees and forest provided by the **party** and **partykit** packages in R; see, for example, `?partykit::varimp`.

---

## 7.6 Casewise proximities

So far in this book, we've mainly discussed tree-based methods for supervised learning problems. However, not every problem is supervised. For example, it is often of interest to understand how the data clusters—that is, whether the rows of the data form any “interesting” groups. (This is an application of unsupervised learning.) Many clustering methods rely on computing the pairwise distances between any two rows in the data, but the challenge becomes choosing the right distance metric. Euclidean distance (i.e., the “ordinary” straight-line, or “as the crow flies” distance between two points), for example, is quite sensitive to the scale of the inputs. It's also rather awkward to compute the Euclidean distance between two rows of data when the features are a mix of both numeric and categorical types. Fortunately, other distance (or distance-like) measures are available which more naturally apply to mixed data types.

Another useful output that can be obtained from an RF, provided it's implemented, are pairwise case *proximities*. RF proximities are distance-like measures of how similar any two observations are, and can be used for

- clustering in supervised and unsupervised (Section 7.6.3) settings;

- detecting outliers/novel cases ([Section 7.6.1](#));
- imputing missing values ([Section 7.6.2](#)).

To compute the proximities between all pairs of training observations in an RF, do the following:

- 1) pass all of the data, both training and OOB, down each tree;
- 2) every time records  $i$  and  $j$  cohabit in the same terminal node of a tree, increase their proximity by one;
- 3) At the end, normalize the proximities by dividing by the number of trees in the forest.

So how does this measure similarity between cases? Recall that RFs (and bagged decision trees in general) intentionally build deep, overgrown decision trees. In order for two observations to land in the same terminal node, they have to satisfy all of the same conditions leading to it. If two observations occupy the same terminal node across a majority of the trees in the forest, then they are likely very similar to each other in terms of feature values. Note that using all the training data can lead to unrealistic proximities. To circumvent this, proximities can be computed on only the OOB cases. It is also possible to compute proximities for new cases (an example application is given in [Section 7.6.4](#)).

The end result is an  $N \times N$  proximity matrix, where  $N$  is the sample size of the data set proximities are being computed for. As it turns out, this matrix is symmetric (since  $\text{prox}(i, j) = \text{prox}(j, i)$ ), positive definite (i.e., has all positive eigenvalues), and bounded above by one, with the diagonal elements equal to one (since  $\text{prox}(i, i) = 1$ ). Consequently, for any two cases  $i$  and  $j$ , we can treat  $1 - \text{prox}(i, j)$  as a squared distance-like metric, which can be used as input into any distance-based clustering algorithm. For example, Shi et al. [2005] used RF proximities to help identify fundamental subtypes of cancer. A brief example using the Swiss banknote data is provided in [Section 7.6.3.1](#).

The proximities from an RF provide a natural measure of similarity between records when the predictor variables are of mixed types (e.g., numeric and categorical) and measured on different scales; they are invariant to monotone transformations and naturally support categorical variables. The biggest drawback, as with any pairwise distance-like metric, is that it requires storing an  $N \times N$  matrix; although, since the casewise proximity matrix is symmetric, you only need to store the upper or lower triangular part (see, for example, `?treemisc::proximity`). Proximities are also not implemented in most open source RF software. However, if you can obtain the  $N \times B$  matrix of terminal node assignments (which is available in most open source RF software), then it is rather straightforward to compute the proximities yourself; an example, specific to the R package **ranger** [Wright et al., 2021], can be found at <https://mnwright.github.io/ranger/r/oob-proximity-matrix/>, while a

C++ implementation is available in `treemisc`'s `proximity()` function. The next two sections discuss more specific uses of proximities that are useful in a supervised learning context.

### 7.6.1 Detecting anomalies and outliers

Outliers (or anomalies) are generally defined as cases that are removed from the main body of the data. In the context of an RF, Leo Breiman defined outliers as cases whose proximities to all other cases in the data are generally small. For classification, he proposed a simple measure of “outlyingness” based on the RF proximity values. Define the average proximity from case  $m$  in class  $j$  to the rest of the training data in class  $j$  as

$$\text{prox}^*(m) = \sum_{k \in \text{class } j} \text{prox}^2(m, k),$$

where the sum is over all training instances belonging to class  $j$ . The outlyingness of case  $m$  in class  $j$  to all other cases in class  $j$  is defined as

$$\text{out}(m, j) = \frac{N}{\text{prox}^*(m)},$$

where  $N$  is the number of training instances. Generally, a value above 10 is reason to suspect the case of being an outlier [Breiman, 2002]. Obviously, this measure is limited to smaller data sets and RF implementations where proximities can be efficiently computed. In [Section 7.8.5](#), I'll look at a specialized RF extension that's more suitable for detecting outliers and anomalies, especially in higher dimensions. An interesting use case for the proximity-based outlyingness measure is presented in the next section.

#### 7.6.1.1 Example: Swiss banknotes

An interesting use case for the proximity-based outlyingness measure is in detecting potentially mislabeled response classes in classification problems. Consider, for example, the Swiss banknote data from [Section 1.4.1](#). Before fitting a default RF, I switched the label for observation 101; this observation is supposedly a counterfeit banknote ( $y = 1$ ), but I switched the class label to genuine ( $y = 0$ ). The proximity-based outlier scores are displayed in [Figure 7.9](#). There are two obvious potential outliers, labeled with their corresponding row number. Here, you can see that the counterfeit banknote I mislabeled as genuine (observation 101) received the largest outlier score. Observation 70 is also interesting and worth investigating; perhaps it was also mislabeled?

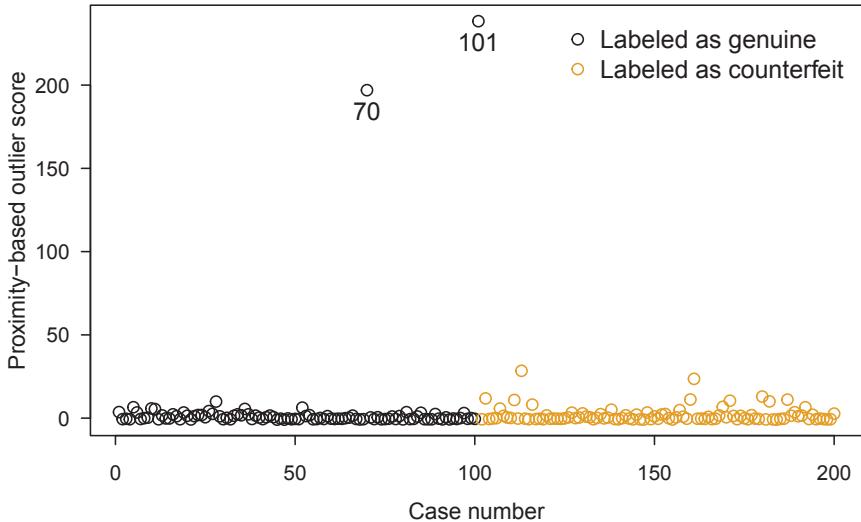


FIGURE 7.9: Proximity-based outlier scores for the Swiss banknote data. The largest outlier score corresponds to observation 101, which was a counterfeit banknote that was mislabeled as genuine.

### 7.6.2 Missing value imputation

Many decision tree algorithms can naturally handle missing values; CART and CTree, for example, employ surrogate splits to handle missing values ([Section 2.7](#)). Unfortunately, the idea does not carry over to RFs. I suppose that makes sense: searching for surrogates would greatly increase the computation time of the RF algorithm. Although some RF software can handle missing values without casewise deletion (e.g., the **h2o** package in both R and Python), most often they have to be imputed or otherwise dealt with.

Breiman also developed a clever way to use RF proximities for imputing missing values. The idea is to first impute missing values with a simple method (such as using the mean or median for numeric predictors and the most common value for categorical ones). Next, fit an initial RF to the  $N$  complete observations and generate the  $N \times N$  proximity matrix. For a numeric feature, the initial imputed values can be updated using a weighted mean over the non-missing values where the weights are given by the proximities. For categorical variables, the imputed values are updated using the most frequent non-missing values where frequency is weighted by the proximities. Then just iterate until some convergence criterion is met (typically 4–6 runs). In other words, this method just imputes missing values using a weighted mean/mode with more weight on non-missing cases.

Breiman [2002] noted that the OOB estimate of error in RFs tends to be overly optimistic when fit to training data that has been imputed. As with proximity-based outlier detection, this approach to imputation does not scale well (especially since it requires fitting multiple RFs and computing proximities). Further, this imputation method is often not as accurate as more contemporary techniques, like those implemented in the R package **mice** [van Buuren and Groothuis-Oudshoorn, 2021].

Perhaps the biggest drawback to proximity-based imputation, like many other imputation methods, is that it only generates a single completed data set. As discussed in van Buuren [2018, Chap. 1], our level of confidence in a particular imputed value can be expressed as the variation across a number of completed data sets. In Section 7.9.3, I'll use the CART-based multiple imputation procedure discussed in Section 2.7.1 and show how we can have confidence in the interpretation of the RF output by incorporating the variability associated with multiple imputation runs.

### 7.6.3 Unsupervised random forests

As it turns out, RFs can be used in unsupervised settings as well (i.e., when there is no defined response variable). In this case, the goal is to cluster the data, that is, see if the rows from the learning sample form any ‘interesting’ groups.

In an unsupervised RF, the idea is formulate a two-class problem. The first class corresponds to the original data, while the second class corresponds to a synthetic data set generated from the original sample. There are two ways to generate the synthetic data corresponding to the second class [Liaw and Wiener, 2002]:

- 1) a bootstrap sample is generated from each predictor column of the original data;
- 2) a random sample is generated uniformly from the range of each predictor column of the original data.

These two data sets are then stacked on top of each other, and an ordinary RF is used to build a binary classifier to try and distinguish between the real and synthetic data. (A necessary drawback here is that the resulting data set is twice as large as the original learning sample.) If the OOB misclassification error rate in the new two-class problem is, say,  $\geq 40\%$ , then the columns look too much like independent variables in the eyes of the RF; in other words, the dependencies among the columns do not play a large role in discriminating between the two classes. On the other hand, if the OOB misclassification rate is lower, then the dependencies are playing an important role. If there is some discrimination between the two classes, then the resulting proximity matrix

can be used as an input into any distance-based clustering algorithm (like  $k$ -means or hierarchical clustering).

### 7.6.3.1 Example: Swiss banknotes

Continuing with the Swiss banknote example, I generated a synthetic version of the data set using the bootstrap approach outlined in the previous section, and then stacked the data together into a two-class problem:

```
bn <- treemisc::banknote
X.original <- subset(bn, select = -y) # features only
X.synthetic <- X.original
set.seed(1034)
for (i in seq_len(ncol(X.original))) {
  X.synthetic[[i]] <- sample(X.synthetic[[i]], replace = TRUE)
}
X <- rbind(X.original, X.synthetic)

# Add binary indicator (doesn't)
X$y <- rep(c("original", "synthetic"), each = nrow(bn))
```

I then fit an RF of 1000 trees using the newly created binary indicator  $y$  and generated proximities for the original (i.e., first 200) observations. So how well did the unsupervised RF cluster the data? Well, we could convert the proximity matrix into a dissimilarity matrix and feed it into any distance-based clustering algorithm. Another approach, which I'll take here, is to visualize the dissimilarities using *multidimensional scaling* (MDS). MDS is one of many methods for displaying (transformed) multidimensional data in a lower-dimensional space; for details, see Johnson and Wichern [2007, Sec. 12.6]. Essentially, MDS takes a set of dissimilarities—one minus the proximities, in this case—and returns a set of points such that the distances between the points are approximately equal to the dissimilarities. Figure 7.10 shows the best-fitting two-dimensional representation. Here you can see a clear separation between the genuine bills (black) and counterfeit bills (yellow).

### 7.6.4 Case-specific random forests

The *case-specific RF* [Xu et al., 2016] is another interesting application of RF proximities (Section 7.6). The idea is to build a new RF to more accurately predict each individual observation in the test set. The individual RFs give more weight to the training observations that have higher proximity to the observations in the test set.

Let  $\mathbf{d}_{trn}$  and  $\mathbf{d}_{tst}$  be the train and test data sets with  $N$  and  $N_{tst}$  observations, respectively. The general steps for growing a case-specific RF are outlined in Algorithm 7.3 below:

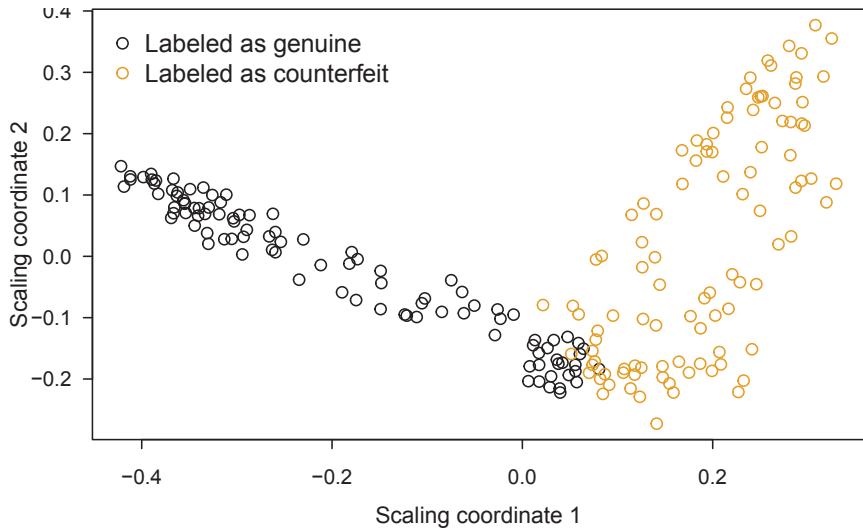


FIGURE 7.10: MDS coordinates of the proximities from an unsupervised RF fit to the Swiss banknote data. As can be seen, there are two noticeable clusters, although not perfectly separated. The genuine banknotes (black circles) generally fall into one cluster, while the counterfeit banknotes (yellow circles) tend to fall in the other.

---

**Algorithm 7.3** Case-specific random forest algorithm.

---

- 1) Grow an ordinary RF of size  $B$  to the training data  $\mathbf{d}_{trn}$ .
  - 2) For each observation in the test set, say,  $\mathbf{x}_0$ , do the following:
    - a) Compute the proximities between  $\mathbf{x}_0$  and each observation in  $\mathbf{d}_{trn}$  using the initial RF from Step 1); let  $\{\text{prox}_i(\mathbf{x}_0)\}_{i=1}^N$  be the proximities between  $\mathbf{x}_0$  and each case from  $\mathbf{d}_{trn}$ ; that is, the fraction of times  $\mathbf{x}_0$  cohabitates with each training instance across the  $B$  trees in the initial RF.
    - b) Define the case weight for training case  $i$  relative to  $\mathbf{x}_0$  as  $w_i^* = \text{prox}_i(\mathbf{x}_0) / \sum_{i=1}^N \text{prox}_i(\mathbf{x}_0)$ .
    - c) Predict  $\mathbf{x}_0$  with a new RF grown to  $\mathbf{d}_{trn}$  using case weights  $\{w_i^*\}_{i=1}^N$ .
- 

In essence, a case-specific RF predicts a new case  $\mathbf{x}_0$  using a new RF that gives more weight to the original training observations that have higher

proximity (i.e., are more similar) to  $\mathbf{x}_0$ . Note that most open source RF software provide the option to specify case weights for the training observations, which are used to weight each row when taking bootstrap samples (but many implementations do not provide proximities). While the idea of case-specific RFs makes sense, it has a couple of limitations. First off, it requires fitting  $N_{tst} + 1$  RFs, which can be expensive whenever  $N$  or  $N_{tst}$  are large. Second, it requires computing  $N \times N_{tst}$  proximities from an RF, which aren't always available from software.

Case-specific RFs are relatively straightforward to implement with traditional RF software, provided you can compute proximity scores<sup>f</sup>. The R package **ranger** provides an implementation of case-specific RFs. I applied this methodology to the Ames housing example (Section 1.4.7), which actually resulted in a slight increase to the test RMSE when compared to a traditional RF; the code to reproduce the example is available on the companion website for this book.

---

## 7.7 Prediction standard errors

Using a similar technique to OOB error estimation, Wager et al. [2014] proposed a method for estimating the variance of an RF prediction using a technique called the *jackknife*. The jackknife procedure is very similar to LOOCV, but specifically used for estimating the variance of a statistic of interest. If we have a statistic,  $\hat{\theta}$ , estimated from  $N$  training records, then the jackknife estimate of the variance of  $\hat{\theta}$  is given by:

$$\hat{V}_{jack}(\hat{\theta}) = \frac{N-1}{N} \sum_{i=1}^N (\hat{\theta}_{(i)} - \hat{\theta}_{(.)})^2, \quad (7.4)$$

where  $\hat{\theta}_{(i)}$  is the statistic of interest using all the  $N$  training observations except observation  $i$ , and  $\hat{\theta}_{(.)} = \sum_{i=1}^N \hat{\theta}_{(i)}/N$ .

For brevity, let  $\hat{f}(\mathbf{x}) = \hat{f}_B^{rf}(\mathbf{x})$ , for some arbitrary observation  $\mathbf{x}$  (see Algorithm 7.1). A natural jackknife variance estimate for the RF prediction  $\hat{f}(\mathbf{x})$  is given by

---

<sup>f</sup>Even if you don't have access to an implementation of RFs that can compute proximities, they're still obtainable as long as you can compute terminal node assignments for new observations (i.e., compute which terminal node a particular observation falls in for each of the  $B$  trees), which is readily available in most RF software. See Section 7.6 for details.

$$\hat{V}_{jack}(\hat{f}(\mathbf{x})) = \frac{N-1}{N} \sum_{i=1}^N \left( \hat{f}_{(i)}(\mathbf{x}) - \hat{f}(\mathbf{x}) \right)^2. \quad (7.5)$$

This is derived under the assumption that  $B = \infty$  trees were averaged together in the forest, which, of course, is never the case. Consequently, (7.5) has a positive bias. Fortunately, the same  $B$  bootstrap samples used to derive the forest can also be used to provide the bias corrected variance estimate

$$\hat{V}_{jack}^{BC}(\hat{f}(\mathbf{x})) = \hat{V}_{jack}(\hat{f}(\mathbf{x})) - (e-1) \frac{N}{B} \hat{v}(\mathbf{x}), \quad (7.6)$$

where  $e = 2.718\dots$  is Euler's constant and

$$\hat{v}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^B \left( \hat{f}_b(X) - \hat{f}(\mathbf{x}) \right)^2 \quad (7.7)$$

is the bootstrap estimate of the variance of a prediction from a single RF tree. Fortunately, all of the required quantities for computing (7.6) are readily available in the output from most open source RF software. This procedure is implemented in the R packages **ranger** and **grf** [Tibshirani et al., 2021]; the latter is a pluggable package for nonparametric statistical estimation and inference based on RFs, also known as *generalized random forests* [Athey et al., 2019]. The **forestci** package [Polimis et al., 2017] provides a Python implementation compatible with scikit-learn RF objects.

Once the estimated standard error of a prediction  $\hat{y}$  is obtained, it can be useful to summarize it using a Gaussian-based confidence interval of the form  $\hat{y} \pm z_\alpha \hat{\sigma}$ , where  $z_\alpha$  is a quantile from a standard normal distribution and  $\hat{\sigma}$  is the estimated standard error of  $\hat{y}$ ; see Wager et al. [2014] for details. Zhang et al. [2020] further discuss the use of confidence/prediction intervals for RF predictions using several methods, including *split conformal prediction* [Lei et al., 2018] and *quantile regression forests* (Section 7.8.2).

### 7.7.1 Example: predicting email spam

Switching back to the email spam data, let's compute jackknife-based standard errors for the test set predicted class probabilities. Following Wager et al. [2014], I fit an RF using  $B = 20,000$  trees and three different values for  $m_{try}$ : 5, 19 (based on Breiman's default for classification), and 57 (an ordinary bagged tree ensemble).

The predicted class probabilities for `type = "spam"`, based on the test data, from each RF are displayed in Figure 7.11 (*x*-axis), along with their bias-corrected jackknife estimated standard errors (*y*-axis). Notice how the misclassified cases (solid black points) tend to correspond to observations where the predicted class probability is closer to 0.5. It also appears that the more constrained RF with  $m_{try} = 5$  produced smaller standard errors, while the default RF ( $m_{try} = 19$ ) and bagged tree ensemble ( $m_{try} = 57$ ) produced noticeably larger standard errors, with the bagged tree ensemble performing the worst.

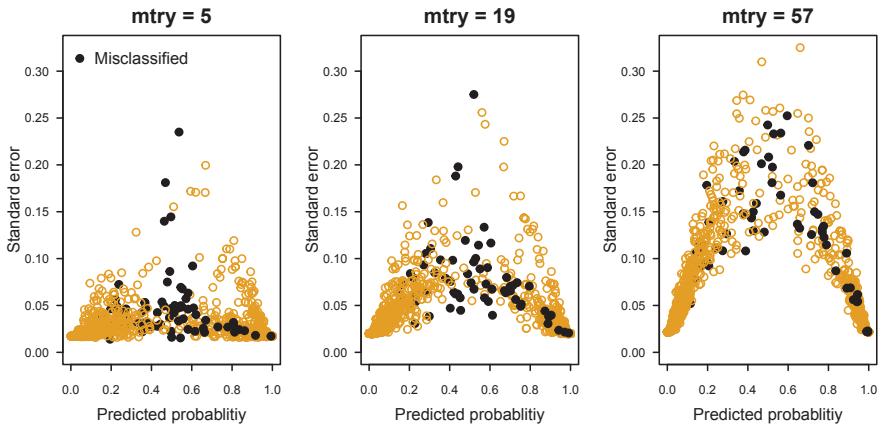


FIGURE 7.11: Bias-corrected jackknife variance estimates for the predicted probabilities using the spam training data. The solid black points correspond to the misclassified cases which appear to be more closely concentrated near the center (i.e., 0.5).

## 7.8 Random forest extensions

The following subsections highlight some notable extensions to the RF algorithm. Note that this is by no means an exhaustive list, so I tried to choose extensions that have been shown to be practically useful and that also have available (and currently maintained) open source implementations.

### 7.8.1 Oblique random forests

As briefly mentioned in Section 4.3.1, linear (or oblique) splits—which are splits based on linear combinations of the predictors—can sometimes improve

predictive accuracy, even if they do make the tree harder to interpret. Many decision tree algorithms support linear splits (e.g., CART and GUIDE) and Breiman [2001] even proposed a variant of RFs that employed linear splits based on random linear combinations of the predictors. This approach did not gain the same traction as the traditional RF algorithm based on univariate splits. In fact, I'm not aware of any open source RF implementations that support his original approach based on random coefficients.

Menze et al. [2011] proposed a variant, called *oblique random forests* (ORFs), that explicitly learned optimal split directions at internal nodes using linear discriminative models<sup>g</sup>, as opposed to random linear combinations. Similar to *random rotation ensembles* (Section 7.8.3), ORFs tend to have a smoother topology; see, for example, Figure 7.16. Menze et al. [2011] even go as far as to recommend the use of ORFs over the traditional RF when applied to mostly numeric features. Nonetheless, the idea of non-axis oriented splits in RFs has still not caught on. The only open source implementation of ORFs that I'm aware of is in the R package **obliqueRF** [Menze and Splitthoff, 2012], which has not been updated since 2012.

A more recent approach, called *projection pursuit random forest* (PPforest) [da Silva et al., 2021a] uses splits based on linear combinations of randomly chosen inputs. Each linear combination is found by optimizing a projection pursuit index [Friedman and Tukey, 1974] to get a projection of the features that best separates the classes; hence, this method is also only suitable for classification. PPforests are implemented in the R package **PPforest** [da Silva et al., 2021b]. Individual *projection pursuit trees* (PPtrees) [Lee et al., 2013], which are used as the base learners in a PPforest, can be fit using the R package **PPtreeViz** [Lee, 2019] (which seems to have superseded the older **PPtree** package).

## 7.8.2 Quantile regression forests

The goal of many supervised learning algorithms is to infer something about the relationship between the response and a set of predictors. In regression, for example, the goal is often to estimate the conditional mean  $E(Y|\mathbf{x})$ , for some observation  $\mathbf{x}$ . In a typical regression tree, an estimate of the conditional mean is given by the mean response associated with the terminal node observation  $\mathbf{x}$  falls into. In an RF, the terminal node means are simply averaged across all the trees in the forest.

The conditional mean response, however, provides only a limited summary of the conditional distribution function  $F(y|\mathbf{x}) = \Pr(Y \leq y|\mathbf{x})$ . Denote the

---

<sup>g</sup>Similar to the LDA-based approach used in GUIDE (Section 4.3.1). However, GUIDE restricts itself to linear splits in only two features at a time to help with interpretation and reduce the impact of missing values.

$\alpha$ -quantile of  $Y|\mathbf{x}$  as  $Q_\alpha(\mathbf{x})$ . In other words,  $\Pr(Y \leq y|Q_\alpha(\mathbf{x})) = \alpha$ . Compared to the conditional mean, the quantiles give a more useful summary of the distribution. For example,  $\alpha = 0.5$  corresponds to the median. If the conditional distribution of  $Y|\mathbf{x}$  were symmetric, then the conditional mean and median would be the same. However, if  $Y|\mathbf{x}$  is skewed, then, compared to the conditional median, the conditional mean can be a misleading summary of what a typical value of  $Y|\mathbf{x}$  is. Furthermore, estimating  $\Pr(Y \leq y|Q_\alpha(\mathbf{x}))$  for various values of  $\alpha$  can give insight into the variability around a single point estimate, like the conditional median. This is the idea behind *quantile regression*.

The same idea can be applied to an RF, and was formerly introduced in Meinshausen [2006] as *quantile regression forests* (QRF). In a QRF, the conditional distribution of  $Y|\mathbf{x}$  is approximated by the weighted mean

$$\hat{F}(y|\mathbf{x}) = \sum_{i=1}^N w_i(\mathbf{x}) I(Y_i \leq y),$$

where  $I(expression)$  is the indicator function that evaluates to one whenever *expression* is true, and zero otherwise. The weights  $w_i(\mathbf{x})$  are estimated from the terminal node observations across all the trees in the forest and are defined in Meinshausen [2006]. In contrast to a traditional regression forest, this requires storing all the observations in each node, as opposed to just the mean.

As it turns out, there's not much difference between QRFs and RFs, aside from what information gets stored from each tree and how fitted values and predictions are obtained. RFs only need to keep track of the terminal node means. To estimate the full conditional distribution of  $Y|\mathbf{x}$ , QRFs need to retain all observations across all terminal nodes.

### 7.8.2.1 Example: predicting home prices (with prediction intervals)

To illustrate, I fit a traditional RF and a QRF to the Ames housing data using the same train/test split discussed in [Section 1.4.7](#); similar to before, I used `Sale_Price/1000` as the response. For each house in the test set, the predicted 0.025, 0.5, and 0.975 quantiles were obtained (which corresponds to a predicted median sale price and 95% prediction interval). Following the Boston housing example in Meinshausen [2006], the test set observations were ordered according to the length of the corresponding prediction intervals, and each observation was centered by subtracting the midpoint from the corresponding prediction interval. The observed sale prices are shown in black and the estimated conditional medians are given in yellow. The black lines

show the corresponding 95% prediction bounds. (Note that the prediction intervals here are pointwise prediction intervals.)

The most expensive house in the test set sold for a (rescaled) sale price of \$610. A traditional RF estimated a conditional mean sale price of \$472.59, whereas the QRF produced a conditional median sale price of \$479.07 with a 0.025 quantile of \$255.24 and a 0.975 quantile of \$745. Here, the QRF gives a much better sense of the variability in the predicted outcome, as well as a sense of the skewness of its distribution.

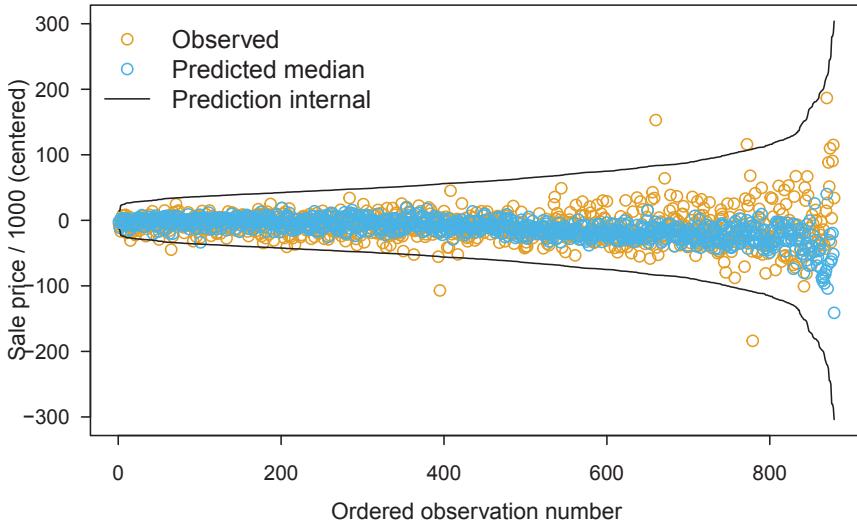


FIGURE 7.12: Rescaled sale prices for each home in the test set, along with the predicted 0.025, 0.5, and 0.975 quantiles from a QRF. To enhance visualization, the observations were ordered according to the length of the corresponding prediction intervals, and the mean of the upper and lower end of the prediction interval is subtracted from all observations and prediction intervals.

### 7.8.3 Rotation forests and random rotation forests

Before talking about *rotation forests* and random rotation ensembles, it might help to briefly discuss *rotation matrices*. A rotation matrix  $\mathbf{R}$  of dimension  $p$  is a  $p \times p$  square transformation matrix that's used to perform a rotation in  $N$ -dimensional Euclidean space.

A common application of rotation matrices in statistics is *principal component analysis* (PCA). The details of PCA are beyond the scope of this book, but the interested reader is pointed to Johnson and Wichern [2007, Chap. 8], among others. While PCA has many use cases, it is really just an unsupervised

dimension reduction technique that seeks to explain the variance-covariance structure of a set of variables through a few linear combinations of these variables.

To illustrate, consider the  $N = 100$  data points shown in Figure 7.13 (left); the axes for  $x_1$  and  $x_2$  are shown using dashed yellow and blue lines, respectively. The data were generated from a simple linear regression defined by

$$X_{2i} = X_{1i} + \epsilon_i, \quad i = 1, 2, \dots, 100, \quad (7.8)$$

where  $X_{1i} \stackrel{iid}{\sim} \mathcal{U}(0, 1)$  and  $\epsilon_i \stackrel{iid}{\sim} N(0, 1)$ . Further, let  $\mathbf{X}$  be the  $100 \times 2$  matrix whose first and second columns are given by  $X_{1i}$  and  $X_{2i}$ , respectively. As a rotation in two dimensions, PCA finds the rotation of the axes that yields maximum variance. The rotated axes for this example are shown in Figure 7.13 (middle). Notice that the first (i.e., yellow) axis is aligned with the direction of maximum variance in the sample. An alternative would be to rotate the data points themselves (right side of Figure 7.13). In this case, the variable loadings from PCA form a  $2 \times 2$  rotation matrix,  $\mathbf{R}$ , that can be used to rotate  $\mathbf{X}$  so that the direction of maximal variance aligns with the first (i.e., yellow) axis; this is shown in the right side of Figure 7.13. The rotated matrix is given by  $\mathbf{X}' = \mathbf{X}\mathbf{R}$ . Notice how the relative position of the points between  $x_1$  and  $x_2$  is preserved, albeit rotated about the axes.

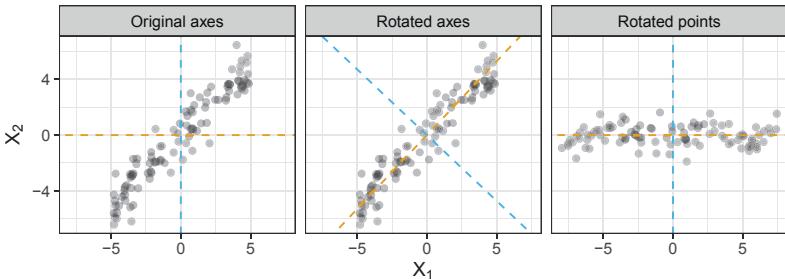


FIGURE 7.13: Data generated from a simple linear regression model. Left: Original data points and axes. Middle: Original data with rotated axes (notice how the first/yellow axis aligns with the direction of maximum variance in the sample). Right: Rotated data points on the original axes (here the data are rotated so that the direction of maximal variance aligns with the first/yellow axis).

So what does any of this have to do with RFs? Recall that the key to accuracy with model averaging is diversity. In an RF, diversity is achieved by choosing a random subset of predictors prior to each split in every tree. A rotation forest [Rodríguez et al., 2006], on the other hand, introduces diversity to a bagged tree ensemble by using PCA to construct a rotated feature space prior

to the construction of each tree. Rotating the feature space allows adaptive nonparametric learning algorithms, like decision trees, to learn potentially interesting patterns in the data that might have gone unnoticed in the original feature space. Applying PCA to all the predictors prior to the construction of each tree, even when using sampling with replacement, won't be enough to diversify the ensemble. Instead, prior to the construction of each tree, the predictor set is randomly split into  $K$  subsets, PCA is run separately on each, and a new set of linearly extracted features is constructed by pooling all the principal components (i.e., the rotated data points).  $K$  is treated as a tuning parameter, but the value of  $K$  that results in roughly three features per subset seems to be the suggested default [Kuncheva and Rodríguez, 2007]. Rotation forests can be thought of as a bagged tree ensemble with a random feature transformation applied to the predictors prior to constructing each tree. In this sense, PCA can be thought of as a feature extraction method. By performing PCA on random subsets of features prior to fitting each tree, rotation forests can improve the performance of a bagged tree ensemble. In this case, the derived features come from PCA applied to random subsets of the data, and while other feature extraction methods have also been considered, PCA was found to be the most suitable [Kuncheva and Rodríguez, 2007].

Rotation forests have been shown to be competitive with RFs and can achieve better performance on data sets with mostly quantitative variables; although, this seems to be true mostly for smaller ensemble sizes [Rodríguez et al., 2006, Kuncheva and Rodríguez, 2007]. However, most comparative studies I've seen seem to focus on classification accuracy for comparison, which we know is not the most appropriate metric for comparing models in classification settings. Rotation forests are available in the R package **rotationForest** [Ballings and Van den Poel, 2017].

### 7.8.3.1 Random rotation forests

A similar approach, called random rotation ensembles [Blaser and Fryzlewicz, 2016], applies a random rotation to all the features prior to constructing each tree. Blaser and Fryzlewicz [2016] discuss two algorithms for generating random rotation matrices and provide general R and C++ code for doing so. The **treemisc** function **rrm()** uses the *indirect method* discussed in Blaser and Fryzlewicz [2016] and is shown below. Note that rotations are only applied to numeric features and can be sensitive to both scale and outliers, hence, rescaling the numeric features is often required; see Blaser and Fryzlewicz [2016] for several recommendations. Random rotation forests are not generally available in open source software, but you can find my poor man's implementation in package **treemisc**, which I'll demonstrate in the next section; see **?treemisc::rforest** for details.

Let  $\mathbf{X}_c$  be the subset of numeric/continuous features from the full feature set  $\mathbf{X}$ . In a random rotation forest, before fitting the  $b$ -th tree, the numeric features are randomly transformed using  $\mathbf{X}_{c,b} = \mathbf{X}_c \mathbf{R}_b$  (for  $b = 1, 2, \dots, B$ ), where  $\mathbf{R}_b$  is a randomly generated rotation matrix of dimension equal to the number of columns of  $\mathbf{X}_c$ .

```
treemisc::rrm

#> function(n) {
#>   QR <- qr(matrix(rnorm(n ^ 2), ncol = n)) # A = QR
#>   M <- qr.Q(QR) %*% diag(sign(diag(qr.R(QR))))
#>   if (det(M) < 0) M[, 1L] <- -M[, 1L] # det(M) = +1
#>   M
#> }
#> <bytecode: 0x7ff296e1fe90>
#> <environment: namespace:treemisc>
```

To illustrate the effect of applying random rotations to a set of features, let's continue with the simulated data from the previous section. In the code chunk below, I re-generate the same  $N = 100$  points from (7.8); the original data are displayed in [Figure 7.14](#) (black points), along with the observations under various random rotations, including PCA (orange points). Such rotations preserve the inter-relationships between predictors, but cast them into a different space resulting in equally accurate, but more diverse trees.

```
set.seed(1038)
X1 <- runif(100, min = -5, max = 5)
X2 <- X1 + rnorm(length(X1))
X <- cbind(X1, X2)
palette("Okabe-Ito") # colorblind-friendly color palette
plot(X, xlim = c(-8, 8), ylim = c(-8, 8), col = 1, las = 1,
     xlab = expression(x[1]), ylab = expression(x[2]))
pcR <- loadings(princomp(X, cor = FALSE, fix_sign = FALSE)) # PCA
points(X %*% pcR, col = 2) # plot PCA rotation
abline(0, 1, lty = 2, col = 1) # original axis
abline(h = 0, lty = 2, col = 2) # axis after PCA rotation
for (i in 3:5) { # plot random rotations
  R <- treemisc::rrm(2) # generate a random 2x2 rotation matrix
  points(X %*% R, col = adjustcolor(i, alpha.f = 0.5))
}
legend("topleft", legend = "Original sample", pch = 1, col = 1,
       inset = 0.01, bty = "n")
palette("default")
```

### 7.8.3.2 Example: Gaussian mixture data

In this section, I'll use the Gaussian mixture data from Hastie et al. [2009] to compare the results of an RF, rotation forest, and random rotation forest. The data for each class come from a mixture of ten normal distributions,

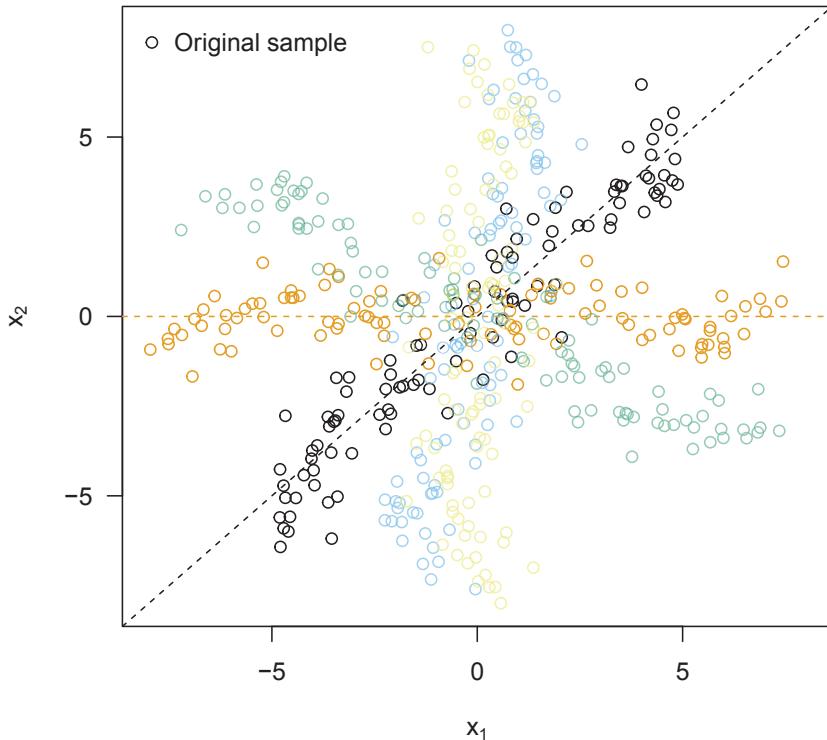


FIGURE 7.14: Scatterplot of  $x_1$  vs.  $x_2$ . The original data are shown in black and a dashed black line gives the direction of maximum variance. The rotated points under PCA are shown in dark yellow along with the new axis of maximal variance; notice how in two dimensions this shifts the points to have maximal variance along the  $x$ -axis. The rest of the colors display the data points under random rotations.

whose individual means are also normally distributed. A full description of the data generating process can be found in Hastie et al. [2009, Sec. 2.3.3] and an application to RFs is provided in Hastie et al. [2009, Sec. 15.4.3]. The raw data are available at <https://web.stanford.edu/~hastie/ElemStatLearn/data.html>. For convenience, the data are also available in the **treemisc** R package that accompanies this book, and can be read in using:

```
library(treemisc)
```

```

eslmix <- load_eslmix()
class(eslmix) # should be a list
#> [1] "list"
names(eslmix) # names of components
#> [1] "x"         "y"         "xnew"      "prob"
#> [5] "marginal"  "px1"      "px2"      "means"

```

Note that this is not a data frame, but rather a list with several components; for a description of each, see [?treemisc::load\\_eslmix](#).

The code chunk below constructs a scatterplot of the training data (i.e., component `x`) along with the Bayes decision boundary<sup>h</sup>; see [Figure 7.15](#). The Bayes error rate for these data—that is, the theoretically optimal error rate—is 0.210.

```

x <- as.data.frame(eslmix$x) # training data
xnew <- as.data.frame(eslmix$xnew) # evenly spaced grid of points
x$y <- as.factor(eslmix$y) # coerce to factor for plotting
xnew$prob <- eslmix$prob # Pr(Y = 1 | xnew)

# Colorblind-friendly palette
oi.cols <- unname(palette.colors(8, palette = "Okabe-Ito"))

# Construct scatterplot of training points
p <- ggplot(x, aes(x = x1, y = x2, color = y)) +
  geom_point(alpha = 1, show.legend = FALSE) +
  scale_colour_manual(values = oi.cols) +
  theme_bw()

# Add optimal (i.e., Bayes) decision boundary
p + geom_contour(data = xnew, aes(x = x1, y = x2, z = prob),
                  breaks = 0.5, color = oi.cols[4],
                  inherit.aes = FALSE, linetype = 2)

```

Next I fit three tree-based ensembles: a traditional RF, a rotation forest, and a random rotation forest. The rotation forest was fit using the **rotationForest** package, while the RF and random rotation forest were fit using **treemisc**'s **rforest()** function. Note that this is a poor man's implementation of Breiman's RF algorithm I wrote that optionally rotates the features at random prior to the construction of each tree. It is based on the well-known **randomForest** package [Breiman et al., 2018] and is missing many of the bells and whistles, hence not recommended for general use, but it works. Also, this

---

<sup>h</sup>As noted in Hastie et al. [2009, [Chap. 2](#)], since the data generating mechanism is known for each of the two classes, the theoretically optimal decision boundary can be computed exactly. This makes it useful to compare classifiers visually in terms of their estimated decision boundaries.

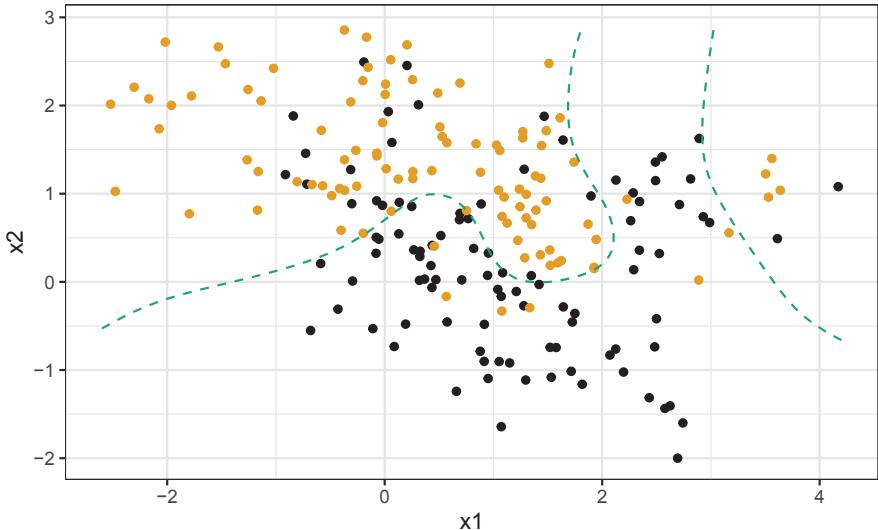


FIGURE 7.15: Simulated mixture data with optimal (i.e., Bayes) decision boundary.

implementation only uses regression trees for the base learners, hence, only regression and binary classification are supported. For the latter, the probability machine approach discussed in [Section 7.2.1](#) is implemented.

The resulting decision boundaries from each forest are in [Figure 7.16](#). Here, you can see that the axis-oriented nature of the individual trees in a traditional RF leads to a decision boundary with an axis-oriented flavor (i.e., the decision boundary is rather “boxy”). The RF also exhibits more signs of overfitting, as suggested by the little islands of decision boundaries. On the other hand, using feature rotation (with PCA or random rotations) prior to building each tree results in a noticeably smoother and non-axis-oriented decision boundary. The test error rates for the RF, rotation forest, and random rotation forest, under this random seed, are 0.235, 0.239, and 0.226, respectively. (As always, the code to reproduce this example is available on the companion website for this book.)

#### 7.8.4 Extremely randomized trees

Just as RFs offer an additional layer of randomization to bagged decision trees, *extremely randomized trees* (or extra-trees) [Geurts et al., 2006] are essentially an RF with an additional randomization step. In particular, the split point for any feature at each node in a tree is essentially selected at random from a uniform distribution. The extra randomization can further

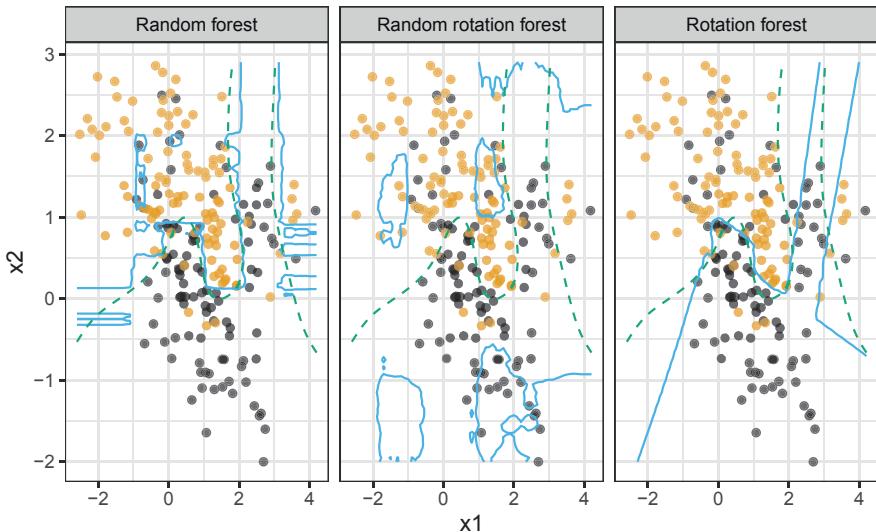


FIGURE 7.16: Traditional RF vs. random rotation forest on the mixture data from Hastie et al. [2009]. The random rotation forest produces a noticeably smoother decision boundary than the axis-oriented decision boundary from a traditional RF.

decrease variance, but sometimes at the cost of additional bias [Geurts et al., 2006]; this is especially true if the data contain irrelevant features. To combat the extra bias, extra-trees utilize the full learning sample to grow each tree, rather than bootstrap replicas (another subtle difference from the bagging and RF algorithms). Note that bootstrap sampling can be used in extra-trees ensembles, but Geurts et al. argue that it can often lead significant drops in accuracy<sup>i</sup>.

The primary tuning parameters for an extra-trees ensemble are  $K$  and  $n_{min}$ , where  $K$  is the number of random splits to consider for each candidate splitter, and  $n_{min}$  is the minimum node size, which is a common parameter in many tree-based models and can act as a smoothing parameter. A common default for  $K$  is

$$K = \begin{cases} \sqrt{p} & \text{for classification} \\ p & \text{for regression} \end{cases},$$

where  $p$  is the number of available predictors. The optimal split is chosen from the sample of  $K$  splits in the usual way (i.e., the split that results in the

---

<sup>i</sup>Most open source implementations of extra-trees optionally allow for bootstrap sampling.

largest reduction in node impurity). Note that  $n_{min}$  has the same defaults as it does in an RF (Section 7.2).

The extra-trees ensemble still makes use of the RF  $m_{try}$  parameter, but note that, in the extreme case where  $K = 1$ , an extra-trees tree is unsupervised in that the response variable is not needed in determining any of the splits. Such a *totally randomized tree* [Geurts et al., 2006] can be useful in detecting potential outliers and anomalies, as will be discussed in Section 7.8.5. Extra-trees can be fit in R via the **ranger** package. In Python, an implementation of the extra-trees algorithm is provided by the **sklearn.ensemble** module.

### 7.8.5 Anomaly detection with isolation forests

While the proximities of an RF can be used to detect novelties and potential outliers, they’re rather computationally expensive to compute and store, especially for large data sets; also, as previously mentioned, many RF implementations do not support proximities. A more general approach to anomaly detection, called an *isolation forest*, was proposed in Liu et al. [2008]. An isolation forest is essentially an ensemble of *isolation trees* (IsoTrees). IsoTrees are similar to extra-trees with  $K = 1$  (Section 7.8.4), except that the splitting variables are also chosen at random; hence, IsoTrees are unsupervised in the sense that the tree building process does not make use of any response variable information.

So how does it work? Isolation forests are quite simple actually. The core idea is to “isolate” anomalous observations, rather than creating a profile for “normal” ones—the latter seems to be the more common approach taken by other methods in practice. Isolation forests assume that

- 1) anomalies are rare in comparison to “normal” observations;
- 2) anomalies differ from “normal” instances in terms of the values of their features.

In other words, isolation forests assume anomalies are “few and different” [Liu et al., 2008]. If anomalies are “few and different”, then they are susceptible to *isolation* (i.e., easy to separate from the rest of the observations).

In an IsoTree, observations are recursively separated until all instances are isolated to their own terminal node; since the split variables and split points are determined completely at random, no response information is needed. Anomalous observations tend to be easier to isolate with fewer random partitions compared to normal instances. That is to say, the relatively few instances of anomalies tend to have shorter path lengths in an IsoTree when compared to normal observations. The path length to each observation can be computed for a forest of independently grown trees and aggregated into a single *anomaly score*. The anomaly score for an arbitrary observation  $\mathbf{x}$  is given by

$$s(\mathbf{x}, N) = 2^{-\frac{1}{B} \sum_{b=1}^B h_b(\mathbf{x})/c(N)}, \quad (7.9)$$

where  $N$  is the sample size,  $h_b(\mathbf{x})$  is the path length to  $\mathbf{x}$  in the  $b$ -th tree, and  $c(N)$  is the *average path length of unsuccessful searches*; in a binary tree constructed from  $N$  observations,  $c(N)$  is given by

$$c(N) = 2H(N - 1) - 2(N - 1)/N,$$

where  $H(i)$  is the  $i$ -th *harmonic number* (<https://mathworld.wolfram.com/HarmonicNumber.html>).

Let  $\bar{h}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^B h_b(\mathbf{x})$  be the average path length for instance  $\mathbf{x}$  across all trees in the forest, and note that  $0 < s(\mathbf{x}, N) \leq 1$  and  $0 < \bar{h}(\mathbf{x}) \leq N - 1$ . A few useful relationships regarding (7.9) are worth noting:

- $s(\mathbf{x}, N) \rightarrow 0.5$  as  $\bar{h}(\mathbf{x}) \rightarrow c(N)$ ;
- $s(\mathbf{x}, N) \rightarrow 1$  as  $\bar{h}(\mathbf{x}) \rightarrow 0$ ;
- $s(\mathbf{x}, N) \rightarrow 0$  as  $\bar{h}(\mathbf{x}) \rightarrow N - 1$ .

Since the assumption is that anomalies are easier to isolate, they are likely to have shorter path lengths on average. Hence, any instance  $\mathbf{x}$  with values of  $s(\mathbf{x}, N)$  close to one tend to be highly anomalous. If  $s(\mathbf{x}, N)$  is much smaller than 0.5, then it is safe to regard  $\mathbf{x}$  as a “normal” instance. If all the instances return a value close to 0.5, then it is safe to say the sample does not really contain any anomalies. Note that these are just guidelines.

Isolation forests are a top-performing unsupervised method for detecting potential outliers and anomalies [Domingues et al., 2018]. Compared to other outlier detection algorithms, isolation forests are scalable (e.g., they have relatively little computational and memory requirements), fully nonparametric, and do not require a distance-like matrix. The Anti-Abuse AI Team at LinkedIn uses isolation forests to help detect abuse on LinkedIn (e.g., fake accounts, account takeovers, and profile scraping) [Verbus, 2019].

Let’s illustrate the overall idea with a simple simulated example. Consider the data in Figure 7.17 (left). Here, the points  $\{(x_{1i}, x_{2i})\}_{i=1}^{2,000}$  were independently sampled from a standard normal distribution. However, I changed the first observation to  $(x_{11}, x_{21}) = (5.5, 5.5)$  (purple point), which is quite anomalous compared to the rest of the sample.

A single IsoTree from an isolation forest is displayed in Figure 7.17 (right), along with the path taken by  $\mathbf{x}$  (in purple). Here  $\mathbf{x}$  has a path length of two and you can see that the path  $\mathbf{x}$  takes in the tree is relatively shorter than most of the other available paths.

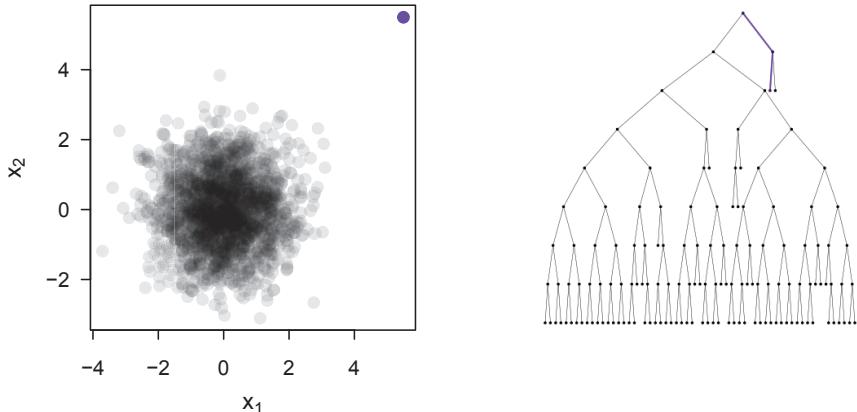


FIGURE 7.17: Isolation tree example. Left: scatterplot of two independent samples from a standard normal distribution; the observation with coordinates (5.5, 5.5) is a clear anomaly. Right: isolation tree diagram; the branch isolating the purple point is also highlighted in purple.

#### 7.8.5.1 Extended isolation forests

An *extended isolation forest* [Hariri et al., 2021] improves the consistency and reliability of the anomaly score produced by a standard isolation forest by using random oblique splits (in this case, hyperplanes with random slopes)—as opposed to axis-oriented splits—which often results in improved anomaly scores. The tree in Figure 7.17 is from an extended isolation forest fit using the `eif` package [Hariri et al., 2021], which is also available in R.

#### 7.8.5.2 Example: detecting credit card fraud

To illustrate the basic use of an isolation forest, I'll use a data set from Kaggle<sup>j</sup> containing anonymized credit card transactions, labeled as fraudulent or genuine, obtained over a 48 hour period in September of 2013; the data can be downloaded from Kaggle at

<https://www.kaggle.com/mlg-ulb/creditcardfraud>.

Recognizing fraudulent credit card transactions is an important task for credit card companies to ensure that customers are not charged for items that they did not purchase. Since fraudulent transactions are relatively rare (as one

<sup>j</sup>Kaggle is an online community of data scientists and machine learning practitioners who can find and publish data sets and enter competitions to solve data science challenges; for more, visit <https://www.kaggle.com/>.

would hope), the data are highly imbalanced, with 492 frauds (0.17%) out of the  $N = 284,807$  transactions.

For reasons of confidentiality, the original features have been transformed using PCA, resulting in 28 numeric features labeled V1, V2, ..., V28. Two additional variables, **Time** (the number of seconds that have elapsed between each transaction and the first transaction in the data set) and **Amount** (the transaction amount), are also available. These are labeled data, with the binary outcome **Class** taking on values of 0 or 1, where a 1 represents a fraudulent transaction. While this can certainly be framed as a supervised learning problem, I'll only use the class label to measure the performance of our isolation forest-based anomaly detection, which will be unsupervised. I would argue that it is probably more often that you will be dealing with unlabeled data of this nature, as it is rather challenging to accurately label each transaction in a large database.

To start, I'll split the data into train/test samples using only  $N = 10,000$  observations (3.51%) for training; the remaining 274,807 observations (96.49%) will be used as a test set. However, before doing so, I'm going to shuffle the rows just to make sure they are in random order first. (Assume I've already read the data into a data frame called **ccfraud**.)

```
# ccfraud <- data.table::fread("some/path/to/ccfraud.csv")

# Randomly permute rows
set.seed(2117) # for reproducibility
ccfraud <- ccfraud[sample(nrow(ccfraud)), ]

# Split data into train/test sets
set.seed(2013) # for reproducibility
trn.id <- sample(nrow(ccfraud), size = 10000, replace = FALSE)
ccfraud.trn <- ccfraud[trn.id, ]
ccfraud.tst <- ccfraud[-trn.id, ]

# Check class distribution in each
proportions(table(ccfraud.trn$Class))
proportions(table(ccfraud.tst$Class))

#>
#>      0      1
#> 0.9982 0.0018
#>
#>      0      1
#> 0.99828 0.00172
```

Next, I'll use the **isotree** package [Cortes, 2022] to fit a default isolation forest to the training set and provide anomaly scores for the test set. (Notice how I exclude the true class labels (column 31) when constructing the isolation forest!)

```

library(isotree)

# Fit a default isolation forest
ccfraud.info <- isolation.forest(ccfraud.trn[, -31], nthreads = 1,
                                  seed = 2223)

# Compute anomaly scores for the test observations
head(scores <- predict(ccfraud.info, newdata = ccfraud.tst))

#>     1     2     3     4     5     6
#> 0.320 0.341 0.324 0.325 0.340 0.325

```

Although this isn't necessarily a classification problem, we can treat the anomaly scores as probabilities and construct informative graphics. While a *precision-recall (PR) curve*<sup>k</sup> could be useful here, I think a simple *cumulative lift chart*<sup>l</sup> would be more informative. Below I compute both; see Figure 7.18. Looking at the lift chart, for example, we can see that if we were to audit 5% of the highest scoring transactions in the test set, then we will have found roughly 87% of the fraudulent cases.

The PR curve doesn't look good, as you might expect after looking at the lift chart. For example, even though we can identify roughly 87% of the fraudulent transactions by looking at only 5% of the test sample, that still leaves more than 1300 non-fraudulent transactions that also have to be audited. In this example, it seems that we're not able to detect the majority of frauds without accepting a large number of false positives.

```

#cutoff <- sort(unique(scores))
# Compute precision and recall across various cutoffs
cutoff <- seq(from = min(scores), to = max(scores), length = 999)
cutoff <- c(0, cutoff)
precision <- recall <- numeric(length(cutoff))
for (i in seq_along(cutoff)) {
  yhat <- ifelse(scores >= cutoff[i], 1, 0)
  tp <- sum(yhat == 1 & ccfraud.tst$Class == 1) # true positives
  tn <- sum(yhat == 0 & ccfraud.tst$Class == 0) # true negatives
  fp <- sum(yhat == 1 & ccfraud.tst$Class == 0) # false positives
  fn <- sum(yhat == 0 & ccfraud.tst$Class == 1) # false negatives
  precision[i] <- tp / (tp + fp) # precision (or PPV)
  recall[i] <- tp / (tp + fn) # recall (or sensitivity)
}
precision <- c(precision, 0)
recall <- c(recall, 0)

```

<sup>k</sup>Precision (or *positive predictive value*) is directly proportional to the prevalence of the positive outcome. PR curves are not appropriate for case-control studies (e.g., which also includes case-control sampling—like down sampling—with imbalanced data sets) and should only be used when the true class priors are reflected in the data.

<sup>l</sup>The cumulative gains (or lift) chart shows the fraction of the overall number of cases in a given category “gained” by targeting a percentage of the total number of cases.

```

head(cbind(recall, precision))

#>      recall precision
#> [1,]      1  0.00172
#> [2,]      1  0.00172
#> [3,]      1  0.00172
#> [4,]      1  0.00172
#> [5,]      1  0.00173
#> [6,]      1  0.00173

# Compute data for lift chart
ord <- order(scores, decreasing = TRUE)
y <- ccfraud.tst$Class[ord] # order according to sorted scores
prop <- seq_along(y) / length(y)
lift <- cumsum(y) / sum(ccfraud.tst$Class) # convert to proportion
head(cbind(prop, lift))

#>      prop    lift
#> [1,] 3.64e-06 0.00000
#> [2,] 7.28e-06 0.00000
#> [3,] 1.09e-05 0.00000
#> [4,] 1.46e-05 0.00000
#> [5,] 1.82e-05 0.00000
#> [6,] 2.18e-05 0.00211

```

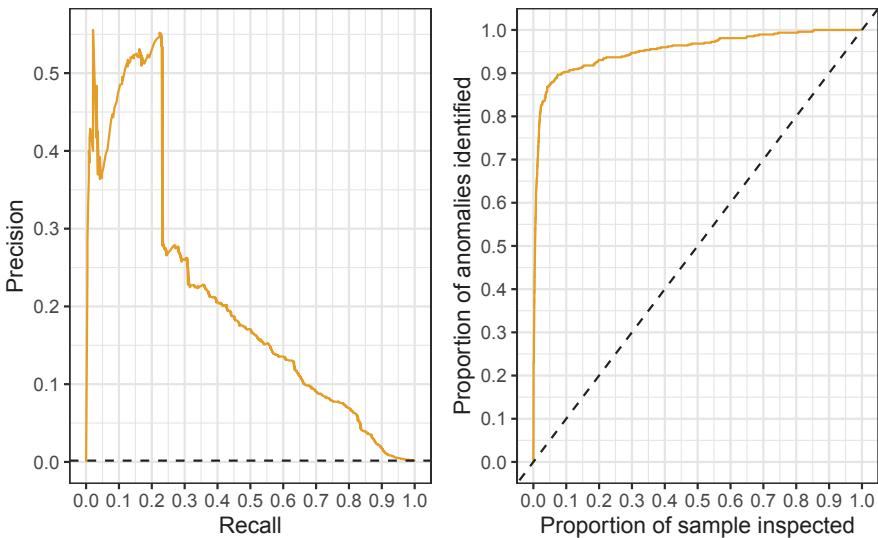


FIGURE 7.18: Precision-recall curve (left) and lift/cumulative gain chart (right) for the isolation forest applied to the credit card fraud detection test set.

We can take the analysis a step further by using Shapley values (Section 6.3.1) to help explain the observations with the highest/lowest anomaly scores, whichever is of more interest. To illustrate, let's estimate the feature contributions for the test observation with the highest anomaly score. Keep in mind that the features in this data set have been anonymized using PCA, so we won't be able to understand much of the output from a contextual perspective, but the idea applies to any application of anomaly detection based on a model that produces anomaly scores, like isolation forests. I'm just treating the scores as ordinary predictions and applying Shapley values in the usual way.

In the code chunk below, I find the observation in the test data that corresponds to the highest anomaly score. Here, we see that `max.x` corresponds to an actual instance of fraud (`Class = 1`) and was assigned an anomaly score of 0.843. The average anomaly score on the training data is 0.336, for a difference of 0.507. The question we want to try and answer is: how did each feature contribute to the difference 0.507? This is precisely the type of question that Shapley values can help with.

```
max.id <- which.max(scores) # row ID for max anomaly score
(max.x <- ccfraud.tst[max.id, ])

#>      Time    V1    V2    V3    V4    V5    V6    V7    V8
#> 1: 166198 -35.5 -31.9 -48.3 15.3 -114 73.3 121 -27.3
#>          V9 V10   V11   V12   V13   V14   V15   V16   V17
#> 1: -3.87 -12  6.85 -9.19 7.13 -6.8  8.88 17.3 -7.17
#>          V18 V19   V20   V21   V22   V23   V24   V25   V26
#> 1: -1.97  5.5 -54.5 -21.6 5.71 -1.58  4.58  4.55  3.42
#>          V27 V28 Amount Class
#> 1: 31.6 -15.4  25691     0

max(scores)

#> [1] 0.843
```

Next, I'll use **fastshap** to generate Shapley-based feature contributions using the Monte Carlo approach discussed in Section 6.3.2.2 with 1000 repetitions. Note that we have to tell **fastshap** how to generate scores from an **isotree** **isolation.forest()** model by providing a helper prediction function. The estimated contributions are displayed in Figure 7.19. Here you can see that `Amount=25691.16` had the largest (positive) contribution (well above the 99-th percentile for the entire data set) to this observation having a higher than average anomaly score.

```
library(fastshap)

X <- ccfraud.trn[, 1:30] # feature columns only
max.x <- max.x[, 1:30] # feature columns only!
pfun <- function(object, newdata) { # prediction wrapper
  predict(object, newdata = newdata)
}
```

```
# Generate feature contributions
set.seed(1351) # for reproducibility
ex <- explain(ccfraud.ifo, X = X, newdata = max.x,
              pred_wrapper = pfun, adjust = TRUE,
              nsim = 1000)
sum(ex) # should sum to f(x) - baseline whenever `adjust = TRUE`
#> [1] 0.507
```

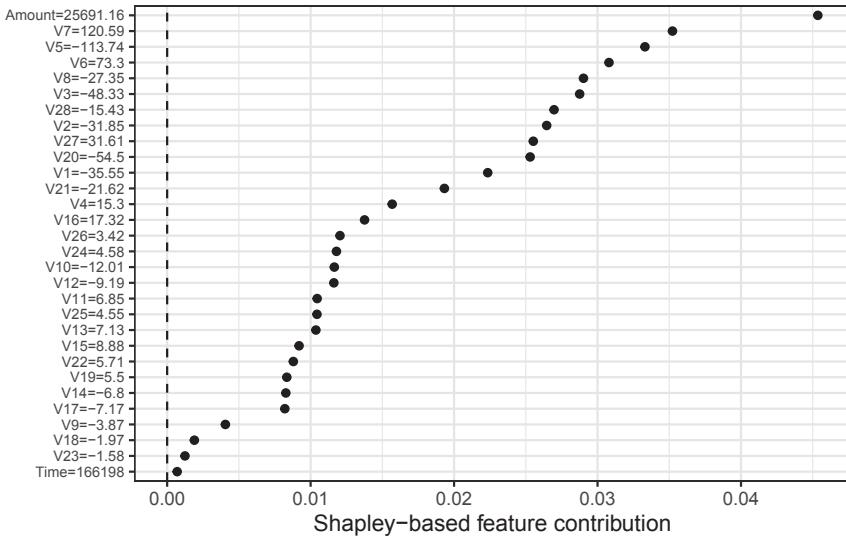


FIGURE 7.19: Estimated feature contributions for the test observation with highest anomaly score. There's a dashed vertical line at zero to differentiate between features with a positive/negative contribution. In this case, all feature values contributed positively to the difference  $f(x^*) - E[\hat{f}(x)] = 0.501$ .

## 7.9 Software and examples

RFs are available in numerous software, both open source and proprietary. The R packages **randomForest**, **ranger**, and **randomForestSRC** [Ishwaran and Kogalur, 2022] implement the traditional RF algorithm for classification and regression; the latter two also support survival analysis, as well as several other extensions. It's important to point out that **ranger**'s implementation of the RF algorithm treats categorical variables as ordered by default; for

details, see the description of the `respect.unordered.factors` argument in `?ranger::ranger`. The **party** and **partykit** packages offer an implementation using conditional inference trees as the base learners (Section 3.4) for the base learners; our `crforest()` function from Section 7.2.3 follows the same approach. The CRAN task view on “Machine Learning & Statistical Learning” includes a section dedicated to RFs in R, so be sure to check that out as well: <https://cran.r-project.org/view=MachineLearning>.

While **randomForest** is a close port of Breiman’s original Fortran code, the **ranger** package is far more scalable and implements a number of modern extensions and improvements discussed in this chapter (e.g., RF as a probability machine, Gini-corrected importance, quantile regression, case-specific RFs, extra-trees, etc.). Another scalable implementation is available from **h2o** [LeDell et al., 2021]. RFs are also part of Spark’s MLLib library [Meng et al., 2016], which includes several R and Python interfaces (in particular, **SparkR**, **sparklyr**, and **pyspark**; an example using **SparkR** is provided in Section 7.9.5).

In Python, RFs, extra-trees, and isolation forests are available in the **sklearn.ensemble** module. Julia users can fit RFs via the **DecisionTree.jl** package. The official GUIDE software (Section 4.9) has the option to construct an RF from individual GUIDE trees; see Loh et al. [2019] and Loh [2020] for details.

Let’s now work through several problems using random forest software.

### 7.9.1 Example: mushroom edibility

No! These data are easy and an ensemble would be overkill here. Remember, the original goal of the problem was to come up with an accurate but simple rule for determining the edibility of a mushroom. This was easily accomplished using a single decision tree (e.g., CART with some manual pruning) or a rule-based model like CORELS; see, for example, Figure 2.22.

### 7.9.2 Example: “deforesting” a random forest

In Section 5.5, I showed how the LASSO can be used to effectively post-process a tree-based ensemble by essentially zeroing out the predictions from some of the trees and reweighting the rest. The idea is that we can often reduce the number of trees quite substantially without sacrificing much in the way of performance. A smaller number of trees means we could, at least in theory, compute predictions faster, which has important implications for model deployment (e.g., when trying to score large data sets on a regular basis). However, unless we have a way to remove the zeroed out trees from

the fitted RF object, we can't really reap all the benefits. This is the purpose of the new `deforest()` function in the `ranger` package<sup>m</sup>, which I'll demonstrate in this section using the Ames housing example.

Keep in mind that this method of post-processing is not specific to bagged tree ensembles and RFs, and can be fruitfully applied to other types of ensembles as well; see [Section 8.9.3](#) for an example using a *gradient boosted tree ensemble*.

To start, I'll load a few packages, prep the data, and create a helper function for computing the RMSE as a function of the number of trees in a `ranger`-based RF:

```
library(ranger)
library(treemisc) # for isle_post() function

# Load the Ames housing data and split into train/test sets
ames <- as.data.frame(AmesHousing::make_ames())
ames$Sale_Price <- ames$Sale_Price / 1000 # rescale response
set.seed(2101) # for reproducibility
trn.id <- sample.int(nrow(ames), size = floor(0.7 * nrow(ames)))
ames.trn <- ames[trn.id, ] # training data/learning sample
ames.tst <- ames[-trn.id, ] # test data
xtst <- subset(ames.tst, select = -Sale_Price) # test features only

# Function to compute RMSE as a function of number of trees
rmse <- function(object, X, y) { # only works with "ranger" objects
  p <- predict(object, data = X, predict.all = TRUE)$predictions
  sapply(seq_len(ncol(p)), FUN = function(i) {
    pred <- rowMeans(p[, seq_len(i)], drop = FALSE])
    sqrt(mean((pred - y) ^ 2))
  })
}
```

Next, I'll fit two different RFs:

**RFO** a default RF with  $B = 1,000$  maximal depth trees;

**RFO.4.5** an RF with  $B = 1,000$  shallow (depth-4) trees, where each tree is built using only a 5% random sample (with replacement) from the training data.

I'll record the computation time of each fit using `system.time()` (this function will also be used later to measure scoring time), which will provide some insight into the potential computational savings offered by this post-processing method<sup>n</sup>:

---

<sup>m</sup>The `deforest()` function is not available in versions of `ranger`  $\leq 0.13.0$ .

<sup>n</sup>Note that there are better ways to benchmark and time expressions in R; see, for example, the `microbenchmark` package [Mersmann, 2021].

```
# Fit a default RF with 1,000 maximal depth trees
set.seed(942) # for reproducibility
system.time({
  rfo <- ranger(Sale_Price ~ ., data = ames.trn, num.trees = 1000)
})

#>    user  system elapsed
#>  5.845   0.112   1.899

# Fit an RF with 1,000 shallow (depth=4) trees on 5% bootstrap samples
set.seed(1021) # for reproducibility
system.time({
  rfo.4.5 <- ranger(Sale_Price ~ ., data = ames.trn, num.trees = 1000,
                     max.depth = 4, sample.fraction = 0.05)
})

#>    user  system elapsed
#>  0.275   0.009   0.113

# Test set MSE as a function of the number of trees
rmse.rfo <- rmse(rfo, X = xtst, y = ames.tst$Sale_Price)
rmse.rfo.4.5 <- rmse(rfo.4.5, X = xtst, y = ames.tst$Sale_Price)
c("Test RMSE (RFO)" = rmse.rfo[1000],
  "Test RMSE (RFO.4.5)" = rmse.rfo.4.5[1000])

#>      Test RMSE (RFO) Test RMSE (RFO.4.5)
#> 24.8          36.7
```

The test RMSE for the RFO model is comparable to the test RMSE from the conditional RF fit in [Section 7.2.3](#). In comparison, the RFO.4.5 model has a much larger test RMSE, which we might have expected given the shallowness of each tree and the tiny fraction of the learning sample each was built from. Consequently, the RFO.4.5 model finished training in only a fraction of the time it took the RFO model. As we'll see shortly, post-processing will help improve the performance of RFO.4.5 so that it is comparable to RFO in terms of performance, while substantially reducing the number of trees (i.e., comparable performance, faster training time, and fewer trees in the end).

Next, I'll obtain the individual tree predictions from each forest and post-process them using the LASSO via `treemisc`'s `isle_post()` function. Note that  $k$ -fold cross-validation can be used here instead of (or in conjunction with) a test set; see `?treemisc::isle_post` for details. For brevity, I'll use a simple prediction wrapper, called `treepreds()`, to compute and extract the individual tree predictions from each RF model:

```
treepreds <- function(object, newdata) {
  p <- predict(object, data = newdata, predict.all = TRUE)
  p$predictions # return predictions component
}

# Post-process RFO ensemble using an independent test set
```

```

preds.trn <- treepreds(rfo, newdata = ames.trn)
preds.tst <- treepreds(rfo, newdata = ames.tst)
rfo.post <- treemisc::isle_post(
  X = preds.trn,
  y = ames.trn$Sale_Price,
  newX = preds.tst,
  newy = ames.tst$Sale_Price,
  family = "gaussian"
)

# Post-process RFO.4.5 ensemble using an independent test set
preds.trn.4.5 <- treepreds(rfo.4.5, newdata = ames.trn)
preds.tst.4.5 <- treepreds(rfo.4.5, newdata = ames.tst)
rfo.4.5.post <- treemisc::isle_post(
  X = preds.trn.4.5,
  y = ames.trn$Sale_Price,
  newX = preds.tst.4.5,
  newy = ames.tst$Sale_Price,
  family = "gaussian"
)

```

The results are plotted in [Figure 7.20](#). Here, we can see that both models benefited from post-processing, but the RFO model only experienced a marginal increase in performance compared to RFO.4.5. Is the slightly better performance in the default RFO model enough to justify its larger training time? Maybe in this particular example, but for larger data sets, the difference in training time can be huge, making it extremely worthwhile. For the post-processed RFO.4.5 model, the test RMSE is minimized using only 93 (reweighted) trees.

```

palette("Okabe-Ito")
plot(rmse.rfo, type = "l", ylim = c(20, 50),
      lty = 1, xlab = "Number of trees", ylab = "Test RMSE")
lines(rmse.rfo.4.5, col = 2)
lines(sqrt(rfo.post$results$mse), col = 1, lty = 2)
lines(sqrt(rfo.4.5.post$results$mse), col = 2, lty = 2)
legend("topright", col = c(1, 2, 1, 2), lty = c(1, 1, 2, 2),
       legend = c("RFO", "RFO.4.5", "RFO (post)", "RFO.4.5 (post)"),
       inset = 0.01, bty = "n")
palette("default")

```

To make this useful in practice, we need a way to remove trees from a fitted RF (i.e., to “deforest” the forest of trees). This could vastly speed up prediction time and reduce the memory footprint of the final model. Fortunately, the **ranger** package includes such a function; see [?ranger::deforest](#) for details.

In the code snippet below, I “deforest” the RFO.4.5 ensemble by removing trees corresponding to the zeroed-out LASSO coefficients, which requires es-

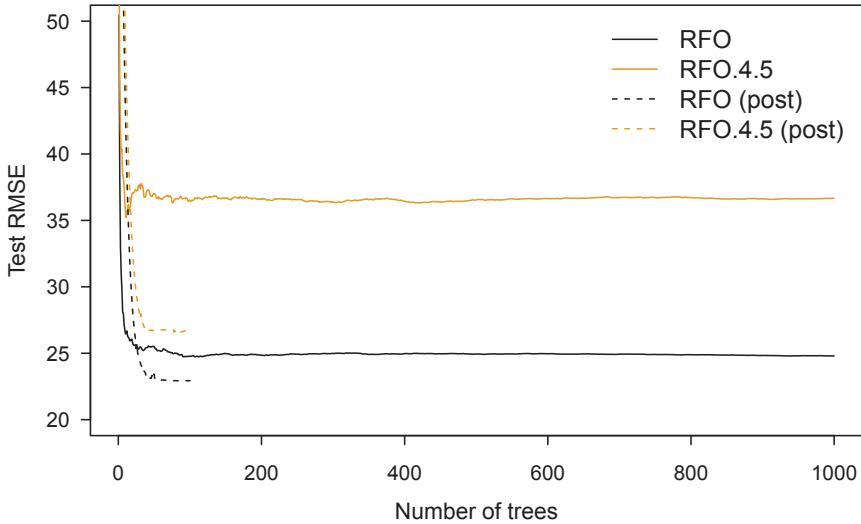


FIGURE 7.20: Test RMSE for the RFO and RFO.4.5 fits. The dashed lines correspond to the post-processed versions of each model. Note how the RFO model only experienced a marginal increase in performance compared to the RFO.4.5 model.

imating the optimal value for the penalty parameter  $\lambda$  (it might be helpful to read the help page for `?glmnet::coef.glmnet`):

```
res <- rfo.4.5.post$results # post-processing results on test set
lambda <- res[which.min(res$mse), "lambda"] # optimal penalty parameter
coefs <- coef(rfo.4.5.post$lasso.fit, s = lambda)[, 1L]
int <- coefs[1L] # intercept
tree.coefs <- coefs[-1L] # no intercept
trees <- which(tree.coefs == 0) # trees to remove

# Remove trees corresponding to zeroed-out coefficients
rfo.4.5.def <- deforest(rfo.4.5, which.trees = trees)

# Check size of each object
c(
  "RFO.4.5" = format(object.size(rfo.4.5), units = "MB"),
  "RFO.4.5 (deforested)" = format(object.size(rfo.4.5.def), units = "MB")
)
#> RFO.4.5 RFO.4.5 (deforested)
#> "1 Mb"           "0.1 Mb"
```

Notice the impact this had on reducing the overall size of the fitted model. This can often lead to a much more compact model that's easier to save and load when memory requirements are a concern.

We can't just use the "deforested" tree ensemble directly; remember, the estimated LASSO coefficients imply a reweighting of the remaining trees! To obtain the reweighted predictions from the "deforested" model, we need to do a bit more work. Here, I'll create a new prediction function, called `predict.def()`, that will compute the reweighted predictions from the remaining trees using the estimated LASSO coefficients—similar to how predictions in a linear model are computed.

To test it out, I'll stack the learning sample (`ames.trn`) on top of itself 100 times, resulting in  $N = 205,100$  observations for scoring. Below, I compare the prediction times for both the original (i.e., non-processed) and "deforested" RFO.4.5 fits:

```
ames.big <- # stack data on top of itself 100 times
do.call("rbind", args = replicate(100, ames.trn, simplify = FALSE))

# Compute reweighted predictions from a ``deforested'' ranger object
predict.def <- function(rf.def, weights, newdata, intercept = TRUE) {
  preds <- predict(rf.def, data = newdata,
                    predict.all = TRUE)$predictions
  res <- if (isTRUE(intercept)) { # returns a one-column matrix
    cbind(1, preds) %*% weights
  } else {
    preds %*% weights
  }
  res[, 1, drop = TRUE] # coerce to atomic vector
}

# Scoring time for original RFO.4.5 fit
system.time({
  # full random forest
  preds <- predict(rfo.4.5, data = ames.big)
})

#>   user  system elapsed
#> 37.15    2.64   13.35

# Scoring time for post-processed RFO.4.5 fit using updated weights
weights <- coefs[coefs != 0] # LASSO-based weights for remaining trees
system.time({
  preds.post <- predict.def(rfo.4.5.def, weights = weights,
                             newdata = ames.big)
})

#>   user  system elapsed
#>  4.17    0.73   4.47
```

The final model contains only 93 trees and achieved a test RMSE of 26.59, while also being orders of magnitude faster to initially train. The computational advantages are easier to appreciate on even larger data sets.

In summary, I used the LASSO to post-process and “deforest” a large ensemble of shallow trees (which trained relatively fast), producing a much smaller ensemble with fewer trees that scores faster compared to the default RFO. While the default RFO model had a slightly smaller test RMSE of 24.72 compared to the “deforested” RFO.4.5 test RMSE of 111.29, the difference is arguably negligible (especially when you take the differences in both training and scoring time into account).

### 7.9.3 Example: survival on the Titanic

In this example, I’ll walk through a simple RF analysis of the well-known Titanic data set, where the goal is to understand survival probability aboard the ill-fated Titanic. A more thoughtful analysis using logistic regression and spline-based techniques is provided in Harrell [2015, Chap. 12].

Several versions of this data set are publicly available; for example, in the R package **titanic** [Hendricks, 2015]. Here, I’ll use a more complete version of the data<sup>o</sup> which can be loaded using the `getHdata()` from package **Hmisc** [Harrell, 2021]; the raw data can also be downloaded from <https://hbiostat.org/data/>. In this example, I’ll only consider a handful of the original variables:

```
t3 <- read.csv("https://hbiostat.org/data/repo/titanic3.csv",
                stringsAsFactors = TRUE)
keep <- c("survived", "pclass", "age", "sex", "sibsp", "parch")
t3 <- t3[, keep] # only retain key variables
```

Note that roughly 20.09% of the values for `age`, the age in years of the passenger, are missing:

```
sapply(t3, FUN = function(x) mean(is.na(x)))
#> survived    pclass      age       sex     sibsp     parch
#>   0.000     0.000    0.201     0.000     0.000     0.000
```

Following Harrell [2015, Sec. 12.4], I use a decision tree to investigate which kinds of passengers tend to have a missing value for `age`. In the example below, I use the **partykit** package to apply the CTree algorithm (Chapter 3) using a missing value indicator for `age` as the response. From the tree output we can see that third-class passengers had the highest rate of missing `age` values (29.3%), followed by first-class male passengers with no siblings or

---

<sup>o</sup>A description of the original source of these data is provided in Harrell [2015, p. 291].

spouses aboard (22.8%). This makes sense, since males and third-class passengers supposedly had the least likelihood of survival (“women and children first”).

```
library(partykit)

# Fit a conditional inference tree using missingness as response
temp <- t3 # temporary copy
temp$age <- as.factor(ifelse(is.na(temp$age), "y", "n"))
(t3.ctree <- ctree(age ~ ., data = temp))

#>
#> Model formula:
#> age ~ survived + pclass + sex + sibsp + parch
#>
#> Fitted party:
#> [1] root
#> |   [2] pclass <= 2
#> |   |   [3] sibsp <= 0
#> |   |   |   [4] sex in female: n (n = 135, err = 6%)
#> |   |   |   [5] sex in male
#> |   |   |   |   [6] pclass <= 1: n (n = 123, err = 23%)
#> |   |   |   |   [7] pclass > 1: n (n = 122, err = 11%)
#> |   |   |   [8] sibsp > 0: n (n = 220, err = 3%)
#> |   |   [9] pclass > 2: n (n = 709, err = 29%)
#>
#> Number of inner nodes:    4
#> Number of terminal nodes: 5

# plot(t3.ctree) # plot omitted
```

### 7.9.3.1 Missing value imputation

Next, I'll use the CART-based multiple imputation procedure outlined in [Section 2.7.1](#) to perform  $m = 21$  separate imputations for each missing `age` value. Why did I choose  $m = 21$ ? White et al. [2011] propose setting  $m \geq 100f$ , where  $f$  is the fraction of incomplete cases<sup>P</sup>. Since `age` is the only missing variable, with  $f = 0.201$ , I chose  $m = 21$ . Using multiple different imputations will give us an idea of the sensitivity of the results of our (yet to be fit) RF.

```
library(mice)

set.seed(1125) # for reproducibility
imp <- mice(t3, method = "cart", m = 21, minbucket = 5,
            printFlag = FALSE)

# Display nonparametric densities
densityplot(imp)
```

---

<sup>P</sup>When  $f \geq 0.03$ , Harrell [2015, p. 57] suggests setting  $m = \max(5, 100f)$

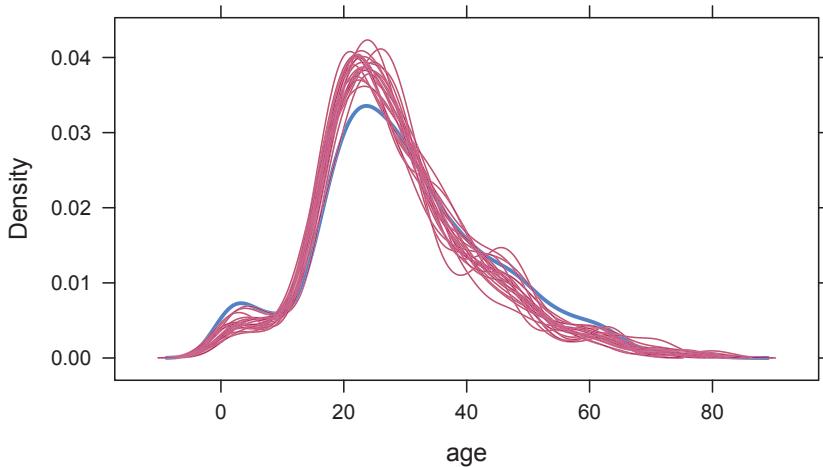


FIGURE 7.21: Nonparametric density estimate of `age` for the complete cases (blue line) and 15 imputed data sets.

Nonparametric densities for passenger age are given in Figure 7.21. There is one density for each of the imputed data sets (red curves) and one density for the original complete case (blue curve). The overall distributions are comparable, but there is certainly some variance across the  $m = 21$  imputation runs. Our goal is to run an RF analysis for each of the  $m = 21$  completed data sets and inspect the variability of the results. For instance, I might graphically show the  $m = 21$  variable importance scores for each feature, along with the mean or median.

Next, I call `complete()` (from package **mice**) to produce a list of the  $m = 21$  completed data sets which I can use to carry on with the analysis. The only difference is that I'll perform the same analysis on each for the completed data sets.<sup>q</sup>

```
t3.mice <- complete(
  data = imp,      # "mids" object (multiply imputed data set)
  action = "all",  # return list of all imputed data sets
  include = FALSE  # don't include original data (i.e., data with NAs)
)
length(t3.mice)  # returns a list of completed data sets

#> [1] 21
```

<sup>q</sup>This approach is probably not ideal in situations where the analysis is expensive (e.g., because the data are “big” and the model is expensive to tune). In such cases, you may have to settle for a smaller, less optimal value for  $m$ .

For comparison, let's look at the results from using the proximity-based RF imputation procedure discussed in [Section 7.6.2](#). The code snippet below uses `rfImpute()` from package **randomForest** to handle the proximity-based imputation. The results are plotted along with those from MICE in [Figure 7.22](#).

```
# Generate completed data set using RF's proximity-based imputation
set.seed(2121) # for reproducibility
t3.rfimpute <-
  randomForest::rfImpute(as.factor(survived) ~ ., data = t3,
                        iter = 5, ntree = 500)

#> ntree      OOB      1      2
#> 500: 20.70% 12.36% 34.20%
#> ntree      OOB      1      2
#> 500: 19.17% 10.75% 32.80%
#> ntree      OOB      1      2
#> 500: 19.56% 11.62% 32.40%
#> ntree      OOB      1      2
#> 500: 19.71% 11.74% 32.60%
#> ntree      OOB      1      2
#> 500: 19.10% 10.75% 32.60%

# Construct matrix of imputed values
m <- imp$m # number of MICE-based imputation runs
na.id <- which(is.na(t3$age))
x <- matrix(NA, nrow = length(na.id), ncol = m + 1)
for (i in 1:m) x[, i] <- t3.mice[[i]]$age[na.id]
x[, m + 1] <- t3.rfimpute$age[na.id]

# Plot results
palette("Okabe-Ito")
plot(x[, 1], type = "n", xlim = c(1, length(na.id)), ylim = c(0, 100),
     las = 1, ylab = "Imputed value")
for (i in 1:m) {
  lines(x[, i], col = adjustcolor(i, alpha.f = 0.1))
}
lines(rowMeans(x[, 1:m]), col = 1, lwd = 2)
lines(x[, m + 1], lwd = 2, col = 2)
legend("topright", legend = c("MICE: CART", "RF: proximity"), lty = 1,
       col = 1:2, bty = "n")
palette("default")
```

Here, you can see that the imputed values from both procedures are similar, but that multiple imputations provide a range of plausible values. Also, there are a few instances where there's a bit of a gap between the imputed values for the two procedures. For example, consider observations 956 and 959, whose records are printed below. The first passenger is recorded to be a third-class female with three siblings (or spouses) and one parent (or child) aboard. This individual is likely a child. The proximity-based imputation imputed the age

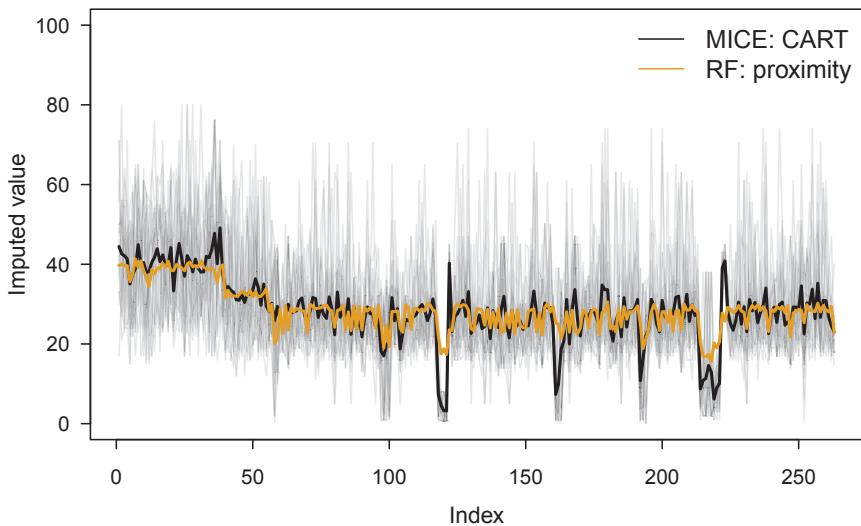


FIGURE 7.22: Imputed values for the 263 missing `age` values. The yellow line corresponds to the proximity-based imputation. The light gray lines correspond to the 15 different imputation runs using MICE, and the black line corresponds to their average.

for this passenger as 17.522 years, whereas MICE gives an average of 4.571 years and a plausible range of 0.75–8.00 years. Similarly, the proximity method imputed the age for case 959—a third-class female with three children—as 23.52 whereas MICE gave an average of 40.238. Which imputations do you think are more plausible?

```
t3[c(956, 959), ]  
#>   survived pclass age      sex sibsp parch  
#> 956       0     3  NA female     3     1  
#> 959       0     3  NA female     0     4
```

### 7.9.3.2 Analyzing the imputed data sets

With the  $m = 21$  completed data sets in hand, I can proceed with the RF analysis. The idea here is to fit an RF separately to each completed data set. I'll then look at the variance of the results (e.g., OOB error, variable importance scores, etc.) to judge its sensitivity to the different plausible imputations. Below, I use the **ranger** package to fit a (default) RF/probability machine to each of the  $m = 21$  completed data sets. In anticipation of looking at the sensitivity of the variable importance scores, I set `importance` =

"permutation" to employ the OOB-based permutation variable importance procedure discussed in [Section 7.5.2](#).

```
library(ranger)

# Obtain a list of probability forests, one for each imputed data set
set.seed(2147) # for reproducibility
rfos <- lapply(t3.mice, FUN = function(x) {
  ranger(as.factor(survived) ~ ., data = x, probability = TRUE,
         importance = "permutation")
})

# Check OOB errors (Brier-score, in this case)
sapply(rfos, FUN = function(forest) forest$prediction.error)

#>      1     2     3     4     5     6     7     8     9
#> 0.134 0.133 0.135 0.134 0.134 0.134 0.135 0.134 0.132
#>    10    11    12    13    14    15    16    17    18
#> 0.133 0.132 0.133 0.133 0.135 0.135 0.135 0.133 0.134
#>    19    20    21
#> 0.134 0.134 0.133
```

The OOB errors from each model are comparable; that's a good start! The average OOB Brier score is 0.134, with a standard deviation of 0.001.

Next, I'll look at variable importance. With multiple imputation I think the most sensible thing to do is to just plot the variable importance scores from each run together, so that you can see the variability in the results:

```
# Compute list of VI scores, one for each model. Note: can use
#`FUN = ranger::importance` to be safe
vis <- lapply(rfos, FUN = importance)

# Stack into a data frame
head(vis <- as.data.frame(do.call(rbind, args = vis)))

#>   pclass    age    sex  sibsp  parch
#> 1 0.0531 0.0370 0.122 0.0130 0.0149
#> 2 0.0529 0.0404 0.125 0.0143 0.0145
#> 3 0.0504 0.0323 0.124 0.0126 0.0139
#> 4 0.0498 0.0336 0.125 0.0149 0.0131
#> 5 0.0540 0.0393 0.122 0.0135 0.0161
#> 6 0.0533 0.0407 0.123 0.0137 0.0142

# Display boxplots of results
boxplot(vis, las = 1)
```

[Figure 7.23](#) shows a boxplot of the  $m = 21$  variable importance scores for each feature; these are the OOB-based permutation scores discussed in [Section 7.5.2](#). The results do not vary too much, so we can be somewhat confident in the overall ranking of the features. Clearly `sex` is the most important

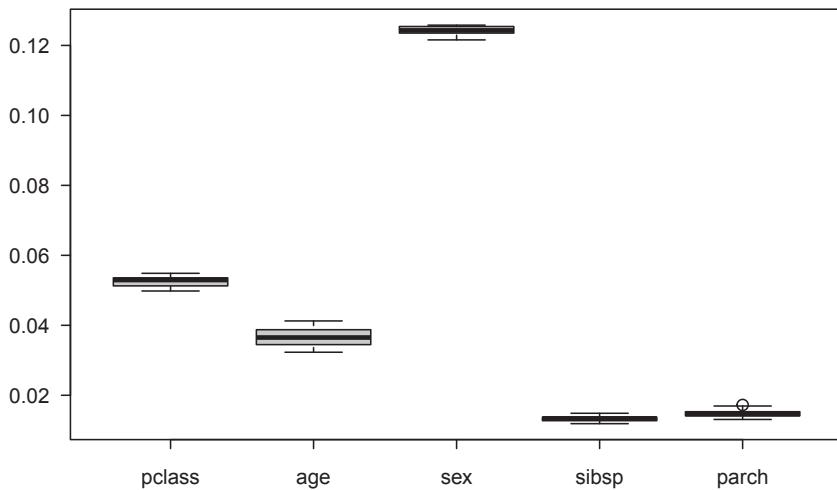


FIGURE 7.23: Boxplot of variable importance scores across all  $m = 21$  RF fits.

predictor of survivability in these models, followed by passenger class (**pclass**) and passenger age (**age**). The remaining features are comparatively less important, with no discernible difference between the number of siblings/spouses aboard (code**sibsp**) and the number of parents/children aboard (**parch**).

It's natural to look at feature effect plots after examining the importance of each variable (i.e., main effects). A common feature effect plot is the partial dependence plot (Section 6.2.1), or PD plot for short. Following a similar strategy, we can compute the partial dependence for each feature across all  $m = 21$  fitted RFs, and display the results on the same graph.

The code snippets below rely on the **pdp** package for constructing each partial dependence curve; for details, see Greenwell [2021b]. To start, I define a simple prediction wrapper for extracting predictions from a fitted **ranger** model; see `?ranger::predict.ranger` for details on computing and extracting predictions from a "ranger" object. Note that PD plots are essentially constructed from averaged predictions, so the function below returns the average predicted probability of survival:

```
pfun <- function(object, newdata) { # mean(prob(survived=1|x))
  mean(predict(object, data = newdata)$predictions[, "1"])
}
```

Since there are  $m = 21$  imputed data sets, I'm essentially computing  $m = 21$  3-way PD plots (i.e., visualizing a 3-way interaction effect), which can

be quite cumbersome computationally. Because we only have one numeric predictor (`age`), it will take only a few minutes in this example. For larger problems, or problems with many numeric features, you may want to consider computing PD plots for each feature individually (i.e., main effects), or using parallel computing (or other computational tricks). Below, I instruct `pdp`'s `partial()` function to plot over an evenly spaced grid of 19 percentiles for `age` (from 5-th to 95-th) within each unique combination of `pclass`<sup>r</sup> and `sex`, giving a total of  $19 \times 3 \times 2 = 114$  plotting points.

```
library(pdp)

# Construct PD plots for each model
pdps <- lapply(1:m, FUN = function(i) {
  partial(rfoss[[i]], pred.var = c("age", "pclass", "sex"),
          pred.fun = pfun, train = t3.mice[[i]], cats = "pclass",
          quantiles = TRUE, probs = 1:19/20)
})

# Stack into a single data frame for plotting
for (i in seq_along(pdps)) {
  pdps[[i]]$m <- i
}
head(pdps <- do.call(rbind, args = pdps))

#>   age pclass    sex yhat m
#> 1  5.0      1 female 0.848 1
#> 2 14.5      1 female 0.915 1
#> 3 18.0      1 female 0.935 1
#> 4 19.0      1 female 0.936 1
#> 5 21.0      1 female 0.939 1
#> 6 22.0      1 female 0.934 1
```

Next, I plot the results. There's some R-ninja trickery happening in the code chunk below in order to get the plot I want. Using `ggplot2`, I want to group a set of line plots by two variables, but color by just one of them. We can paste the two grouping variables together into a new column. However, base R's `interaction()` function can accomplish this for us; see `?interaction` for details.

The results are displayed in [Figure 7.24](#); compare this to Figure 12.22 in Harrell [2015, p. 308]. I also included a rug representation (i.e., 1-d plot) in each panel showing the deciles (i.e., the 10-th percentile, 20-th percentile, etc.) of passenger age from the original (incomplete) training set. This helps guide where the plots are potentially extrapolating. Using deciles means that 10% of the observations lie between any two consecutive rug marks; see Greenwell [2017] for some remarks on the importance of avoiding extrapolation when

---

<sup>r</sup>I'm using `cats = "pclass"` here to treat `pclass` as categorical since it's restricted to  $\text{pclass} \in \{1, 2, 3\}$ .

interpreting PD plots, as well as some mitigation strategies (e.g., using rug plots and convex hulls).

```
library(ggplot2)

# Plot results
deciles <- quantile(t3$age, prob = 1:9/10, na.rm = TRUE)
ggplot(pdps, aes(age, yhat, color = sex,
                  group = interaction(m, sex))) +
  geom_line(alpha = 0.3) +
  geom_rug(aes(age), data = data.frame("age" = deciles),
            sides = "b", inherit.aes = FALSE) +
  labs(x = "Age (years)", y = "Survival probability") +
  facet_wrap(~ pclass) +
  scale_colour_manual(values = c("black", "orange")) + # Okabe-Ito
  theme_bw() +
  theme(legend.title = element_blank(),
        legend.position = "top")
```

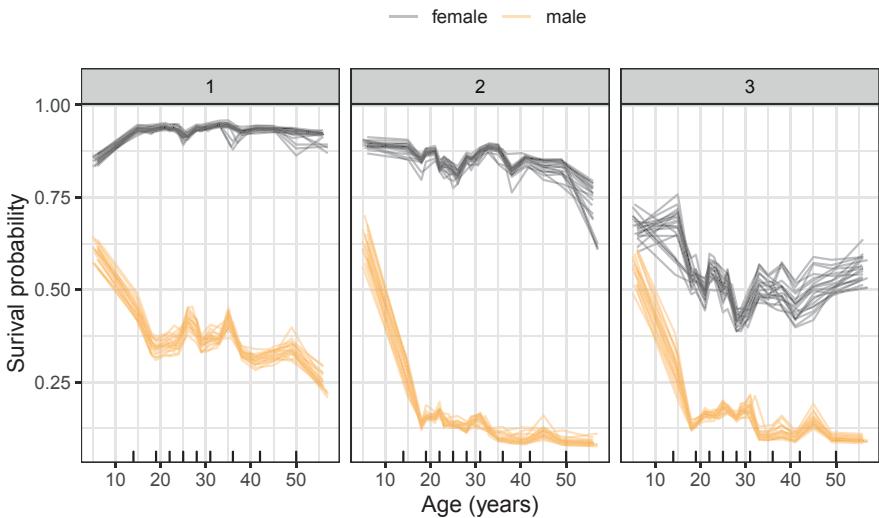


FIGURE 7.24: Partial dependence of the probability of surviving on passenger age, class, and sex. There's one curve for each of the  $m = 21$  completed data sets.

While there's some variability in the results, the overall patterns are clear. First-class females had the best chances of surviving, regardless of age or class. Passenger age seems to have a stronger negative effect on passenger survivability for males compared to females, regardless of class. The difference in survivability between males and females is less pronounced for third-class

passengers. Do you agree? What other conclusions can you draw from this plot?

Finally, let's use Shapley values (Section 6.3.1) to help us understand individual passenger predictions. To illustrate, let's focus on a single (hypothetical/fictional) passenger. Everyone, especially those who haven't seen the movie, meet Jack<sup>s</sup>:

```
jack.dawson <- data.frame(
  survived = 0L, # in case you haven't seen the movie
  pclass = 3L, # using `3L` instead of `3` to treat as integer
  age = 20.0,
  sex = factor("male", levels = c("female", "male")),
  sibsp = 0L,
  parch = 0L
)
```

Here, I'll use the **fastshap** package for computing Shapley values, but you can use any Shapley value package you like (e.g., R package **iml**). First, we need to set up a prediction wrapper—this is a function that tells **fastshap** how to extract predictions from the fitted **ranger** model, which is the purpose of function **pfun()** below. Next, I compute approximate feature contributions for Jack's predicted probability of survival using 1000 Monte-Carlo repetitions, which is done for each of the  $m = 21$  completed data sets:

```
library(fastshap)

# Prediction wrapper for `fastshap::explain()`; has to return a single
# (atomic) vector of predictions
pfun <- function(object, newdata) { # compute prob(survived=1|x)
  predict(object, data = newdata)$predictions[, 2]
}

# Estimate feature contributions for each imputed training set
set.seed(754)
ex.jack <- lapply(1:21, FUN = function(i) {
  X <- subset(t3.mice[[i]], select = -survived)
  explain(rfes[[i]], X = X, newdata = jack.dawson, nsim = 1000,
          adjust = TRUE, pred_wrapper = pfun)
})

# Bind together into one data frame
ex.jack <- do.call(rbind, args = ex.jack)

# Add feature values to column names
names(ex.jack) <- paste0(names(ex.jack), "=" , t(jack.dawson))
print(ex.jack)
```

---

<sup>s</sup>I guesstimated some of Jack's inputs, based on the movie I saw in seventh grade.

```
#> # A tibble: 21 x 5
#>   `pclass=3` `age=20` `sex=male` `sibsp=0` `parch=0`
#>   <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
#> 1 -0.0836 -0.0136    -0.141  0.00721 -0.0174
#> 2 -0.0796 -0.0222    -0.144  0.0109  -0.00967
#> 3 -0.0743 -0.000271   -0.144  0.00995 -0.0170
#> 4 -0.0709 -0.0132    -0.139  0.00740 -0.0126
#> 5 -0.0807 -0.0192    -0.134  0.00768 -0.0159
#> 6 -0.0807 -0.0134    -0.136  0.0103  -0.0159
#> 7 -0.0840 -0.00355   -0.145  0.00999 -0.0147
#> 8 -0.0874  0.0110    -0.136  0.0103  -0.0254
#> 9 -0.0754 -0.00982   -0.143  0.00449 -0.0233
#> 10 -0.0663 -0.000338  -0.144  0.00519 -0.0165
#> # ... with 11 more rows
```

Fortunately, again, the results are relatively stable across imputations. A summary of the overall Shapley explanations, along with Jack's predictions, is shown in Figure 7.25. Here, we can see that Jack being a third-class male contributed the most to his poor predicted probability of survival, aside from him not being able to fit on the floating door that Rose was hogging...

```
# Jack's predicted probability of survival across all imputed
# data sets
pred.jack <- data.frame("pred" = sapply(rfos, FUN = function(rfo) {
  pfun(rfo, jack.dawson)
}))

# Plot setup (e.g., side-by-side plots)
par(mfrow = c(1, 2), mar = c(4, 4, 2, 0.1),
  las = 1, cex.axis = 0.7)

# Construct boxplots of results
boxplot(pred.jack, col = adjustcolor(2, alpha.f = 0.5))
mtext("Predicted probability of surviving", line = 1)
boxplot(ex.jack, col = adjustcolor(3, alpha.f = 0.5), horizontal = TRUE)
mtext("Feature contribution", line = 1)
abline(v = 0, lty = "dashed")
```

We just walked through a simple analysis of the well-known Titanic data set, with a focus on using RFs to understand which passengers were most likely to survive and why. The analysis was complicated by the fact that one variable, `age`, contained many missing values. As a result, I performed multiple imputation, followed by an RF analysis on each of the plausible imputed data sets to gauge the sensitivity of the resulting imputations. In this example, the results seemed relatively stable across imputations, so we can be confident in our conclusions.

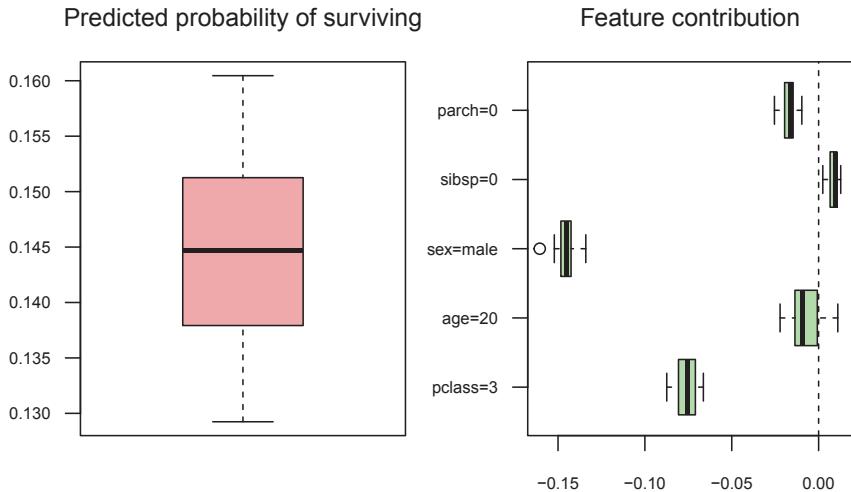


FIGURE 7.25: Predicted probability of survival for Jack across all imputed data sets (left) and their corresponding Shapley-based feature contributions (right).

#### 7.9.4 Example: class imbalance (the good, the bad, and the ugly)

As discussed in Section 7.2.1, we often don't want to predict a class label, but rather the probability of belonging to a particular class—the latter has the benefit of providing some indication of uncertainty (e.g., if we predicted Jack to have a 15% chance of surviving his voyage on the Titanic, then we also estimate that he has an 85% chance of not surviving). Some models, however, can give you poor estimates of the class probabilities. *Calibration* addresses the issue by allowing you to better calibrate the probabilities of a fitted model, or provide estimated probabilities for models that are unable to naturally produce them in the first place (support vector machines, for example).

Calibration refers to the degree of agreement between observed and predicted probabilities, and its utility is well-discussed in the statistical literature (even if not commonly practiced); see, for example, Rufibach [2010], Austin and Steyerberg [2014], and Niculescu-Mizil and Caruana [2005]—the latter discusses calibration in a broader context and includes some discussion on calibrating RFs. A well calibrated (binary) probability model should produce accurate class probabilities such that among those probability estimates close to, say, 0.7, approximately 70% actually belong to the positive class, and so on.

Unless the trees are grown to purity, RFs generally produce consistent and *well-calibrated* probabilities [Malley et al., 2012, Niculescu-Mizil and Caruana, 2005], while boosted trees (Chapter 8) do not<sup>t</sup>. However, we'll see shortly that that's not always the case. Furthermore, *calibration curves* can be useful in comparing the performance of fitted probability models.

Real binary data are often unbalanced. For example, in modeling loan defaults, the target class (default on a loan) is often underrepresented. This is expected since we would hope that most people don't default on their loan over the course of paying it off. However, many practitioners perceive class imbalance as an issue that affects "accuracy." In actuality, the problem is usually that the data are balanced [Matloff, 2017, p. 193].

In this example, I'm going to simulate some unbalanced data. In particular, I'm going to convert the Friedman 1 benchmark regression data (Section 1.4.3) into a binary classification problem using a *latent variable model*. Essentially, I'll treat the observed response values as the linear predictor of a logistic regression model and convert them to probabilities. We can then use a binomial random number generator to simulate the observed class labels. The important thing to remember about this simulation study is that we have access to the true underlying class probabilities!

A simple function to convert the Friedman 1 regression problem into a binary classification problem, as described above, is given below. (Note that the line `d$y <- d$y - 23` shifts the intercept term and effectively controls the balance of the generated 0/1 outcomes—here, it was chosen to obtain a 0/1 class balance of roughly 0.95/0.05.)

```
gen_binary <- function(...) {
  d <- treemisc::gen_friedman1(...) # regression data
  d$y <- d$y - 23 # shift intercept
  d$prob <- plogis(d$y) # inverse logit to obtain class probabilities
  #d$prob <- exp(d$y) / (1 + exp(d$y)) # same as above
  d$y <- rbinom(nrow(d), size = 1, prob = d$prob) # 0/1 outcomes
  d
}

# Generate samples
set.seed(1921) # for reproducibility
trn <- gen_binary(100000) # training data
tst <- gen_binary(100000) # test data
```

Let  $N = N_0 + N_1$  be the number of observations in the learning sample, where  $N_0$  and  $N_1$  represent the number of observations that belong to class 0 and class 1, respectively. If the learning sample is a random (i.e., representative) sample from the population of interest, then we can estimate the true

---

<sup>t</sup>Surprisingly, in contrast to RFs, bagging decision trees grown to purity produces consistent probability estimates [Malley et al., 2012, Biau et al., 2008].

class priors from the data using  $\pi_i = N_i/N$ , for  $i = 1, 2$ ; this was discussed for CART-like decision trees in [Section 2.2.4](#). There are three scenarios to consider:

- a) the data form a representative sample, and the observed class frequencies reflect the true class priors in the population (**the good**);
- b) the class frequencies have been artificially balanced, but the true class frequencies/priors are known (**the bad**);
- c) the class frequencies have been artificially balanced, and the true class frequencies/priors are unknown (**the ugly**).

In the code chunk below I use an independent sample of size  $N = 10^6$  to estimate  $\pi_1$  (i.e., the *prevalance* of observations in class 1 in the population):

```
(pi1 <- proportions(table(gen_binary(1000000)$y))["1"])

#>      1
#> 0.0498
```

Next, I'll define a simple calibration function that can be used for *isotonic calibration*; for a brief overview of different calibration methods, see Niculescu-Mizil and Caruana [2005], Kull et al. [2017]. Note that there are many R and Python libraries for calibration; for example, `val.prob()` from R package `rms` [Harrell, Jr., 2021] and the `sklearn.calibration` module.

```
isocal <- function(prob, y) { # isotonic calibration function
  ord <- order(prob)
  prob <- prob[ord] # put probabilities in increasing order
  y <- y[ord]
  prob.cal <- isoreg(prob, y)$yf # fitted values
  data.frame("original" = prob, "calibrated" = prob.cal)
}
```

To start, let's fit a default RF to the original (i.e., unbalanced) learning sample. Note that I exclude the `prob` column when specifying the model formula.

```
library(ranger)

# Fit a probability forest (omitting the prob column)
set.seed(1446) # for reproducibility
(rfo1 <- ranger(y ~ . - prob, data = trn, probability = TRUE,
                 verbose = FALSE))

#> Ranger result
#>
#> Call:
#>   ranger(y ~ . - prob, data = trn, probability = TRUE, verbose = FA...
#>
```

```
#> Type:                               Probability estimation
#> Number of trees:                  500
#> Sample size:                     100000
#> Number of independent variables: 10
#> Mtry:                            3
#> Target node size:                10
#> Variable importance mode:       none
#> Splitrule:                      gini
#> OOB prediction error (Brier s.): 0.0256
```

The OOB prediction error (in this case, the Brier score) is 0.026. The Brier score on the test data can also be computed, but since I have access to the true probabilities, I might as well compare them with the predictions too (for this, I'll compute the MSE between the predicted and true probabilities). In this case, we see that the Brier score on the test set is comparable to the OOB Brier score.

```
prob1 <- predict(rfo1, data = tst)$predictions[, 2]

mean((prob1 - tst$y) ^ 2) # Brier score
#> [1] 0.0255

mean((prob1 - tst$prob) ^ 2) # MSE between predicted and true probs
#> [1] 0.00319
```

Looking at a single metric (or metrics) does not paint a full picture, so it can be helpful to look at specific visualizations, like calibration curves, to further assess the accuracy of the model's predicted probabilities (lift charts can also be useful). The leftmost plot in [Figure 7.26](#) shows the actual vs. predicted probabilities for the test set, as well as the isotonic-based calibration curve from the above RF. In this case, the RF seems to be doing a reasonable job in terms of accuracy. The model seems well-calibrated for probabilities below 0.5, but seems to have a slight negative bias for probabilities above 0.5, which makes sense since most of the probability mass is concentrated near zero (as we might have expected given the true class frequencies).

To naively combat the perceived issue of unbalanced class labels, the learning sample is often artificially rebalanced (e.g., using down sampling) so that the class outcomes have roughly the same distribution. In general, THIS IS A BAD IDEA for probability models, and can lead to serious bias in the predicted probabilities—in fact, any algorithm that requires you to remove good data to optimize performance is suspect. Nonetheless, sometimes the data have been artificially rebalanced in a preprocessing step outside of our control, or maybe you decided to down sample the data to reduce computation time (in which case, you should try to preserve the original class frequencies, or at least store them for adjustments later). In any case, let's see what happens to our predictions when we down sample the majority class.

In scenarios b)–c), we cannot estimate  $\pi_0$  and  $\pi_1$  from the learning sample; however, in scenario b) we might have estimates of  $\pi_0$  and  $\pi_1$ , perhaps from historical data. If the data have been artificially balanced, then it's possible to use good estimates of  $\pi_0$  and  $\pi_1$  to “correct” (or adjust) the output predicted probabilities. With CART and GUIDE, it's possible to provide the true priors and let the tree algorithm handle the adjustment (we saw how this is handled in CART in [Section 2.2.4](#) and provided an example with **rpart** using the letter image recognition example in [Section 2.9.5](#)). What if no `priors` argument is available in your software? Fortunately, you can apply a simple adjustment to the output predicted probabilities, as discussed in Matloff [2017, pp. 197–198]. A simple function to adjust the predicted probabilities is given below. Here, `p` is a vector of predicted probabilities for the positive class (i.e.,  $\widehat{\Pr}(Y = 1|\mathbf{x})$ ), `observed.ratio` is the ratio of the observed class frequencies (i.e.,  $N_0/N_1$ ), and `true.ratio` is the ratio of the true class priors (i.e.,  $\pi_0/\pi_1$ ).

```
prob.adjust <- function(p, observed.ratio, true.ratio) {
  f.ratio <- (1 / p - 1) * (1 / observed.ratio)
  1 / (1 + true.ratio * f.ratio)
}
```

Let's try this out on an RF fit to a down sampled version of the training data. Below I artificially balance the classes by removing rows corresponding to the dominant class (i.e.,  $y = 0$ ):

```
trn.1 <- trn[trn$y == 1, ]
trn.0 <- trn[trn$y == 0, ]
trn.down <- rbind(trn.0[seq_len(nrow(trn.1)), ], trn.1)
table(trn.down$y)

#>
#>     0      1
#> 5018 5018
```

Next, I'll fit another (default) RF, but this time to the down samples training set. I then apply the adjustment formula to the predicted probabilities for the positive class in the test set:

```
set.seed(1146) # for reproducibility
rfo2 <- ranger(y ~ . - prob, data = trn.down, probability = TRUE)

# Predicted probabilities for the positive class: P(Y=1|x)
prob2 <- predict(rfo2, data = tst)$predictions[, 2]
mean((prob2 - tst$y) ^ 2) # Brier score

#> [1] 0.0756
mean((prob2 - tst$prob) ^ 2) # MSE
#> [1] 0.0538
```

```

prob3 <- prob.adjust(prob2, observed.ratio = 1,
                      true.ratio = (1 - pi1) / pi1)
mean((prob3 - tst$y) ^ 2) # Brier score
#> [1] 0.0285
mean((prob3 - tst$prob) ^ 2) # MSE between predicted and true probs
#> [1] 0.00609

```

Figure 7.26 shows the predicted vs. true probabilities across three different cases: 1) predicted probabilities (`prob1`) from an RF applied to the original training data (left display), 2) predicted probabilities (`prob2`) from an RF applied to a down-sampled version of the original training data, but adjusted using the original class frequencies (middle display), and 3) predicted probabilities (`prob3`) from an RF applied to a down-sampled version of the original training data with no adjustment (right display).

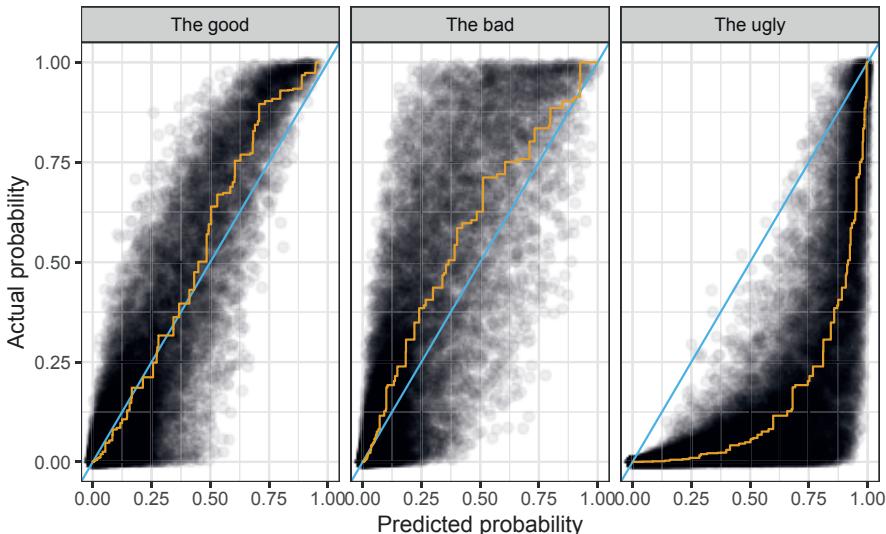


FIGURE 7.26: True vs. predicted probabilities for the test set from an RF fit to the original and down sampled (i.e., artificially balanced) training sets. Left: probabilities from the original RF. Middle: Probabilities from the down sampled RF with post-hoc adjustment. Right: Probabilities from the down sampled RF. The calibration curve is shown in orange, with the blue curve representing perfect calibration.

Compared to the predicted probabilities from an RF fit to the original (i.e., unbalanced) learning sample, down sampling appears to have produced biased and poorly calibrated probabilities, although, the adjustment formula seems

to provide some relief but requires access to the true class priors (or good estimates thereof).

Basically, it is ill-advised to choose a model based on a metric that forces a classification based on an arbitrary threshold. Instead, choose a model using a *proper scoring rule* (e.g., or the Brier score) that makes use of the full range of predicted probabilities and is optimized when the true probabilities are recovered. Down sampling, adjusted or not, seems to highly under or over estimate the true class probabilities.

### 7.9.5 Example: partial dependence with Spark MLlib

In this section, I'll look at a well-known bank marketing data set available from the UC Irvine Machine Learning Repository [Moro et al., 2014]. The data concern the direct marketing campaigns of a Portuguese banking institution, which were based on phone calls. Often, more than one contact to the same client was required, in order to assess if the product, a bank term deposit, would be subscribed or not; hence, this is a binary classification problem with response variable  $y$  taking on values yes/no depending on whether or not the client did/did not subscribe to the bank term deposit. For details and a description of the different columns, visit <https://archive.ics.uci.edu/ml/datasets/Bank+Marketing>.

Furthermore, I'll do the analysis via one of the R front ends to Apache Spark and MLlib: **SparkR**<sup>u</sup> [Venkataraman et al., 2016]. Starting with Spark 3.1.1, **SparkR** also provides a distributed data frame implementation that supports common data wrangling operations like selection, filtering, aggregation, etc. (similar to using **data.table** or **dplyr** with R data frames, but on large data sets that don't fit into memory). For instructions on installing Spark, visit <https://spark.apache.org/docs/latest/sparkr.html>.

While the bank marketing data contain 21 columns on 41,188 records, this is by no means "Spark territory." However, you may find yourself in a situation where you need to use Spark MLlib for scalable analytics, so I think it's useful to show how to perform common statistical and machine learning tasks in Spark and MLlib, like fitting RFs, assessing performance, and computing PD plots.

To start, I'll download a zipped file containing a directory with the full bank marketing data set. The code downloads the zipped file into a temporary directory, unzips it, and reads in the CSV file of interest. (If the following code does not work for you, then you may find it easier to just manually download the `bank-additional-full.csv` and read it into R on your own.)

---

<sup>u</sup>This analysis can easily be translated to any other front end to Spark, including **sparklyr** or **pyspark**.

```
url <- paste0("https://archive.ics.uci.edu/ml/machine-learning",
              "-databases/00222/bank-additional.zip")
temp <- tempfile(fileext = ".zip") # to store zipped file
download.file(url, destfile = temp)
bank <- read.csv(unz(temp, "bank-additional/bank-additional-full.csv"),
                 sep = ";", stringsAsFactors = FALSE)
unlink(temp) # delete temporary file
```

Next, I'll clean up the data a bit. First off, I'll replace the dots in the column names with underscores; Spark does not like dots in column names! Second, I'll coerce the response (`y`) from a factor (`no/yes`) to a binary indicator (0/1) and treat it as numeric to fit a probability forest/machine. Finally, I'll remove the column called `duration`. Too often have I seen online analyses of the same data, only for the analyst to be fooled into thinking that `duration` is a useful indicator of whether or not a client will subscribe to a bank term deposit. If you take care and read the data documentation, you'd notice that the value of `duration` is not known before a call is made to a client. In other words, the value of `duration` is not known at prediction time and therefore cannot be used to train a model. This is a textbook example of target leakage. KNOW YOUR DATA! Finally, the data are split into train/test sets using a 50/50 split; I could do this manually, but here I'll use the `caret` package's `createDataPartition()` function, which uses stratified sampling to ensure that the distribution of classes is similar between the resulting partitions<sup>v</sup>:

```
names(bank) <- gsub("\\.", replacement = "_", x = names(bank))
bank$y <- ifelse(bank$y == "yes", 1, 0)
bank$duration <- NULL # remove target leakage

# Split data into train/test sets using a 50/50 split
set.seed(1056)
trn.id <- caret::createDataPartition(bank$y, p = 0.5, list = FALSE)
bank.trn <- bank[trn.id, ] # training data
bank.tst <- bank[-trn.id, ] # test data
```

Next, I'll load `ggplot2`, `SparkR`, and initialize a `SparkSession`—the entry point into `SparkR` (see `?SparkR::sparkR.session` for details and the various Spark properties that can be set). If you're new to Spark, start with the online documentation: <https://spark.apache.org/docs/latest/index.html>; you can also find links to `SparkR` here as well. Note that `SparkR` is not on CRAN, but is included with a standard install of Apache Spark. To load `SparkR` in an existing R session<sup>w</sup>, say, in RStudio, you need

---

<sup>v</sup>Several other packages could also be used here, like `rsample` [Silge et al., 2021], for example.

<sup>w</sup>You can also start an R session with `sparkR` already available from the terminal by running `./bin/sparkR` from your Spark home folder; for details, see <https://spark.apache.org/docs/latest/sparkr.html>

to tell ) the location of the package. (Note that the code snippet below may need to change for you depending on where you have Spark installed; for me, it's in C:\spark\spark-3.0.1-bin-hadoop2.7\R\lib.)

```
library(SparkR, lib.loc = "C:\\spark")
library(ggplot2)

# Start a local connection to Spark using all available cores
sparkR.session(master = "local[*]")
```

Next, I'll apply MLlib's Spark-enabled RF algorithm by calling `spark.randomForest()`; here, I'll use  $B = 500$  trees with a max depth of 10. Note that **SparkR** works with Spark DataFrames, not R data frames, so I have to coerce our train/test sets to Spark DataFrames using `createDataFrame()` before applying any Spark operations (ideally, I'd read the original data into a Spark DataFrame directly and process the data using Spark operations, but I was being lazy):

```
bank.trn.sdf <- createDataFrame(bank.trn)
bank.tst.sdf <- createDataFrame(bank.tst)

# Fit a regression/probability forest
bank.rfo <- spark.randomForest(
  bank.trn.sdf, y ~ ., type = "regression",
  numTrees = 500, maxDepth = 10, seed = 1205
)
```

To assess the performance of the probability forest, I can compute the Brier score on the test set. A couple of things are worth noting about the code chunk below. First, the `predict()` method, when applied to a **SparkR** MLlib model, returns the predictions along with the original columns from the supplied Spark DataFrame. Second, note that I have to compute the Brier score using Spark DataFrame operations, like **SparkR**'s `summarize()` function, in this case).

```
p <- predict(bank.rfo, newData = bank.tst.sdf) # Pr(Y=yes|x)
head(summarize(p, brier_score = mean((p$prediction - p$y)^2)))
```

```
#>   brier_score
#> 1  0.07815544
```

The AUC on the test set for this model, if you care purely about discrimination, is 0.798<sup>x</sup>, which is in line with some of the even more advanced analyses I've seen on these data. Nice! In addition, [Figure 7.27](#) shows an isotonic regression-based calibration curve (left) and cumulative gains chart, both com-

---

<sup>x</sup>Even if discrimination is the goal, AUC does not take into account the prior class probabilities and is not necessarily appropriate in situations with severe class imbalance; in this case the area under the PR curve would be more informative [Davis and Goadrich, 2006].

puted from the test data. The model seems reasonably calibrated (as we would hope from a probability forest). The cumulative gains chart tells us, for example, that we could expect roughly 1,500 subscriptions by contacting the top 20% of clients with the highest predicted probability of subscribing.

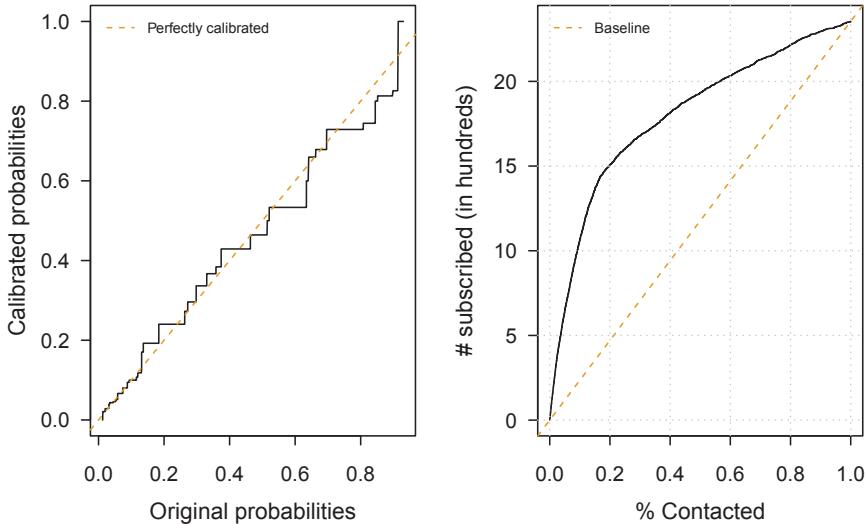


FIGURE 7.27: Graphical assessment of the performance on the test set. Left: isotonic regression-based calibration curve. Right: Cumulative gains chart.

Next, I'll look at the RF-based variable importance scores. Unfortunately, **SparkR** does not return the variable importance scores from tree-based models in a friendly format; it just gives one long nasty string, as can be seen below<sup>y</sup>:

```
rfo.summary <- summary(rfo) # extract summary information
(vi <- rfo.summary$featureImportances) # gross...
#> [1] "(52,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...
```

Nonetheless, I can use some *regular expression* (regex) magic to parse the output into a more friendly data frame. (By no means do I claim this to be the best solution; I'm certainly no regexpert—ba dum tsh.)

```
vi <- substr(vi, start = regexpr(", ", text = vi)[1] + 1,
             stop = nchar(vi) - 1)
vi <- gsub("\\\\[", replacement = "c(", x = vi)
```

<sup>y</sup>It is not clearly documented which variable importance metric MLlib uses in its implementation of RFs, but I suspect it's the impurity-based metric (Section 7.5) since, as far as I'm aware, MLlib's RF implementation does not support the notion of OOB observations (Section 7.3).

```

vi <- gsub("\\]", replacement = ")",
vi <- paste0("cbind(", vi, ")")
vi <- as.data.frame(eval(parse(text = vi)))
names(vi) <- c("feature.id", "importance")
vi$feature.name <- rfo.summary$features[vi[, 1] + 1]
head(vi[order(vi$importance, decreasing = TRUE), ], n = 10)

#>   feature.id importance      feature.name
#> 52       51     0.1915    nr_employed
#> 51       50     0.1447    euribor3m
#> 44       43     0.1141      pdays
#> 1        0      0.0711      age
#> 50       49     0.0496  cons_conf_idx
#> 48       47     0.0469  emp_var_rate
#> 49       48     0.0341  cons_price_idx
#> 43       42     0.0329    campaign
#> 45       44     0.0192    previous
#> 47       46     0.0171 poutcome_failure

```

The output suggests that the number of employees (`nr_employed`), a quarterly economic indicator, the Euribor 3 month rate (`euribor3m`<sup>z</sup>), a daily economic indicator, and the number of days passed since the client was last contacted from a previous campaign (`pdays`) are important predictors.

To further investigate the effect of these features on the model output, we can look at feature effect plots (such as PD and ICE plots). Here, I'll construct a PD plot for the economic indicator `euribor3m`<sup>aa</sup>. The trick to computing PD plots in Spark, if you can afford the memory, is to generate all the necessary data up front so that you only need one call to a scoring function. Once you have all the predictions, you can just post-process the results into partial dependence values by averaging the predictions within each unique value of the feature of interest; the same idea works for ICE plots as well (Section 6.2.3). We saw how to do this in base R in Section 6.2.5 using the Ames data. Even with large training data sets that don't fit into memory, the aggregated partial dependence values will be small enough to bring into memory as an ordinary R data frame and plotted using your favorite plotting library.

Following the same recipe outlined in Section 6.2.5, I'll start by creating a grid of values we want the PD plot for `euribor3m` to cover. For example, we can use an evenly spaced grid of points that covers the range of the predictor values of interest, or the sample quantiles; the latter has the benefit of potentially excluding outliers/extremes from the resulting plot. Since the data resides in a Spark data frame, we can't just use base R functionality. Luckily, **SparkR** provides the functionality we need via `approxQuantile()`, which we use to

---

<sup>z</sup>The 3 month Euribor rate is the interest rate at which a selection of European banks lend one another funds (denominated in euros) whereby the loans have a 3 month maturity.

<sup>aa</sup>You can find a similar example using `dplyr` with the `sparklyr` front end to Spark here: <https://github.com/bgreenwell/pdp/issues/97>.

construct a new Spark DataFrame containing only the plotting values for `euribor3m`. Then, we just need to create a Cartesian product with the original training data (excluding the variable `euribor3m`), or representative sample thereof. This is accomplished in the next code chunk.

A word of caution is in order. Even though Spark is designed to work with large data sets in a distributed fashion, Cartesian products can still be costly! Hence, if your learning sample is quite large (e.g., in the millions), which is probably the case if you're using MLlib, then keep in mind that you don't necessarily need to utilize the entire training sample for computing partial dependence and the like. If you have 50 million training records, for example, then consider only using a small fraction, say, 10,000, for constructing feature effect plots.

```
euribor3m.grid <- as.DataFrame(unique( # DataFrame of unique quantiles
  approxQuantile(bank.trn.sdf, cols = "euribor3m",
                  probabilities = 1:29 / 30, relativeError = 0)
))
names(euribor3m.grid) <- "euribor3m"

# Training data without euribor3m
trn.wo.euribor3m <- bank.trn.sdf # copy of training data
trn.wo.euribor3m$euribor3m <- NULL # remove euribor3m

# Create a Cartesian product
pd <- crossJoin(euribor3m.grid, trn.wo.euribor3m)
dim(pd) # nrow(euribor3m.grid) * nrow(trn.wo.euribor3m)

#> [1] 514850      20
```

Finally, we can compute the partial dependence values by aggregating the predictions using a simple grouping operator combined with a summary function (for PD plots, we just average the predictions). The results are displayed in [Figure 7.28](#). Here you can see that the relative frequency of exclamation marks is positively associated with spam (note that the *y*-axis is on the probability scale).

```
ggplot(pd, aes(x = euribor3m, y = yhat)) +
  geom_line() +
  geom_rug(data = as.data.frame(euribor3m.grid),
            aes(x = euribor3m), inherit.aes = FALSE) +
  xlab("Euribor 3 month rate") +
  ylab("Partial dependence") +
  theme_bw()

sparkR.stop() # stop the Spark session
```

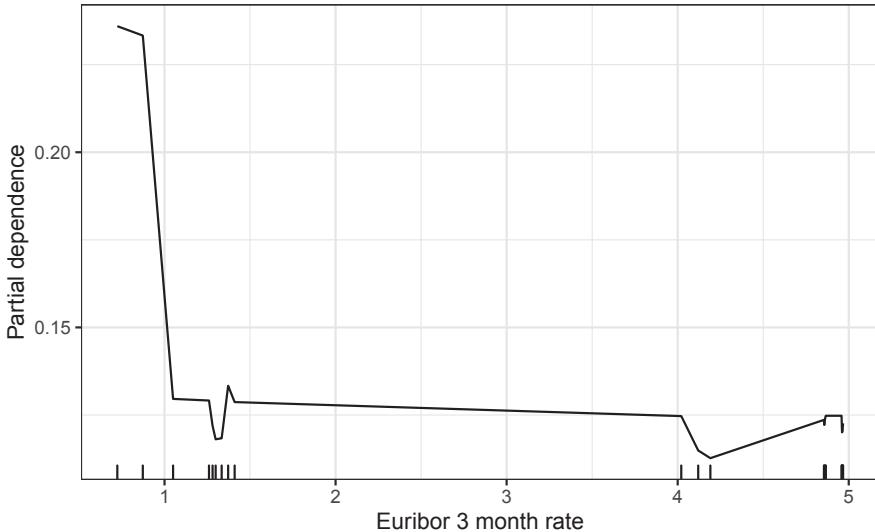


FIGURE 7.28: Partial dependence of subscription probability on Euribor 3 month rate from the bank marketing probability forest. The rug display along the  $x$ -axis summarizes the distribution of `euribor3m` via a grid of 29 evenly spaced quantiles.

## 7.10 Final thoughts

Yikes, that was a long chapter, but necessarily so. While RFs were originally introduced in Breiman [2001], many of the ideas have been seen before. For example, the term “random forest” was actually coined by Ho [1995], who used the *random subspace* method to combine trees grown in random subspaces of the original features. Breiman [2001] references several other attempts to further improve bagging by introducing more diversity among the trees in a “forest.”

Leo Breiman was a phenomenal statistician (and theoretical probabilist) who had a profound impact on the field of statistical and machine learning. If you’re interested in more of his work, especially on the development of RF, and the many collaborations it involved, see Cutler [2010]. Adele Cutler, a close collaborator with Breiman on RFs, still maintains their original RF website at <https://www.stat.berkeley.edu/~breiman/RandomForests/>. This website is still one of the best references to understanding Breiman’s original RF and includes links to several relevant papers and the original Fortran source code.

To end this chapter, I'll leave you with a quote listed under the philosophy section of Breiman and Cutler's RF website, which applies more generally than just to RFs:

---

Random forest is an example of a tool that is useful in doing analyses of scientific data. But the cleverest algorithms are no substitute for human intelligence and knowledge of the data in the problem. Take the output of random forests not as absolute truth, but as smart computer generated guesses that may be helpful in leading to a deeper understanding of the problem.

Leo Breiman and Adele Cutler

<https://www.stat.berkeley.edu/~breiman/RandomForests/>

---



Taylor & Francis  
Taylor & Francis Group  
<http://taylorandfrancis.com>

# 8

---

## Gradient boosting machines

---

One step at a time is all it takes to get you there.

Emily Dickinson

---

I like RFs because they’re powerful and flexible, yet conceptually simple: average a bunch of “de-correlated” trees together in the hopes of producing an accurate prediction. However, RF is not always the most efficient or most accurate tree ensemble to use. Gradient tree boosting provides another rich and flexible class of tree-based ensembles that, at a high level, I think is also conceptually simple. However, with gradient tree boosting, the devil is in the details.

In [Section 5.2.1](#), we were introduced to AdaBoost.M1, a particularly simple boosting algorithm for binary classification problems. Boosting initially started off as a way to improve the performance of weak binary classifiers. Over time, boosting has evolved into an incredibly flexible procedure that, like RFs, can handle a wide array of supervised learning problems.

In this chapter, I’ll walk through the basics of the most currently popular flavor of boosting: *stochastic gradient boosting*, also known as a *gradient boosting machine* (or GBM for short)<sup>a</sup>. Although GBMs are meant to be more general, in this book, GBM generally refers to stochastic gradient boosted decision trees.

---

<sup>a</sup>This flavor of boosting goes by several names in the literature. For example, the R package **gbm** [Greenwell et al., 2021b] fits this class of models, but stands for *generalized boosted models*.

## 8.1 Steepest descent (a brief overview)

In parametric modeling (i.e., where the form of the prediction function is known in advance), we are often concerned with estimating the parameters of a prediction function  $f(\mathbf{x}; \boldsymbol{\theta})$ , where  $\boldsymbol{\theta} \in \mathbb{R}^p$  is a  $p$ -dimensional vector of fixed but unknown parameters (e.g., the coefficients in a linear regression model). We do this by minimizing the “loss” in using  $f(\mathbf{x}; \boldsymbol{\theta})$  to predict  $y$  on a set of training data, where the loss function is defined as

$$L(\boldsymbol{\theta}) = \sum_{i=1}^N L[y_i, f(\mathbf{x}_i; \boldsymbol{\theta})].$$

The goal is to minimize  $L(\boldsymbol{\theta})$  with respect to  $\boldsymbol{\theta}$ . A simple example is least squares regression, where loss is defined as the sum of squared residuals (which I'll refer to henceforth as LS loss) and is minimized as a function of the coefficients  $\boldsymbol{\theta}$ . LS loss is analytically tractable and easy to solve for linear models. A more general approach for any differentiable loss function is to use numerical optimization methods to solve

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}). \quad (8.1)$$

A popular method for solving (8.1) is the method of *steepest descent*, a special case of the more general method of *gradient descent*. Gradient descent is a general class of iterative optimization algorithms that express the solution to (8.1) as a sum of components:

$$\boldsymbol{\theta}^* = \sum_{b=0}^B \boldsymbol{\theta}_b,$$

where  $\boldsymbol{\theta}_0$  is some initial guess and  $\{\boldsymbol{\theta}_b\}_{b=1}^B$  are successive “steps” or “boosts” towards the optimal solution  $\boldsymbol{\theta}^*$ , and are found through the update equation

$$\boldsymbol{\theta}_b = \boldsymbol{\theta}_{b-1} + \gamma \Delta \boldsymbol{\theta}_{b-1},$$

where  $\Delta \boldsymbol{\theta}_{b-1} = -\partial L(\boldsymbol{\theta}) / \partial \boldsymbol{\theta}$  is the *negative gradient* of  $L(\boldsymbol{\theta})$  with respect to  $\boldsymbol{\theta}$  and represents the direction of “steepest descent” of  $L(\boldsymbol{\theta})$ , and  $\gamma > 0$  is the *step size* taken in that direction. Steepest descent methods differ in how  $\gamma$  is determined.

Here's a pretty common analogy for explaining gradient descent without any math or fancy notation. Imagine trying to reach the bottom of a large hill (i.e., trying to find the global minimum) blindfolded. Without being able to see, and assuming you're not playing Marco Polo, you'll have to rely on what you can feel on the ground around you (i.e., local information) to find your way to the bottom. Ideally you'd feel around the ground at your current location ( $\boldsymbol{\theta}_{b-1}$ ) to get a sense of the direction of steepest descent ( $\Delta\boldsymbol{\theta}_{b-1}$ )—the fastest way down—and proceed in that direction while periodically reassessing which direction to go using the current local information available. How far you go in each direction is determined by your step size  $\gamma_b$ .

This simple analogy glosses over a number of details, like getting stuck in a hole (i.e., finding a local minimum), but hopefully the basic idea of gradient descent is relatively clear: to find the global minimum of  $L(\boldsymbol{\theta})$ , we take incremental steps in the direction of steepest descent, provided by the negative gradient of  $L(\boldsymbol{\theta})$  evaluated at the current point. The step size to take at each iteration can be fixed or estimated by solving another minimization problem:

$$\gamma_b = \arg \min_{\gamma} L(\boldsymbol{\theta}_{b-1} - \gamma \Delta\boldsymbol{\theta}_b). \quad (8.2)$$

In the latter case, finding  $\gamma_b$  by minimizing (8.2) is referred to as the “line search” along the direction of  $\Delta\boldsymbol{\theta}_b$ , and the overall procedure is referred to as the method of steepest descent. It's worth noting that oftentimes the solution to (8.2) is found via a simple approximation (e.g., using a single Newton-Raphson step).

---

## 8.2 Gradient tree boosting

Now, what does steepest descent have to do with boosting decision trees? Imagine trying to find some generic prediction function  $f$  such that

$$f^* = \arg \min_f L(f),$$

where  $L(f) = \sum_{i=1}^N L[y_i, f(\mathbf{x}_i)]$  is a loss function evaluated over the learning sample and encourages  $f$  to fit the data well [James et al., 2021, p. 302].

In contrast to (8.1), the parameters to be optimized here are the  $N$  fitted values  $\mathbf{f} \in \mathbb{R}^N$  from  $f(\mathbf{x}_i)$  found at each iteration evaluated at the training data  $\mathbf{x}_i$ :

$$\mathbf{f} = \{f(\mathbf{x}_i)\}_{i=1}^N.$$

Steepest descent in “function space” can be used to find the optimal  $\hat{\mathbf{f}}$  as the sum of  $B$   $N$ -dimensional component vectors:

$$\mathbf{f}_B = \sum_{b=0}^B \mathbf{f}_b, \quad \mathbf{f}_b \in \mathbb{R}^N,$$

where, similar to before,  $\mathbf{f}_0$  is an initial guess, and each component  $\mathbf{f}_b$  depends on the current estimate  $\mathbf{f}_{b-1}$ , and so forth. Steepest descent finds the next update using

$$\mathbf{f}_b = \mathbf{f}_{b-1} - \gamma \mathbf{g}_b,$$

where

$$\mathbf{g}_b = \left\{ \left[ \frac{\partial L(f)}{\partial f} \right]_{f=f_{b-1}(\mathbf{x}_i)} \right\}_{i=1}^N \quad (8.3)$$

is a length  $N$  column vector whose  $j$ -th component ( $g_{jb}$ ) is the gradient of  $L(\mathbf{f})$  evaluated at  $f = f_{b-1}(\mathbf{x}_j)$ .

One drawback is that the gradient components  $\mathbf{g}_b$  in (8.3) are only defined at the observed training observations  $\mathbf{x}_i$  ( $i = 1, 2, \dots, N$ ), whereas we want the final prediction function  $f_M(\mathbf{x})$  to be defined at new data points; otherwise, how would we make new predictions? Further, the procedure does not take into account the fact that observations with similar feature values are likely to have similar predictions [Ridgeway, 1999]. To this end, Friedman [2001] proposed using a class of functions that make use of the predictor information to approximate the gradient at each step. In gradient tree boosting (the focus of this chapter), for example, a regression tree is used to approximate the gradient at each step. In particular, at each step, we fit a  $J$ -terminal node regression tree<sup>b</sup>, which has the form

$$f(\mathbf{x}_i; \boldsymbol{\theta}, \mathbf{R}) = \sum_{j=1}^J \theta_j I(\mathbf{x}_i \in R_j),$$

where  $\boldsymbol{\theta} = \{\theta_j\}_{j=1}^J$  represents the terminal node estimates (i.e., the mean response in each terminal node),  $\mathbf{R} = \{R_j\}_{j=1}^J$  represents the disjoint regions

---

<sup>b</sup>Typically, a CART-like tree, but any regression tree would, in theory, work here.

that form the  $J$  terminal nodes, and  $I(\cdot)$  is the usual indicator function that evaluates to one whenever its argument is true (and zero otherwise). By fitting a model—a regression tree, in this case—to the observed negative gradient means we can define it at new data points. Using regression trees, the update becomes

$$\begin{aligned} f_b(\mathbf{x}) &= f_{b-1}(\mathbf{x}) + \gamma_b \sum_{j=1}^J \theta_{jb} I(\mathbf{x} \in R_{jb}) \\ &= f_{b-1}(\mathbf{x}) + \sum_{j=1}^J \gamma_{jb} I(\mathbf{x} \in R_{jb}). \end{aligned}$$

Consequently, the line search for choosing the step size  $\gamma_b$  is equivalent to updating the terminal node estimates using the specified loss function:

$$\{\gamma_{jb}\}_{j=1}^J = \arg \min_{\{\gamma_j\}_{j=1}^J} \sum_{i=1}^N L \left[ y_i, f_{b-1}(\mathbf{x}_i) + \sum_{j=1}^J \gamma_j I(\mathbf{x}_i \in R_{jb}) \right]. \quad (8.4)$$

Following Friedman [2001], since the  $J$  terminal node regions are disjoint, we can rewrite (8.4) as

$$\gamma_{jb} = \arg \min_{\gamma} \sum_{\mathbf{x}_i \in R_{jb}} L[y_i, f_{b-1}(\mathbf{x}_i) + \gamma], \quad (8.5)$$

which is the optimal constant update for each terminal node region,  $\{R_{jb}\}_{j=1}^J$ , based on the specified loss function  $L$  and the current iteration  $f_{b-1}(\mathbf{x}_i)$ . Solving (8.5) is equivalent to fitting a *generalized linear model* with an *offset*<sup>c</sup> [Efron and Hastie, 2016, p. 349]. This step is quite important since, for some loss functions, the original terminal node estimates will not be accurate enough. For example, with *least absolute deviation* (LAD) loss (see Section 8.2.0.1), the observed negative gradient,  $\mathbf{g}_b$ , only takes on integer values in  $\{-1, 1\}$ ; hence, the fitted values are not likely to be very accurate. In summary, the “line search” step (8.5) modifies the terminal node estimates of the current fit to minimize loss.

---

<sup>c</sup>Roughly speaking, an offset is an adjustment term (in this case, a fixed constant) to be added to the predictions in a model; this is more common in generalized linear models where it’s added to the *linear predictor* with a fixed coefficient of one (rather than an estimated coefficient).

For LS loss, (8.5) becomes

$$\begin{aligned}\gamma_{jb} &= \arg \min_{\gamma} \sum_{\mathbf{x}_i \in R_{jb}} ([y_i - f_{b-1}(\mathbf{x}_i)] - \gamma)^2 \\ &= \arg \min_{\gamma} \sum_{\mathbf{x}_i \in R_{jb}} (r_{i,b-1} - \gamma)^2 \\ &= \frac{1}{N_{jb}} \sum_{\mathbf{x}_i \in R_{jb}} r_{i,b-1}\end{aligned},$$

where  $r_{i,b-1}$  and  $N_{jb}$  are the  $i$ -th residual and number of observations in the  $j$ -th terminal node for the  $b$ -th iteration, respectively. This results in the mean of the residuals in each terminal node at the  $b$ -th iteration, which is precisely what the original regression tree induced at iteration  $b$  uses for prediction (i.e., the terminal node summaries). In other words, for the special case of LS loss, the original terminal node estimates at the  $b$ -th iteration are already optimal, and so no update (i.e., line search) is needed.

For LAD loss,  $L(f) = |y - f(\mathbf{x})|$ , and (8.5) results in the median of the current residuals in the  $j$ -th terminal node at the  $b$ -th iteration. Solving (8.5) can be difficult for general loss functions, like those often used in binary or multinomial classification settings, and fast approximations are often employed (see [Section 8.2.0.1](#)).

The full gradient tree boosting algorithm is presented in Algorithm 8.1. Note that this is the original gradient tree boosting algorithm proposed in Friedman [2001]. Several variations have been proposed in the literature, each with their own enhancements, and I'll discuss some of these modifications in the sections that follow.

### 8.2.0.1 Loss functions

Various boosting algorithms can be defined by specifying different (surrogate) loss functions  $L(y, f)$  in Algorithm 8.1. While not a tuning parameter, it is important to use an appropriate loss function for the problem at hand—LS loss, for example, is not appropriate for every regression problem. There are several common loss functions often used in GBMs and their implementations, and a handful of these are described in [Table 8.1](#).

For regression, LS and LAD loss are common choices. LAD loss has the benefit of being robust in the presence of long-tailed error distributions and response outliers. Regardless of loss, gradient tree boosting is already robust to long-tailed distributions or outliers in the feature space due to the robustness of the individual base learners (in this case, regression trees). Recall that trees are invariant to strictly monotone transformations of the predictors (e.g., using  $x_j$ ,  $e^{x_j}$ , or  $\log(x_j)$  for the  $j$ -th predictor all produce the same results).

---

**Algorithm 8.1** Vanilla gradient tree boosting algorithm.

---

- 1) Initialize  $f_0(\mathbf{x}) = \arg \min_{\theta} \sum_{i=1}^N L(y_i, \theta)$  (a constant).
  - 2) For  $b$  in  $1, \dots, B$  do the following:
    - a) Compute the negative gradient, evaluated at the training data, to be used as the current working response
 
$$y_{ib}^* = - \left[ \frac{\partial L(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i)} \right]_{f(\mathbf{x}_i)=f_{b-1}(\mathbf{x}_i)}, \quad i = 1, 2, \dots, N.$$
    - b) Fit a regression tree with  $J_b$  terminal node regions using CART's level-wise tree growing strategy:  $R_{jb}$ ,  $j = 1, 2, \dots, J_b$ .
    - c) Update the terminal node predictions using
 
$$\gamma_{jb} = \arg \min_{\gamma} \sum_{\mathbf{x}_i \in R_{jb}} L(y_i, f_{b-1}(\mathbf{x}_i) + \gamma), \quad j = 1, 2, \dots, J_b.$$
    - d) Update  $f_b(\mathbf{x})$  as
 
$$f_b(\mathbf{x}) \leftarrow f_{b-1}(\mathbf{x}) + \sum_{j=1}^{J_b} \gamma_{jb} I(\mathbf{x} \in R_{jb}).$$
  - 3) Return  $\hat{f}(\mathbf{x}) = f_B(\mathbf{x})$ .
- 

Consequently, there is little need to be concerned with transformations of the features in most tree-based ensembles. For outcomes with normally distributed errors (or at least approximately so), LAD loss will be less efficient than LS loss and generalization performance will suffer. A happy compromise is provided by the Huber loss function for *Huber M-regression* described in Friedman [2001] and Hastie et al. [2009, p. 360]. The Huber loss function provides resistance to outliers and long-tailed error distributions while maintaining high efficiency in cases where errors are more normally distributed.

Compared to exponential loss (Section 5.2.4), using *binomial deviance* (or log loss) for binary outcomes provides some robustness to mislabeled examples [Hastie et al., 2009, Section 10.6 ]. For the binary case with  $y \in \{0, 1\}$ , the binomial deviance can be written as

$$\begin{aligned} L(y, f) &= -[y \log(p) + (1-y) \log(1-p)] \\ &= \log[1 + \exp(-2\tilde{y}f)], \end{aligned}$$

where  $\tilde{y} = 2y - 1 \in \{-1, +1\}$ <sup>d</sup> and  $f$  refers to half the log odds for  $y = +1$ . With binomial deviance, there is no closed-form solution to the line search (8.5) in Algorithm 8.1, and approximations are often used instead. For example, a single Newton-Raphson step yields

$$\gamma_{jb} = \sum_{\mathbf{x}_i \in R_{jb}} \tilde{y}_i / \sum_{\mathbf{x}_i \in R_{jb}} |\tilde{y}_i| (2 - |\tilde{y}_i|).$$

The final approximation  $\hat{f}(\mathbf{x})$ , which is half the logit for  $y = +1$ , can be inverted to produce a predicted probability. The binomial deviance can also be generalized to the case of multiclass classification [Friedman, 2001].

More specialized loss functions also exist when dealing with other types of outcome variables. For example, Poisson loss (Table 8.1) can be used when modeling counts (which are always positive integers). Ridgeway [1999] showed how gradient boosting is extendable to the exponential family<sup>e</sup>, via likelihood-based loss functions, as well as Cox proportional hazards regression models for censored outcomes. Greg Ridgeway is also the original creator of the R package **gbm**, which is arguably the first open source implementations of gradient boosted decision trees.

TABLE 8.1: Common loss functions for gradient tree boosting. The top and bottom sections list common loss functions used for ordered and binary outcomes, respectively.

Loss name	Loss function	Negative gradient
Least squares	$\frac{1}{2} [y_i - f(\mathbf{x}_i)]^2$	$y_i - f(\mathbf{x}_i)$
Least absolute deviation	$ y_i - f(\mathbf{x}_i) $	$\text{sign}[y_i - f(\mathbf{x}_i)]$
Poisson deviance <sup>f</sup>	$y_i f(\mathbf{x}_i) - e^{f(\mathbf{x}_i)}$	$y_i - e^{f(\mathbf{x}_i)}$
Exponential	$\exp[-y_i f(\mathbf{x}_i)]$	$y_i \exp[-y_i f(\mathbf{x}_i)]$
Binomial deviance <sup>g</sup>	$\log[1 + \exp(-2\tilde{y}_i f(\mathbf{x}_i))]$	$2\tilde{y}_i / (1 + \exp[2\tilde{y}_i f(\mathbf{x}_i)])$

<sup>d</sup>This re-encoding is done for computational efficiency and also results in the same population minimizer as exponential loss (i.e., Adaboost.M1 from Section 5.2.4); see Bühlmann and Hothorn [2007] for details.

<sup>e</sup>The exponential family includes many common loss functions as a special case; for example, the Gaussian family is equivalent to using LS loss, the Laplace distribution is equivalent to using LAD loss, and the Bernoulli/binomial family is equivalent to using binomial deviance (or log loss).

<sup>f</sup>Here,  $y_i \in \{0, 1, 2, \dots\}$  is a non-negative integer (e.g., the number of people killed by mule or horse kicks in the Prussian army per year, or the number of calls to a customer support center on a particular day).

<sup>g</sup>Here,  $\tilde{y}_i \in \{-1, 1\}$  and  $f(\mathbf{x}_i)$  refers to half the log odds for  $y = +1$ .

### 8.2.0.2 Always a regression tree?

Another subtle difference from bagging or RFs is that GBMs always use regression trees for the base learner, even for classification problems! It makes sense if you think about it: gradient tree boosting involves fitting a tree to the negative gradient of the loss function (i.e., pseudo residuals) at each iteration, which is always ordered and treated as continuous.

### 8.2.0.3 Priors and missclassification cost

In [Section 2.2.4](#), I discussed how CART can naturally incorporate specific class priors and unequal misclassification costs (through a loss/cost matrix) when used for classification. Unfortunately, these concepts do not carry over to GBMs since the base learners are always regression trees; hence, there is no concept of false positives or negatives. One workaround is to use loss functions that incorporate case weights, which would allow us to give more weight to different subsets of the data (e.g., the underrepresented class). For example, we can incorporate case weights into the LS loss function by using

$$\frac{1}{\sum_{i=1}^N w_i} \sum_{i=1}^N w_i [y_i - f(\mathbf{x}_i)]^2,$$

where  $\{w_i\}_{i=1}^N$  are positive case weights that affect the terminal node estimates. For example, a terminal node in a regression tree with values 2, 3, and 8 would be given estimate of  $(2 + 3 + 8) / 3 = 4.333$  under equal case weights (the default). However, if the corresponding case weights were 1, 5, and 1, then the terminal node estimate would become  $[(1 \times 2) + (5 \times 3) + (1 \times 8)] / (1 + 5 + 1) = 3.571$ . See Kriegler and Berk [2010] for an example on boosted quantile regression with case weights for small area estimation. The discussion in Berk [2008, [Sec. 6.5](#)] is also worth reading.

## 8.3 Hyperparameters and tuning

Gradient tree boosting provides a powerful and flexible approach to predictive modeling. The flexibility in choosing a loss function allows you to fit a rich class of models to all kinds of response outcomes. The flexibility and state-of-the-art performance come at the price of a relatively large number of tuning parameters, which fall into two categories: boosting-specific and tree-specific hyperparameters. These will be discussed in the next two sections. Several modern implementations of GBMs include even more tunable

parameters, especially around regularization. These will be discussed briefly in [Section 8.8](#).

### 8.3.1 Boosting-specific hyperparameters

The two boosting-specific parameters associated with gradient tree boosting are the number of trees in the ensemble ( $B$ ) and a *shrinkage* parameter  $\nu$  which Friedman also discussed in Friedman [2001]. These are arguably the two most impact tuning parameters associated with GBMs, and will be discussed first, starting with  $B$ .

#### 8.3.1.1 The number of trees in the ensemble: $B$

---

If it's not using early stopping, it's crap.

Pafka [2020]

---

Unlike bagging and RFs, GBMs can overfit as the number of trees in the ensemble ( $B$ ) increases, all else held constant. This is evident from [Figure 8.1](#), which shows the training error (black curve) and 5-fold CV error (yellow curve) for a sequence of boosted regression stumps fit to the Ames housing data ([Section 1.4.7](#)) using LS loss. While the training error will continue to decrease to zero as the number of trees ( $B$ ) increases, at some point the validation error will start to increase. Consequently, it is important to tune the number of boosting iterations  $B$ .

So how many boosting iterations should you try? In part, it depends on a number of things, including the values set for other hyperparameters. In essence, you want  $B$  large enough to adequately minimize loss and small enough to avoid overfitting. For smaller data sets, like the PBC data ([Section 1.4.9](#)), it's easy enough to fix  $B$  to an arbitrarily large value ( $B = 2000$ , say) and use a method like cross-validation to select an optimal value, assuming  $B$  was large enough to begin with. In the baseball hitters example, the optimal number of trees found by 5-fold CV is 52 (dashed blue line). For larger data sets, this approach can be wasteful, which is where *early stopping* comes in.

The idea of early stopping is rather simple. At each iteration, we keep track of the overall performance using some form of cross-validations or a separate validation set. When the model stops improving by a prespecified amount, the

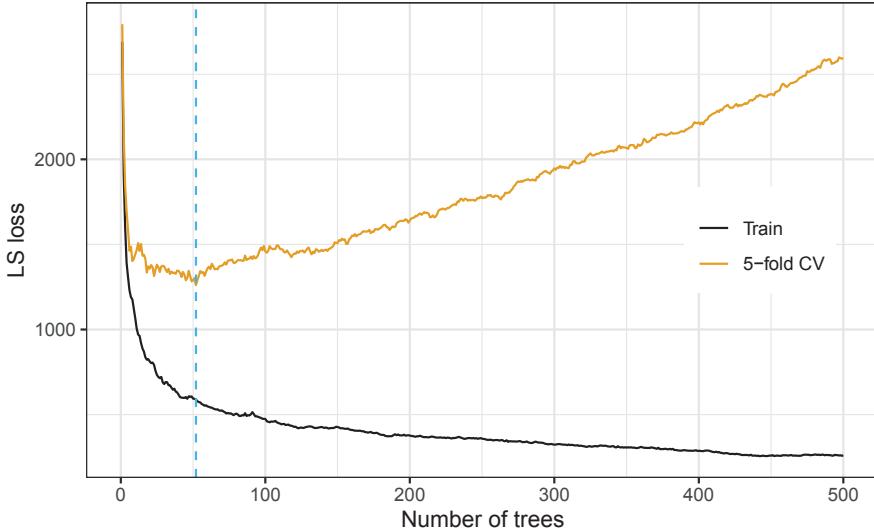


FIGURE 8.1: Gradient boosted decision stumps for the Ames housing example. The training error (black curve) continues to decrease as the number of trees increases, while the error based on 5-fold CV eventually starts to increase after  $B = 83$  trees (dashed blue vertical line), indicating a problem with overfitting for larger values of  $B$ .

process “stops early” and the model is considered to have converged. Early stopping is really a mechanic of the implementation, not the GBM algorithm itself, and so it is not necessarily supported in all implementations of gradient tree boosting. Note that the concept of early stopping can be applied to other iterative or ensemble methods as well, like RFs (Chapter 7). Implementations of GBMs that support early stopping are discussed in Section 8.9. The next section deals with a tuning parameter that’s intimately connected to the number of boosted trees, the *learning rate*.

### 8.3.1.2 Regularization and shrinkage

Regularization methods attempt to prevent overfitting by constraining the fitting procedure. There are two main approaches to regularization in GBMs: 1) controlling the number of terms/base learners  $B$ , which we discussed in Section 8.3.1.1, and 2) an explicit shrinkage parameter; the latter was introduced by Friedman [2001] to help prevent overfitting. It effectively reduces the influence of each individual tree leaving more room for future trees to improve the model. In particular, step 2d) of Algorithm 8.1 is replaced with

$$f_b(\mathbf{x}) \leftarrow f_{b-1}(\mathbf{x}) + \nu \sum_{j=1}^{J_b} \gamma_{jb} I(\mathbf{x} \in R_{jb}),$$

where  $\nu \in (0, 1]$  is a shrinkage, or regularization parameter, sometimes also referred to as the learning rate. The two parameters  $B$  and  $\nu$  are not independent. Each one can control the degree of fit and thus affect the best value of the other. Decreasing  $\nu$  increases the best value for  $B$  but can also result in an appreciable increase in generalization performance. All else held constant, by decreasing the learning rate, you have to fit more trees to reach optimal performance, which results in a smoother performance curve. It is generally the case that for small shrinkage parameters, say,  $\nu = 0.001$ , there is a fairly long plateau in which predictive performance is at its best, making it harder to overfit compared to using a relatively larger learning rate.

### 8.3.1.3 Example: predicting ALS progression

Here, I'll look at a brief example using the ALS data from Efron and Hastie [2016, p. 349]. A description of the data, along with the original source and download instructions, can be found at

<https://web.stanford.edu/~hastie/CASI/>.

The data concern  $N = 1,822$  observations on *amyotrophic lateral sclerosis* (ALS or Lou Gehrig's disease) patients. The goal is to predict ALS progression over time, as measured by the slope (or derivative) of a functional rating score (**dFRS**), using 369 available predictors obtained from patient visits. The data were originally part of the DREAM-Phil Bowen ALS Predictions Prize4Life challenge. The winning solution [Küffner et al., 2015] used a tree-based ensemble quite similar to an RF, while Efron and Hastie [2016, Chap. 17] analyzed the data using GBMs (as I'll do in this chapter). I'll show a fuller analysis of these data in [Sections 8.9.2–8.9.3](#).

[Figure 8.2](#) shows the performance of a (very) basic implementation of gradient tree boosting with LS loss using **treemisc**'s **lsoobst()** function (see [Section 8.5](#)) applied to the ALS data. Here, we can see the test MSE as a function of the number of trees using two different learning rates: 0.02 (black curve) and 0.50 (yellow curve) (following Efron and Hastie [2016, p. 339], these are boosted regression trees of depth three). Using  $\nu = 0.50$  results in overfitting much quicker. The performance curve for  $\nu = 0.50$  is also less smooth than for  $\nu = 0.02$ . While not spectacularly different, using  $\nu = 0.02$  results in a slightly more accurate model (in terms of MSE on the test set), but requires far more trees. For comparison (and as a sanity check against **treemisc**'s overly simplistic **lsoobst()** function), I also included the results from a popular open source implementation of gradient boosting called

XGBoost (to be discussed further in [Section 8.8.1](#)). For comparison, a default RF using 250 trees produced a test MSE of 0.261.

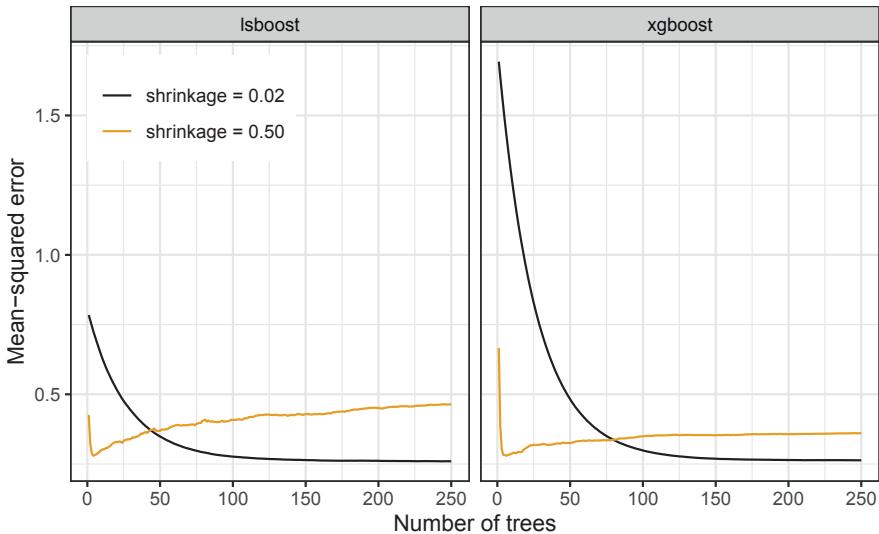


FIGURE 8.2: Gradient boosted depth-three regression trees for the ALS data using two different learning rates: 0.02 (black curve) and 0.50 (yellow curve). Left: results from our own `lsboost()` function. Right: results from XGBoost.

### 8.3.2 Tree-specific hyperparameters

The number of terminal nodes  $J$  controls the size (i.e., complexity) of each tree in gradient tree boosting and plays the role of an important tuning parameter for capturing interaction effects. Alternatively, you can control tree size by specifying the maximum depth. In general, a tree with maximum depth  $d$  can capture interactions up to order  $d$ . Note that a tree of depth  $d$  will have at most  $2^d$  terminal nodes and  $2^d - 1$  splits. A binary tree with  $J$  terminal nodes contains  $J - 1$  splits and can capture interactions up to order  $J - 1$ . (A  $J - 1$ -th order interaction is known as a  $J$ -way interaction effect; hence,  $J = 1$  corresponds to an additive model with no interaction effects). The documentation for scikit-learn's implementation of GBMs<sup>h</sup> notes that controlling tree size with  $J$  seems to give comparable results to using  $d = J - 1$  "...but is significantly faster to train at the expense of a slightly higher training error."

<sup>h</sup>See the “Controlling the tree size” section of the scikit-learn documentation at <https://scikit-learn.org/stable/modules/ensemble.html>.

Hastie et al. [2009, p. 363] suggest that  $4 \leq J \leq 8$  (or  $2 \leq d \leq 3$ ) works well in general (with results not being too sensitive to different values in this range) and that  $J > 10$  is rarely necessary. In many cases, a simpler additive model (i.e.,  $d = 1$  or  $J = 2$ ) is sufficient.

### 8.3.3 A simple tuning strategy

I don't think that grid searches are all that useful for GBMs, and tend to be too costly for large data sets, especially if early stopping is not available. A simple and effective tuning strategy for GBMs is to leave the tree-specific hyperparameters at their defaults (discussed in the previous sections) and tune the boosting parameters. A rule of thumb proposed by Greg Ridgeway<sup>i</sup> is to set shrinkage as small as possible while still being able to fit the model in a reasonable amount of time and storage. For example, aim for 3,000–10,000 iterations with shrinkage rates between 0.01–0.001; use early stopping, if it's available. More elaborate tuning strategies for GBMs are discussed in Boehmke and Greenwell [2020, Chap. 12].

---

## 8.4 Stochastic gradient boosting

Friedman [2002] proposed a minor modification to the original GBM algorithm in Friedman [2001], where he showed that both generalization performance and execution speed of GBMs can often be improved dramatically by incorporating additional randomization into the procedure, similar to how bagging can improve the performance of a single tree. The idea is to forgo using the entire training sample to fit each subsequent tree in the ensemble and instead use a subsample of the training data drawn at random without replacement; typically a 50% random subsample is used to induce each tree (i.e., roughly half the original training data), although, for larger data sets, a smaller fraction can be used. Due to the extra randomization step, the full procedure is referred to as stochastic gradient boosting and is the most common flavor of gradient boosting seen in practice today. Friedman [2002] suggests that  $0.5 \leq f \leq 0.8$  generally leads to an improvement for small to moderate sized data sets, where  $f$  is the fraction of the original training data sampled at random before building each tree.

As with bootstrap sampling in bagging and RFs, a happy by-product of subsampling in GBMs is the ability to produce an OOB estimate of the generalization error (Section 7.3). OOB estimates of error are similar to that obtained

---

<sup>i</sup>See the **gbm** package vignette: `vignette("gbm", package = "gbm")`.

using  $N$ -fold (or leave-one-out) cross-validation and computed at virtually no extra cost to the fitting algorithm. However, as stated in Section 7.3, the OOB approach can provide overly pessimistic estimates of the true error, but can still be used for hyperparameter tuning [Janitza and Hornung, 2018]. Since GBMs have lots of tuning parameters, the OOB approach provides a computationally feasible solution to selecting a reasonable learning rate, number of trees, etc.

It's important to note that Janitza and Hornung [2018] refer specifically to OOB-based error estimates for RFs, not GBMs. To this day, I have yet to see an extensive study on the usefulness of OOB-based error estimates in GBMs compared to more traditional cross-validation approaches.

#### 8.4.1 Column subsampling

Column subsampling is another technique that can be used to improve model performance and speed up fitting. Similar to column subsampling in an RF, a subsample of columns can be used for building each individual tree<sup>j</sup>. Apparently subsampling the columns prior to building each tree, can reduce the chances of overfitting even more than traditional row subsampling [Chen and Guestrin, 2016].

To illustrate, consider the test MSE curves for the ALS data displayed in Figure 8.3. In this example, subsampling the columns appears to outperform subsampling the rows (here, I arbitrarily chose a subsampling rate of 0.3). In practice these parameters need to be tuned, but it's probably safe and more computationally efficient in practice to just deal with one of these two hyperparameters. If you're dealing with a really wide data set, it may be more efficient to consider column subsampling, or both column subsampling and row subsampling if you have many rows as well.

---

### 8.5 Gradient tree boosting from scratch

Let's implement a quick-and-dirty gradient tree boosting function based on LS loss. The function, called `lsboost()`, is available in package `treemisc` (see `?treemisc::lsboost` for details and a description of the arguments), but the code is relatively straightforward and reproduced in the code chunk below. It's

---

<sup>j</sup>While similar, an RF chooses a random subsample of features prior to each split of every tree.

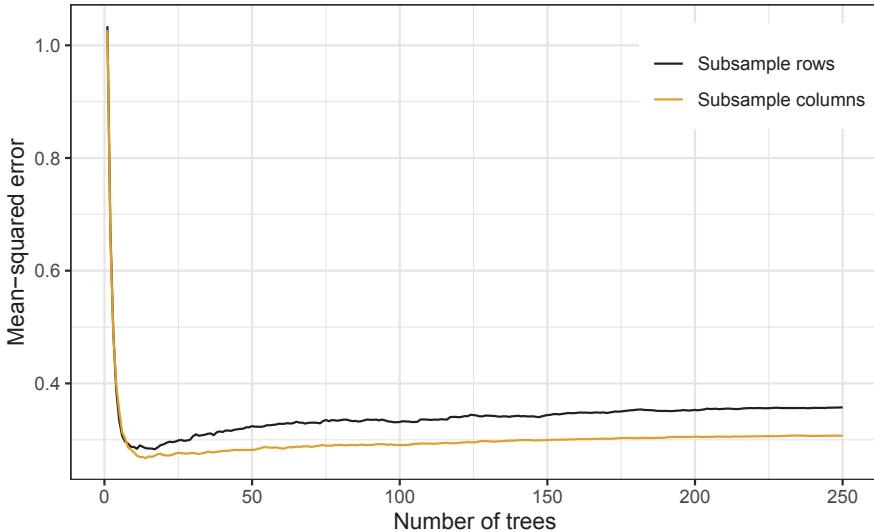


FIGURE 8.3: Effect of subsampling in GBMs on the ALS data. In this case, randomly subsampling the columns (yellow curve) slightly outperforms randomly subsampling the rows (black curve).

worth noting a few things about `lsboost()` and `predict.lsboost()`, before we continue:

- the code uses R's built-in S3 *object-oriented (OO) programming* system [Wickham, 2019, Chap. 13], which allows us to extend R's built-in `predict()` generic via the `predict.lsboost()` function (e.g., so we can compute predictions with, say, `predict(my.lsboost.model, newdata = some.data)`);
- `lsboost()` uses `rpart` to fit the individual regression trees, but other implementations could be used instead (e.g., CTrees via `partykit`);
- `lsboost()` returns an object of class "lsboost", which is essentially a list of `rpart` trees that the `predict()` function knows how to combine;
- these functions are for illustration and not meant for serious use—they are not optimized in any sense.

If you cut out the fluff, gradient tree boosting, at least with LS loss, can be implemented in as little as 10 lines of code (probably less):

```
lsboost <- function(X, y, ntree = 100, shrinkage = 0.1, depth = 6,
                     subsample = 0.5, init = mean(y)) {
  yhat <- rep(init, times = nrow(X)) # initialize fit; f_0(x)
  trees <- vector("list", length = ntree) # to store each tree
  ctrl <- # control tree-specific parameters
```

```

rpart::rpart.control(cp = 0, maxdepth = depth, minbucket = 10)
for (tree in seq_len(ntree)) { # Step 2) of Algorithm 8.1
  id <- sample.int(nrow(X), size = floor(subsample * nrow(X)))
  samp <- X[id, ] # random subsample
  samp$pr <- y[id] - yhat[id] # pseudo residual
  trees[[tree]] <- # fit tree to current pseudo residual
    rpart::rpart(pr ~ ., data = samp, control = ctrl)
  yhat <- yhat + shrinkage * predict(trees[[tree]], newdata = X)
}
res <- list("trees" = trees, "shrinkage" = shrinkage,
            "depth" = depth, "subsample" = subsample, "init" = init)
class(res) <- "lsboost"
res
}

# Extend R's generic predict() function to work with "lsboost" objects
predict.lsboost <- function(object, newdata, ntree = NULL,
                             individual = FALSE, ...) {
  if (is.null(ntree)) {
    ntree <- length(object[["trees"]]) # use all trees
  }
  shrinkage <- object[["shrinkage"]] # extract learning rate
  trees <- object[["trees"]][seq_len(ntree)]
  pmat <- sapply(trees, FUN = function(tree) { # all predictions
    shrinkage * predict(tree, newdata = newdata)
  }) # compute matrix of (shrunken) predictions; one for each tree
  if (isTRUE(individual)) {
    pmat # return matrix of (shrunken) predictions
  } else {
    rowSums(pmat) + object$init # return boosted predictions
  }
}

```

Gradient tree boosting with LS loss is simpler to implement because there's no need to perform the line search step in Algorithm 8.1 (i.e., the terminal node estimates are already optimal). A slightly more complicated function that also implements gradient tree boosting with **rpart**, but using LAD loss, is shown below; this function is also part of **treemisc** (see `?treemisc::ladboost` for details). Here, care needs to be taken to update the terminal node summaries accordingly (see the commented section starting with `# Line search`). For LAD loss, we simply use the terminal node sample medians, as discussed in [Section 8.2](#); here, I update the **frame** component of the **rpart** tree, but **partykit** could also be used, as illustrated in the commented out section. Also, note that the initial fit (`init`) defaults to the median response as well.

```

ladboost <- function(X, y, ntree = 100, shrinkage = 0.1, depth = 6,
                      subsample = 0.5, init = median(y)) {
  yhat <- rep(init, times = nrow(X)) # initialize fit
  trees <- vector("list", length = ntree) # to store each tree

```

```

ctrl <- # control tree-specific parameters
rpart::rpart.control(cp = 0, maxdepth = depth, minbucket = 10)
for (tree in seq_len(ntree)) {
  id <- sample.int(nrow(X), size = floor(subsample * nrow(X)))
  samp <- X[id, ]
  samp$pr <- sign(y[id] - yhat[id]) # use signed residual
  trees[[tree]] <-
    rpart::rpart(pr ~ ., data = samp, control = ctrl)
#-----
# Line search; update terminal node estimates using median
#-----
where <- trees[[tree]]$where # terminal node assignments
map <- tapply(samp$pr, INDEX = where, FUN = median)
trees[[tree]]$frame$yval[where] <- map[as.character(where)]
#
# Could use partykit instead:
#
# trees[[tree]] <- partykit::as.party(trees[[tree]])
# med <- function(y, w) median(y) # see ?partykit::predict.party
# yhat <- yhat +
#   shrinkage * partykit::predict.party(trees[[tree]],
#                                         newdata = X, FUN = med)
#-----
yhat <- yhat + shrinkage * predict(trees[[tree]], newdata = X)
}
res <- list("trees" = trees, "shrinkage" = shrinkage,
           "depth" = depth, "subsample" = subsample, "init" = init)
class(res) <- "ladboost"
res
}

```

### 8.5.1 Example: predicting home prices

Let's apply the `lsboost()` function to the Ames housing data. Below, I use the same train/test split for the Ames housing data we've been using throughout this book, then call `lsboost()` to fit a GBM to the training set; here, I'll use a shrinkage factor of  $\nu = 0.1$ :

```

library(treemisc)

# Split Ames data into train/test sets using a 70/30 split
ames <- as.data.frame(AmesHousing::make_ames())
ames$Sale_Price <- ames$Sale_Price / 1000 # rescale response
set.seed(4919) # for reproducibility
id <- sample.int(nrow(ames), size = floor(0.7 * nrow(ames)))
ames.trn <- ames[id, ]
ames.tst <- ames[-id, ]

```

```
# Fit a gradient tree boosted ensemble with 500 trees
set.seed(1110) # for reproducibility
ames.bst <-
  lsboost(subset(ames.trn, select = -Sale_Price), # features only
          y = ames.trn$Sale_Price, ntree = 500, depth = 4,
          shrinkage = 0.1)
```

The test RMSE as a function of the number of trees in the ensemble is computed below using the previously defined `predict()` method; the results are shown in [Figure 8.4](#) (black curve). For brevity, the code uses `sapply()` to essentially iterate cumulatively through the  $B = 500$  trees and computes the test RMSE for the first tree, first two trees, etc. For comparison, the test RMSEs from a default RF are also computed and displayed in [Figure 8.4](#) (yellow curve). In this example, the GBM slightly outperforms the RF.

```
set.seed(1128) # for reproducibility
ames.rfo <- # fit a default RF for comparison
  randomForest(subset(ames.trn, select = -Sale_Price),
               y = ames.trn$Sale_Price, ntree = 500,
               # Monitor test set performance (MSE, in this case)
               xtest = subset(ames.tst, select = -Sale_Price),
               ytest = ames.tst$Sale_Price)

# Helper function for computing RMSE
rmse <- function(pred, obs, na.rm = FALSE) {
  sqrt(mean((pred - obs)^2, na.rm = na.rm))
}

# Compute RMSEs from both models on the test set as a function of the
# number of trees in each ensemble (i.e.,  $B = 1, 2, \dots, 500$ )
rmses <- matrix(nrow = 500, ncol = 2) # to store results
colnames(rmses) <- c("GBM", "RF")
rmses[, "GBM"] <- sapply(seq_along(ames.bst$trees), FUN = function(B) {
  pred <- predict(ames.bst, newdata = ames.tst, ntree = B)
  rmse(pred, obs = ames.tst$Sale_Price)
}) # add GBM results
rmses[, "RF"] <- sqrt(ames.rfo$test$mse) # add RF results
```

---

## 8.6 Interpretability

Interpreting GBMs is no different from any other nonparametric model. For example, [Section 5.4](#) discussed how the individual tree-based importance scores ([Section 2.8](#)) can be aggregated across all the trees in an ensemble

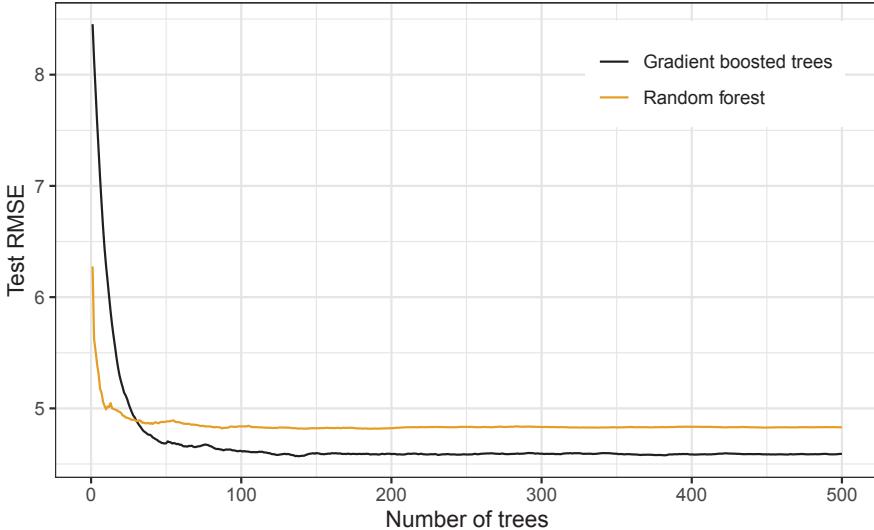


FIGURE 8.4: Root mean-squared error for the Ames housing test set as a function of  $B$ , the number of trees in the ensemble. Here, I show both a GBM (black curve) and a default RF (yellow curve). In this case, gradient tree boosting with LS loss, a shrinkage of  $\lambda = 0.1$ , and a maximum tree depth of  $d = 4$  (black curve) slightly outperforms a default RF (yellow curve).

to form a more stable measure of predictor importance; however, as with CART and RFs, this measure is also biased for GBMs, although, the permutation importance method (Section 6.1.1) applies equally well to GBMs, or any supervised learning model, for that matter. PDPs and ICE plots can be used to visualize the global and local effect that subsets of features have on the model’s predictions, respectively. Shapley values, among other techniques, can be used to infer the contribution each feature value has on the difference between its associated prediction and the model’s baseline (or average training) prediction, which can also be used to generate global measures of both feature importance and feature effects. The next two sections discuss specialized interpretability techniques often associated with GBMs.

### 8.6.1 Faster partial dependence with the recursion method

For regression trees based on single-variable splits, Friedman [2001] described a fast procedure for computing the partial dependence of  $\hat{f}(\mathbf{x})$  on a subset of features using a weight traversal of each tree (henceforth referred to as the

*recursion method*). In particular<sup>k</sup>, if a split node involves an input feature from the interest set ( $\mathbf{z}_c$ ), the corresponding left or right branch is followed; otherwise both branches are followed, each branch being weighted by the fraction of training observations that entered that branch. Finally, the partial dependence is given by a weighted average of all the visited terminal node values.

The idea is not specific to gradient tree boosting (e.g., it could also be applied to a RF), but as far as I'm aware, it's only implemented in a couple of open source packages: R's **gbm** package (which **pdp** takes full advantage of) and the **sklearn.inspection** module, which supports the recursion method only for certain tree-based estimators. Most other implementations rely on the brute force method described in [Section 6.2.1](#).

### 8.6.1.1 Example: predicting email spam

Let's illustrate with the email spam example ([Section 1.4.5](#)). Here, I used the R package **gbm** to fit a GBM using log loss,  $B = 4,043$  depth-2 regression trees (found using 5-fold cross-validation), a shrinkage factor of  $\nu = 0.01$ .

To gain an appreciation for the computational speed-up of the recursion method (which is implemented in **gbm**), I computed Friedman's  $H$ -statistic for all 1,596 pairwise interactions, which took roughly five minutes! The largest pairwise interaction occurred between **address** and **receive**. The partial dependence of the log-odds of spam on the joint frequencies of **address** and **receive** is displayed in [Figure 8.5](#). Using the fast recursion method, this took roughly a quarter of a second to compute, compared to the brute force method, which took almost 500 seconds.

## 8.6.2 Monotonic constraints

Increasing the interpretability of a model without sacrificing too much in the way of accuracy is useful in many real-world applications. For example, prior knowledge may be available (e.g., from subject matter experts or historical data) indicating that a given feature should in general have a positive (or negative) effect on the expected outcome. With GBMs, we can often increase interpretability by enforcing such *monotonic constraints*.

In gradient tree boosting, monotonic constraints enforce a specific splitting strategy in each of the constituent regression trees, where binary splits of a variable in one direction either always increase (monotone increasing) or decrease (monotone decreasing) the mean response in the resulting child node. For example, in a model with just two features,  $x_1$  and  $x_2$ , if we specified a

---

<sup>k</sup>Deets taken from the partial dependence documentation on scikit-learn's website: [https://scikit-learn.org/stable/modules/partial\\_dependence.html](https://scikit-learn.org/stable/modules/partial_dependence.html).

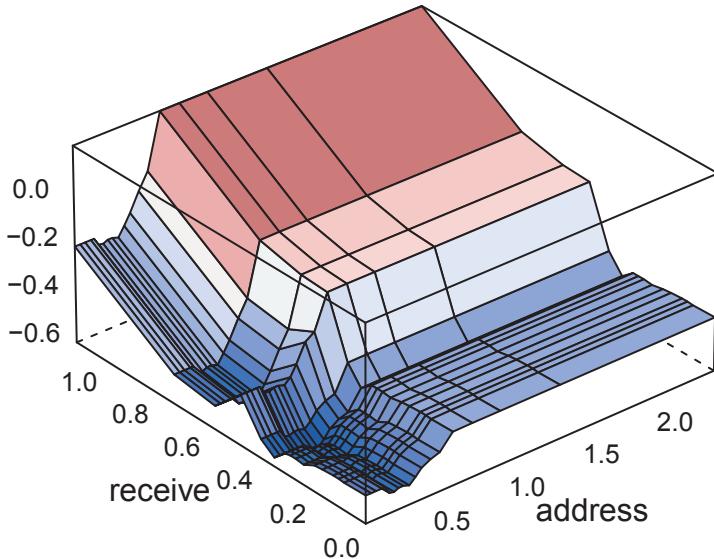


FIGURE 8.5: Partial dependence of log-odds of spam on joint frequency of `address` and `receive`.

monotonic increasing constraint on  $x_1$ , then for all  $x_1 \leq x'_1$  we would have  $f(x_1, x_2) \leq f(x'_1, x_2)$ . Such constraints are quite easy to visualize using partial dependence or ICE plots (Section 6.2), as briefly illustrated in the next section.

Enforcing monotonicity, where it makes sense, can make predictions more interpretable. For example, credit score is often used in determining whether to reject a loan or credit card application. If all the relevant features between two applicants are the same, aside from their current credit score, it might make sense to force the model to predict a lower probability of default for the applicant with the higher credit score. Such constraints are suitable for use in more regulated applications; for example, the likelihood of a loan approval is often higher with a better credit score. Gill et al. [2020] propose a mortgage lending workflow based on GBMs with monotonicity constraints, *explainable neural networks*, and Shapley values (Section 6.3.1), which gives careful consideration to US adverse action notice and anti-discrimination requirements.

### 8.6.2.1 Example: bank marketing data

Returning to the bank marketing example from Section 7.9.5, I fit a GBM with and without a decreasing monotonic constraint on `euribor3m`, the Euribor 3

month rate<sup>1</sup>. In both cases, I used 5-fold cross-validation to fit a GBM with a maximum of 3,000 trees using a shrinkage rate of  $\nu = 0.01$  and a maximum depth of  $d_{max} = 3$ . The partial dependence of the probability of subscribing on `euribor3m` from each model is displayed in Figure 8.6. Both figures tell the same story: the predicted probability of subscribing tends to decrease as the euribor 3 month rate increases. However, it may make sense here to assume the relationship to be monotonic decreasing, as in the left side of Figure 8.6. This can help increase interpretation and understanding by incorporating domain expertise, for example, by removing some of the noise like the little spike in the right side of Figure 8.6 near  $euribor3m = 1 = 1$ . Compare these to the RF-based PDP from Figure 7.28.

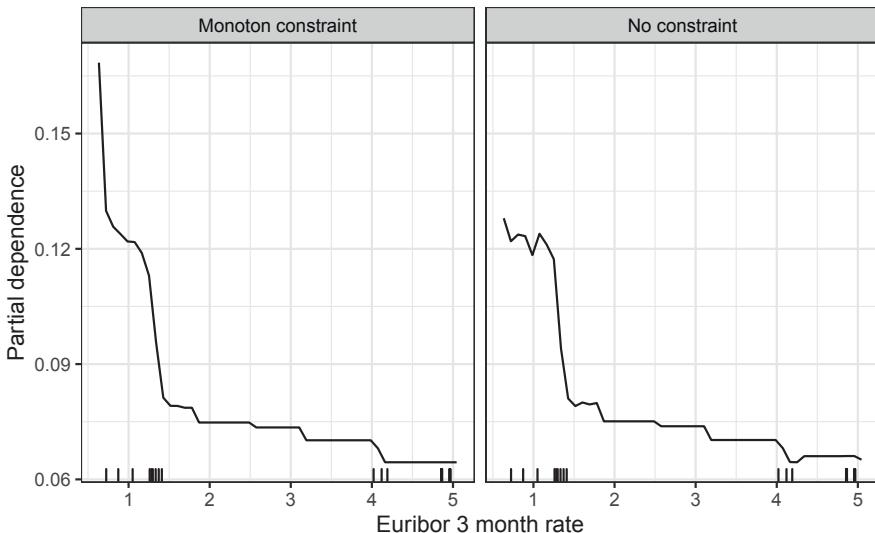


FIGURE 8.6: Partial dependence of subscription probability on euribor 3 month rate from the bank marketing probability forest. Left: with monotonic constraint on `euribor3m`. Right: without constraint. The rug display along the  $x$ -axis summarizes the distribution of `euribor3m` via a grid of 29 evenly spaced quantiles.

---

<sup>1</sup>Recall that the 3 month Euribor rate is the interest rate at which a selection of European banks lend each other funds (denominated in euros) whereby the loans have a 3 month maturity.

## 8.7 Specialized topics

### 8.7.1 Level-wise vs. leaf-wise tree induction

There are two strategies to consider when growing an individual decision tree that we have yet to discuss:

- *Level-wise* (also referred to as *depth-wise* or *depth first*) tree induction is used by many common decision tree algorithms (e.g., CART and C4.5/C5.0, but this probably depends on the implementation) and grows a tree level by level in a fixed order; that is, each node splits the data by prioritizing the nodes closer to the root node.
- *Leaf-wise* tree induction (also referred to as *best-first* splitting), on the other hand, grows a tree by splitting the node whose split leads to the largest reduction of impurity.

When grown to maximum depth, both strategies result in the same tree structure; the difference occurs when trees are restricted to a maximum depth or number of terminal nodes. Leaf-wise tree induction, while not specific to boosting, has primarily only been evaluated in that context; see, for example, Friedman [2001] and Shi [2007].

Figure 8.7 gives an example of a tree grown level-wise (left) and leaf-wise (right). Notice how the overall tree structures are the same, but the order in which the splits are made (i.e.,  $S_1$ – $S_4$ ) is different. In general, level-wise growth tends to work better for smaller data sets, whereas leaf-wise tends to overfit. Leaf-wise growth tends to excel in larger data sets where it is considerably faster than level-wise growth. This is why some modern GBM implementations—like LightGBM (Section 8.8)—default to growing trees leaf-wise.

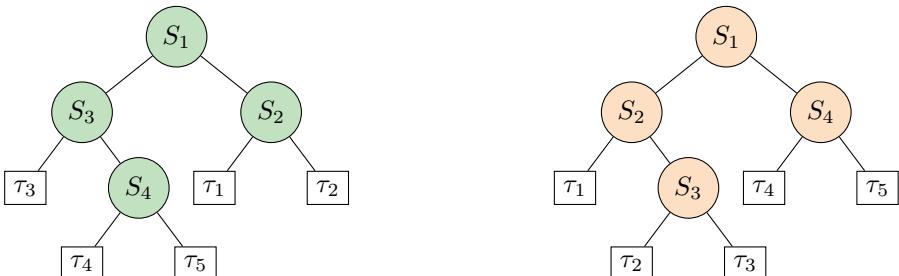


FIGURE 8.7: Hypothetical decision tree grown level-wise/depth-first (left) and leaf-wise/best-first (right).

### 8.7.2 Histogram binning

Finding the optimal split for a numeric feature in a decision tree can be slow when dealing with many unique values; the more unique values a numeric predictor has, the more split points the tree algorithm has to search through. A much faster alternative is to bucket the numeric features into bins using histograms.

The idea is to first bin the input features into integer-valued bins (255–256 bins seems to be the default across many implementations) which can tremendously reduce the number of split points to search through. Histogram binning is implemented in a number of popular GBM implementations, including XGBoost (Section 8.8.1), LightGBM (Section 8.8.2), and the `sklearn.ensemble` module. LightGBM’s online documentation lists several earlier references to this approach; visit <https://lightgbm.readthedocs.io/en/latest/Features.html> for details.

### 8.7.3 Explainable boosting machines

Explainable boosting machine (EBM) is an interpretable model developed at Microsoft Research [Nori et al., 2019]. In particular, an EBM is a tree-based, cyclic gradient boosting *generalized additive model* (GAM) with automatic interaction detection. The authors claim that EBMs are often as accurate as current state-of-the-art algorithms (like RFs and GBMs) while remaining completely interpretable. And while EBMs are often slower to train than other modern algorithms, they are extremely compact and fast at prediction time, which makes them attractive for deploying in a production process.

In essence, an EMB fits a GAM of the form:

$$g(\mathbb{E}[y|\mathbf{x}]) = \beta_0 + \sum_j f_j(x_j), \quad (8.6)$$

where  $g$  is a *link function* that connect the random and systematic component (e.g., adapts the GAM to different settings such as classification, regression, or Poisson regression), and  $f_j$  is a function of predictor  $x_j$ . Compared to a traditional GAM, an EBM:

- estimates each feature function  $f_j(x_j)$  using tree-based ensembles, like gradient tree boosting or bagging;
- can automatically detect and include pairwise interaction terms of the form  $f_{ij}(x_i, x_j)$ .

The overall boosting procedure is restricted to train on one feature at a time in a “round-robin” fashion using a small learning rate to ensure that feature

order does not matter, which helps limit the effect of collinearity or strong dependencies among the features.

EBMs are considered “glass box” or highly interpretable models because the contribution of each feature (or pairwise interaction) to a final prediction can be visualized and understood by plotting  $f_j(x_j)$ , similar to a PD plot (Section 6.2.1). And since EBMs are additive models, each feature contributes to predictions in a modular way that makes it easy to reason about the contribution of each feature to the prediction [Nori et al., 2019]. The simple additive structure of an EBM comes at the cost of longer training times. However, at the end of model fitting, the individual trees can be dropped and only the  $f_j(x_j)$  and  $f_{ij}(x_i, x_j)$  need to be retained, which makes EBMs faster at execution time. EBMs are available in the **interpret** package for Python. For more info, check out the associated GitHub repository at <https://github.com/interpretml/interpret>.

#### 8.7.4 Probabilistic regression via natural gradient boosting

Many classification tasks are inherently probabilistic. For example, probability forests (Section 7.2.1) can be used to obtain consistent probability estimates for the different class outcomes (i.e.,  $\Pr(y = j|\mathbf{x})$ ). Regression tasks, on the other hand, are typically not probabilistic and the predictions correspond to some location estimate of  $y|\mathbf{x}$ ; that is, the distribution of  $y$  conditional on a set of predictor values  $\mathbf{x}$ . For instance, the terminal nodes in a regression tree—which are used to compute fitted values and predictions—provide an estimate of the conditional mean  $E(y|\mathbf{x})$ . Often, it is of scientific interest to know about the probability of specific events conditional on a set of features, rather than a single point estimate like  $E(y|\mathbf{x})$ . In the ALS example, rather than using an estimate of the conditional mean  $\hat{f}(\mathbf{x}) = \hat{E}(\text{dFRS}|\mathbf{x})$  to predict ALS progression for a new patient, it might be more useful to estimate  $\Pr(\text{dFRS} < c|\mathbf{x})$ , for some constant  $c$ . This is where probabilistic regression/forecasting comes in.

Probabilistic regression models provide estimates of the entire probability distribution of the response conditional on a set of predictors, denoted  $\mathcal{D}_{\boldsymbol{\theta}}(y|\mathbf{x})$ , where  $\boldsymbol{\theta}$  represents the parameters of the conditional distribution. For example, the normal distribution has  $\boldsymbol{\theta} = (\mu, \sigma)$ ; examples include *generalized additive models for shape, scale, and location* (GAMLSS) [Rigby and Stasinopoulos, 2005], *Bayesian additive regression trees* (BART) [Chipman et al., 2010], and Bayesian deep learning. While several approaches to probabilistic regression exist, many of them are inflexible (e.g., GAMLSS), computationally expensive (e.g., BART), or inaccessible to non-experts (e.g., Bayesian deep learning) [Duan et al., 2020]. *Natural gradient boosting* (NGBoost) extends the simple ideas of gradient boosting to probabilistic regression by treating

the parameters  $\theta$  as targets for a multiparameter boosting algorithm similar to gradient boosting (Algorithm 8.1). We say “multiparameter” because NGBoost fits a separate model for each parameter at every iteration.

The “natural” in “natural gradient boosting” refers to the fact that NGBoost uses something called the *natural gradient*, as opposed to the ordinary gradient. The natural gradient provides the direction of steepest descent in *Riemannian space*; this is necessary since gradient descent in the parameter space is not gradient descent in the distribution space because distances don’t correspond. The important thing to remember is that NGBoost approximates the gradient of a proper scoring rule—similar to a loss function, but for predicted probabilities and probability distributions of the observed data—as a function of  $\theta$ . Compared to alternative probabilistic regression methods, NGBoost is fast, flexible, scalable, and easy to use. An example, albeit in Python, is given in Section 8.9.2. NGBoost is available in the **ngboost** package for Python. For more info, check out the NGBoost GitHub repository at <https://github.com/stanfordmlgroup/ngboost>.

---

## 8.8 Specialized implementations

In this section, I’ll take a look at three specialized implementations of GBMs that are quite popular for supervised learning tasks, probably due to their availability across platforms and ability to scale to incredibly large data sets, even on a single machine where the data cannot fit into memory.

### 8.8.1 eXtreme Gradient Boosting: XGBoost

XGBoost [Chen and Guestrin, 2016] is one of the most popular and scalable implementations of GBMs. While XGBoost follows the same principles as the standard GBM algorithm, there are some important differences, a few of which are listed below:

- more stringent regularization to help prevent overfitting;
- a novel sparsity-aware split finding algorithm;
- weighted quantile sketch for fast and approximate tree learning;
- parallel tree building (across nodes within a tree);
- exploits out-of-core processing for maximum scalability on a single machine;

- employs the deep-learning concept of *dropout* to mitigate the problem of *overspecialization* [Vinayak and Gilad-Bachrach, 2015].

With these differences, XGBoost can scale to billions of rows in distributed or memory-limited settings (e.g., a single machine), hence the “extreme” in extreme gradient boosting. It has also been shown to be more accurate than the more traditional implementation of GBM (e.g., the R package **gbm**. There are a number of interfaces to XGBoost, including R, Python (with a scikit-learn interface), Julia, Scala, Java, Ruby, and C++. There’s also a command-line interface. General tuning strategies for XGBoost (with examples in R) are given in Boehmke and Greenwell [2020, Section 12.5.2]. For more details, including installation, visit the XGBoost GitHub repository at <https://github.com/dmlc/xgboost>.

In contrast to ordinary GBMs, which use CART-like regression trees for the base learners (i.e., splits are determined using the sum of squares criteria described in [Section 2.3](#)), XGBoost uses a regularized second-order approximation to the loss function in its split search algorithm. Let  $g_{jb}$  (defined previously in [Section 8.2](#)) and  $h_{jb} = \partial^2 L(y_j, f_{b-1}) / \partial f_{b-1}^2$  be the gradient and *hessian* (i.e., second-order gradient) values of the loss function  $L$  at the  $b$ -th iteration evaluated at the  $j$ -th observation (what a mouthful!). Further, let  $I_L$  and  $I_R$  be the set of observations in the left and right daughter nodes resulting from split  $s$ . The set of instances in the parent node is denoted by  $I = I_L \cup I_R$ . In XGBoost, splits are selected that result in the largest gain (or reduction in loss) which can be written as

$$\mathcal{L}_{split} = \frac{1}{2} \left[ \frac{\left( \sum_{i \in I_L} g_i \right)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{\left( \sum_{i \in I_R} g_i \right)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{\left( \sum_{i \in I} g_i \right)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma. \quad (8.7)$$

For brevity, I dropped the  $b$  subscript in  $g_i$  and  $h_i$ . Here,  $\gamma \in [0, \infty)$  is the minimum gain required to make a further split in the tree, with larger values resulting in a more conservative algorithm (i.e., fewer splits). Similarly,  $\lambda \in [0, \infty)$  can be viewed as an  $L_2$  penalty (similar to the one used in *ridge regression*), with larger values resulting in a more conservative algorithm. Both parameters provide a form regularization that constrains the model’s complexity and guards against overfitting. Eventually, XGBoost included an additional regularization parameter,  $\alpha \in [0, 1]$ , which acts as an  $L_1$  penalty (similar to the one used in the LASSO). Consequently, these can be viewed as additional hyperparameters to be tuned.

Aside from the exhaustive greedy search employed by CART, XGBoost provides an optional approximate search algorithm that is more efficient in distributed settings or when the data cannot easily fit into memory. In essence, the continuous feature values are mapped to buckets formed by the percentiles of each feature’s distribution in the training data. The best split for each

feature is found by searching through this reduced set of candidate split values. For details, see Chen and Guestrin [2016]. A modification for weighted data, called a *weighted quantile sketch*, is also discussed in Chen and Guestrin [2016].

Around early 2017, XGBoost introduced *fast histogram binning* (Section 8.7.2) to even further push the boundaries of scale and computation speed. In contrast to the original approximate tree learning strategy, which generates a new set of bins for each iteration, the histogram method re-uses the bins over multiple iterations, and therefore is far better suited for large data sets. XGBoost also introduced the option to grow trees leaf-wise, as opposed to just level-wise (the default), which can also speed up fitting, albeit, at the risk of potentially overfitting the training data (Section 8.7.1).

Sparse data are common in many situations, including the presence of missing values and one-hot encoding. In such cases, efficiency can be obtained by making the algorithm aware of any sparsity patterns. XGBoost handles sparsity by learning an optimal “default” direction at each split in a tree. When an observation is missing for one of the split variables, for example, it is simply passed down the default branch. For details, see Chen and Guestrin [2016].

One drawback of XGBoost is that it does not currently handle categorical variables—they have to be re-encoded numerically (e.g., using one-hot encoding). However, at the time of writing this book, XGBoost has experimental support for categorical variables, although it’s currently quite limited. An example of using XGBoost is given in Section 8.9.4. Note that XGBoost can also be used to fit RFs in a distributed fashion; see the XGBoost documentation for details.

### 8.8.2 Light Gradient Boosting Machine: LightGBM

LightGBM [Ke et al., 2017] offers many of the same advantages as XGBoost, including sparse optimization, parallel tree building, a plethora of loss functions, enhanced regularization, bagging, histogram binning, and early stopping. A major difference between the two is that LightGBM defaults to building trees leaf-wise (or best-first). Unlike XGBoost, LightGBM can more naturally handle categorical features in a way similar to what’s described in Section 2.4. In addition, the LightGBM algorithm utilizes two novel techniques, *gradient-based one-side sampling* (GOSS) and *exclusive feature bundling* (EFB).

GOSS reduces the number of observations by excluding rows with small gradients, while the remaining instances are used to estimate the information gain for each split; the idea is that observations with larger gradients play a more important role in split selection. EFB, on the other hand, reduces the

number of predictors by bundling mutually exclusive features together (i.e., they rarely take nonzero values simultaneously). Both of these features allow LightGBM to speed up the training process, while maintaining accuracy.

LightGBM is available in both C, R, and Python. For details, visit the LightGBM GitHub repository at <https://github.com/microsoft/LightGBM>. Without access to GPUs, XGBoost and LightGBM are among the most efficient GBM implementations. If you have access to GPUs, XGBoost currently seems to have a slight edge. Like XGBoost, LightGBM can also be run in random forest mode; consult the LightGBM documentation for details.

### 8.8.3 CatBoost

While XGBoost and LightGBM seem to be the most popular implementations of GBMs, they didn't initially<sup>m</sup> handle categorical variables as well as another GBM variant called CatBoost [Dorogush et al., 2018, Prokhorenkova et al., 2017]. One of the main selling points of CatBoost is the ability to handle categorical variables without the need for numerically encoding them. From the CatBoost website:

---

Improve your training results with CatBoost that allows you to use non-numeric factors, instead of having to pre-process your data or spend time and effort turning it to numbers.

<https://catboost.ai/>

---

They also claim that CatBoost works reasonably well out of the box with less time needed to be spent on hyperparameter tuning.

In CatBoost, a process called *quantization* is applied to numeric features, whereby values are divided into disjoint ranges or buckets—this is similar to the approximate tree growing algorithm in XGBoost whereby numeric features are binned. Before each split, categorical variables are converted to numeric using a strategy similar to mean target encoding, called *ordered target statistics*, which avoids the problem of target leakage and reduces overfitting. CatBoost is currently available as a command line application in C++, but R and Python interfaces are also available. For further details and resources, visit the CatBoost website at <https://catboost.ai/>.

---

<sup>m</sup>Both XGBoost and LightGBM now support categorical features.

---

## 8.9 Software and examples

GBMs are implemented in a number of open source software packages. The original implementation of GBMs was called MART(tm), for *multiple additive regression trees*<sup>n</sup>. The R package **gbm** was probably one of the first available open source implementations of Friedman's original GBM algorithm, with extensions to boosting likelihood-based models, like exponential family and proportional hazards regression models. Another general R package for boosting is **mboost** [Hothorn et al., 2021a], which implements boosting for optimizing general risk functions utilizing component-wise (penalized) least squares estimates as base-learners for fitting various kinds of generalized linear and generalized additive models to potentially high-dimensional data (i.e., with **mboost** you can specify different base-learners for individual predictors).

The **sklearn.ensemble** module implements Friedman's original GBM algorithm with some additional optimizations, like early stopping and the option to use histogram binning.

XGBoost, LightGBM, and CatBoost are available in both R and Python, and all three support early stopping; the R packages **xgboost** and **lightgbm** are both available on CRAN (at least at the time of writing this book). H2O's implementation of GBM also supports early stopping. Several benchmarks [Pafka, 2019, 2021] comparing these as well as several other GBM implementations, are available from Szilard Pafka at <https://github.com/szilard/GBM-perf>. Check out his other repositories on GitHub for fantastic talks on GBMs and other machine learning algorithms and their implementations.

### 8.9.1 Example: Mayo Clinic liver transplant data

In this example, I'll return to the PBC data introduced in Section 1.4.9, where the goal is to model survival in patients with the autoimmune disease PBC. In Section 3.5.3, I fit a CTree model to the randomized subjects using log-rank scores. Here, I'll use the GBM framework to boost a *Cox proportional hazards* (Cox PH) model; see Ridgeway [1999] for details.

The Cox PH model is one of the most widely used models for the analysis of survival data. It is a semi-parametric model in the sense that it makes a parametric assumption regarding the effect of the predictors on the hazard

---

<sup>n</sup>MART, which evolved into a product called TreeNet(tm), is proprietary software available from Salford Systems, which is currently owned by MiniTab; visit <https://www.minitab.com/en-us/predictive-analytics/treenet/> for details.

function (or *hazard rate*) at time  $t$ , often denoted  $\lambda(t)$ , but makes no assumption regarding the shape of  $\lambda(t)$ ; since little is often known about  $\lambda(t)$  in practice, the Cox PH model is quite useful. The hazard rate—also referred to as the *force of mortality* or *instantaneous failure rate*—is related to the probability that the event (e.g., death or failure) will occur in the next instant of time, given the event has not yet occurred [Harrell, 2015, Sec. 17.3]; it's not a true probability since  $\lambda(t)$  can exceed one. Studying the hazard rate helps understand the nature of risk of time.

Extending Algorithm 8.1 to maximize Cox's log-partial likelihood (which is akin to minimizing an appropriate loss function) allows us to relax the linearity assumption, which assumes that the (possibly transformed) predictors are linearly related to the log hazard, and fit a richer class of models based on regression trees. Below, I load the **survival** package and recreate the same **pb2** data frame I used in [Section 3.5.3](#):

```
library(survival)

# Prep the data a bit
pb2 <- pbc[!is.na(pbc$trt), ] # use randomized subjects
pb2$id <- NULL # remove ID column
# Consider transplant patients to be censored at day of transplant
pb2$status <- ifelse(pb2$status == 2, 1, 0)
facs <- c("sex", "spiders", "hepato", "ascites", "trt", "edema")
for (fac in facs) { # coerce to factor
  pb2[[fac]] <- as.factor(pb2[[fac]])
}
```

Next, I'll fit a boosted PH regression model using the **gbm** package; details on the deviance/loss function used in Algorithm 8.1 can be found in the **gbm** package vignette: `vignette("gbm", package = "gbm")`. Here, I use  $B = 3000$ , a maximum tree depth of three, and a learning rate of  $\nu = 0.001$ . The optimal number of trees (`best.iter`) is determined using 5-fold cross-validation.

```
library(gbm)

set.seed(1551) # for reproducibility
pb2.gbm <- gbm(Surv(time, status) ~ ., data = pb2,
                 distribution = "coxph", n.trees = 3000,
                 interaction.depth = 3, shrinkage = 0.001,
                 cv.folds = 5)
(best.iter <- gbm.perf(pbc2.gbm, method = "cv", plot.it = FALSE))

#> [1] 1934
```

Below, I construct a Cleveland dot plot of the overall variable importance scores ([Section 5.4](#)) using **gbm**'s `summary` method; the results are displayed in [Figure 8.8](#):

```
vi <- summary(pbc2.gbm, n.trees = best.iter, plotit = FALSE)
dotchart(vi$rel.inf, labels = vi$var, xlab = "Variable importance")
```

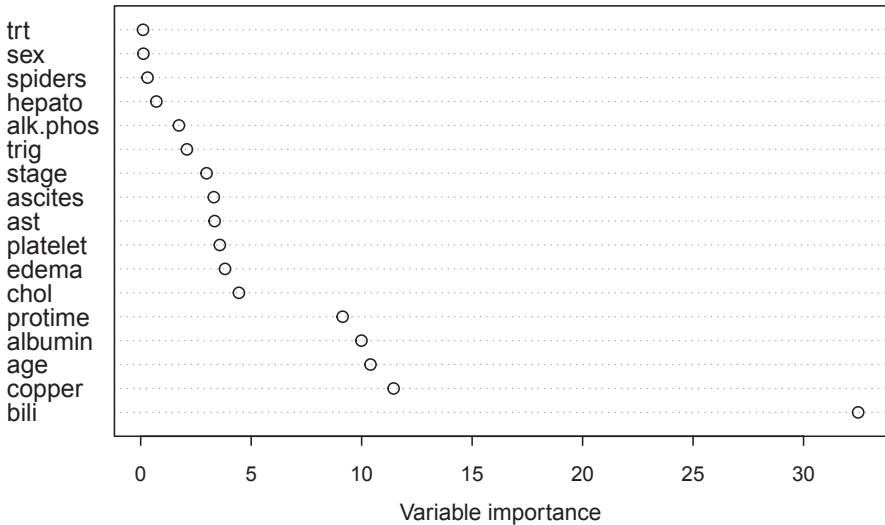


FIGURE 8.8: Variable importance plot from the boosted Cox PH model applied to the PBC data.

It looks as though serum bilirubin (mg/dl) (`bili`) is the most influential feature on the fitted model. We can easily investigate a handful of important features by constructing PDPs or ICE plots. In the next code chunk, I construct c-ICE plots (Section 6.2.3) for the top four features. The results are displayed in Figure 8.9; the average curve (i.e., partial dependence, albeit centered) is shown in red. Note that while `gbm` has built-in support for partial dependence using the recursion method (Section 8.6.1), it does not support ICE plots; hence, I'm using the brute force approach (`recursive = FALSE`) via the `pdp` package. The code essentially creates a list of plots, which is displayed in a 2-by-2 grid using the `gridExtra` package [Auguie, 2017]:

```
library(ggplot2)
library(pdp)

# Create list of c-ICE/PD plots for top 4 predictors
top4 <- c("bili", "copper", "age", "albumin")
pdps.top4 <- lapply(top4, FUN = function(x) {
  partial(pbc2.gbm, pred.var = x, check.class = FALSE,
          recursive = FALSE, n.trees = best.iter, ice = TRUE,
          center = TRUE, plot = TRUE, plot.engine = "ggplot2",
          rug = TRUE, alpha = 0.1) +
  ylab("Log hazard") # change default y-axis label
})
```

```
})
```

```
# Display list of plots in a grid
gridExtra::grid.arrange(grobs = pdps.top4, nrow = 2)
```

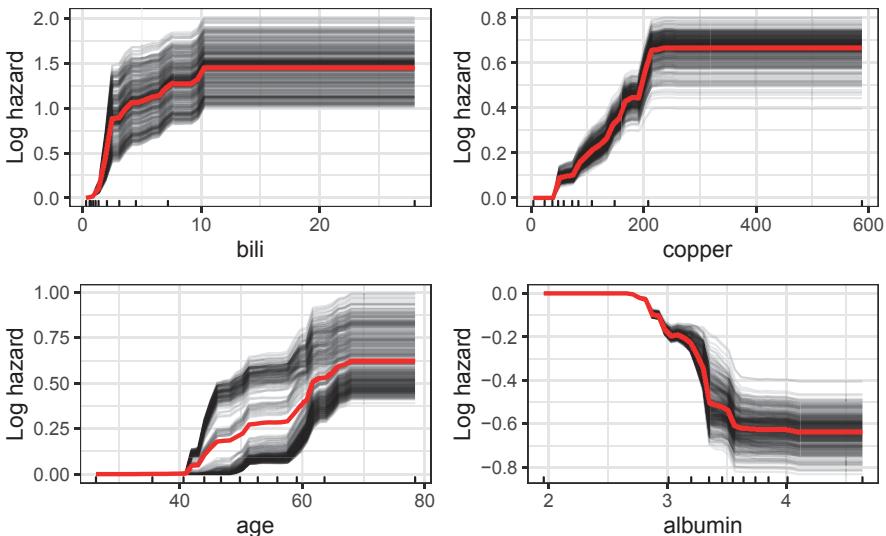


FIGURE 8.9: Main effect plots from a gradient boosted Cox PH model on the PBC data.

Each plot shows a relatively nonlinear monotonic effect on the predicted log hazard rate. For example, the predicted log hazard tends to increase with increasing serum bilirubin (mg/dl). The heterogeneity in the individual c-ICE curves for some of the predictors (e.g., `age`) suggests the presence of potential interaction effects. We can use Friedman's  $H$ -statistic (Section 6.2.2) to quantify the strength of potential pairwise interaction effects. The `gbm` function `interact.gbm()` can be used to obtain these statistics; see `?gbm:::interact.gbm` for details. Here, I create a simple wrapper function, called `gbm.2way()`, which uses `interact.gbm()` with R's built-in `combn()` function to measure the interaction strength between all possible pairs of predictors:

```
gbm.2way <- function(object, data, var.names = object$var.names,
                      n.trees = object$n.trees) {
  var.pairs <- combn(var.names, m = 2, simplify = TRUE)
  h <- combn(var.names, m = 2, simplify = TRUE, FUN = function(x) {
    interact.gbm(object, data = data, i.var = x, n.trees = n.trees)
  })
  res <- as.data.frame(t(var.pairs))
  res$h <- h
```

```

names(res) <- c("var1", "var2", "h")
res[order(h, decreasing = TRUE), ]
}

# Compute H-statistics for all pairs of predictors
pbc2.h <- gbm.2way(pbc2.gbm, data = pbc2, n.trees = best.iter)
head(pbc2.h, n = 5) # look at top 5

#>      var1     var2     h
#> 22     age     bili 0.1351
#> 29     age platelet 0.0749
#> 99     bili   protime 0.0703
#> 114    albumin protime 0.0701
#> 135 platelet   stage 0.0642

```

According to the  $H$ -statistic, the strongest interaction effects appear to occur between `age` and `bili`. This should not be surprising since the CTree fit to the same data in Section 3.5.3 showed `bili` as the first splitter, but `age` split below that, with `bili` again splitting below `age`, which suggest a potential interaction effect between the two. We can visualize this effect using a two-dimensional PDP, which I accomplish below using the `pdp` package. The result is displayed in Figure 8.10. Here, you can see that the effect of increasing serum bilirubin (mg/dl) on the predicted log hazard is stronger for older individuals.

```

pd <- partial(pbc2.gbm, pred.var = c("bili", "age"), chull = TRUE,
               check.class = FALSE, n.trees = best.iter)
autoplot(pd, legend.title = "PD") +
  xlab("Serum bilirubin (mg/dl)") +
  ylab("Age (years)")

```

Now suppose we wanted to understand why the model predicted a relatively high log hazard rate for a particular patient. One way to accomplish this, as we've already seen, is through Shapley-based feature contributions. To illustrate, I'll use the `fastshap` package to help understand which feature values contributed the most (and how) to the subject in the learning sample with the largest predicted log hazard.

To start, I'll grab the fitted values from the model (i.e., the predictions from the learning sample) and determine which subject had the highest predicted log hazard, before using `fastshap`'s `explain()` function to estimate the Shapley-based feature contributions (recall that we need to define and pass a prediction wrapper to `explain()` so it knows how to compute new predictions from the fitted model):

```

library(fastshap)

p <- predict(pbc2.gbm, newdata = pbc2, n.trees = best.iter)
max.id <- which.max(p) # row ID highest predicted log hazard

```

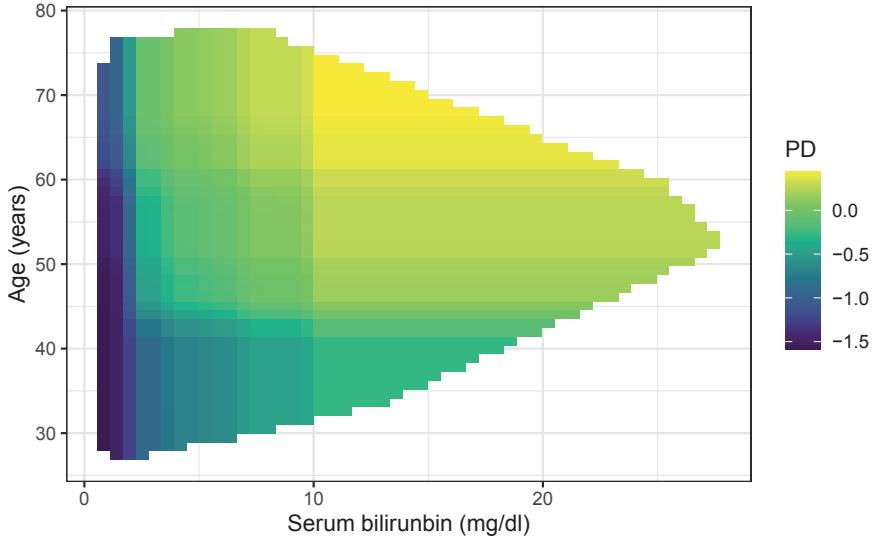


FIGURE 8.10: Partial dependence of log hazard on the joint value of `bili` and `age`.

```
# Define prediction wrapper for explain
pfun <- function(object, newdata) {
  predict(object, newdata = newdata, n.trees = best.iter)
}

# Estimate feature contributions for newx using 1,000 Monte Carlo reps
X <- pbc2[, pbc2.gbm$var.names] # feature columns only
newx <- pbc2[max.id, pbc2.gbm$var.names]
set.seed(1408) # for reproducibility
(ex <- explain(pbc2.gbm, X = X, nsim = 1000, pred_wrapper = pfun,
                newdata = newx))

##> # A tibble: 1 x 17
##>       trt    age      sex ascites hepato spiders
##>       <dbl> <dbl>    <dbl>    <dbl>    <dbl>
##> 1 -0.00168 0.313 -0.000342   0.282 -0.0175 -0.00492
##> # ... with 11 more variables: edema <dbl>, bili <dbl>,
##> #     chol <dbl>, albumin <dbl>, copper <dbl>,
##> #     alk.phos <dbl>, ast <dbl>, trig <dbl>,
##> #     platelet <dbl>, protime <dbl>, stage <dbl>
```

A great way to visualize such contributions is through a *waterfall chart*. I do so below using the R package `waterfall` [Howard, II, 2016]; the results are displayed in Figure 8.11. Here, we can see a bit more clearly the magnitude and direction of each feature value's contribution to the difference between the

current predicted log hazard and the overall average baseline—in essence, it shows how this subject went from the average baseline log hazard of -1 to their much higher prediction of 2.701. Note that the `waterfallchart()` function produces a `lattice` graphic, which behaves differently than base R graphics; hence, I use the `ladd()` function from package `mosaic` [Pruim et al., 2021] to add specific details to the plot (e.g., text labels and additional reference lines).

```
library(waterfall)

# Reshape Shapley values for plotting and include feature values
res <- data.frame("feature" = paste0(names(newx), "=",
                                         t(newx)),
                  "shapley.value" = t(ex))

# Waterfall chart of feature contributions
palette("Okabe-Ito")
waterfallchart(feature ~ shapley.value, data = res, origin = mean(p),
               summaryname = "f(x) - baseline", col = 2:3,
               xlab = "Log hazard")

mosaic::ladd(panel.abline(v = max(p), lty = 2, col = 1))
mosaic::ladd(panel.abline(v = mean(p), lty = 2, col = 1))
mosaic::ladd(panel.text(2.5, 8, labels = "f(x)", col = 1))
mosaic::ladd(panel.text(-0.55, 8, labels = "baseline", col = 1))
palette("default")
```

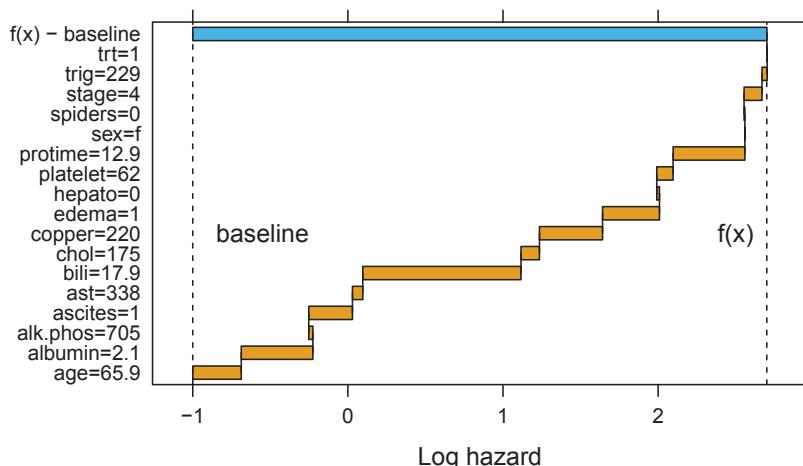


FIGURE 8.11: Waterfall chart showing the individual feature contributions to the subject with the highest predicted log hazard.

### 8.9.2 Example: probabilistic predictions with **NGBoost** (in Python)

In addition to **ngboost**, I'll require some additional packages: **pandas** [The Pandas Development Team, 2020], for reading and manipulating data; **numpy** [Harris et al., 2020], for basic numeric computations on arrays; and **scipy**, for basic statistical functions (like computing probabilities from distributions). In the Python chunk below, I load all the functionality I need before reading in the ALS data and splitting it into the same train/test sets used in the previous examples (note that the test set indicator used for splitting the data is included as a column named `testset`).

```
import numpy as np
import pandas as pd
import scipy.stats
from ngboost import NGBRegressor
from ngboost.distns import Normal

# Read in ALS data and split into train/test sets
url = "https://web.stanford.edu/~hastie/CASI_files/DATA/ALS.txt"
als = pd.read_csv(url, sep = " ")
als_trn = als[als["testset"] == False]
als_tst = als[als["testset"] == True]
X_trn = als_trn.drop(["testset", "dFRS"], axis=1) # features only
X_tst = als_tst.drop(["testset", "dFRS"], axis=1) # features only
```

Next, I initialize a **NGBRegressor** object, called `ngb`, with several parameters. In this example, I'll assume the distribution of `dFRS` conditional on a set of predictor values is normally distributed with some (unknown) mean and standard deviation, but note that **ngboost** supports several other distributions for regression and classification (yes, **ngboost** can also be used for classification tasks as well). As with ordinary GBMs, I also specify the number of base estimators and the learning rate. (The default base learner in **ngboost** is a depth-3 regression tree.)

```
ngb = NGBRegressor(Dist=Normal, n_estimators=2000, learning_rate=0.01,
                     verbose_eval=0, random_state=1601)
```

Now I can fit the actual model by calling the `fit()` on our `ngb` object. Here, I provide the test set to the validation parameters and use early stopping to determine at which iteration (or tree) the procedure should stop. Here, the procedure will stop if there has been no improvement in the negative log-likelihood (the default scoring rule in **ngboost**) in five consecutive rounds.

```
_ = ngb.fit(X_trn, Y=als_trn["dFRS"], X_val=X_tst,
            Y_val=als_tst["dFRS"], early_stopping_rounds=5)

#> == Early stopping achieved.
```

```
#> == Best iteration / VAL145 (val_loss=0.7495)
```

There are two prediction methods for `NGBRRegressor` objects: `predict()`, which returns point predictions as one would expect from a standard regression model, and `pred_dist()`, which returns a distribution object representing the conditional distribution of  $Y|\mathbf{x}_i$  for each observation  $\mathbf{x}_i$  in the scoring data set. First, let's compute point predictions for the entire data set and the corresponding MSE for comparison with our earlier GBM models on these data. Here, the results are rather similar in terms of accuracy.

```
pred = ngb.predict(X_tst)
np.mean(np.square(als_tst["dFRS"].values - pred))
#> 0.26773677346012037
```

Of more interest in probabilistic regression is an estimate of the conditional distribution itself, rather than some point estimate like the mean. In this case, I need estimates of the mean and standard deviation (since those parameters uniquely define a normal distribution). These are easily obtainable from the `params` component of the output from `pred_dist()`.

Below, I estimate these parameters for the first observation in the test set ( $\mathbf{x}_1$ ) and use them to estimate  $\Pr(dFRS < 0|\mathbf{x}_1)$ . The estimated mean and standard deviation (i.e., location and scale parameters of the normal distribution) for each observation of interest can also be used to obtain prediction intervals.

```
dist = ngb.pred_dist(X_tst.head(1)).params
dist
#> {'loc': array([-0.4649206]), 'scale': array([0.43861847])}
scipy.stats.norm(dist["loc"] [0], scale=dist["scale"] [0]).cdf(0)
#> 0.8554199300698616
```

### 8.9.3 Example: post-processing GBMs with the LASSO

Recall from [Section 5.5](#) that it is possible to reduce the number of base learners in a fitted ensemble via post-processing with the LASSO (hopefully without sacrificing accuracy). We saw this using the Ames housing data with a bagged tree ensemble and RF in [Sections 5.5.1](#) and [7.9.2](#), respectively.

It may seem redundant to include another ISLE post-processing example, but there's a subtle difference that can be overlooked with GBMs: the initial fit  $f_0(\mathbf{x})$  in Step 1) of Algorithm 8.1 essentially represents an offset.

To illustrate, I'll continue with the ALS example from [Section 8.9.2](#)<sup>o</sup>. Below, I read in the ALS data and split it into train/test sets using the provided `testset` indicator column; the `-1` keeps the indicator column out of the train/test sets.

```
# Read in the ALS data
url <- "https://web.stanford.edu/~hastie/CASI_files/DATA/ALS.txt"
als <- read.table(url, header = TRUE)

# Split into train/test sets
trn <- als[!als$testset, -1] # training data w/o testset column
tst <- als[als$testset, -1] # test data w/o testset column
X.trn <- subset(trn, select = -dFRS)
X.tst <- subset(tst, select = -dFRS)
y.trn <- trn$dFRS
y.tst <- tst$dFRS
```

Next, I call on our `lsboost()` function to fit a GBM using  $B = 1000$  depth-2 trees with a shrinkage factor of  $\nu = 0.01$  and a subsampling rate of 50%. I also compute test predictions from each individual tree and compute the cumulative MSE as a function of the number of trees. (Warning, the model fitting here will take a few minutes.)

```
library(treemisc)

set.seed(1122) # for reproducibility
lsb.fit <- lsboost(X.trn, y = y.trn, shrinkage = 0.01, ntree = 1000,
                     depth = 2, subsample = 0.5)

# Mean squared error function
mse <- function(y, yhat, na.rm = FALSE) {
  mean((y - yhat)^ 2, na.rm = na.rm)
}

# Compute test MSE as a function of the number of trees
preds.tst <- predict(lsb.fit, newdata = X.tst, individual = TRUE)
mse.boost <- sapply(seq_len(ncol(preds.tst)), FUN = function(ntree) {
  # Only aggregate predictions from first B/ntree trees
  pred.ntree <- rowSums(preds.tst[, seq_len(ntree)], drop = FALSE]) +
    lsb.fit$init # don't forget to add on the initial fit/mean response
  mse(y.tst, yhat = pred.ntree)
})
```

Next, I'll call upon the `isle_post()` function from package **treemisc** to post-process our fitted GBM using the LASSO. There's one important difference between this example and the previous ones applied to bagging and RF: with

---

<sup>o</sup>A special thanks to Trevor Hastie for clarification and sharing code from Efron and Hastie [2016, pp. 346–347], which greatly helped in producing this analysis (which is a detailed recreation of their example) and building out **treemisc**'s `isle_post()` function.

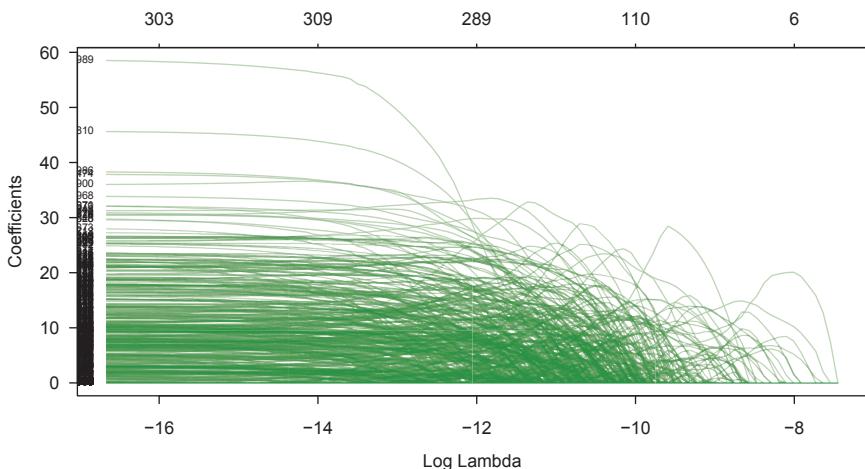


FIGURE 8.12: Coefficient path as a function of the  $L_1$ -penalty as  $\lambda$  varies. The top axis indicates the number of nonzero coefficients (i.e., number of trees) at the current value of  $\lambda$ . Here, we have one coefficient per tree in the full ensemble (the intercept is forced to be zero here).

boosting, we need to make sure we include the initial fit  $f_0(\mathbf{x})$ , which is stored in the "init" component of the output from `lgb.BoostingModel`. Recall that for LS loss  $f_0(\mathbf{x}) = \bar{y}$ , the mean response in the training data. This can be done in one of two ways:

- 1) arbitrarily add it to the predictions from the first tree;
  - 2) include it as an offset when fitting the LASSO and generating predictions.

In this example, I'll include the initial fit as an offset in the call to `isle_post()`.

The results are displayed in Figure 8.12 which contains the coefficient paths as a function of the  $L_1$ -penalty as  $\lambda$  varies. The top axis indicates the number of nonzero coefficients (i.e., number of trees) at the current value of  $\lambda$ . Here, the smallest test error for the LASSO-based post-processed GBM is 25.9% and corresponds to 84 trees; see Figure 8.13. The post-processing has significantly reduced the number of trees in this example resulting in a substantially more parsimonious model while maintaining accuracy. Sweet!

```
library(treemisc)
```

```
# Fit a LASSO model to the individual training predictions
preds.trn <- predict(lsb.fit, newdata = X.trn, individual = TRUE)
```

```

als.boost.post <- isle_post(preds.trn, y = y.trn, offset = lsb.fit$init,
                           newX = preds.tst, newy = y.tst,
                           family = "gaussian")

# Plot the coefficient paths from the LASSO model
plot(als.boost.post$lasso.fit, xvar = "lambda", las = 1, label = TRUE,
      col = adjustcolor("forestgreen", alpha.f = 0.3),
      cex.axis = 0.8, cex.lab = 0.8)

# Plot regularization path
palette("Okabe-Ito")
plot(mse.boost, type = "l", las = 1,
      ylim = range(c(mse.boost, als.boost.post$results$mse)),
      xlab = "Number of trees", ylab = "Test MSE")
lines(als.boost.post$results, col = 2)
abline(h = min(mse.boost), lty = 2)
abline(h = min(als.boost.post$results$mse), col = 2, lty = 2)
palette("default")

```

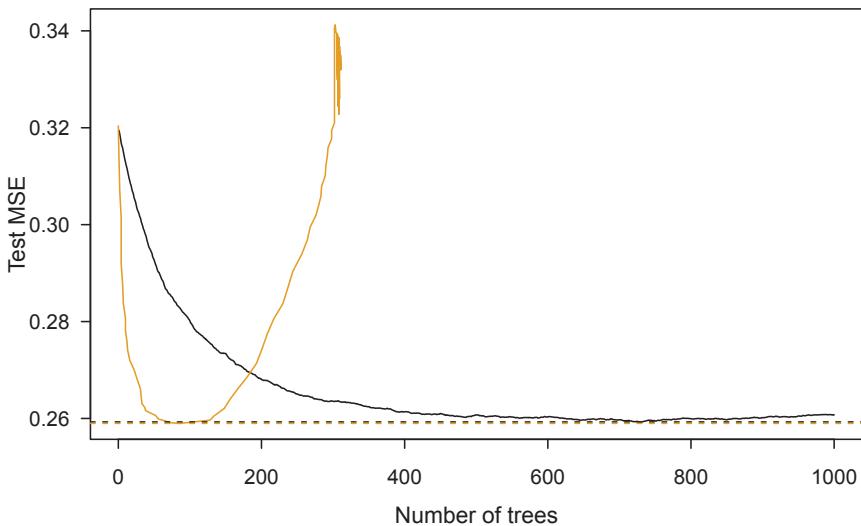


FIGURE 8.13: Test MSE as a function of the number of trees from the full GBM (black curve) and LASSO-based post-processed results (yellow curve). Here, we can see that by re-weighting the trees using an  $L_1$  penalty, which enables some the trees to be dropped entirely, we end up with a smaller more parsimonious model without degrading performance on the test set.

### 8.9.4 Example: direct marketing campaigns with XGBoost

In this example, I'll revisit the bank marketing data I analyzed back in [Section 7.9.5](#) using an RF in Spark. Here, I'll fit a GBM using the scalable XGBoost library and show the benefits to early stopping. For brevity, I'll omit the necessary code already shown in [Section 7.9.5](#). To that end, assume we already have the full data set loaded into a data frame called `bank`.

Next, similar to the RF-based analysis, I'll clean up some of the columns and column names. Since XGBoost requires all the data to be numeric<sup>P</sup>, we have to re-encode the categorical features. The binary variables I'll convert to 0/1, while categorical variables with higher cardinality will be transformed using one-hot encoding (OHE)<sup>q</sup>. First, I'll deal with the binary variables and column names:

```
names(bank) <- gsub("\\.", replacement = "_", x = names(bank))
bank$y <- ifelse(bank$y == "yes", 1, 0)
bank$contact <- ifelse(bank$contact == "telephone", 1, 0)
bank$duration <- NULL # remove target leakage
```

Next, I'll deal with the one-hot encoding. There are several packages that can help with this (e.g., `caret`'s `dummyVars()` function); I'll do the transformation using pure `data.table`. The code below identifies the remaining categorical variables (`cats`) and uses `data.table`'s `melt()` and `dcast()` functions to handle the heavy lifting; the left hand side of the generated formula (`fo`) tells `dcast()` which variables to not OHE (i.e., the binary and non-categorical features):

```
bank$id <- seq_len(nrow(bank)) # need a unique row identifier
cats <- names(which(sapply(bank, FUN = class) == "character"))
lhs <- paste(setdiff(names(bank), cats), collapse = "+")
fo <- as.formula(paste(lhs, "~ variable + value"))
bank <- as.data.table(bank) # coerce to data.table
bank.ohe <- dcast(melt(bank, id.vars = setdiff(names(bank), cats)),
                  formula = fo, fun = length)
bank$id <- bank.ohe$id <- NULL
```

Now that we have the data encoded properly for XGBoost, we can split the data into train/test sets using the same 50/50 split as before:

```
set.seed(1056) # for reproducibility
trn.id <- caret::createDataPartition(bank.ohe$y, p = 0.5, list = FALSE)
bank.trn <- data.matrix(bank.ohe[trn.id, ]) # training data
bank.tst <- data.matrix(bank.ohe[-trn.id, ]) # test data
```

---

<sup>P</sup>While XGBoost has limited (and experimental) support for categorical features, this does not seem to be accessible via the R interface, at least at the time of writing this book.

<sup>q</sup>Several of the categorical features are technically ordinal (e.g., `day_of_week`) and should probably be converted to integers.

XGBoost does not work with R data frames. The `xgb.train()` function, in particular, only accepts data as an "xgb.DMatrix" object. An XGBoost DMatrix is an internal data structure used by XGBoost, which is optimized for both memory efficiency and training speed; see `?xgboost::xgb.DMatrix` for details. We can create such an object using the `xgboost` function `xgb.DMatrix()` (note that I separate the predictors and response in the calls to `xgb.DMatrix()`):

```
library(xgboost)

xnames <- setdiff(names(bank.ohe), "y")
dm.trn <- xgb.DMatrix(bank.trn[, xnames], label = bank.trn[, "y"])
dm.tst <- xgb.DMatrix(bank.tst[, xnames], label = bank.tst[, "y"])
```

XGBoost has a lot of parameters, so it can be helpful to construct a list of such for use when calling the fitting function `xgb.train()`. Below, I create a list containing several boosting and tree-specific parameters:

```
params <- list(
  eta = 0.01, # shrinkage/learning rate
  max_depth = 3,
  subsample = 0.5,
  objective = "binary:logistic", # for predicted probabilities
  eval_metric = "rmse", # square root of Brier score
  nthread = 8
)
```

Finally, I can fit an XGBoost model. I'll fit two in total, one without early stopping and one with, starting with the no early stopping version below. But first, I'll define a “watch list,” which is just a named list of data sets to use for evaluating model performance after each iteration that we can use to determine the optimal number of trees ( $k$ -fold cross-validation could also be used via `xgboost`'s `xgb.cv()` function):

```
watch.list <- list(train = dm.trn, eval = dm.tst)

# Train an XGBoost model without early stopping
set.seed(1100) # for reproducibility
bank.xgb.1 <-
  xgb.train(params, data = dm.trn, nrounds = 3000, verbose = 0,
            watchlist = watch.list)
(best.iter <- which.min(bank.xgb.1$evaluation_log$eval_rmse))

#> [1] 1296
```

Out of 3,000 total iterations, we really only needed to build 1,296 trees, which can be expensive for large data sets (regardless of which fancy scalable implementation you use). While XGBoost is incredibly efficient, it is still wasteful to fit more trees than potentially necessary. To that end, I can turn on early stopping (Section 8.3.1.1) to halt performance once it detects

the potential for overfitting. In XGBoost, early stopping will halt training if model performance has not improved for a specified number of iterations (`early_stopping_rounds`).

In the code chunk below, I fit the same model (random seed and all), but tell XGBoost to stop the training process if the performance on the test set (as specified in the watch list) has not improved for 150 consecutive iterations (5% of the total number of requested iterations)<sup>r</sup>:

```
set.seed(1100) # for reproducibility
(bank.xgb.2 <-
  xgb.train(params, data = dm.trn, nrounds = 3000, verbose = 0,
            watchlist = watch.list, early_stopping_rounds = 150))

## ##### xgb.Booster
## raw: 1.7 Mb
## xgb.attributes:
##   best_iteration, best_msg, best_ntreelimit, best_score, niter
##   niter: 1445
##   best_ntreelimit : 1296
##   best_iteration : 1296
##   best_score : 0.274
##   best_msg : [1296] train-rmse:0.274552 eval-rmse:0.273857
```

In this case, using early stopping resulted in the same optimal number of trees (e.g., 1,296), but only required 1,446 boosting iterations (or trees) in total, a decent savings in terms of both computation time and storage space (1.7 Mb for early stopping compared to 3.6 Mb for the full model)! The overall training results are displayed in [Figure 8.14](#) below.

```
palette("Okabe-Ito")
plot(bank.xgb.1$evaluation_log[, c(1, 2)], type = "l",
      xlab = "Number of trees",
      ylab = "RMSE (square root of Brier score)")
lines(bank.xgb.1$evaluation_log[, c(1, 3)], type = "l", col = 2)
abline(v = best.iter, col = 2, lty = 2)
abline(v = bank.xgb.2$niter, col = 3, lty = 2)
legend("topright", legend = c("Train", "Test"), inset = 0.01, bty = "n",
       lty = 1, col = 1:2)
palette("default")
```

Unlike R's `gbm` package or Python's `sklearn.inspection` module, XGBoost does not support the recursion method for fast PD and ICE plots. However, XGBoost does support Tree SHAP ([Section 6.3.2.1](#)), which we can use to construct Shapley-based variable importance and dependence plots. In R, these can be obtained at prediction time by specifying `predcontrib = TRUE` in the call to `predict()`.

---

<sup>r</sup>I've seen several online blog posts suggest a value of `early_stopping_rounds` equal to 10% of the total number of requested iterations, but no evidence or citations as to why.

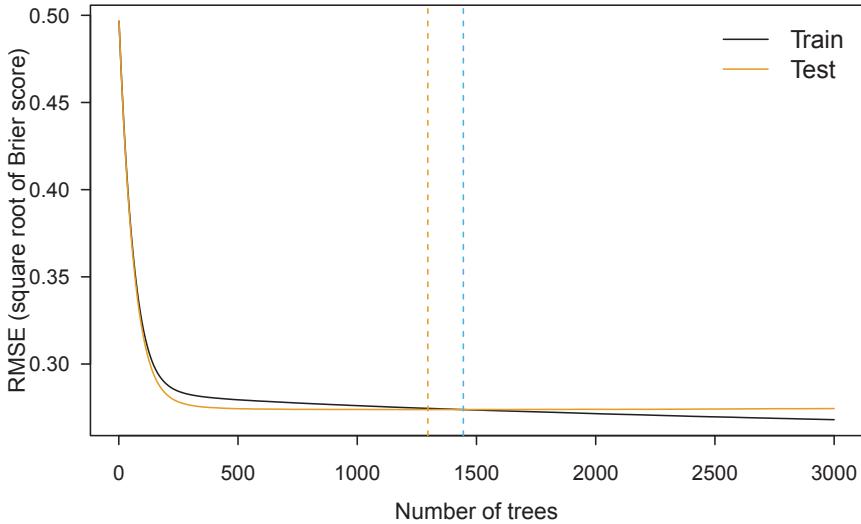


FIGURE 8.14: RMSE (essentially, the square root of the Brier score) from an XGBoost model fit to the bank marketing data. According to the independent test set (yellow curve), the optimal number of trees is 1,296 (vertical dashed yellow line). Early stopping, which reached the same conclusion, would've stopped training at 1,446 trees (vertical dashed blue line), which roughly halves the training time in this case.

Below, I compute Tree SHAP values for the entire training set and use that to form Shapley-based variable importance scores; here, I'll follow Lundberg et al. [2020] and the **shap** module's approach by computing the mean absolute Shapley value for each column. (Note that it is not necessary to use the entire learning sample for doing this, and a large enough subsample should often suffice, especially when dealing with hundreds of thousands or millions of records.) A dot plot of the top 10 Shapley-based importance scores is displayed in Figure 8.15. Note that I need to specify the optimal number of trees (`ntreelimit = best.iter`) when calling `predict()`:

```
shap.trn <- predict(bank.xgb, newdata = dm.trn, ntreelimit = best.iter,
                     predcontrib = TRUE, approxcontrib = FALSE)
shap.trn <- shap.trn[, -which(colnames(shap.trn) == "BIAS")]

# Shapley-based variable importance
shap.vi <- colMeans(abs(shap.trn))
shap.vi <- shap.vi[order(shap.vi, decreasing = TRUE)]
dotchart(shap.vi[1:10], xlab = "mean(|SHAP value|)", pch = 19)
```

For comparison, I computed the usual aggregated tree-based importance scores from the boosted model; this is what's given by the `Gain` column in the

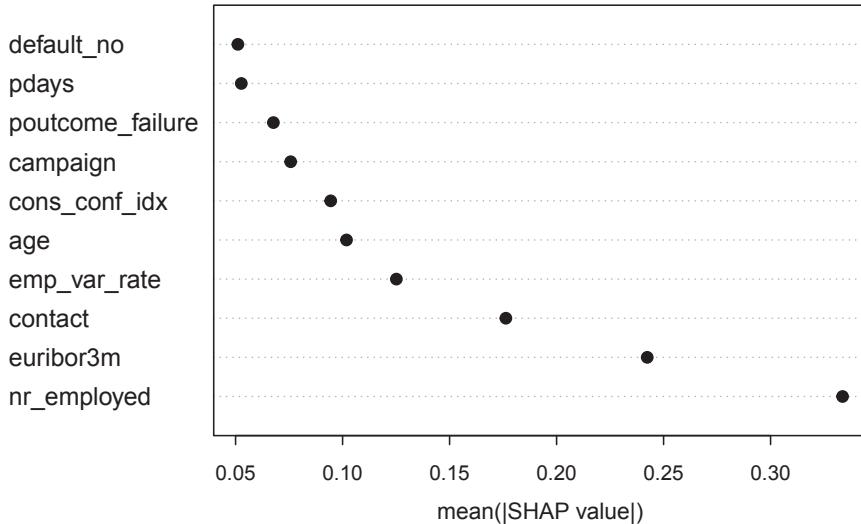


FIGURE 8.15: Shapley-based variable importance scores from an XGBoost model fit to the bank marketing data.

output below (see `?xgboost::xgb.importance` for details on the other two measures `Cover` and `Frequency`). Below, I display the top six features (note that tree indexing in XGBoost, whether using R or not, is zero-based):

```
head(xgb.importance(model = bank.xgb, trees = 0:(best.iter - 1)))

#>          Feature  Gain Cover Frequency
#> 1:      nr_employed 0.3689 0.1012    0.0395
#> 2:      euribor3m 0.1531 0.1693    0.1801
#> 3:      cons_conf_idx 0.0704 0.0960    0.0651
#> 4:          age 0.0665 0.1219    0.1471
#> 5:      pdays 0.0400 0.0420    0.0363
#> 6: poutcome_success 0.0334 0.0178    0.0137
```

To get a sense of a predictor's effect on the model output, in terms of Shapley values, we can construct a Shapley dependence plot; this is nothing more than a plot of a feature's Shapley values against the raw feature values in a particular sample. Below, I construct a Shapley dependence plot for `age` (Figure 8.16); a nonparametric smooth is also displayed (yellow curve). Here, you can see that individuals in the age range 30–50 (roughly) generally correspond to negative Shapley values, meaning they tend to drive the predicted probability of subscribing towards the average baseline. Whereas younger and older individuals tend to have `age` contribute a positive effect. The non-constant variance in the scatter plot suggests potential interaction effects, which could

be explored further by using another feature (or features) to help color the plot.

```
shap.age <- data.frame("age" = bank.trn[, "age"],
                       "shap" = shap.trn[, "age"])

# Shapley dependence plot for age
cols <- palette.colors(3, palette = "Okabe-Ito")
ggplot(shap.age, aes(age, shap)) +
  geom_point(alpha = 0.1) +
  geom_smooth(se = FALSE, color = cols[2]) +
  geom_hline(yintercept = 0, linetype = "dashed", color = cols[3]) +
  xlab("Age (years)") + ylab("Shapley value")
```

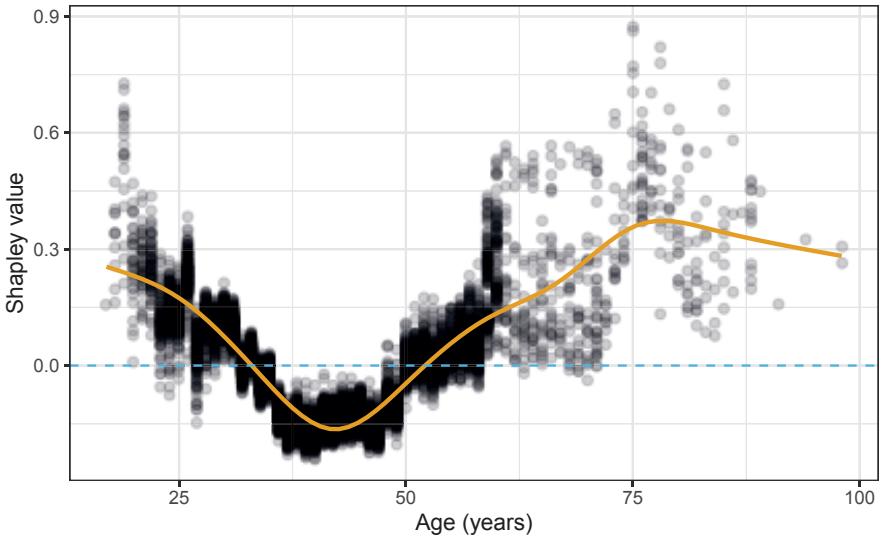


FIGURE 8.16: Shapley dependence plot for `age` from the XGBoost model fit to the bank marketing data; a nonparametric smooth is also shown (yellow curve). Any point below the horizontal dashed blue line corresponds to a negative contribution to the predicted outcome).

---

## 8.10 Final thoughts

GBMs are a powerful class of machine learning algorithms that can achieve state-of-the-art performance, provided you train them properly. Due to the existence of efficient libraries (like XGBoost and Microsoft's LightGBM) and

the shallower nature of the individual trees, GBMs can also scale incredibly well; see, for example, Pafka [2021]. For these reasons, GBMs are quite popular in applied practice and are often used in the winning entries for many supervised learning competitions with tabular data sets. Just keep in mind that, unlike RFs, GBMs are quite sensitive to several tuning parameters (e.g., the learning rate and number of boosting iterations), and these models should be carefully tuned (ideally, with some form of early stopping, especially if you're working with a large learning sample, using a fairly small learning rate with a large number of boosting iterations, and/or tuning lots of parameters).



Taylor & Francis  
Taylor & Francis Group  
<http://taylorandfrancis.com>

---

# Bibliography

---

- Hongshik Ahn and Wei-Yin Loh. Tree-structured proportional hazards regression modeling. *Biometrics*, 50(2):471–485, 1994.
- Alfaro, Esteban; Gamez, Matias, Garcia, and Noelia; with contributions from Li Guo. *adabag: Applies Multiclass AdaBoost.M1, SAMME and Bagging*, 2018. URL <https://CRAN.R-project.org/package=adabag>. R package version 4.2.
- André Altmann, Laura Tološi, Oliver Sander, and Thomas Lengauer. Permutation importance: a corrected feature importance measure. *Bioinformatics*, 26(10):1340–1347, 2010. ISSN 1367-4803. doi: [10.1093/bioinformatics/btq134](https://doi.org/10.1093/bioinformatics/btq134). URL <https://doi.org/10.1093/bioinformatics/btq134>.
- Elaine Angelino, Nicholas Larus-Stone, Daniel Alabi, Margo Seltzer, and Cynthia Rudin. Learning certifiably optimal rule lists for categorical data. *Journal of Machine Learning Research*, 18(234):1–78, 2018. URL <http://jmlr.org/papers/v18/17-716.html>.
- Dan Apley. *ALEPlot: Accumulated Local Effects (ALE) Plots and Partial Dependence (PD) Plots*, 2018. URL <https://CRAN.R-project.org/package=ALEplot>. R package version 1.1.
- Dan Apley and Jingyu Zhu. Visualizing the effects of predictor variables in black box supervised learning models. *Journal of the Royal Statistical Society Series B*, 82(4):1059–1086, September 2020. doi: [10.1111/rssb.12377](https://doi.org/10.1111/rssb.12377). URL <https://ideas.repec.org/a/bla/jorssb/v82y2020i4p1059-1086.html>.
- Susan Athey, Julie Tibshirani, and Stefan Wager. Generalized random forests. *The Annals of Statistics*, 47(2):1148–1178, 2019. doi: [10.1214/18-AOS1709](https://doi.org/10.1214/18-AOS1709). URL <https://doi.org/10.1214/18-AOS1709>.
- Baptiste Auguie. *gridExtra: Miscellaneous Functions for "Grid" Graphics*, 2017. URL <https://CRAN.R-project.org/package=gridExtra>. R package version 2.3.
- Peter C. Austin and Ewout W. Steyerberg. Graphical assessment of internal and external calibration of logistic regression models by using loess smoothers. *Statistics in Medicine*, 33(3):517–535, 2014. doi: [10.1002/sim.5941](https://doi.org/10.1002/sim.5941). URL <https://doi.org/10.1002/sim.5941>.

- Michel Ballings and Dirk Van den Poel. *rotationForest: Fit and Deploy Rotation Forest Models*, 2017. URL <https://CRAN.R-project.org/package=rotationForest>. R package version 0.1.3.
- Richard A. Berk. *Statistical Learning from a Regression Perspective*. Springer Series in Statistics. Springer New York, 2008. ISBN 9780387775012.
- Gérard Biau, Luc Devroye, and Gábor Lugosi. Consistency of random forests and other averaging classifiers. *Journal of Machine Learning Research*, 9(66):2015–2033, 2008. URL <http://jmlr.org/papers/v9/biau08a.html>.
- Przemysław Biecek and Hubert Baniecki. *ingredients: Effects and Importances of Model Ingredients*, 2021. URL <https://CRAN.R-project.org/package=ingredients>. R package version 2.2.0.
- Przemysław Biecek and Tomasz Burzykowski. *Explanatory Model Analysis*. Chapman and Hall/CRC, New York, 2021. ISBN 9780367135591. URL <https://pbiecek.github.io/ema/>.
- Przemysław Biecek, Alicja Gosiewska, Hubert Baniecki, and Adam Izdebski. *iBreakDown: Model Agnostic Instance Level Variable Attributions*, 2021. URL <https://CRAN.R-project.org/package=iBreakDown>. R package version 2.0.1.
- Rico Blaser and Piotr Fryzlewicz. Random rotation ensembles. *Journal of Machine Learning Research*, 17(4):1–26, 2016. URL <http://jmlr.org/papers/v17/blaser16a.html>.
- Bradley Boehmke and Brandon Greenwell. *Hands-On Machine Learning with R*. Chapman & Hall/CRC the R series. CRC Press, 2020. ISBN 9781138495685.
- Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996a. URL <https://doi.org/10.1007/BF00058655>.
- Leo Breiman. Heuristics of instability and stabilization in model selection. *The Annals of Statistics*, 24(6):2350–2383, 1996b.
- Leo Breiman. Technical note: Some properties of splitting criteria. *Machine Learning*, 24:41–47, 1996c.
- Leo Breiman. Pasting small votes for classification in large databases and online. *Machine Learning*, 36(1):85–103, 1999. doi: [10.1023/A:1007563306331](https://doi.org/10.1023/A:1007563306331). URL <https://doi.org/10.1023/A:1007563306331>.
- Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. URL <https://doi.org/10.1023/A:1010933404324>.
- Leo Breiman. Manual on setting up, using, and understanding random forests v3.1. Technical report, 2002. URL [https://www.stat.berkeley.edu/~breiman/Using\\_random\\_forests\\_V3.1.pdf](https://www.stat.berkeley.edu/~breiman/Using_random_forests_V3.1.pdf).

- Leo Breiman, Jerome H. Friedman, Charles J. Stone, and Richard A. Olshen. *Classification and Regression Trees*. Taylor & Francis, 1984. ISBN 9780412048418.
- Leo Breiman, Adele Cutler, Andy Liaw, and Matthew Wiener. *randomForest: Breiman and Cutler's Random Forests for Classification and Regression*, 2018. URL <https://www.stat.berkeley.edu/~breiman/RandomForests/>. R package version 4.6-14.
- Peter Bühlmann and Torsten Hothorn. Boosting Algorithms: Regularization, Prediction and Model Fitting. *Statistical Science*, 22(4):477–505, 2007. doi: [10.1214/07-STS242](https://doi.org/10.1214/07-STS242). URL <https://doi.org/10.1214/07-STS242>.
- Tom Bylander. Estimating generalization error on two-class datasets using out-of-bag estimates. *Machine Learning*, 48(1):287–297, 2002. doi: [10.1023/A:1013964023376](https://doi.org/10.1023/A:1013964023376). URL <https://doi.org/10.1023/A:1013964023376>.
- Angelo Canty and Brian Ripley. *boot: Bootstrap Functions (Originally by Angelo Canty for S)*, 2021. URL <https://CRAN.R-project.org/package=boot>. R package version 1.3-28.
- M. Carlisle. Racist data destruction?, May 2019. URL <https://medium.com/@docintangible/racist-data-destruction-113e3eff54a8>.
- Hugh Chen, Joseph D. Janizek, Scott Lundberg, and Su-In Lee. True to the model or true to the data?, 2020. URL <https://arxiv.org/abs/2006.16234>.
- Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: [10.1145/2939672.2939785](https://doi.org/10.1145/2939672.2939785). URL <https://doi.org/10.1145/2939672.2939785>.
- Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, Rory Mitchell, Ignacio Cano, Tianyi Zhou, Mu Li, Junyuan Xie, Min Lin, Yifeng Geng, and Yutian Li. *xgboost: Extreme Gradient Boosting*, 2021. URL <https://github.com/dmlc/xgboost>. R package version 1.5.0.2.
- Hugh A. Chipman, Edward I. George, and Robert E. McCulloch. BART: Bayesian additive regression trees. *The Annals of Applied Statistics*, 4(1):266–298, 2010. doi: [10.1214/09-AOAS285](https://doi.org/10.1214/09-AOAS285). URL <https://doi.org/10.1214/09-AOAS285>.
- Philip A. Chou. Optimal partitioning for classification and regression trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(04):340–354, 1991. ISSN 1939-3539. doi: [10.1109/34.88569](https://doi.org/10.1109/34.88569).

- William S. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368):829–836, 1979. doi: [10.1080/01621459.1979.10481038](https://doi.org/10.1080/01621459.1979.10481038). URL <https://doi.org/10.1080/01621459.1979.10481038>.
- David Cortes. *isotree: Isolation-Based Outlier Detection*, 2022. URL <https://github.com/david-cortes/isotree>. R package version 0.5.5.
- Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47(4):547–553, 2009. ISSN 0167-9236. doi: <https://doi.org/10.1016/j.dss.2009.05.016>. URL <http://www.sciencedirect.com/science/article/pii/S0167923609001377>.
- Mark Culp, Kjell Johnson, and George Michailidis. *ada: The R Package Ada for Stochastic Boosting*, 2016. URL <https://CRAN.R-project.org/package=ada>. R package version 2.0-5.
- Adele Cutler. Remembering Leo Breiman. *The Annals of Applied Statistics*, 4(4):1621–1633, 2010. doi: [10.1214/10-AOAS427](https://doi.org/10.1214/10-AOAS427). URL <https://doi.org/10.1214/10-AOAS427>.
- Natalia da Silva, Dianne Cook, and Eun-Kyung Lee. A projection pursuit forest algorithm for supervised classification. 0(0):1–13, 2021a. doi: [10.1080/10618600.2020.1870480](https://doi.org/10.1080/10618600.2020.1870480). URL <https://doi.org/10.1080/10618600.2020.1870480>.
- Natalia da Silva, Eun-Kyung Lee, and Di Cook. *PPforest: Projection Pursuit Classification Forest*, 2021b. URL <https://github.com/natydasilva/PPforest>. R package version 0.1.2.
- Data Mining Group. Predictive model markup language, 2014. URL <http://www.dmg.org/>. Version 4.2.
- Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML ’06, pages 233–240, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933832. doi: [10.1145/1143844.1143874](https://doi.org/10.1145/1143844.1143874). URL <https://doi.org/10.1145/1143844.1143874>.
- Anthony C. Davison and David V. Hinkley. *Bootstrap Methods and Their Application*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997. ISBN 9780521574716.
- Dean De Cock. Ames, Iowa: Alternative to the Boston housing data as an end of semester regression project. *Journal of Statistics Education*, 19(3): null, 2011. doi: [10.1080/10691898.2011.11889627](https://doi.org/10.1080/10691898.2011.11889627). URL <https://doi.org/10.1080/10691898.2011.11889627>.

- Luc Devroye, László Györfi, and Gábor Lugosi. *A Probabilistic Theory of Pattern Recognition*. Stochastic Modelling and Applied Probability. Springer New York, 1997. ISBN 9780387946184.
- Stephan Dlugosz. *rpart.LAD: Least Absolute Deviation Regression Trees*, 2020. URL <https://CRAN.R-project.org/package=rpart.LAD>. R package version 0.1.2.
- Rémi Domingues, Maurizio Filippone, Pietro Michiardi, and Jihane Zouaoui. A comparative evaluation of outlier detection algorithms: Experiments and analyses. *Pattern Recognition*, 74:406–421, 2018. ISSN 0031-3203. doi: <https://doi.org/10.1016/j.patcog.2017.09.037>. URL <https://doi.org/10.1016/j.patcog.2017.09.037>.
- Lisa Doove, Stef Van Buuren, and Elise Dusseldorp. Recursive partitioning for missing data imputation in the presence of interaction effects. *Computational Statistics & Data Analysis*, 72:92–104, 2014. ISSN 0167-9473. doi: [10.1016/j.csda.2013.10.025](https://doi.org/10.1016/j.csda.2013.10.025). URL <https://doi.org/10.1016/j.csda.2013.10.025>.
- Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. Catboost: gradient boosting with categorical features support, 2018. URL <https://arxiv.org/abs/1810.11363>.
- Matt Dowle and Arun Srinivasan. *data.table: Extension of ‘data.frame’*, 2021. URL <https://CRAN.R-project.org/package=data.table>. R package version 1.14.2.
- Tony Duan, Anand Avati, Daisy Yi Ding, Khanh K. Thai, Sanjay Basu, Andrew Y. Ng, and Alejandro Schuler. Ngboost: Natural gradient boosting for probabilistic prediction, 2020. URL <https://arxiv.org/abs/1910.03225>.
- Bradley Efron and Trevor Hastie. *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*. Institute of Mathematical Statistics Monographs. Cambridge University Press, 2016. doi: [10.1017/CBO9781316576533](https://doi.org/10.1017/CBO9781316576533).
- Ad Feelders. Handling missing data in trees: Surrogate splits or statistical imputation? In *Principles of Data Mining and Knowledge Discovery*, pages 329–334, 03 2000. ISBN 978-3-540-66490-1. doi: [10.1007/978-3-540-48247-5\\_38](https://doi.org/10.1007/978-3-540-48247-5_38). URL [https://doi.org/10.1007/978-3-540-48247-5\\_38](https://doi.org/10.1007/978-3-540-48247-5_38).
- Aaron Fisher, Cynthia Rudin, and Francesca Dominici. All models are wrong, but many are useful: Learning a variable’s importance by studying an entire class of prediction models simultaneously. 2018. doi: [10.48550/ARXIV.1801.01489](https://doi.org/10.48550/ARXIV.1801.01489). URL <https://arxiv.org/abs/1801.01489>.
- Thomas R. Fleming and David P. Harrington. *Counting Processes and Survival Analysis*. Wiley Series in Probability and Statistics. Wiley, 1991. ISBN 9780471522188.

- Bernhard Flury and Hans Riedwyl. *Multivariate Statistics: A Practical Approach*. Statistics texts. Springer Netherlands, 1988. ISBN 9780412300301.
- Christopher Flynn. *Python bindings for C++ ranger random forests*, 2021. URL <https://github.com/crflynn/skranger>. Python package version 0.3.2.
- Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning*, ICML'96, pages 148–156, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. ISBN 1558604197.
- Peter W. Frey and David J. Slate. Letter recognition using Holland-style adaptive classifiers. *Machine Learning*, 6(2):161–182, 1991. URL <https://doi.org/10.1007/BF00114162>.
- Jerome Friedman, Trevor Hastie, Rob Tibshirani, Balasubramanian Narasimhan, Kenneth Tay, Noah Simon, and James Yang. *glmnet: Lasso and Elastic-Net Regularized Generalized Linear Models*, 2021. URL <https://CRAN.R-project.org/package=glmnet>. R package version 4.1-3.
- Jerome H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–67, 03 1991. doi: [10.1214/aos/1176347963](https://doi.org/10.1214/aos/1176347963). URL <https://doi.org/10.1214/aos/1176347963>.
- Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001. URL <https://doi.org/10.1214/aos/1013203451>.
- Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, 2002. ISSN 0167-9473. doi: [https://doi.org/10.1016/S0167-9473\(01\)00065-2](https://doi.org/10.1016/S0167-9473(01)00065-2). URL [https://doi.org/10.1016/S0167-9473\(01\)00065-2](https://doi.org/10.1016/S0167-9473(01)00065-2).
- Jerome H. Friedman and Peter Hall. On bagging and nonlinear estimation. *Journal of Statistical Planning and Inference*, 137(3):669–683, 2007. ISSN 0378-3758. doi: <https://doi.org/10.1016/j.jspi.2006.06.002>. URL <http://www.sciencedirect.com/science/article/pii/S0378375806001339>. Special Issue on Nonparametric Statistics and Related Topics: In honor of M.L. Puri.
- Jerome H. Friedman and Bogdan E. Popescu. Importance sampled learning ensembles. Technical report, Stanford University, Department of Statistics, 2003. URL <https://statweb.stanford.edu/~jhf/ftp/isle.pdf>.
- Jerome H. Friedman and Bogdan E. Popescu. Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2(3):916–954, 2008. ISSN 19326157. URL <https://doi.org/10.2307/30245114>.

Jerome H. Friedman and John W. Tukey. A projection pursuit algorithm for exploratory data analysis. *IEEE Transactions on Computers*, C-23(9): 881–890, 1974. doi: [10.1109/T-C.1974.224051](https://doi.org/10.1109/T-C.1974.224051). URL <https://doi.org/10.1109/T-C.1974.224051>.

Jerome H. Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors). *The Annals of Statistics*, 28(2):337–407, 2000. doi: [10.1214/aos/1016218223](https://doi.org/10.1214/aos/1016218223). URL <https://doi.org/10.1214/aos/1016218223>.

Giuliano Galimberti, Gabriele Soffritti, and Matteo Di Maso. *rpartScore: Classification trees for ordinal responses*, 2012. URL <https://CRAN.R-project.org/package=rpartScore>. R package version 1.0-1.

Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 2nd edition, 2019. ISBN 9781492032649.

Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006. doi: [10.1007/s10994-006-6226-1](https://doi.org/10.1007/s10994-006-6226-1). URL <https://doi.org/10.1007/s10994-006-6226-1>.

Anil K. Ghosh, Probal Chaudhuri, and Debasis Sengupta. Classification using kernel density estimates. *Technometrics*, 48(1):120–132, 2006. URL <https://doi.org/10.1198/004017005000000391>.

Navdeep Gill, Patrick Hall, Kim Montgomery, and Nicholas Schmidt. A responsible machine learning workflow with focus on interpretable models, post-hoc explanation, and discrimination testing. *Information*, 11(3), 2020. ISSN 2078-2489. doi: [10.3390/info11030137](https://doi.org/10.3390/info11030137). URL <https://doi.org/10.3390/info11030137>.

Alex Goldstein, Adam Kapelner, Justin Bleich, and Emil Pitkin. Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. *Journal of Computational and Graphical Statistics*, 24(1):44–65, 2015. URL <https://doi.org/10.1080/10618600.2014.907095>.

Alex Goldstein, Adam Kapelner, and Justin Bleich. *ICEbox: Individual Conditional Expectation Plot Toolbox*, 2017. URL <https://CRAN.R-project.org/package=ICEbox>. R package version 1.1.2.

Brandon M. Greenwell. pdp: An R package for constructing partial dependence plots. *The R Journal*, 9(1):421–436, 2017. URL <https://journal.r-project.org/archive/2017/RJ-2017-016/index.html>.

Brandon M. Greenwell. fastshap: Fast Approximate Shapley Values, 2021a. URL <https://github.com/bgreenwell/fastshap>. R package version 0.0.7.

- Brandon M. Greenwell. *pdp: Partial Dependence Plots*, 2021b. URL <https://github.com/bgreenwell/pdp>. R package version 0.7.0.9000.
- Brandon M. Greenwell. *treemisc: Data Sets and Functions to Accompany "Tree-Based Methods for Statistical Learning in R"*, 2021c. R package version 0.0.1.
- Brandon M. Greenwell and Bradley C. Boehmke. Variable importance plots—an introduction to the *vip* package. *The R Journal*, 12(1):343–366, 2020. URL <https://doi.org/10.32614/RJ-2020-013>.
- Brandon M. Greenwell, Bradley C. Boehmke, and Andrew J. McCarthy. A simple and effective model-based variable importance measure, 2018. URL <https://arxiv.org/abs/1805.04755>.
- Brandon M. Greenwell, Brad Boehmke, and Bernie Gray. *vip: Variable Importance Plots*, 2021a. URL <https://github.com/koalaverse/vip/>. R package version 0.3.3.
- Brandon M. Greenwell, Bradley Boehmke, Jay Cunningham, and GBM Developers. *gbm: Generalized Boosted Regression Models*, 2021b. URL <https://github.com/gbm-developers/gbm>. R package version 2.1.5.
- Alexander Hapfelmeier, Torsten Hothorn, Kurt Ulm, and Carolin Strobl. A new variable importance measure for random forests with missing data. *Statistics and Computing*, 24(1):21–34, 2014.
- Sahand Hariri, Matias Carrasco Kind, and Robert J. Brunner. Extended isolation forest. *IEEE Transactions on Knowledge and Data Engineering*, 33(4):1479–1489, 2021. doi: [10.1109/TKDE.2019.2947676](https://doi.org/10.1109/TKDE.2019.2947676). URL <https://doi.org/10.1109/TKDE.2019.2947676>.
- Frank Harrell. *Regression Modeling Strategies*. Springer Series in Statistics. Springer International Publishing, 2nd edition, 2015. ISBN 978-3-319-19424-0.
- Frank E Harrell, Jr. *Hmisc: Harrell Miscellaneous*, 2021. URL <https://hbiostat.org/R/Hmisc/>. R package version 4.6-0.
- Frank E. Harrell, Jr. *rms: Regression Modeling Strategies*, 2021. URL <https://CRAN.R-project.org/package=rms>. R package version 6.2-0.
- Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585

- (7825):357–362, September 2020. doi: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2). URL <https://doi.org/10.1038/s41586-020-2649-2>.
- David Harrison and Daniel L. Rubinfeld. Hedonic housing prices and the demand for clean air. *Journal of Environmental Economics and Management*, 5(1):81–102, 1978. URL [https://doi.org/10.1016/0095-0696\(78\)90006-2](https://doi.org/10.1016/0095-0696(78)90006-2).
- Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer-Verlag, 2009. URL <https://web.stanford.edu/~hastie/ElemStatLearn/>.
- Paul Hendricks. *titanic: Titanic Passenger Survival Data Set*, 2015. URL <https://github.com/paulhendricks/titanic>. R package version 0.1.0.
- Lionel Henry and Hadley Wickham. *purrr: Functional Programming Tools*, 2020. URL <https://CRAN.R-project.org/package=purrr>. R package version 0.3.4.
- Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282, 1995. doi: [10.1109/ICDAR.1995.598994](https://doi.org/10.1109/ICDAR.1995.598994). URL <https://doi.org/10.1007/s10115-013-0679-x>.
- Giles Hooker. Generalized functional anova diagnostics for high-dimensional functions of dependent variables. *Journal of Computational and Graphical Statistics*, 16(3):709–732, 2007. URL <https://doi.org/10.1198/106186007X237892>.
- Giles Hooker, Lucas Mentch, and Siyu Zhou. There is no free variable importance, 2019. URL <https://arxiv.org/abs/1905.03151>.
- Kurt Hornik. *RWeka: R/Weka Interface*, 2021. URL <https://CRAN.R-project.org/package=RWeka>. R package version 0.4-44.
- Allison Horst, Alison Hill, and Kristen Gorman. *palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*, 2020. URL <https://CRAN.R-project.org/package=palmerpenguins>. R package version 0.1.0.
- Torsten Hothorn and Achim Zeileis. *partykit: A Toolkit for Recursive Partytioning*, 2021. URL <http://partykit.r-forge.r-project.org/partykit/>. R package version 1.2-15.
- Torsten Hothorn, Berthold Lausen, Axel Benner, and Martin Radespiel-Tröger. Bagging survival trees. *Statistics in Medicine*, 23(1):77–91, 2004. doi: [10.1002/sim.1593](https://doi.org/10.1002/sim.1593). URL <https://doi.org/10.1002/sim.1593>.
- Torsten Hothorn, Peter Buehlmann, Sandrine Dudoit, Annette Molinaro, and Mark Van Der Laan. Survival ensembles. *Biostatistics*, 7(3):355–373, 2006a.

- Torsten Hothorn, Kurt Hornik, Mark A. van de Wiel, and Achim Zeileis. A lego system for conditional inference. *The American Statistician*, 60(3): 257–263, 2006b. URL <https://doi.org/10.1198/000313006X118430>.
- Torsten Hothorn, Kurt Hornik, and Achim Zeileis. Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical Statistics*, 15(3):651–674, 2006c.
- Torsten Hothorn, Peter Buehlmann, Thomas Kneib, Matthias Schmid, and Benjamin Hofner. *mboost: Model-Based Boosting*, 2021a. URL <https://github.com/boost-R/mboost>. R package version 2.9-5.
- Torsten Hothorn, Kurt Hornik, Carolin Strobl, and Achim Zeileis. *party: A Laboratory for Recursive Partitioning*, 2021b. URL <http://party.R-forge.R-project.org>. R package version 1.3-9.
- Torsten Hothorn, Henric Winell, Kurt Hornik, Mark A. van de Wiel, and Achim Zeileis. *coin: Conditional Inference Procedures in a Permutation Test Framework*, 2021c. URL <http://coin.r-forge.r-project.org>. R package version 1.4-2.
- James P. Howard, II. *waterfall: Waterfall Charts*, 2016. URL <https://CRAN.R-project.org/package=waterfall>. R package version 1.0.2.
- Hemant Ishwaran and Udaya B. Kogalur. *randomForestSRC: Fast Unified Random Forests for Survival, Regression, and Classification (RF-SRC)*, 2022. URL <https://luminwin.github.io/randomForestSRC/https://github.com/kogalur/randomForestSRC/https://web.ccs.miami.edu/~hishwaran/>. R package version 3.0.0.
- Hemant Ishwaran, Udaya B. Kogalur, Eugene H. Blackstone, and Michael S. Lauer. Random survival forests. *Ann. Appl. Stat.*, 2(3):841–860, 09 2008. doi: [10.1214/08-AOAS169](https://doi.org/10.1214/08-AOAS169). URL <https://doi.org/10.1214/08-AOAS169>.
- Gareth James, Daniella Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Texts in Statistics. Springer New York, 2nd edition, 2021. ISBN 9781071614174. URL <https://www.statlearning.com/>.
- Silke Janitza and Roman Hornung. On the overestimation of random forest’s out-of-bag error. *PLOS ONE*, 13(8):1–31, 08 2018. doi: [10.1371/journal.pone.0201904](https://doi.org/10.1371/journal.pone.0201904). URL <https://doi.org/10.1371/journal.pone.0201904>.
- Silke Janitza, Ender Celik, and Anne-Laure Boulesteix. A computationally fast variable importance test for random forests for high-dimensional data. *Advances in Data Analysis and Classification*, 12(4):885—915, 2018. ISSN 1862-5347. doi: [10.1007/s11634-016-0276-4](https://doi.org/10.1007/s11634-016-0276-4). URL <https://doi.org/10.1007/s11634-016-0276-4>.

- Dominik Janzing, Lenon Minorics, and Patrick Blöbaum. Feature relevance quantification in explainable ai: A causal problem, 2019. URL <https://arxiv.org/abs/1910.13413>.
- Richard A. Johnson and Dean W. Wichern. *Applied Multivariate Statistical Analysis*. Applied Multivariate Statistical Analysis. Pearson Prentice Hall, 2007. ISBN 9780131877153.
- Zachary Jones. *mmpf: Monte-Carlo Methods for Prediction Functions*, 2018. URL <https://CRAN.R-project.org/package=mmpf>. R package version 0.0.5.
- Alexandros Karatzoglou, Alex Smola, and Kurt Hornik. *kernlab: Kernel-Based Machine Learning Lab*, 2019. URL <https://CRAN.R-project.org/package=kernlab>. R package version 0.9-29.
- Gordon V. Kass. An exploratory technique for investigating large quantities of categorical data. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 29(2):119–127, 1980.
- Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL <https://proceedings.neurips.cc/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf>.
- Hyunjoong Kim and Wei-Yin Loh. Classification trees with unbiased multiway splits. *Journal of the American Statistical Association*, 96(454):589–604, 2001. ISSN 01621459. URL <https://doi.org/10.2307/2670299>.
- John P. Klein and Melvin L. Moeschberger. *Survival Analysis: Techniques for Censored and Truncated Data*. Statistics for Biology and Health. Springer New York, 2003. ISBN 9780387953991.
- Brian Kriegler and Richard Berk. Small area estimation of the homeless in Los Angeles: An application of cost-sensitive stochastic gradient boosting. *The Annals of Applied Statistics*, 4(3):1234–1255, 2010. doi: [10.1214/10-AOAS328](https://doi.org/10.1214/10-AOAS328). URL <https://doi.org/10.1214/10-AOAS328>.
- Robert Küffner, Neta Zach, Raquel Norel, Johann Hawe, David Schoenfeld, Liuxia Wang, Guang Li, Lilly Fang, Lester Mackey, Orla Hardiman, Merit Cudkowicz, Alexander Sherman, Gokhan Ertaylan, Moritz Grosse-Wentrup, Torsten Hothorn, Jules van Ligtenberg, Jakob H. Macke, Timm Meyer, Bernhard Schölkopf, Linh Tran, Rubio Vaughan, Gustavo Stolovitzky, and Melanie L. Leitner. Crowdsourced analysis of clinical trial data to predict

- amyotrophic lateral sclerosis progression. *Nature Biotechnology*, 33(1):51–57, 2015. doi: 10.1038/nbt.3051. URL <https://doi.org/10.1038/nbt.3051>.
- Max Kuhn. *AmesHousing: The Ames Iowa Housing Data*, 2020. URL <https://github.com/topepo/AmesHousing>. R package version 0.0.4.
- Max Kuhn. *modeldata: Data Sets Useful for Modeling Packages*, 2021. URL <https://CRAN.R-project.org/package=modeldata>. R package version 0.1.1.
- Max Kuhn and Kjell Johnson. *Applied Predictive Modeling*. Springer-Verlag, 2013. URL <https://books.google.com/books?id=xYRDAAAQBAJ>. ISBN 978-1-4614-6848-6.
- Max Kuhn and Ross Quinlan. *C50: C5.0 Decision Trees and Rule-Based Models*, 2021. URL <https://topepo.github.io/C5.0/>. R package version 0.1.5.
- Meelis Kull, Telmo M. Silva Filho, and Peter Flach. Beyond sigmoids: How to obtain well-calibrated probabilities from binary classifiers with beta calibration. *Electronic Journal of Statistics*, 11(2):5052–5080, 2017. doi: 10.1214/17-EJS1338SI. URL <https://doi.org/10.1214/17-EJS1338SI>.
- Ludmila I. Kuncheva and Juan J. Rodríguez. An experimental study on rotation forest ensembles. In Michal Haindl, Josef Kittler, and Fabio Roli, editors, *Multiple Classifier Systems*, pages 459–468. Springer Berlin, Heidelberg, 2007. ISBN 978-3-540-72523-7.
- Michael LeBlanc and John Crowley. Relative risk trees for censored survival data. *Biometrics*, 48(2):411–425, 1992.
- Michael Leblanc and John Crowley. Survival trees by goodness of split. *Journal of the American Statistical Association*, 88(422):457–467, 1993. doi: 10.1080/01621459.1993.10476296. URL <https://doi.org/10.1080/01621459.1993.10476296>.
- Erin LeDell, Navdeep Gill, Spencer Aiello, Anqi Fu, Arno Candel, Cliff Click, Tom Kraljevic, Tomas Nykodym, Patrick Aboyoun, Michal Kurka, and Michal Malohlava. *h2o: R Interface for the H2O Scalable Machine Learning Platform*, 2021. URL <https://github.com/h2oai/h2o-3>. R package version 3.34.0.3.
- Eun-Kyung Lee. *PPtreeViz: Projection Pursuit Classification Tree Visualization*, 2019. URL <https://CRAN.R-project.org/package=PPtreeViz>. R package version 2.0.4.
- Yoon Dong Lee, Dianne Cook, Ji won Park, and Eun-Kyung Lee. Pptree: Projection pursuit classification tree. *Electronic Journal of Statistics*, 7:

- 1369–1386, 2013. doi: [10.1214/13-EJS810](https://doi.org/10.1214/13-EJS810). URL <https://doi.org/10.1214/13-EJS810>.
- Jing Lei, Max G’Sell, Alessandro Rinaldo, Ryan J. Tibshirani, and Larry Wasserman. Distribution-free predictive inference for regression. *Journal of the American Statistical Association*, 113(523):1094–1111, 2018. doi: [10.1080/01621459.2017.1307116](https://doi.org/10.1080/01621459.2017.1307116). URL <https://doi.org/10.1080/01621459.2017.1307116>.
- Friedrich Leisch and Evgenia Dimitriadou. *mlbench: Machine Learning Benchmark Problems*, 2021. URL <https://CRAN.R-project.org/package=mlbench>. R package version 2.1-3.
- Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002. URL <https://CRAN.R-project.org/doc/Rnews/>.
- Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, 2008. doi: [10.1109/ICDM.2008.17](https://doi.org/10.1109/ICDM.2008.17). URL <https://doi.org/10.1109/ICDM.2008.17>.
- Wei-Yin Loh. Regression trees with unbiased variable selection and interaction detection. *Statistica Sinica*, 12(1):361–386, 2002. URL <https://doi.org/10.1214/09-AOAS260>.
- Wei-Yin Loh. Regression trees with unbiased variable selection and interaction detection. *Annals of Applied Statistics*, 3(4):1710–1737, 2009.
- Wei-Yin Loh. Classification and regression trees. *WIREs Data Mining and Knowledge Discovery*, 1(1):14–23, 2011. URL <https://doi.org/10.1002/widm.8>.
- Wei-Yin Loh. *Variable Selection for Classification and Regression in Large p, Small n Problems*, volume 205, pages 135–159. 12 2012. ISBN 978-1-4614-1965-5. doi: [10.1007/978-1-4614-1966-2\\_10](https://doi.org/10.1007/978-1-4614-1966-2_10).
- Wei-Yin Loh. Fifty years of classification and regression trees. *International Statistical Review / Revue Internationale de Statistique*, 82(3):329–348, 2014.
- Wei-Yin Loh. User manual for guide ver. 34.0, 2020. URL <http://pages.stat.wisc.edu/~loh/treeprogs/guide/guideman.pdf>.
- Wei-Yin Loh and Yu-Shan Shih. Split selection methods for classification trees. *Statistica Sinica*, 7(4):815–840, 1997.
- Wei-Yin Loh and Nunta Vanichsetakul. Tree structured classification via generalized discriminant analysis. *Journal of the American Statistical Association*, 83:715–728, 1988.

- Wei-Yin Loh and Wei Zheng. Regression trees for longitudinal and multiresponse data. *The Annals of Applied Statistics*, 7(1):495–522, 2013. doi: [10.1214/12-AOAS596](https://doi.org/10.1214/12-AOAS596). URL <https://doi.org/10.1214/12-AOAS596>.
- Wei-Yin Loh and Peigen Zhou. *The GUIDE Approach to Subgroup Identification*, pages 147–165. Springer International Publishing, Cham, 2020. ISBN 978-3-030-40105-4. doi: [10.1007/978-3-030-40105-4\\_6](https://doi.org/10.1007/978-3-030-40105-4_6). URL [https://doi.org/10.1007/978-3-030-40105-4\\_6](https://doi.org/10.1007/978-3-030-40105-4_6).
- Wei-Yin Loh and Peigen Zhou. Variable importance scores. *Journal of Data Science*, 19(4):569–592, 2021. ISSN 1680-743X. doi: [10.6339/21-JDS1023](https://doi.org/10.6339/21-JDS1023). URL <https://doi.org/10.6339/21-JDS1023>.
- Wei-Yin Loh, Xu He, and Michael Man. A regression tree approach to identifying subgroups with differential treatment effects. *Statistics in Medicine*, 34(11):1818–1833, 2015. URL <https://doi.org/10.1002/sim.6454>.
- Wei-Yin Loh, John Eltinge, Moon Jung Cho, and Yuanzhi Li. Classification and regression trees and forests for incomplete data from sample surveys. *Statistica Sinica*, 29(1):431–453, 2019. ISSN 10170405, 19968507. doi: [10.5705/ss.202017.0225](https://doi.org/10.5705/ss.202017.0225). URL <https://doi.org/10.5705/ss.202017.0225>.
- Wei-Yin Loh, Qiong Zhang, Wenwen Zhang, and Peigen Zhou. Imissing data, imputation and regression trees. *Statistica Sinica*, 30:1697–1722, 2020.
- Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017. URL <http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf>.
- Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. From local explanations to global understanding with explainable ai for trees. *Nature Machine Intelligence*, 2(1):2522–5839, 2020.
- Javier Luraschi, Kevin Kuo, Kevin Ushey, J. J. Allaire, Hossein Falaki, Lu Wang, Andy Zhang, Yitao Li, and The Apache Software Foundation. *sparklyr: R Interface to Apache Spark*, 2021. URL <https://spark.rstudio.com/>. R package version 1.7.3.
- Szymon Maksymiuk, Alicja Gosiewska, and Przemyslaw Biecek. Landscape of r packages for explainable artificial intelligence, 2021. URL <https://arxiv.org/abs/2009.13248>.
- James D. Malley, Jochen Kruppa, Abhijit Dasgupta, Karen Godlove Malley, and Andreas Ziegler. Probability machines: consistent probability estimation using nonparametric learning machines. *Methods of Information in*

- Medicine, 51(1):74–81, 2012. URL <https://doi.org/10.3414/ME00-01-0052>.
- Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008. ISBN 0521865719.
- Norm Matloff. *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, 2011. ISBN 9781593273842.
- Norm Matloff. *regtools: Regression and Classification Tools*, 2019. URL <https://github.com/matloff/regtools>. R package version 1.1.0.
- Norman Matloff. *Statistical Regression and Classification: From Linear Models to Machine Learning*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press, 2017. ISBN 9781351645898.
- David Mease, Abraham J. Wyner, and A. Buja. Boosted classification trees and class probability/quantile estimation. *Journal of Machine Learning Research*, 8:409–439, 2007.
- Nicolai Meinshausen. Quantile regression forests. *Journal of Machine Learning Research*, 7(35):983–999, 2006. URL <http://jmlr.org/papers/v7/meinshausen06a.html>.
- Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016. URL <http://jmlr.org/papers/v17/15-237.html>.
- Bjoern Menze and Nico Splitthoff. *obliqueRF: Oblique Random Forests from Recursive Linear Model Splits*, 2012. URL <https://CRAN.R-project.org/package=obliqueRF>. R package version 0.3.
- Bjoern H. Menze, B. Michael Kelm, Daniel N. Splitthoff, Ullrich Koethe, and Fred A. Hamprecht. On oblique random forests. In *Machine Learning and Knowledge Discovery in Databases*, pages 453–469. Springer Berlin, Heidelberg, 2011. ISBN 978-3-642-23783-6.
- Olaf Mersmann. *microbenchmark: Accurate Timing Functions*, 2021. URL <https://github.com/joshuaulrich/microbenchmark/>. R package version 1.4.9.
- Robert Messenger and Lewis Mandell. A modal search technique for predictive nominal scale multivariate analysis. *Journal of the American Statistical Association*, 67(340):768–772, 1972.

- Daniele Micci-Barreca. A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems. *SIGKDD Explor. Newsl.*, 3(1):27–32, July 2001. ISSN 1931-0145. doi: [10.1145/507533.507538](https://doi.org/10.1145/507533.507538). URL <https://doi.org/10.1145/507533.507538>.
- Stephen Milborrow. *earth: Multivariate Adaptive Regression Splines*, 2021a. URL <http://www.milbo.users.sonic.net/earth/>. R package version 5.3.1.
- Stephen Milborrow. *rpart.plot: Plot rpart Models: An Enhanced Version of plot.rpart*, 2021b. URL <http://www.milbo.org/rpart-plot/index.html>. R package version 3.1.0.
- Matthew W. Mitchell. Bias of the random forest out-of-bag (oob) error for certain input parameters. *Open Journal of Statistics*, 1(3):205–211, 10 2011. doi: [10.4236/ojs.2011.13024](https://doi.org/10.4236/ojs.2011.13024). URL <https://doi.org/10.4236/ojs.2011.13024>.
- Christoph Molnar. *Interpretable Machine Learning*. 2019. URL <https://christophm.github.io/interpretable-ml-book/>.
- Christoph Molnar and Patrick Schratz. *iml: Interpretable Machine Learning*, 2020. URL <https://CRAN.R-project.org/package=iml>. R package version 0.10.1.
- Christoph Molnar, Gunnar König, Julia Herbinger, Timo Freiesleben, Susanne Dandl, Christian A. Scholbeck, Giuseppe Casalicchio, Moritz Grosse-Wentrup, and Bernd Bischl. General pitfalls of model-agnostic interpretation methods for machine learning models, 2021.
- James N. Morgan and John A. Sonquist. Problems in the analysis of survey data, and a proposal. *Journal of the American Statistical Association*, 58 (302):415–434, 1963.
- Sérgio Moro, Paulo Cortez, and Paulo Rita. A data-driven approach to predict the success of bank telemarketing. *Decision Support Systems*, 62: 22–31, 2014. ISSN 0167-9236. doi: <https://doi.org/10.1016/j.dss.2014.03.001>. URL <https://www.sciencedirect.com/science/article/pii/S016792361400061X>.
- Stefano Nembrini, Inke R König, and Marvin N. Wright. The revival of the gini importance? *Bioinformatics*, 34(21):3711–3718, 05 2018. ISSN 1367-4803. doi: [10.1093/bioinformatics/bty373](https://doi.org/10.1093/bioinformatics/bty373). URL <https://doi.org/10.1093/bioinformatics/bty373>.
- Anna Neufeld. *treevalues: Selective Inference for Regression Trees*, 2022. URL <https://github.com/anna-neufeld/treevalues>. R package version 0.1.0.

Anna C. Neufeld, Lucy L. Gao, and Daniela M. Witten. Tree-values: selective inference for regression trees, 2021.

Alexandru Niculescu-Mizil and Rich Caruana. Predicting good probabilities with supervised learning. In *Proceedings of the 22nd International Conference on Machine Learning*, ICML '05, pages 625–632, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595931805. doi: [10.1145/1102351.1102430](https://doi.org/10.1145/1102351.1102430). <https://doi.org/10.1145/1102351.1102430>.

Deborah Ann Nolan and Duncan Temple Lang. *Data Science in R: a Case Studies Approach to Computational Reasoning and Problem Solving*. Chapman & Hall/CRC the R series. CRC Press, Boca Raton, 2015. ISBN 9781482234817.

Harsha Nori, Samuel Jenkins, Paul Koch, and Rich Caruana. Interpretml: A unified framework for machine learning interpretability. *arXiv preprint arXiv:1909.09223*, 2019.

Szilard Pafka. benchm-ml. <https://github.com/szilard/benchm-ml>, 2019. URL <https://github.com/szilard/benchm-ml>.

Szilard Pafka. Gradient boosting machines (gbm): From zero to hero (with r and python code). [https://docs.google.com/presentation/d/1WdQajKNeJR5gJs437XUuLksBJPm4rowdzH3i1vEWTHA/edit#slide=id.g58411bbf6a\\_0\\_15](https://docs.google.com/presentation/d/1WdQajKNeJR5gJs437XUuLksBJPm4rowdzH3i1vEWTHA/edit#slide=id.g58411bbf6a_0_15), 2 2020. URL [https://docs.google.com/presentation/d/1WdQajKNeJR5gJs437XUuLksBJPm4rowdzH3i1vEWTHA/edit#slide=id.g58411bbf6a\\_0\\_15](https://docs.google.com/presentation/d/1WdQajKNeJR5gJs437XUuLksBJPm4rowdzH3i1vEWTHA/edit#slide=id.g58411bbf6a_0_15).

Szilard Pafka. Gbm-perf. <https://github.com/szilard/GBM-perf>, 2021. URL <https://github.com/szilard/GBM-perf>.

Terence Parr and James D. Wilson. Technical report: Partial dependence through stratification, 2019. URL <https://arxiv.org/abs/1907.06698>.

Andrea Peters and Torsten Hothorn. ipred: Improved Predictors, 2021. URL <https://CRAN.R-project.org/package=ipred>. R package version 0.9-12.

Kivan Polimis, Ariel Rokem, and Bryna Hazelton. Confidence intervals for random forests in python. *Journal of Open Source Software*, 2(1), 2017.

Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features, 2017. URL <https://arxiv.org/abs/1706.09516>.

Randall Pruim, Daniel T. Kaplan, and Nicholas J. Horton. mosaic: Project MOSAIC Statistics and Mathematics Teaching Utilities, 2021. URL <https://CRAN.R-project.org/package=mosaic>. R package version 1.8.3.

Ross J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in machine learning. Elsevier Science, 1993. ISBN 9781558602380.

- Revolution Analytics and Steve Weston. *foreach: Provides Foreach Looping Construct*, 2020. URL <https://github.com/RevolutionAnalytics/foreach>. R package version 1.5.1.
- Greg Ridgeway. The state of boosting. *Computing Science and Statistics* 31:172–181s, 31:172–181, 1999.
- Robert A. Rigby and Mikis D. Stasinopoulos. Generalized additive models for location, scale and shape. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 54(3):507–554, 2005. doi: <https://doi.org/10.1111/j.1467-9876.2005.00510.x>. URL <https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-9876.2005.00510.x>.
- Brian Ripley. *tree: Classification and Regression Trees*, 2021. URL <https://CRAN.R-project.org/package=tree>. R package version 1.0-41.
- Brian Ripley. *MASS: Support Functions and Datasets for Venables and Ripley's MASS*, 2022. URL <http://www.stats.ox.ac.uk/pub/MASS4/>. R package version 7.3-55.
- Brian D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996. ISBN 9780521717700.
- Juan J. Rodríguez, Ludmila I. Kuncheva, and Carlos J. Alonso. Rotation forest: A new classifier ensemble method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(10):1619–1630, 2006. doi: <10.1109/TPAMI.2006.211>.
- Kaspar Rufibach. Use of brier score to assess binary predictions. *Journal of Clinical Epidemiology*, 63(8):938–939, Aug 2010.
- Marco Sandri and Paola Zuccolotto. A bias correction algorithm for the gini variable importance measure in classification trees. *Journal of Computational and Graphical Statistics*, 17(3):611–628, 2008. doi: <10.1198/106186008X344522>. URL <https://doi.org/10.1198/106186008X344522>.
- Deepayan Sarkar. *lattice: Trellis Graphics for R*, 2021. URL <http://lattice.r-forge.r-project.org/>. R package version 0.20-45.
- Mark Robert Segal. Regression trees for censored data. *Biometrics*, 44(1): 35–47, 1988.
- Stephen J. Senn. Dichotomania: An obsessive compulsive disorder that is badly affecting the quality of analysis of pharmaceutical trials. In *Proceedings of the International Statistical Institute, 55th Session*, Sydney, 2005.
- Juliet P. Shaffer. Multiple hypothesis testing. *Annual Review of Psychology*, 46(1):561–584, 1995. URL <https://doi.org/10.1146/annurev.ps.46.020195.003021>.

- Lloyd S. Shapley. 17. *A Value for n-Person Games*, pages 307–318. Princeton University Press, 2016. URL <https://doi.org/10.1515/9781400881970-018>.
- Haijian Shi. *Best-first Decision Tree Learning*. PhD thesis, Hamilton, New Zealand, 2007. URL <https://hdl.handle.net/10289/2317>. Masters.
- Tao Shi, David Seligson, Arie Belldegrun, Aarno Palotie, and Steve Horvath. Tumor classification by tissue microarray profiling: random forest clustering applied to renal cell carcinoma. *Modern Pathology*, 18:547–57, 05 2005. doi: [10.1038/modpathol.3800322](https://doi.org/10.1038/modpathol.3800322). URL <https://doi.org/10.1038/modpathol.3800322>.
- Yu Shi, Guolin Ke, Damien Soukhavong, James Lamb, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, Tie-Yan Liu, and Nikita Titov. *lightgbm: Light Gradient Boosting Machine*, 2022. URL <https://github.com/Microsoft/LightGBM>. R package version 3.3.2.
- Julia Silge, Fanny Chow, Max Kuhn, and Hadley Wickham. *rsample: General Resampling Infrastructure*, 2021. URL <https://CRAN.R-project.org/package=rsample>. R package version 0.1.1.
- Nora Sleumer. *Hyperplane arrangements: construction visualization and applications*. PhD thesis, Swiss Federal Institute of Technology, 1969. PhD dissertation.
- Helmut Strasser and Christian Weber. On the asymptotic theory of permutation statistics. *Mathematical Methods of Statistics*, 2(27), 1999.
- Carolin Strobl, Anne-Laure Boulesteix, Achim Zeileis, and Torsten Hothorn. Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC Bioinformatics*, 8(25), 2007a. doi: [10.1186/1471-2105-8-25](https://doi.org/10.1186/1471-2105-8-25).
- Carolin Strobl, Anne-Laure Boulesteix, Achim Zeileis, and Torsten Hothorn. Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC Bioinformatics*, 8(25), 2007b. URL <https://doi.org/10.1186/1471-2105-8-25>.
- Carolin Strobl, Anne-Laure Boulesteix, Thomas Kneib, Thomas Augustin, and Achim Zeileis. Conditional variable importance for random forests. *BMC Bioinformatics*, 9(307), 2008a. doi: [10.1186/1471-2105-9-307](https://doi.org/10.1186/1471-2105-9-307).
- Carolin Strobl, Anne-Laure Boulesteix, Thomas Kneib, Thomas Augustin, and Achim Zeileis. Conditional variable importance for random forests. *BMC Bioinformatics*, 9(307), 2008b. URL <https://doi.org/10.1186/1471-2105-9-307>.
- The Pandas Development Team. pandas-dev/pandas: Pandas, February 2020. URL <https://doi.org/10.5281/zenodo.3509134>.

- Terry Therneau and Beth Atkinson. *rpart: Recursive Partitioning and Regression Trees*, 2019. URL <https://CRAN.R-project.org/package=rpart>. R package version 4.1-15.
- Terry M. Therneau. *survival: Survival Analysis*, 2021. URL <https://github.com/therneau/survival>. R package version 3.2-13.
- Julie Tibshirani, Susan Athey, Erik Sverdrup, and Stefan Wager. *grf: Generalized Random Forests*, 2021. URL <https://github.com/grf-labs/grf>. R package version 2.0.2.
- Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. ISSN 00359246. URL <http://www.jstor.org/stable/2346178>.
- Stef van Buuren. *Flexible Imputation of Missing Data*. Chapman & Hall/CRC Interdisciplinary Statistics. CRC Press, Taylor & Francis Group, 2018. ISBN 9781138588318. URL <https://books.google.com/books?id=bLMiItgEACAAJ>.
- Stef van Buuren and Karin Groothuis-Oudshoorn. *mice: Multivariate Imputation by Chained Equations*, 2021. URL <https://CRAN.R-project.org/package=mice>. R package version 3.14.0.
- Mark J. van der Laan. Statistical inference for variable importance. *The International Journal of Biostatistics*, 2(1), 2006. URL <https://doi.org/10.2202/1557-4679.1008>.
- Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki, Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael Franklin, Ion Stoica, and Matei Zaharia. Sparkr: Scaling r programs with spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD ’16, pages 1099–1104, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: [10.1145/2882903.2903740](https://doi.org/10.1145/2882903.2903740). URL <https://doi.org/10.1145/2882903.2903740>.
- James Verbus. Detecting and preventing abuse on linkedin using isolation forests, Aug. 2019. URL <https://engineering.linkedin.com/blog/2019/isolation-forest>.
- Rashmi Korlakai Vinayak and Ran Gilad-Bachrach. DART: Dropouts meet Multiple Additive Regression Trees. In Guy Lebanon and S. V. N. Vishwanathan, editors, *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *Proceedings of Machine Learning Research*, pages 489–497, San Diego, California, USA, 09–12 May 2015. PMLR. URL <http://proceedings.mlr.press/v38/korlakaivinayak15.html>.

- Erik Štrumbelj and Igor Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowledge and Information Systems*, 31(3):647–665, 2014. URL <https://doi.org/10.1007/s10115-013-0679-x>.
- Stefan Wager, Trevor Hastie, and Bradley Efron. Confidence intervals for random forests: The jackknife and the infinitesimal jackknife. *Journal of Machine Learning Research*, 15(48):1625–1651, 2014. URL <http://jmlr.org/papers/v15/wager14a.html>.
- Ian R. White, Patrick Royston, and Angela M. Wood. Multiple imputation using chained equations: Issues and guidance for practice. *Statistics in Medicine*, 30(4):377–399, 2011. doi: 10.1002/sim.4067. URL <https://doi.org/10.1002/sim.4067>.
- Hadley Wickham. *Advanced R, Second Edition*. Chapman & Hall/CRC The R Series. CRC Press, 2019. ISBN 9781351201308. URL <https://adv-r.hadley.nz/>.
- Hadley Wickham and Jennifer Bryan. *readxl: Read Excel Files*, 2019. URL <https://CRAN.R-project.org/package=readxl>. R package version 1.3.1.
- Hadley Wickham, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, and Dewey Dunnington. *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*, 2021a. URL <https://CRAN.R-project.org/package=ggplot2>. R package version 3.3.5.
- Hadley Wickham, Romain François, Lionel Henry, and Kirill Müller. *dplyr: A Grammar of Data Manipulation*, 2021b. URL <https://CRAN.R-project.org/package=dplyr>. R package version 1.0.7.
- Edwin B. Wilson and Margaret M. Hildferty. The distribution of chi-square. *Proceedings of the National Academy of Sciences of the United States of America*, 17(12):684–688, 1931.
- Marvin N. Wright, Stefan Wager, and Philipp Probst. *ranger: A Fast Implementation of Random Forests*, 2021. URL <https://github.com/imbs-hl/ranger>. R package version 0.13.1.
- Paul S. Wright. Adjusted p-values for simultaneous inference. *Biometrics*, 48(4):1005–1013, 1992. doi: 10.2307/2532694. URL <https://doi.org/10.2307/2532694>.
- Yihui Xie. *knitr: A General-Purpose Package for Dynamic Report Generation in R*, 2021. URL <https://yihui.org/knitr/>. R package version 1.36.
- Ruo Xu, Dan Nettleton, and Daniel J. Nordman. Case-specific random forests. *Journal of Computational and Graphical Statistics*, 25(1):49–65,

2016. doi: [10.1080/10618600.2014.983641](https://doi.org/10.1080/10618600.2014.983641). URL <https://doi.org/10.1080/10618600.2014.983641>.
- I-Cheng Yeh and Che hui Lien. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Expert Systems with Applications*, 36(2, Part 1):2473–2480, 2009. doi: <https://doi.org/10.1016/j.eswa.2007.12.020>. URL <https://doi.org/10.1016/j.eswa.2007.12.020>.
- Achim Zeileis, Friedrich Leisch, Kurt Hornik, and Christian Kleiber. *strucchange: Testing, Monitoring, and Dating Structural Changes*, 2019. URL <https://CRAN.R-project.org/package=strucchange>. R package version 1.5-2.
- Haozhe Zhang, Joshua Zimmerman, Dan Nettleton, and Daniel J. Nordman. Random forest prediction intervals. *The American Statistician*, 74(4):392–406, 2020. doi: [10.1080/00031305.2019.1585288](https://doi.org/10.1080/00031305.2019.1585288). URL <https://doi.org/10.1080/00031305.2019.1585288>.
- Heping Zhang and Burton H. Singer. *Recursive Partitioning and Applications*. Springer New York, New York, NY, 2010. ISBN 978-1-4419-6824-1. URL [https://doi.org/10.1007/978-1-4419-6824-1\\_3](https://doi.org/10.1007/978-1-4419-6824-1_3).
- Huan Zhang, Si Si, and Cho-Jui Hsieh. Gpu-acceleration for large-scale tree boosting, 2017. URL <https://arxiv.org/abs/1706.08359>.

---

# Index

---

- K*-sample test, 115, 121, 133  
*k*-nearest neighbor, 9, 161–163, 173  
*p*-value, 116, 117, 120–126, 128, 132–134  
1-SE rule, 78, 84, 93, 95–97, 102, 103, 107, 155, 159, 170  
accumulated local effect (ALE) plots, 226  
AdaBoost, 188–194  
AID, *see* automatic interaction detection  
amyotrophic lateral sclerosis (ALS) or Lou Gerig’s disease, 320, 323, 334, 346, 348  
analysis of covariance, 3  
analysis of variance, 59  
    *F*-test, 115, 121, 148  
anomaly detection, 269, 272, 275  
anomaly score, 269  
ANOVA, *see* analysis of variance  
asymptotic, 116, 117, 120, 123  
automatic interaction and detection, 13  
backward elimination, 71, 108  
bagging, *see* bootstrap aggregating, 36  
bandwidth, 162  
base learner, 180–182, 187–189, 192–194  
Bayesian additive regression trees, 334  
best-subset selection, 106  
bias-variance tradeoff, 2, 8, 9  
binomial deviance, 315, 316  
Bonferroni adjustment, 122, 123, 128, 133  
boosting, 36, 180, 187–190, 192, 194, 195, 197  
bootstrap, 81, 132, 157  
bootstrap aggregating, 164, 180–189, 191, 192, 195–198, 201  
branch, 14  
Brier score, 288, 297, 300, 302  
C4.5, 13, 20, 35, 111, 132  
C5.0, 13, 35, 111, 132, 194  
calibration, 294–297, 302  
canonical analysis, 148  
canonical variate, 148  
CART, *see* classification and regression trees  
case-specific random forest, 254  
CatBoost, 338, 339  
censoring  
    interval censoring, 32  
    left censoring, 32  
    right censoring, 2, 32, 33  
censoring indicator, 31, 32  
CHAID, *see* chi-squared automatic interaction detection  
chi-square automatic interaction detection, 13, 113  
chi-square distribution, 120, 152  
chi-square test, 112, 114, 148, 150–154, 157, 163, 172  
child node, 13  
class imbalance, 8  
classification

- binary classification, 7
- multiclass classification, 7
- classification and regression trees, 13, 15, 35, 36, 40–45, 47, 52–54, 58, 60, 67, 69, 71, 74, 78, 80–84, 104–108, 111–113, 125, 126, 128, 130–132, 136, 143, 144, 147–155, 157, 159, 161–164, 171, 172, 180, 184, 194, 195, 202, 232, 243, 245–247, 252, 253, 259, 277, 284, 296, 298, 312, 315, 317, 328, 332, 336
- classification rule with unbiased interaction selection and estimation, 14, 148, 149, 165
- complexity parameter, 73, 74, 76, 78, 89, 91, 95
- conditional expectation, 5
- conditional inference, 111, 113–115, 118, 121–123, 126, 133, 142
- conditional inference trees, 17, 39, 112, 113, 121, 125–129, 131–133, 135, 138, 139, 141–144, 147, 149–155, 159, 161, 163, 164, 172, 202, 232, 237–239, 242, 247, 252, 283, 324, 339, 343
- conditional permutation importance, 249
- conditional random forest (CRF), 237–240, 247
- convex hull, 214, 291
- correlation test, 114, 115, 123, 133
- Cox proportional hazards (Cox PH) model, 339, 340
- cross-join, 213
- cross-validation, 51, 71, 72, 76–78, 89, 91, 93–95, 97, 102, 103, 107, 130, 136, 144, 155, 159, 170, 205, 239, 242–244, 279, 318, 323, 329, 331, 340, 352
- CRUISE, *see* classification rule with unbiased interaction
- selection and estimation
- CTree, *see* conditional inference trees
- cumulative lift chart, 273
- curvature test, 150, 151, 154
- data wrangling, 213
- decision boundary, 9
  - Bayes decision boundary, 9
- deep learning, 223
- description, 5
- dropout, 336
- dummy encoding, 68
- early stopping, 318, 319, 322, 337, 339, 346, 351–353, 357
- edge, 14
- effect encoding, 68
- elastic net, 196
- empirical distribution function, 33
- ensemble, 11, 159, 164, 179–185, 187–189, 192, 195–199, 201, 202
- exclusive feature bundling (EFB), 337
- eXtreme Gradient Boosting (XGBoost), 321, 333, 335–339, 351–353, 355, 356
- extremely randomized trees, 232
- extremely randomized trees (extra-trees), 267–269, 277
- FACT, *see* fast and accurate classification tree
- false color level plot, 213
- family-wise error rate, 122

- fast and accurate classification tree, 148  
fast histogram binning, 337  
feature contribution, 204, 217, 220, 223  
forward selection, 108  
forward-stepwise selection, 106  
function approximation, 4  
*GBM*, *see* gradient boosting  
generalized additive model (GAM), 333  
generalized additive models for shape, scale, and location (GAMLSS), 334  
generalized linear model, 313  
generalized random forests, 257  
generalized, unbiased, interaction detection, and estimation, 147–159, 161–166, 168–173, 202, 232, 259, 277, 298  
gradient boosting, 159, 243  
gradient boosting machines (GBM), 309, 314, 317–323, 326–333, 335, 336, 338, 339, 346–349, 351, 356, 357  
gradient descent, 310, 311, 335  
gradient-based one-side sampling (GOSS), 337  
GUIDE, *see* generalized, unbiased, interaction detection, and estimation  
hard voting, 232, 234  
hessian, 336  
heteroscedasticity, 29  
holdout variable importance, 248  
Huber M-regression, 315  
importance sampled learning ensemble, 196  
imputation, 81, 253, 284–288, 293  
independence test, 114, 115, 121, 122, 128, 129, 133  
independently and identically distributed, 4  
individual conditional expectation (ICE) plots, 209, 211, 214, 215, 225, 226, 304  
influence function, 114, 115, 119, 142  
interaction effect, 150, 153, 161, 163, 172  
internal node, *see* node, internal  
interpretable machine learning, 203  
interquartile range, 162  
IQR, *see* interquartile range  
ISLE, *see* importance sampled learning ensemble  
isolation forest, 269–272, 275, 277  
isolation tree (IsoTree), 269, 270  
J4.8, 20  
jackknife, 256–258  
jujitsu, 116  
**Julia packages**  
    **DecisionTree.jl**, 84, 277  
kernel density estimate, 161, 162  
Kronecker product, 115, 119, 120  
Kruskal-Wallis test, 115, 121  
largest discriminant coordinate, 148  
LASSO, *see* least absolute shrinkage and selection operator  
latent variable model, 295  
LDA, *see* linear discriminant analysis  
leaf-wise tree induction, 332  
learning rate, 319, 320, 323, 333, 340, 346, 357  
least absolute deviation (LAD) loss, 313–316, 325  
least absolute shrinkage and

- selection operator, 196, 199, 201, 277, 279, 280, 282, 283, 336, 347–349
- least squares, 192
- least squares (LS) loss, 310, 314–318, 320, 323–325, 349
- leave-one-covariate-out, 204
- level-wise tree induction, 332
- lift chart, 297
- LightGBM, 332, 333, 337–339, 356
- linear discriminant analysis, 148, 158, 159
- linear discriminant analysis (LDA), 148
- linear predictor, 313
- linear splits, 148, 157–159, 161, 172
- link function, 333
- log-likelihood, 139
- log-loss, 300
- log-rank scores, 116, 141, 142
- log-rank test, 34
- loss function, 192, 193, 310, 311, 313–317, 335–337, 340
- LOWESS smooth, 22
- M5, 20
- main effect, 148–151, 161, 172, 226
- majority vote classifier, 188
- MARS, *see* multivariate adaptive regression splines
- missing values, 78–81, 128
- model-agnostic, 204, 210, 227
- monotonic constraints, 329
- Monte Carlo, 129
- mosaic plot, 25
- multidimensional scaling (MDS), 254
- multiple additive regression trees (MART), 339
- multivariate adaptive regression splines, 11–13, 35, 180, 184
- multiway splits, 13
- natural gradient, 335
- natural gradient boosting (NGBoost), 334, 335
- neural networks, 6
- node, 14
  - internal, 14
  - root, 14
- null hypothesis, 114, 115, 117, 118, 122, 123, 127–129, 133, 142
- object-oriented (OO) programming, 324
- oblique random forest (ORF), 259
- oblique splits, *see* linear splits
- off-the-shelf, 11, 243
- offset, 313, 347
- one-hot encoding, 206
- one-hot-encoding, 68
- OOB, *see* out-of-bag
- ordered target statistics, 338
- ordinary least squares, 157
- out-of-bag, 182, 183, 233, 239–245, 247, 248, 250, 253, 256, 287, 288, 297, 303
- outliers, 123, 129
- overfitting, 8
- overspecialization, 336
- partial dependence (PD) plots, 208–216, 289, 304
- pasting, 187
- permutation importance, 132, 204
- permutation test, 114–116, 121, 129, 132
- polynomial regression, 9, 22, 163, 173
- positive predictive value, 273
- prediction, 5
- prevalance, 296
- principal component analysis (PCA), 261–264, 267, 272, 275

- probability machine, 234, 267, 277, 287  
projection pursuit random forest (PPforest), 259  
projection pursuit trees (PPtrees), 259  
proper scoring rule, 300, 335  
proximities, 249–256, 269  
pruning, 128, 130, 143, 144, 151, 155, 159, 163, 170
- Python functions**  
`fit`, 346  
`pred_dist`, 347  
`predict`, 347
- Python modules**  
`eif`, 271  
`forestci`, 257  
`inspection`, 206, 211  
`interpret`, 334  
`ngboost`, 335, 346  
`numpy`, 346  
`pandas`, 346  
`pyspark`, 277, 300  
`scipy`, 346  
`shap`, 220, 223, 354  
`sklearn.calibration`, 296  
`sklearn.datasets`, 24  
`sklearn.ensemble`, 188, 194, 269, 277, 333, 339  
`sklearn.inspection`, 329, 353  
`sklearn.tree`, 40, 59, 69, 84, 108, 237  
`skranger`, 247
- QDA, *see* quadratic discriminant analysis  
quadratic discriminant analysis, 148, 159  
quantile regression, 260  
quantile regression forest (QRF), 260, 261  
quantization, 338  
QUEST, *see* quick, unbiased, and efficient statistical tree
- quick, unbiased, and efficient statistical tree, 14, 148, 149, 165
- R functions**  
`apply`, 19  
`approxQuantile`, 304  
`as.vector`, 118  
`assess.glmnet`, 200  
`autoplot`, 223  
`cart`, 81  
`cforest`, 237  
`combn`, 342  
`complete`, 285  
`createDataFrame`, 302  
`createDataPartition`, 301  
`crforest`, 237, 238, 247, 277  
`ctree_control`, 133  
`ctree`, 132–134, 237  
`dcast`, 351  
`deforest`, 278  
`dummyVars`, 351  
`err`, 185, 191  
`expand.grid`, 213  
`explain`, 223, 343  
`fac2num`, 191  
`find_best_split`, 50, 64, 66  
`for`, 18, 19, 65  
`getHdata`, 283  
`gi.test`, 123–125, 129, 133–135  
`ginv`, 119  
`glmnet`, 199, 200  
`identity`, 123  
`independence_test`, 118, 120, 123, 125  
`interact.gbm`, 342  
`interaction`, 290  
`isle_post`, 199, 201, 279, 348, 349  
`isolation.forest`, 275  
`kronecker`, 119

ladd, 345  
 lapply, 18, 19  
 list, 190  
 logrank\_trafo, 116, 141  
 lsboost, 320, 321, 323, 324,  
     326, 348, 349  
 melt, 351  
 model.matrix, 120  
 nodes, 135  
 ordinalize, 65, 67  
 p.adjust, 123, 134  
 palette.colors, xv  
 partial, 214, 216, 290  
 par, 20  
 pfun, 292  
 plotcp, 93, 94  
 plot, 88, 134, 136, 143  
 predict.def, 282  
 predict.lsboost, 324  
 predict.rpart, 191  
 predict, 222, 302, 324, 327,  
     353, 354  
 printcp, 94  
 proximity, 251  
 prune\_se, 96  
 prune, 95  
 qr, 120  
 rfiImpute, 286  
 rforest, 266  
 rowMeans, 241  
 rpart.control, 91, 93  
 rpart.plot, 88  
 rpart, 59, 85, 89–93, 103, 104,  
     181, 185  
 rrm, 263  
 sample.shap, 222  
 sapply, 18, 19, 61, 327  
 scetest, 134  
 spark.randomForest, 302  
 splits, 48, 50, 60, 87  
 summarize, 302  
 summary, 86, 88, 92  
 system.time, 198, 278  
 text, 88  
 theme\_bw, 20  
 theme\_set, 20  
 tree\_diagram, 88, 90  
 treepreds, 279  
 val.prob, 296  
 vapply, 18  
 varimp, 139  
 vote, 185  
 waterfallchart, 345  
 xgb.DMatrix, 352  
 xgb.cv, 352  
 xgb.train, 352  
 xtabs, 118

**R packages**

- ALEPlot, 226
- AmesHousing, 29
- C5.0, 132, 194
- C50, 13
- Hmisc, 283
- ICEbox, 211
- MASS, 119
- PPforest, 259
- PPtreeViz, 259
- PPtree, 259
- RWeka, 19
- SparkR, 213, 277,  
    300–304
- adabag, 188, 194
- ada, 194
- bar, 19
- base, 19
- boot, 132
- caret, 301, 351
- coin, 114, 116, 118, 120, 123,  
    125, 129, 133, 141
- data.table, 213, 300, 351
- datasets, 21, 215
- dplyr, 213, 300, 304
- earth, 11
- fastshap, 222, 223, 225, 275,  
    292, 343
- foreach, 183, 238
- gbm, 194, 210, 211, 309, 316,  
    322, 329, 336, 339–342,  
    353
- ggplot2, 20, 223, 290, 301

- glmnet**, 196, 199, 200  
**grDevices**, xv  
**graphics**, 20  
**grf**, 257  
**gridExtra**, 341  
**h2o**, 244, 252, 277  
**iBreakDown**, 223  
**iml**, 206, 207, 211, 223, 226, 292  
**ingredients**, 206, 211  
**ipred**, 188, 197  
**isotree**, 272, 275  
**kernlab**, 25, 184  
**knitr**, 17  
**lattice**, 20, 213, 345  
**lightgbm**, 220, 339  
**mboost**, 339  
**mice**, 81, 253, 285  
**microbenchmark**, 198, 278  
**mlbench**, 24, 103  
**mmpf**, 206  
**modeldata**, 28  
**mosaic**, 345  
**obliqueRF**, 259  
**partykit**, 88, 128, 132–134, 136, 138, 139, 237, 247, 249, 277, 283, 324, 325  
**party**, 128, 132–135, 237, 247, 249, 277  
**pdp**, 98, 99, 211, 214, 216, 223, 289, 290, 329, 341, 343  
**purrr**, 18  
**randomForestSRC**, 276  
**randomForest**, 197, 211, 215, 266, 276, 277, 286  
**ranger**, 247, 248, 250, 256, 257, 269, 276–278, 280, 287, 289, 292  
**regtools**, 104  
**rms**, 296  
**rotationForest**, 263, 266  
**rpart.LAD**, 84  
**rpart.plot**, 88  
**rpartScore**, 84  
**rpart**, 19, 27, 39, 40, 42, 55, 59, 63, 67, 73, 75, 77, 80, 83–85, 88, 89, 91–96, 100, 102, 107, 108, 136, 188, 190, 194, 237, 298, 324, 325  
**rsample**, 301  
**sparkR**, 301  
**sparklyr**, 213, 277, 300, 304  
**strucchange**, 134  
**survival**, 31, 34, 140, 340  
**titanic**, 283  
**treemisc**, xvi, 20, 24, 25, 30, 84, 88, 90, 96, 138, 152, 199, 251, 263, 265, 266, 279, 320, 323, 325, 348  
**treevalues**, 84  
**tree**, 83  
**utils**, 19  
**vip**, 206, 207  
**waterfall**, 344  
**xgboost**, 220, 339, 352  
random forest, 36, 159, 164, 182  
random forest (RF), 229, 231–234, 236, 237, 239, 243–269, 276–280, 283–287, 289, 293–300, 302, 303, 306, 307, 309, 317–323, 327–329, 331, 333, 337, 347, 348, 351, 357  
random subspace, 306  
random variable, 4  
regression, 7  
regular expression, 303  
reliability analysis, 32  
ridge regression, 336  
rotation forests, 261  
rotation matrix, 261, 262, 264  
rule-based models, 36  
rules, 28  
sensitivity, 172  
Shapley value, 217–221, 223, 226, 275, 292

- shrinkage, 192, 318–320, 322, 326, 329, 331, 348
- soft voting, 234
- Spark, 277, 300–302, 304, 305
- split conformal prediction, 257
- stagewise additive modeling, 193
- statistical power, 128
- steepest descent, 310–312, 335
- stepwise linear regression, 155
- stochastic gradient boosting, 309
- sum of squared errors (SSE), 58–60, 63, 64, 71, 151, 154
- support vector machines, 6, 294
- surrogate, 128
- surrogate splits, 79
- survival function, 33
- survival time, 31, 32
- survival trees, 181
- target leakage, 12
- THAID, *see theta automatic interaction detection*
- theta automatic interaction detection, 13
- time-to-event, 32
- true positive rate, 172
- twenty questions, 1
- two-sample test, 115, 126, 133
- type I error, 128, 130
- unbiased, 111–113, 121, 122, 133, 143, 147, 148, 152, 157, 158, 164
- underfitting, 8
- unsupervised, 5
- unsupervised learning, 10, 249
- variable importance, 204–206, 211, 217, 226, 236, 245–249, 285, 287–289, 303
- variable types
  - factor, 4
  - nominal categorical, 3
  - ordered categorical, 3
  - ordered numeric, 3
- waterfall chart, 223, 344
- weighted quantile sketch, 337
- Wilson-Hilferty transformation, 151, 152
- wisdom of the crowd, 179