McGraw Hill connect®

Modern Business Analytics

Taddy | Hendrix | Harding

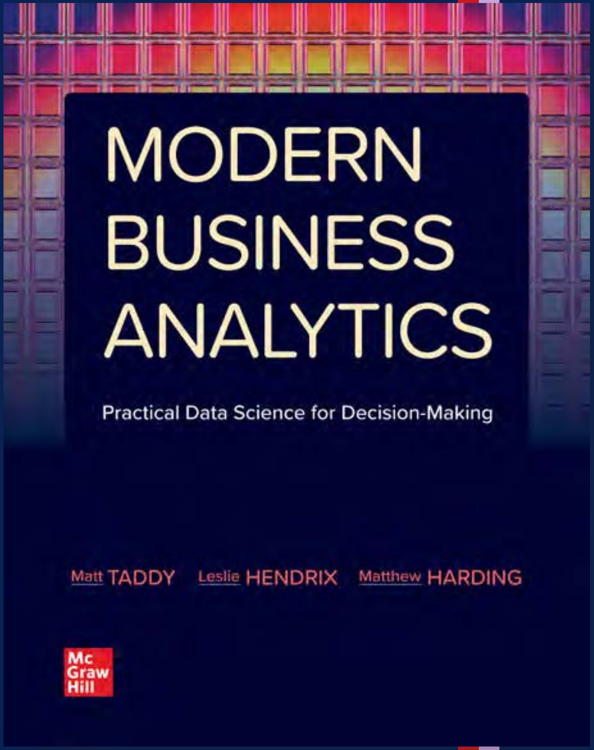

MODERN BUSINESS ANALYTICS
Practical Data Science for Decision-Making
Matt TADDY Leslie HENDRIX Matthew HARDING
Mc Graw Hill

Table of Contents

MODERN BUSINESS ANALYTICS

Practical Data Science for Decision-Making

Matt Taddy
Amazon, Inc.

Leslie Hendrix
University of South Carolina

Matthew C. Harding
University of California, Irvine

Mc Graw Hill

MODERN BUSINESS ANALYTICS

Published by McGraw Hill Education, 1325 Avenue of the Americas, New York, NY 10019. Copyright ©2023 by McGraw Hill Education. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of McGraw Hill Education, including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 LKV 27 26 25 24 23 22

ISBN 978-1-264-07167-8 (SE)
MHID 1-264-07167-1 (SE)
ISBN 978-1-264-07165-4 (Loose leaf)
MHID 1-264-07165-5 (Loose leaf)

Portfolio Manager: *Rebecca Olson*
Product Development Manager: *Michele Janicek*
Product Developer: *Christina Verigan*
Digital Product Developer: *Katherine Ward*
Marketing Manager: *Harper Christopher*
Content Project Managers: *Amy Gehl (Core), Emily Windelborn (Assessment)*
Buyer: *Laura Fuller*
Design: *Matt Diamond*
Content Licensing Specialist: *Melissa Homer*
Cover Image Credit: *MirageC/Getty Images*
Compositor: *Straive*

Library of Congress Cataloging-in-Publication Data

mheducation.com/highered

ABOUT THE AUTHORS

Courtesy of Matt Taddy

Matt Taddy is the author of *Business Data Science* (McGraw Hill, 2019). From 2008–2018 he was a professor of econometrics and statistics at the University of Chicago Booth School of Business, where he developed their Data Science curriculum. Prior to and while at Chicago Booth, he has also worked in a variety of industry positions including as a principal researcher at Microsoft and a research fellow at eBay. He left Chicago in 2018 to join Amazon as a vice president.

Courtesy of Leslie Hendrix

Leslie Hendrix is a clinical associate professor in the Darla Moore School of Business at the University of South Carolina. She received her PhD in statistics in 2011 and a BS in mathematics in 2005 from the University of South Carolina. She has received two university-wide teaching awards for her work in teaching business analytics and statistics courses and is active in the research and teaching communities for analytics. She was instrumental in founding the Moore School's newly formed Data Lab and currently serves as the assistant director.

Courtesy of Matthew C. Harding

Matthew C. Harding is a professor of economics and statistics at the University of California, Irvine. He holds a PhD from MIT and an M.Phil. from Oxford University. Dr. Harding conducts research on econometrics, consumer finance, health policy, and energy economics and has published widely in leading academic journals. He is the founder of Ecometricx, LLC, a big data and machine learning consulting company, and cofounder of FASTlab.global Institute, a nonprofit focusing on education and evidence-based policies in the areas of fair access and sustainable technologies.

MODERN BUSINESS ANALYTICS
Practical Data Science for Decision-Making
Matt TADDY Leslie HENDRIX Matthew HARDING
Matt Taddy
author, Modern Business Analytics

Click below to watch a video for the author:

[Modern Business Analytics]

BRIEF CONTENTS

CONTENTS

PREFACE

What Is This Book About?

The practice of data analytics is changing and modernizing. Innovations in computation and machine learning are creating new opportunities for the data analyst: exposing previously unexplored data to scientific analysis, scaling tasks through automation, and allowing deeper and more accurate modeling. Spreadsheet models and pivot tables are being replaced by code scripts in languages like R and Python. There has been massive growth in digitized information, accompanied by development of systems for storage and analysis of this data. There has also been an intellectual convergence across fields—machine learning and computer science, statistics, and social sciences and economics—that has raised the breadth and quality of applied analysis everywhere. This is the *data science approach to analytics*, and it allows leaders to go deeper than ever to understand their operations, products, and customers.

This book is a primer for those who want to gain the skills to use data science to help make decisions in business and beyond. The modern business analyst uses tools from machine learning, economics, and statistics to not only track what has happened but predict the future for their businesses. Analysts may need to identify the variables important for business policy, run an experiment to measure these variables, and mine social media for information about public response to policy changes. A company might seek to connect small changes in a recommendation system to changes in customer experience and use this information to estimate a demand curve. And any analysis will need to scale to companywide data, be repeatable in the future, and quantify uncertainty about the model estimates and conclusions.

This book focuses on business and economic applications, and we expect that our core audience will be looking to apply these tools as data scientists and analysts inside companies. But we also cover examples from health care and other domains, and the practical material that you learn in this book applies far beyond any narrow set of business problems.

This is not a book about *one of* machine learning, economics, or statistics. Rather, this book pulls from all of these fields to build a toolset for modern business analytics. The material in this book is designed to be useful for *decision making*. Detecting patterns in past data can be useful—we will cover a number of pattern recognition topics—but the necessary analysis for deeper business problems is about *why* things happen rather than what has happened. For this reason, this book will spend the time to move beyond correlation to causal analysis. This material is closer to economics than to the mainstream of data science, which should help you have a bigger practical impact through your work.

We can't cover everything here. This is not an encyclopedia of data analysis. Indeed, for continuing study, there are a number of excellent books covering different areas of contemporary machine learning and data science. For example, Hastie et al. (2009) is a comprehensive modern statistics reference and James et al. (2021) is a less advanced text from a similar viewpoint. You can view this current text as a stepping stone to a career of continued exploration and learning in statistics and machine learning. We want you to leave with a set of best practices that make you confident in what to trust, how to use it, and how to learn more.

GUIDED TOUR

This book is based on the *Business Data Science* text by Taddy (2019), which was itself developed as part of the MBA data science curriculum at the University of Chicago Booth School of Business. This new adaptation creates a more accessible and course-ready textbook, and includes a major expansion of the examples and content (plus an appendix tutorial on computing with R). Visit Connect for digital assignments, code, datasets, and additional resources.

Practical Data Science for Decision Making

Our target readership is anyone who wants to get the skills to use modern large-scale data to make decisions, whether they are working in business, government, science, or anywhere else.

In the past 10 years, we've observed the growth of a class of generalists who can understand business problems and also dive into the (big) data and run their own analyses. There is a massive demand for people with these capabilities, and this book is our attempt to help grow more of these sorts of people. You may be reading this book from a quantitative undergraduate course, as part of your MBA degree at a business school, or in a data science or other graduate program. Or, you may just be reading the book on your own accord. As data analysis has become more crucial and exciting, we are seeing a boom in people switching into data analysis careers from a wide variety of backgrounds. Those self-learners and career-switchers are as much our audience here as students in a classroom.

All of this said, this is not an *easy* book. We have tried to avoid explanations that require calculus or advanced linear algebra, but you will find the book a tough slog if you do not have a solid foundation in first-year mathematics and probability. Since the book includes a breadth of material that spans a range of complexity, we begin each chapter with a summary that outlines each section and indicates their difficulty according to a *ski-hill* scale:

- ● The easiest material, requiring familiarity with some transformations like logarithms and exponents, and an understanding of the basics of probability.
- ■ Moderately difficult material, involving more advanced ideas from probability and statistics or ideas that are going to be difficult to intuit without some linear algebra.
- ◆ The really tough stuff, involving more complex modeling ideas (and notation) and tools from linear algebra and optimization.

The black diamond material is not necessary for understanding future green or blue sections, and so instructors may wish to set their courses to cover the easy and moderately difficult sections while selecting topics from the hardest sections.

The book is designed to be self-contained, such that you can start with little prerequisite background in data science and learn as you go. However, the pace of content on introductory probability and statistics and regression is such that you may struggle if this is your first-ever course on these ideas. If you find this to be the case, we recommend spending some time working through a dedicated introductory statistics book to build some of this understanding before diving into the more advanced data science material.

It is also important to recognize that data science can be learned only by doing. This means writing the code to run analysis routines on really messy data. We will use the R scripting language for all of our examples. All example code and data is available online, and one of the most important skills you will get out of this book will be an advanced education in this powerful and widely used statistical software. For those who are completely new to R, we have also included an extensive R primer. The skills you learn here will also prepare you well for learning how to program in other languages, such as Python, which you will likely encounter in your business analysis career.

This is a book about how to *do* modern business analytics. We will lay out a set of core principles and best practices that come from statistics, machine learning, and economics. You will be working through many real data analysis examples as you learn by doing. It is a book designed to prepare scientists, engineers, and business professionals to use data science to improve their decisions.

An Introductory Example

Before diving into the core material, we will work through a simple finance example to illustrate the difference between data processing or description and a deeper *business analysis.* Consider the graph in Figure 0.1. This shows seven years of monthly returns for stocks in the S&P 500 index (a return is the difference between the current and previous price divided by the prior value). Each line ranging from bright yellow to dark red denotes an individual stock's return series. Their weighted average—the value of the S&P 500—is marked with a bold line. Returns on three-month U.S. treasury bills are in gray.

This is a fancy plot. It looks cool, with lots of different lines. It is the sort of plot that you might see on a computer screen in a TV ad for some online brokerage platform. *If only I had that information, I'd be rich!*

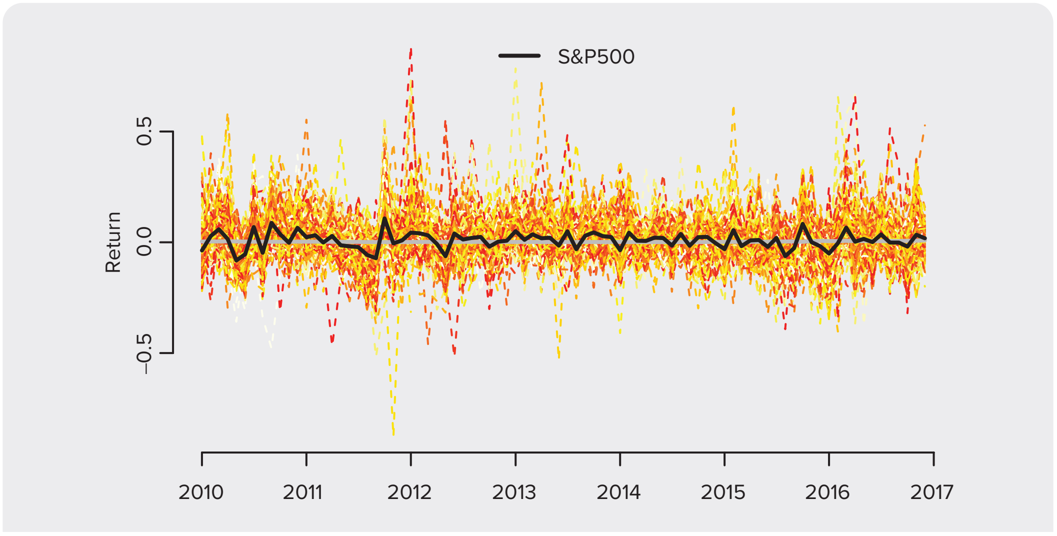


FIGURE 0.1 A fancy plot: monthly stock returns for members of the S&P 500 and their average (the bold line). *What can you learn?*

But what can you actually learn from Figure 0.1? You can see that returns do tend to bounce around near zero (although the long-term average is reliably much greater than zero). You can also pick out periods of higher volatility (variance) where the S&P 500 changes more from month to month and the individual stock returns around it are more dispersed. That's about it. You don't learn *why* these periods are more volatile or when they will occur in the future. More important, you can't pull out useful information about any individual stock. There is a ton of *data* on the graph but little useful information.

Instead of plotting raw data, let's consider a simple *market model* that relates individual stock returns to the market average. The capital asset pricing model (CAPM) regresses the returns of an individual asset onto a measure of overall market returns, as shown here:

$$r_{jt} = \alpha_j + \beta_j m_t + \varepsilon_{jt} \tag{0.1}$$

The *output* r_{jt} is equity j return at time t. The *input* m_t is a measure of the average return—the "market"—at time t. We take m_t as the return on the S&P 500 index that weights 500 large companies according to their market capitalization (the total value of their stock). Finally, ε_{jt} is an *error* that has mean zero and is uncorrelated with the market.

Equation (0.1) is the first regression model in this book. You'll see many more. This is a simple linear regression that should be familiar to most readers. The Greek letters define a line relating each individual equity return to the market, as shown in Figure 0.2. A small β_j, near zero, indicates an asset with low market sensitivity. In the extreme, fixed-income assets like treasury bills have $\beta_j = 0$. On the other hand, a $\beta_j > 1$ indicates a stock that is more volatile than the market, typically meaning growth and higher-risk stocks. The α_j is free money: assets with $\alpha_j > 0$ are adding value regardless of wider market movements, and those with $\alpha_j < 0$ destroy value.

Figure 0.3 represents each stock "ticker" in the two-dimensional space implied by the market model's fit on the seven years of data in Figure 0.1. The tickers are sized proportional to each firm's market capitalization. The two CAPM parameters—[α, β]—tell you a huge amount about the behavior and performance of individual assets. This picture immediately allows you to assess market sensitivity and arbitrage opportunities. For example, the big tech stocks of Facebook (FB), Amazon (AMZN), Apple (AAPL), Microsoft (MSFT), and Google (GOOGL) all have market sensitivity β values close to one. However, Facebook, Amazon, and Apple generated more money independent of the market over this time period compared to Microsoft and Google (which have nearly identical α values and are overlapped on the plot). Note

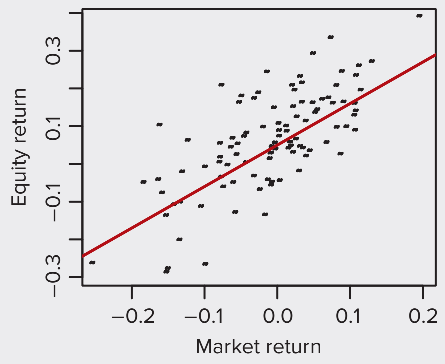


FIGURE 0.2 A *scatterplot* of a single stock's returns against market returns, with the fitted *regression line* for the model of Equation (0.1) shown in red.

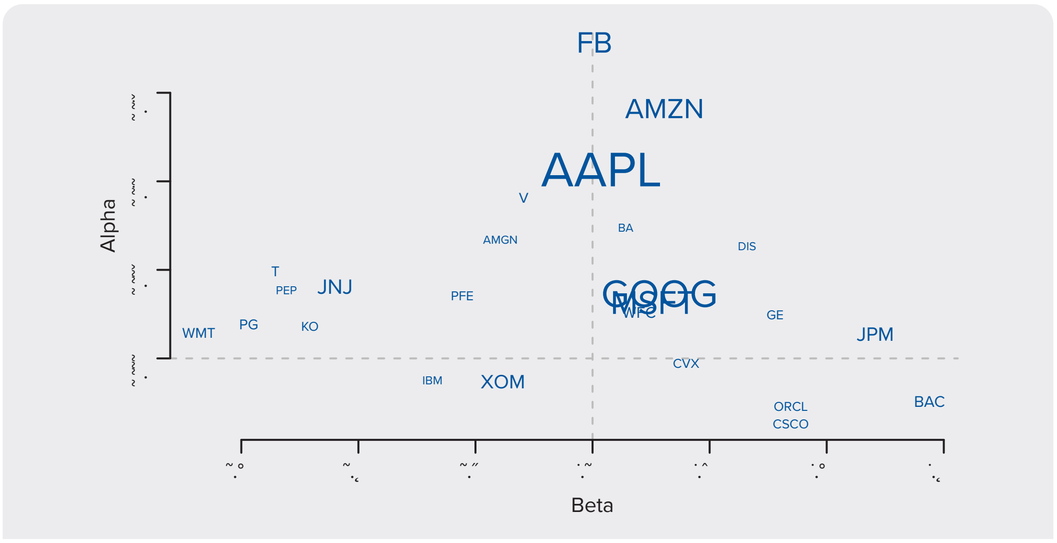


FIGURE 0.3 Stocks positioned according to their fitted market model, where α is money you make regardless of what the market does and β summarizes sensitivity to market movements. The tickers are sized proportional to market capitalization. Production change alpha to α and beta to β in the plot axis labels.

that Facebook's CAPM parameters are estimated from a shorter time period, since it did not have its IPO until May of 2012. Some of the older technology firms, such as Oracle (ORCL), Cisco (CSCO), and IBM, appear to have destroyed value over this period (negative alpha). Such information can be used to build portfolios that maximize mean returns and minimize variance in the face of uncertain future market conditions. It can also be used in strategies like pairs-trading where you find two stocks with similar betas and buy the higher alpha while "shorting" the other.

CAPM is an old tool in financial analysis, but it serves as a great illustration of what to strive toward in practical data science. An interpretable model translates raw data into information that is directly relevant to decision making. The challenge in data science is that the data you'll be working with will be larger and less structured (e.g., it will include text and image data). Moreover, CAPM is derived from assumptions of efficient market theory, and in many applications you won't have such a convenient simplifying framework on hand. But the basic principles remain the same: you want to turn raw data into useful information that has direct relevance to business policy.

Machine Learning

Machine learning (ML) is the field of using algorithms to automatically detect and predict patterns in complex data. The rise of machine learning is a major driver behind data science and a big part of what differentiates today's analyses from those of the past. ML is closely related to modern statistics, and indeed many of the best ideas in ML have come from statisticians. But whereas statisticians have often focused on *model inference*—on understanding the parameters of their models (e.g., testing on individual coefficients in a regression)—the ML community has historically been more focused on the single goal of maximizing *predictive performance* (i.e., predicting future values of some response of interest, like sales or prices).

A focus on prediction tasks has allowed ML to quickly push forward and work with larger and more complex data. If all you care about is predictive performance, then you don't need to worry about whether your model is "true" but rather just test how well it performs when predicting future values. This single-minded focus allows rapid experimentation on alternative models and estimation algorithms. The result is that ML has seen massive success, to the point that you can now expect to have available for almost any type of data an algorithm that will work out of the box to recognize patterns and give high-quality predictions.

However, this focus on prediction means that ML on its own is less useful for many *decision-making* tasks. ML algorithms learn to predict *a future that is mostly like the past.* Suppose that you build an ML algorithm that looks at how customer web browser history predicts how much they spend in your e-commerce store. A purely prediction-focused algorithm will discern what web traffic tends to spend more or less money. It will not tell you what will happen to the spending if you *change* a group of those websites (or your prices) or perhaps make it easier for people to browse the Web (e.g., by subsidizing broadband). That is where this book comes in: we will use tools from economics and statistics in combination with ML techniques to create a platform for using data to make decisions.

Some of the material in this book will be focused on pure ML tasks like prediction and pattern recognition. This is especially true in the earlier chapters on regression, classification, and regularization. However, in later chapters you will use these prediction tools as parts of more structured analyses, such as understanding subject-specific treatment effects, fitting consumer demand functions, or as part of an artificial intelligence system. This typically involves a mix of domain knowledge and analysis tools, which is what makes the data scientist such a powerful figure. The ML tools are useless for policy making without an understanding of the business problems, but a policy maker who can deploy ML as part of their analysis toolkit will be able to make better decisions faster.

Computing with `R`

You don't need to be a software engineer to work as a data scientist, but you need to be able to write and understand computer code. To learn from this book, you will need to be able to read and write in a high-level *scripting* language, in other words, flexible code that can be used to describe recipes for data analysis. In particular, you will need to have a familiarity with `R` (`r-project.org`).

The ability to interact with computers in this way—by typing commands rather than clicking buttons or choosing from a menu—is a basic data analysis skill. Having a script of commands allows you to rerun your analyses for new data without any additional work. It also allows you to make small changes to existing scripts to adapt them for new scenarios. Indeed, making small changes is how we recommend you work with the material in this book. The code for every in-text example is available on-line, and you can alter and extend these scripts to suit your data analysis needs. In the examples for this book, all of the analysis will be conducted in `R`. This is an open-source high-level language for data analysis. `R` is used widely throughout industry, government, and academia. Companies like RStudio sell enterprise products built around `R`. This is not a toy language used simply for teaching purposes—`R` is the real industrial-strength deal.

For the fundamentals of statistical analysis, `R` is tough to beat: all of the tools you need for linear modeling and uncertainty quantification are mainstays. `R` is also relatively forgiving for the novice programmer. A major strength of R is its ecosystem of contributed packages. These are add-ons that increase the capability of core R. For example, almost all of the ML tools that you will use in this book are available via packages. The quality of the packages is more varied than it is for R's core functionality, but if a package has high usage you should be confident that it works as intended.

The Appendix of this book contains a tutorial that is dedicated to getting you started in R. It focuses on the topics and algorithms that are used in the examples in this book. You don't need to be an expert in R to learn from this book; you just need to be able to understand the fundamentals and be willing to mess around with the coded examples. If you have no formal background in coding, worry not: many in the field started out in this position. The learning curve can be steep initially, but once you get the hang of it, the rest will come fast. The tutorial in the Appendix should help you get started. We also provide extensive examples throughout the book, and all code, data, and homework assignments are available through Connect. Every chapter ends with a *Quick Reference* section containing the basic R recipes from that chapter. When you are ready to learn more, there are many great places where you can supplement your understanding of the basics of R. If you simply search for *R* or *R statistics* books on-line, you will find a huge variety of learning resources.

ACKNOWLEDGMENTS

We are grateful for the reviewers who provided feedback on this first edition:

Sue Abdinnour, Wichita State University
Anil Aggarwal, University of Baltimore
Goutam Chakraborty, Oklahoma State University
Rick Cleary, Babson College
John Daniels, Central Michigan University
John Draper, Ohio State University
Janet Frasier, West Virginia University
Phillip C. Fry, Boise State University
Marina Girju, California Baptist University
Richard Hauser, East Carolina University
Kuo-Ting “Ken” Hung, Suffolk University
Aimee Jacobs, California State University, Fresno
Jaehwan Jeong, Radford University
Patrick Johanns, Univeristy of Iowa
Barry King, Butler University
Lauren Kleitz, Xavier University
Su “Sue” Kong, Kutztown University
Min Li, California State University, Sacramento
Jiajuan Liang, University of New Haven
Vic Matta, Ohio University
Ebrahim Mortaz, Pace University
Bob Myers, Georgia Tech University
Robert Nauss, University of Missouri-St. Louis
Arash Negahban, California State University, Chico
Yasin Ozcelik, Fairfield University
Brad Price, West Virginia University
Xingye Qiao, Binghamton University
Roman Rabinovich, Boston University
Bharatendra Rai, UMass–Dartmouth
Balaraman Rajan, California State University, East Bay
Rouzbeh Razavi, Kent State University
John Repede, Queens University of Charlotte
Thomas R. Robbins, East Carolina University
Wendy Swenson Roth, Georgia State University
Seokwoo Song, Weber State University
John R. Stevens, Utah State University
Jie Tao, Fairfield University
Vicar S. Valencia, Indiana University South Bend
Nikhil Varaiya, San Diego State University
Gang Wang, UMass - Dartmouth
K. Matthew Wong, St. John’s University
Chase Wu, New Jersey Institute of Technology
Yajiong “Lucky” Xue, East Carolina University

SUPPORT AT every step

Students: Get Learning that Fits You

Effective tools for efficient studying

Connect is designed to help you be more productive with simple, flexible, intuitive tools that maximize your study time and meet your individual learning needs. Get learning that works for you with Connect.

Study anytime, anywhere

Download the free ReadAnywhere app and access your online eBook, SmartBook 2.0, or Adaptive Learning Assignments when it's convenient, even if you're offline. And since the app automatically syncs with your Connect account, all of your work is available every time you open it. Find out more at **www.mheducation.com/readanywhere**

"I really liked this app—it made it easy to study when you don't have your text-book in front of you."

- Jordan Cunningham, Eastern Washington University

Calendar: owattaphotos/Getty Images

Everything you need in one place

Your Connect course has everything you need—whether reading on your digital eBook or completing assignments for class, Connect makes it easy to get your work done.

Learning for everyone

McGraw Hill works directly with Accessibility Services Departments and faculty to meet the learning needs of all students. Please contact your Accessibility Services Office and ask them to email accessibility@mheducation.com, or visit **www.mheducation.com/about/accessibility** for more information.

Top: Jenner Images/Getty Images, Left: Hero Images/Getty Images, Right: Hero Images/Getty Images

Proctorio
Remote Proctoring & Browser-Locking Capabilities

Remote proctoring and browser-locking capabilities, hosted by Proctorio within Connect, provide control of the assessment environment by enabling security options and verifying the identity of the student.

Seamlessly integrated within Connect, these services allow instructors to control students' assessment experience by restricting browser activity, recording students' activity, and verifying students are doing their own work.

Instant and detailed reporting gives instructors an at-a-glance view of potential academic integrity concerns, thereby avoiding personal bias and supporting evidence-based claims.

ReadAnywhere

Read or study when it's convenient for you with McGraw Hill's free ReadAnywhere app. Available for iOS or Android smartphones or tablets, ReadAnywhere gives users access to McGraw Hill tools including the eBook and SmartBook 2.0 or Adaptive Learning Assignments in Connect. Take notes, highlight, and complete assignments offline – all of your work will sync when you open the app with WiFi access. Log in with your McGraw Hill Connect username and password to start learning – anytime, anywhere!

OLC-Aligned Courses

Implementing High-Quality Online Instruction and Assessment through Preconfigured Courseware

In consultation with the Online Learning Consortium (OLC) and our certified Faculty Consultants, McGraw Hill has created pre-configured courseware using OLC's quality scorecard to align with best practices in online course delivery. This turnkey courseware contains a combination of formative assessments, summative assessments, homework, and application activities, and can easily be customized to meet an individual's needs and course outcomes. For more information, visit https://www.mheducation.com/highered/olc.

Tegrity: Lectures 24/7

Tegrity in Connect is a tool that makes class time available 24/7 by automatically capturing every lecture. With a simple one-click start-and-stop process, you capture all computer screens and corresponding audio in a format that is easy to search, frame by frame. Students can replay any part of any class with easy-to-use, browser-based viewing on a PC, Mac, iPod, or other mobile device.

Educators know that the more students can see, hear, and experience class resources, the better they learn. In fact, studies prove it. Tegrity's unique search feature helps students efficiently find what they need, when they need it, across an entire semester of class recordings. Help turn your students' study time into learning moments immediately supported by your lecture. With Tegrity, you also increase intent listening and class participation by easing students' concerns about note-taking. Using Tegrity in Connect will make it more likely you will see students' faces, not the tops of their heads.

Test Builder in Connect

Available within Connect, Test Builder is a cloud-based tool that enables instructors to format tests that can be printed, administered within a Learning Management System, or exported as a Word document of the test bank. Test Builder offers a modern, streamlined interface for easy content configuration that matches course needs, without requiring a download.

Test Builder allows you to:

- access all test bank content from a particular title.
- easily pinpoint the most relevant content through robust filtering options.
- manipulate the order of questions or scramble questions and/or answers.
- pin questions to a specific location within a test.
- determine your preferred treatment of algorithmic questions.
- choose the layout and spacing.
- add instructions and configure default settings.

Test Builder provides a secure interface for better protection of content and allows for just-in-time updates to flow directly into assessments.

Writing Assignment

Available within Connect and Connect Master, the Writing Assignment tool delivers a learning experience to help students improve their written communication skills and conceptual understanding. As an instructor you can assign, monitor, grade, and provide feedback on writing more efficiently and effectively.

Application-Based Activities in Connect

Application-Based Activities in Connect are highly interactive, assignable exercises that provide students a safe space to apply the concepts they have learned to real-world, course-specific problems. Each Application-Based Activity involves the application of multiple concepts, allowing students to synthesize information and use critical thinking skills to solve realistic scenarios.

Mc Graw Hill create® Your Book, Your Way

McGraw Hill's Content Collections Powered by Create® is a self-service website that enables instructors to create custom course materials—print and eBooks—by drawing upon

McGraw Hill's comprehensive, cross-disciplinary content. Choose what you want from our high-quality textbooks, articles, and cases. Combine it with your own content quickly and easily, and tap into other rights-secured, third-party content such as readings, cases, and articles. Content can be arranged in a way that makes the most sense for your course and you can include the course name and information as well. Choose the best format for your course: color print, black-and-white print, or eBook. The eBook can be included in your Connect course and is available on the free ReadAnywhere app for smartphone or tablet access as well. When you are finished customizing, you will receive a free digital copy to review in just minutes! Visit McGraw Hill Create®— www.mcgrawhillcreate.com — today and begin building!

1 REGRESSION

This chapter develops the framework and language of regression: building models that predict response outputs from feature inputs.

Section 1.1 Linear R egression: Specify, estimate, and predict from a linear regression model for a quantitative response y as a function of inputs $\mathbf{x}$. Use log transforms to model multiplicative relationships and *elasticities*, and use interactions to allow the effect of inputs to depend on each other.

Section 1.2 R esiduals: Calculate the residual errors for your regression fit, and understand the key fit statistics *deviance*, R^2, and *degrees of freedom.*

Section 1.3 L ogistic Regression: Build logistic regression models for a binary response variable, and understand how logistic regression is related to linear regression as a *generalized linear model.* Translate the concepts of deviance, likelihood, and R^2 to logistic regression, and be able to interpret logistic regression coefficients as effects on the log odds that $y = 1$.

Section 1.4 Lik elihood and Deviance: Relate likelihood maximization and deviance minimization, use the generalized linear models to determine residual deviance, and use the `predict` function to integrate new data with the same variable names as the data used to fit your regression.

Section 1.5 Time Series: Adapt your regression models to allow for dependencies in data that has been observed over time, and understand time series concepts including seasonal trends, autoregression, and panel data.

Section 1.6 Spatial Data: Add spatial fixed effects to your regression models and use Gaussian process models to estimate spatial dependence in your observations.

The vast majority of problems in applied data science require regression modeling. You have a *response* variable (y) that you want to model or predict as a function of a vector of *input features*, or covariates ($\mathbf{x}$). This chapter introduces the basic framework and language of regression. We will build on this material throughout the rest of the book.

Regression is all about understanding the *conditional* probability distribution for "y given $\mathbf{x}$," which we write as $p(y|\mathbf{x})$. Figure 1.1 illustrates the conditional distribution in contrast to a *marginal* distribution, which is so named because it corresponds to the unconditional distribution for a single margin (i.e., column) of a data matrix.

A variable that has a probability distribution (e.g., number of bathrooms in Figure 1.1) is called a *random variable*. The *mean* for a random variable is the average of random draws from its probability distribution. While the marginal mean is a simple number, the conditional mean is a function. For example, from Figure 1.1b, you can see that the average home selling price takes different values indexed by the number of bathrooms. The data is distributed randomly around these means, and the way that you model these distributions drives your estimation and prediction strategies.

Conditional Expectation

A basic but powerful regression strategy is to build models in terms of *averages* and *lines*. That is, we will model the conditional mean for our output variable as a linear function of inputs. Other regression strategies can sometimes be useful, such as *quantile regression* that models percentiles of the conditional distribution. However for the bulk of applications you will find that *mean* regression is a good approach.

There is some important notation that you need to familiarize yourself with for the rest of the book. We model the conditional mean for y given $\mathbf{x}$ as

$$\mathbb{E}[y\,|\,\mathbf{x}] = f(\mathbf{x}'\boldsymbol{\beta}) \qquad \textbf{(1.1)}$$

where

- $\mathbb{E}[\cdot]$ denotes the taking of the expectation or average of whatever random variable is inside the brackets. It is an extremely important operation, and we will use this notation to define many of our statistical models.

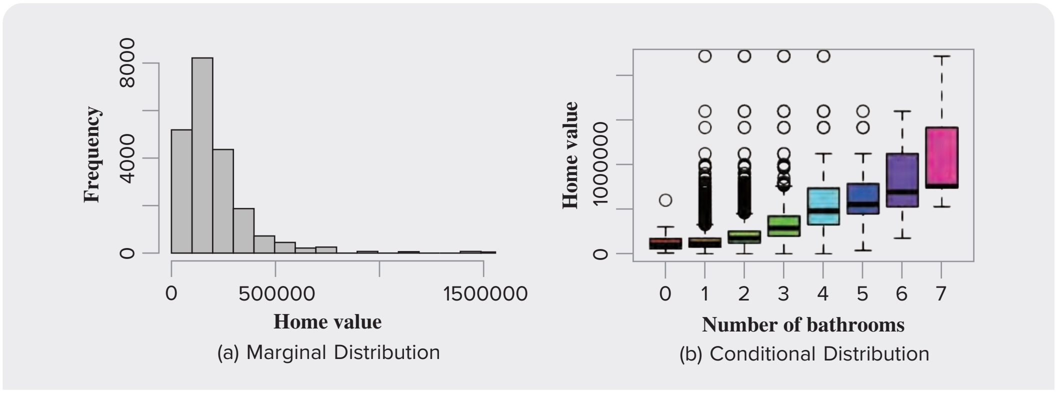


FIGURE 1.1 Illustration of marginal versus conditional distributions for home prices. On the left, we have the marginal distribution for all of the home prices. On the right, home price distributions are conditional on the number of bathrooms.

- The vertical bar | means "given" or "conditional upon," so that $\mathbb{E}[y|\mathbf{x}]$ is read as "the average for y given inputs $\mathbf{x}$."
- $f(\cdot)$ is a "link" function that transforms from the linear model to your response.
- $\mathbf{x} = [1, x_1, x_2, \ldots x_p]$ is the vector of covariates and $\boldsymbol{\beta} = [\beta_0, \beta_1, \beta_2, \ldots \beta_p]$ are the corresponding coefficients.

The *vector* notation, $\mathbf{x}'\boldsymbol{\beta}$, is shorthand for the sum of elementwise products:

$$\mathbf{x}'\boldsymbol{\beta} = [1 x_1\ x_2 \cdots x_p] \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix} = \beta_0 + x_1\beta_1 + x_2\beta_2 + \ldots + x_p\beta_p \tag{1.2}$$

This shorthand notation will be used throughout the book. Here we have used the convention that $x_0 = 1$, such that β_0 is the intercept.

The link function, $f(\cdot)$, defines the relationship between your linear function $\mathbf{x}'\boldsymbol{\beta}$ and the response. The link function gives you a huge amount of modeling flexibility. This is why models of the kind written in Equation (1.1) are called *generalized linear models* (GLMs). They allow you to make use of linear modeling strategies after some simple transformations of your output variable of interest. In this chapter we will outline the two most common GLMs: linear regression and logistic regression. These two models will serve you well for the large majority of analysis problems, and through them you will become familiar with the general principles of GLM analysis.

1.1 Linear Regression

Linear regression is the workhorse of analytics. It is fast to fit (in terms of both analyst and computational time), it gives reasonable answers in a variety of settings (so long as you know how to ask the right questions), and it is easy to interpret and understand. The model is as follows:

$$\mathbb{E}[y|\mathbf{x}] = \beta_0 + x_1\beta_1 + x_2\beta_2 + \ldots + x_p\beta_p \tag{1.3}$$

This corresponds to using the link function $f(z) = z$ in Equation (1.1).

With just one input x, you can write the model as $\mathbb{E}[y|x] = \beta_0 + x\beta_1$ and plot it as in Figure 1.2. β_0 is the intercept. This is where the line crosses the y axis when x is 0. β_1 is the

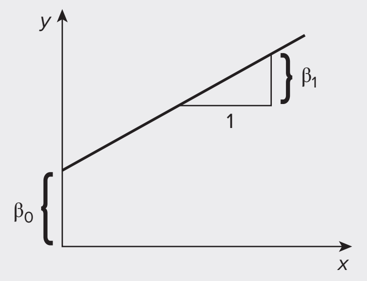


FIGURE 1.2 Simple linear regression with a positive slope β_1. The plotted line corresponds to $\mathbb{E}[y|x]$.

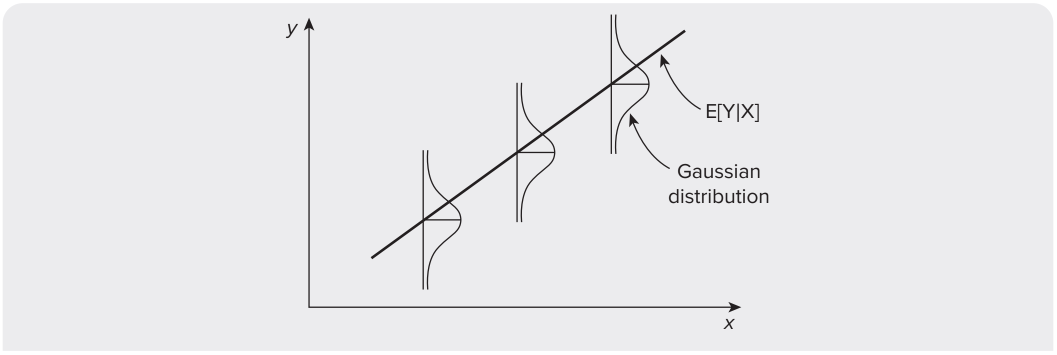


FIGURE 1.3 Using simple linear regression to picture the Gaussian conditional distribution for $y|x$. Here $\mathbb{E}[y|x]$ are the values on the line and the variation parallel to the y axis (i.e., within each narrow vertical strip) is assumed to be described by a Gaussian distribution.

slope and describes how $\mathbb{E}[y|x]$ changes as x changes. If x increases by 1 unit, $\mathbb{E}[y|x]$ changes by β_1. For a two predictor model, we are fitting a plane. Higher dimensions are more difficult to imagine, but the basic intuition is the same.

When fitting a regression model—i.e., when estimating the $\boldsymbol{\beta}$ coefficients—you make some assumptions about the conditional distribution beyond its mean at $\mathbb{E}[y|\mathbf{x}]$. Linear regression is commonly fit for Gaussian (normal) conditional distributions. We write this conditional distribution as

$$y \mid \mathbf{x} \sim \mathrm{N}(\mathbf{x}'\boldsymbol{\beta}, \sigma^2) \tag{1.4}$$

This says that the distribution for y as a function of $\mathbf{x}$ is normally distributed around $\mathbb{E}[y|\mathbf{x}] = \mathbf{x}'\boldsymbol{\beta}$ with variance σ^2. The same model is often written with an additive error term:

$$y = \mathbf{x}'\boldsymbol{\beta} + \varepsilon,\ \varepsilon \sim \mathrm{N}(0, \sigma^2) \tag{1.5}$$

where ε are the "independent" or "idiosyncratic" errors. These errors contain the variations in y that are not correlated with $\mathbf{x}$. Equations (1.4) and (1.5) describe the same model. Figure 1.3 illustrates this model for single-input simple linear regression. The line is the average $\mathbb{E}[y|x]$ and vertical variation around the line is what is assumed to have a normal distribution.

You will often need to transform your data to make the linear model of Equation (1.5) realistic. One common transform is that you need to take a *logarithm* of the response, say, "r," such that your model becomes

$$\log(r) = \mathbf{x}'\boldsymbol{\beta} + \varepsilon,\ \varepsilon \sim \mathrm{N}(0, \sigma^2) \tag{1.6}$$

Of course this is the same as the model in Equation (1.5), but we have just made the replacement $y = \log(r)$. You will likely also consider transformations for the input variables, such that elements of $\mathbf{x}$ include logarithmic and other functional transformations. This is often referred to as *feature engineering*.

Example 1.1 Orange Juice Sales: Exploring Variables and the Need for a Log-Log Model As a concrete example, consider sales data for orange juice (OJ) from Dominick's grocery stores. Dominick's was a Chicago-area chain. This data was collected in the 1990s and is publicly available from the Kilts Center at the University of Chicago's Booth School of Business. The data includes weekly prices and unit sales (number of cartons sold) for three OJ brands—Tropicana, Minute Maid, Dominick's—at 83 stores in the Chicago area, as well as an indicator, `ad`, showing whether each brand was advertised (in store or flyer) that week.

```
> oj <- read.csv("oj.csv",strings=T)
> head(oj)
  sales price     brand  ad
1  8256  3.87 tropicana   0
2  6144  3.87 tropicana   0
3  3840  3.87 tropicana   0
4  8000  3.87 tropicana   0
5  8896  3.87 tropicana   0
6  7168  3.87 tropicana   0
> levels(oj$brand)
[1] "dominicks"  "minute.maid" "tropicana"
```

Notice the argument `strings=T` in `read.csv` as shorthand for "`stringsAsFactors = TRUE`." This converts our `brand` column into a factor variable. This was the default behavior of `read.csv` prior to version 4.0.0 of `R`, but you now need to specify it explicitly. Otherwise you will get an error when you try to make the plots or fit the regression models below.

The code-printout above is our first example showing `R` code and output. We will include a ton of code and output snippets like this throughout the book: they are an integral part of the material. If this output looks unfamiliar to you, you should break here and take the time to work through the `R`-primer in the appendix.

Figure 1.4 shows the prices and sales broken out by brand. You can see in Figure 1.4a that each brand occupies a different price range: Dominick's is the budget option, Tropicana is the luxury option, and Minute Maid lives between. In Figure 1.4c, sales are clearly decreasing with

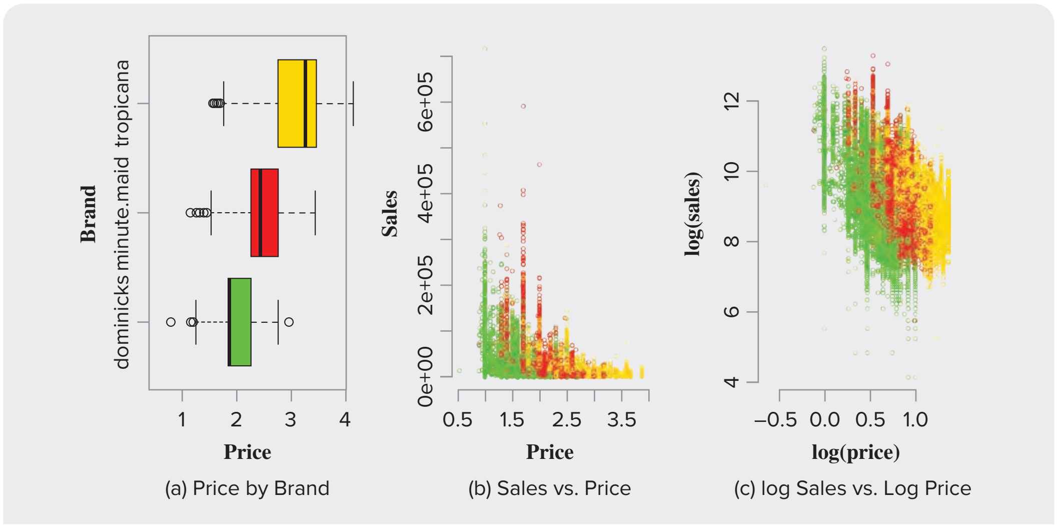


FIGURE 1.4 Dominick's OJ prices by brand and monthly sales, both raw and after a log transformation.

price. This makes sense: demand is *downward* sloping, and if you charge more, you sell less. More specifically, it appears that *log* sales has a roughly linear relationship with *log* price. This is an important point. Whenever you are working with linear (i.e., additive) models, it is crucial that you try to work in the space where you expect to find linearity. For variables that change *multiplicatively* with other factors, this is usually the log scale (see the nearby box for a quick review on logarithms). For comparison, the raw (without log) values in Figure 1.4b show a nonlinear relationship between prices and sales.

Log-Log Models and Elasticities

Another common scenario models against each other two variables that *both* move multiplicatively. For example, Figure 1.5 shows the national gross domestic product (GDP) versus imports for several countries. Fitting a line to the left panel would be silly; its slope will be entirely determined by small changes in the U.S. values. In contrast, the right panel shows that GDP and imports follow a neat linear relationship in log space.

Returning to our OJ example, Figure 1.4c indicates that this *log-log* model might be appropriate for the orange juice sales versus price analysis. One possible regression model is

$$\log(\texttt{sales}) = \beta_0 + \beta_1 \log(\texttt{price}) + \varepsilon \tag{1.7}$$

Here, `log(sales)` increase by β_1 for every unit increase in `log(price)`. Conveniently, log-log models have a much more intuitive interpretation: sales increase by $\beta_1\%$ for every 1% increase in price. To see this, you need a bit of calculus. Write $y = \exp[\beta_0 + \beta_1 \log(x) + \varepsilon]$ and differentiate with respect to x:

$$\frac{dy}{dx} = \frac{\beta_1}{x} e^{\beta_0+\beta_1 \log(x)+\varepsilon} = \frac{\beta_1}{x} y \quad \Rightarrow \quad \beta_1 = \frac{dy/y}{dx/x} \tag{1.8}$$

This shows that β_1 is the proportional change in y over the proportional change in x. In economics there is a special name for such an expression: *elasticity*. The concept of elasticity will play an important role in many of our analyses.

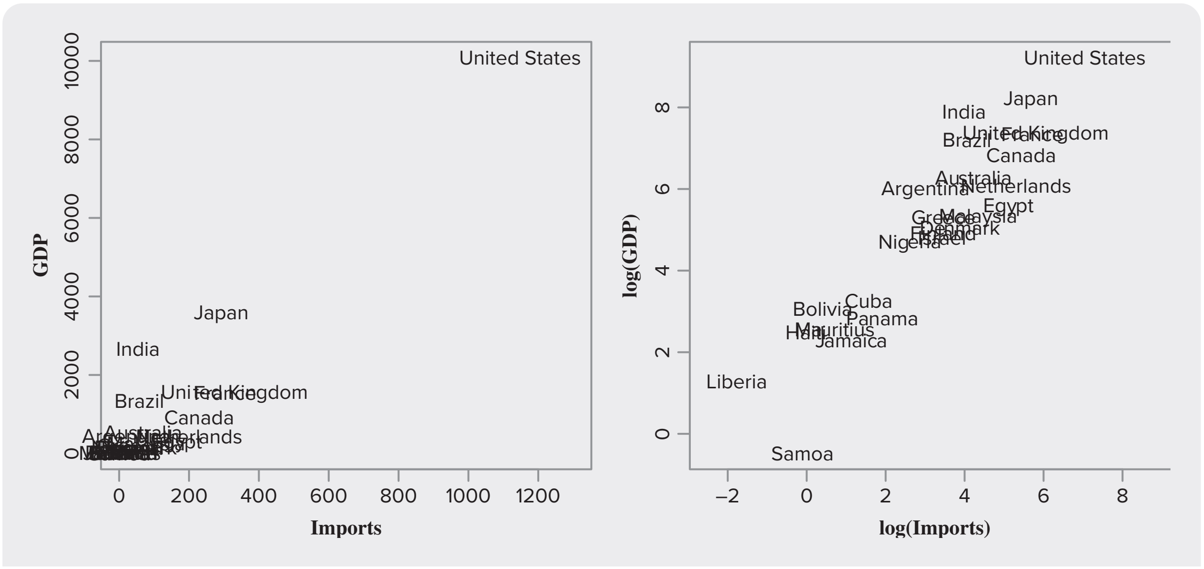


FIGURE 1.5 National GDP against imports, in original and log scale.

Logarithms and Exponents

Recall the logarithm definition:

$$\log(a) = b \Leftrightarrow a = e^b \qquad \textbf{(1.9)}$$

Here, $e \approx 2.72$ is "Euler's number" and we refer to e^b as "e to the power of b" or, simply, 'b exponentiated.' We will sometimes write $\exp[b]$ instead of e^b; they mean the same thing. There are other types of logarithms (sometimes base 10 is used in introductory classes), but we will always use the *natural log* defined in Equation (1.9). The base e plays a central role in science and modeling of systems because e^x is its own derivative: $de^x/dx = e^x$ for those readers who know their calculus.

In a simple linear regression for $\log(y)$ on x, β_1 is added to the expected value for $\log(y)$ for each unit increase in x:

$$\log(y) = \beta_0 + \beta_1 x + \epsilon. \qquad \textbf{(1.10)}$$

The fact that we are taking the log of y makes this model *multiplicative*. Recall some basic facts about logs and exponents: $\log(ab) = \log(a) + \log(b)$, $\log(a^b) = b\log(a)$, and $e^{a+b} = e^a e^b$. Thus, exponentiating both sides of Equation (1.10) yields

$$y = e^{\beta_0+\beta_1 x+\epsilon} = e^{\beta_0} e^{\beta_1 x} e^{\epsilon} \qquad \textbf{(1.11)}$$

Considering $x^* = x + 1$, you get that

$$y^* = e^{\beta_0+\epsilon} e^{\beta_1 x^*} = e^{\beta_0+\epsilon} e^{\beta_1(x+1)} = e^{\beta_0+\beta_1 x+\epsilon} e^{\beta_1} = ye^{\beta_1} \qquad \textbf{(1.12)}$$

Therefore, each unit increase in x leads $\mathbb{E}[y|x]$ to be *multiplied* by the factor e^{β_1}.

Example 1.2 Orange Juice Sales: Linear Regression Now that we have established what a *log-log* model will do for us, let's add a bit of complexity to the model from (1.7) to make it more realistic. If you take a look at Figure 1.4c, it appears that the three brands have log-log sales-price relationships that are concentrated around three separate lines. If you suspect that each brand has the same β_1 elasticity but a different intercept (i.e., if all brands have sales that move with price the same way but at the same price some brands will sell more than others), then you would use a slightly more complex model that incorporates both `brand` and `price`:

$$\log(\texttt{sales}) = \alpha_{\texttt{brand}} + \beta \log(\texttt{price}) + \varepsilon \qquad \textbf{(1.13)}$$

Here, $\alpha_{\texttt{brand}}$ is shorthand for a separate intercept for each OJ brand, which we could write out more fully as

$$\alpha_{\texttt{brand}} = \alpha_0 1_{[\texttt{dominicks}]} + \alpha_1 1_{[\texttt{minute.maid}]} + \alpha_2 1_{[\texttt{tropicana}]}. \qquad \textbf{(1.14)}$$

The indicator functions, $1_{[v]}$, are one if v is the true factor level and zero otherwise. Hence, Equation (1.13) says that, even though their sales all have the same elasticity to price, the brands can have different sales at the same price due to brand-specific intercepts.

Fitting Regressions with `glm`

To fit this regression in `R` you will use the `glm` function, which is used to estimate the class of generalized linear models that we introduced in Equation (1.1). There is also a `lm` function that fits only linear regression models, so you could use that here also (it takes the same arguments), but we will get in the habit of using `glm` since it works for many different GLMs. The function is straightforward to use: you give it a data frame with the `data` argument and provide a `formula` that defines your regression.

```
> fit <- glm( y ~ var1 + ... + varP, data=mydata)
```

The fitted object `fit` is a list of useful things (type `names(fit)` to see them), and there are functions to access the results. For example,

- `summary(fit)` prints the model, information about residual errors, the estimated coefficients and uncertainty about these estimates (we will cover the uncertainty in detail in the next chapter), and statistics related to model fit.
- `coef(fit)` supplies just the coefficient estimates.
- `predict(fit, newdata=mynewdata)` predicts y where `mynewdata` is a data frame with the same variables as `mydata`.

The formula syntax in the `glm` call is important. The ~ symbol is read as "regressed onto" or "as a function of." The variable you want to predict, the y response variable, comes before the ~, and the input features, $\mathbf{x}$, come after. This model formula notation will be used throughout the remainder of the book, and we note some common specifications in Table 1.1.

The `R` formula for (1.13) is `log(sales) ~ brand + log(price)`. You can fit this with `glm` using the `oj` data, and then use the `coef` function to view the fitted coefficients. (More on this in Section 1.4.)

```
> fit<-glm( log(sales) ~ brand + log(price), data=oj)
> coef(fit) # fitted coefficients
     (Intercept) brandminute.maid   brandtropicana   log(price)
      10.8288216        0.8701747        1.5299428   -3.1386914
```

There are a few things to notice here. First, you can see that $\hat{\beta} = -3.1$ for the estimated coefficient on log price. Throughout this book we use the convention that $\hat{\theta}$ denotes the estimated value for some parameter θ. So $\hat{\beta}$ is the estimated "sales-price elasticity," and it says that expected sales drop by about 3% for every 1% price increase. Second, notice that there are distinct model coefficients for Minute Maid and Tropicana but not for Dominick's. This is due

`y ~ x1`	model by x_1
`y ~ .`	include all other columns
`y ~ .-x3`	include all except x_3
`y ~ .-1`	include all, but no intercept
`y ~ 1`	intercept only
`y ~ x1*x2`	include interaction for x_1 and x_2 and lower order terms
`y ~ x1:x2`	include interaction only
`y ~ .^2`	all possible 2 way interactions and lower order terms

TABLE 1.1 Some common syntax for use in formulas.

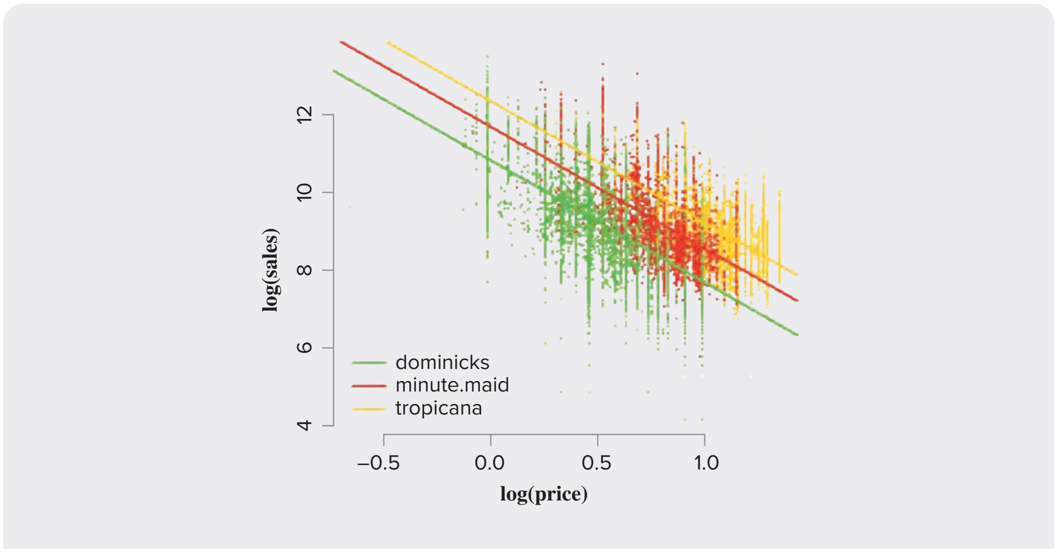


FIGURE 1.6 OJ data and the fitted regression lines (i.e., conditional expectations) for our model from (1.13) that regresses `log(sales)` on `log(price)` and `brand`.

to the way that `R` creates a numeric representation of the factor variables. It treats one of the factor levels as a 'reference level' that is subsumed into the intercept. For details, see the box on model matrices (i.e. design matrices).

The fitted values from the regression in Equation (1.13) are shown in Figure 1.6 alongside the original data. You see three lines shifted according to brand identity. *At the same price,* Tropicana sells more than Minute Maid, which in turn sells more than Dominick's. This makes sense: Tropicana is a luxury product that is preferable at the same price.

Model (Design) Matrices in R

When you regress onto a factor variable, `glm` converts the factor levels into a specific numeric representation. Take a look at rows 100, 200, and 300 from the `oj` data and notice that the `brand` column contains brand names, not numbers.

```
> oj[c(100,200,300),]
    sales price       brand  ad
100  4416  3.19   tropicana   0
200  5440  2.79 minute.maid   0
300 51264  1.39   dominicks   1
```

The first step of `glm` is to create a *model matrix* (also called a *design matrix*) that defines the numeric inputs **x**. It does this with a call to the `model.matrix` function, and you can pull that step out to see what happens.

```
> x <- model.matrix( ~ log(price) + brand, data=oj)
> x[c(100,200,300),]
    (Intercept)  log(price) brandminute.maid brandtropicana
100           1   1.1600209                0              1
200           1   1.0260416                1              0
300           1   0.3293037                0              0
```

The `model.matrix` function has expanded these brand factor levels into a couple of binary, or "dummy," variables that are one when the observation is from that brand and zero otherwise. For example, `brandtropicana` is 1 for the Tropicana observation in row 100 and zero otherwise. There is no `branddominicks` indicator because you need only two variables to represent three categories: when both `brandminute.maid` and `brandtropicana` are zero, the *intercept* gives the value for Dominick's expected log sales at a log price of zero. Each factor's reference level is absorbed by the intercept and the other coefficients represent "change relative to reference" (here, Dominick's). To check the reference level of your factors, type `levels(myfactor)`. The first level is the reference and by default this will be the first in the alphabetized list of levels. To change this, you can do `myfactor = relevel(myfactor, "myref")`.

1.1.1 Interactions

All of the lines in Figure 1.6 have the same slope. In economic terms, the model assumes that consumers of the three brands have the same price sensitivity. This seems unrealistic: money is probably less of an issue for Tropicana customers than it is for the average Dominick's consumer. You can build this information into your regression by having log price *interact* with brand.

An interaction term is the product of two inputs. Including an interaction between, say, x_j and x_k inputs, implies that your linear regression equation includes the product x_jx_k as an input.

$$\mathbb{E}[y|\mathbf{x}] = \beta_0 + \ldots + \beta_k x_k + \beta_j x_j + x_j x_k \beta_{jk} \tag{1.15}$$

Here, "..." just denotes whatever else is in your multiple linear regression model. Equation (1.15) says that the effect on the expected value for y due to a unit increase in x_j is $\beta_j + x_k\beta_{jk}$, such that it depends upon x_k.

Interactions are central to scientific and business questions. For example,

- How does drug effectiveness change with patient age?
- Does gender change the effect of education on wages?
- How does consumer price sensitivity change across brands?

In each case here, you want to know whether one variable changes the effect of another. You don't want to know the average price sensitivity of customers; you want to know whether they are more price sensitive for one product versus another.

Example 1.3 **OJ Sales: Interaction** In the OJ sales regression, to get brand-specific price elasticity terms you need to include an interaction between each of the brand indicator terms and the log price. We can write this as a model with a separate intercept and slope for each brand:

$$\log(\texttt{sales}) = \alpha_{\texttt{brand}} + \beta_{\texttt{brand}} \log(\texttt{price}) + \varepsilon \tag{1.16}$$

We can also expand this notation out to write the exact model that `glm` will be estimating:

$$\begin{aligned}\log(\texttt{sales}) = \; & \alpha_0 + \alpha_1 \mathbb{1}_{[\texttt{minute.maid}]} + \alpha_2 \mathbb{1}_{[\texttt{tropicana}]} + \\ & (\beta_0 + \beta_1 \mathbb{1}_{[\texttt{minute.maid}]} + \beta_2 \mathbb{1}_{[\texttt{tropicana}]}) \log(\texttt{price}) + \varepsilon\end{aligned} \tag{1.17}$$

As before, `dominicks` is the reference level for `brand` and so it is absorbed into both the intercept and baseline slope on log price. For an observation from Dominick's, the indicator functions are all zero so that $\mathbb{E}[\log(\texttt{sales})] = \alpha_0 + \beta_0 \log(\texttt{price})$.

You can fit this model in `glm` with the `*` symbol, which is syntax for "interacted with." Note that `*` also adds the *main effects*—all of the terms from our earlier model in Equation (1.13).

```
> fit2way <- glm(log(sales) ~ log(price)*brand, data=oj)
> coef(fit2way)
                (Intercept)                 log(price)
                10.95468173                -3.37752963
           brandminute.maid             brandtropicana
                 0.88825363                 0.96238960
log(price):brandminute.maid   log(price):brandtropicana
                 0.05679476                 0.66576088
```

The fitted regression is pictured in Figure 1.7.

In the `glm` output, the `log(price):brand` coefficients are the interaction terms. Plugging in 0 for both Tropicana and Minute Maid indicators yields the equation of the line for Dominick's:

$$\mathbb{E}[\log(\texttt{sales})] = 10.95 - 3.38 \log(\texttt{price})$$

Plugging in one for Minute Maid terms and zero for Tropicana terms yields the equation for Minute Maid:

$$\begin{aligned}\mathbb{E}[\log(\texttt{sales})] &= 10.95 - 3.38 \log(\texttt{price}) + 0.89 + 0.06 \log(\texttt{price}) \\ &= 11.84 - 3.32 \log(\texttt{price})\end{aligned}$$

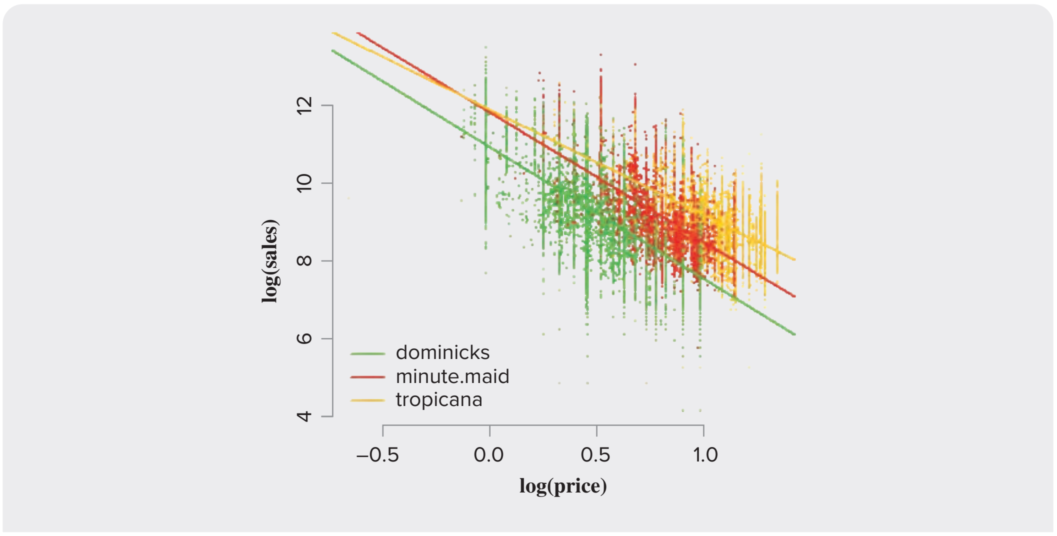


FIGURE 1.7 Fit for the model where we allow interaction between price and brand. Note that if you extrapolate too far, the linearity assumption implies Tropicana selling less than Minute Maid at the same price. This is a reminder that linear models are approximations and should be used with care away from the center of the observed data.

And plugging in one for Tropicana and zero for Minute Maid yields the regression line for Tropicana:

$$\begin{aligned}\mathbb{E}[\log(\texttt{sales})] &= 10.95 - 3.38\log(\texttt{price}) + 0.96 + 0.67\log(\texttt{price})\\ &= 11.91 - 2.71\log(\texttt{price})\end{aligned}$$

We see that Tropicana customers are indeed less sensitive than the others: they have a sales-price elasticity of −2.7 versus around −3.3 for both Dominick's and Minute Maid. This means, for example, that the store should expect a smaller sales increase for price cuts or coupons on Tropicana relative to use of the same promotion on the other brands. The price sensitivity that we estimated for model Equation (1.13), −3.1, was the result of averaging across the three distinct brand elasticities.

Advertising and Price Elasticity

We conclude this introduction to linear regression—and the study of orange juice—with a look at the role of advertising in the relationship between sales and prices. Recall that the OJ data includes an `ad` dummy variable, indicating that a given brand was promoted with either an in-store display or a flier ad during the week that sales and prices were recorded. The ads can increase sales at all prices, they can change price sensitivity, and they can do both of these things in a brand-specific manner. To model this, we specify a three-way interaction between price, brand, and `ad`:

$$\log(\texttt{sales}) = \alpha_{\texttt{brand, ad}} + \beta_{\texttt{brand, ad}}\log(\texttt{price}) + \varepsilon \qquad \textbf{(1.18)}$$

By subsetting on `brand, ad` we are indicating that there are different intercepts and slopes for each combination of the two factors. To fit this model with `glm`, you interact `brand`, `ad`, and `log(price)` with each other in the formula. Again, `glm` automatically includes the lower-level interactions and main effects—all of the terms from our model in (1.18)—in addition to the new three-way interactions.

```
> fit3way <- glm(log(sales) ~ log(price)*brand*ad, data=oj)
> coef(fit3way)
                    (Intercept)                      log(price)
                    10.40657579                     -2.77415436
                brandminute.maid                  brandtropicana
                     0.04720317                      0.70794089
                             ad     log(price):brandminute.maid
                     1.09440665                      0.78293210
       log(price):brandtropicana                   log(price):ad
                     0.73579299                     -0.47055331
             brandminute.maid:ad               brandtropicana:ad
                     1.17294361                      0.78525237
    log(price):brandminute.maid:ad    log(price):brandtropicana:ad
                    -1.10922376                     -0.98614093
```

The brand and ad specific elasticities are compiled in Table 1.2. We see that being featured always leads to more price sensitivity. Minute Maid and Tropicana elasticities drop from −2 to below −3.5 with ads, while Dominick's drops from −2.8 to −3.2. Why does this happen? One possible explanation is that advertisement increases the population of consumers who are considering your brand. In particular, it can increase your market beyond brand loyalists, to people who will be more price sensitive than those who reflexively buy your orange juice every week. Indeed, if you observe increased price sensitivity, it can be an indicator that your marketing efforts are expanding your consumer base. This is why Marketing 101 dictates that ad campaigns should usually be accompanied by price cuts. There is also an alternative explanation. Since the featured products are often also discounted, it could be that at lower price points the average consumer is more price sensitive (i.e., that the price elasticity is also a function of price). The truth is probably a combination of these effects.

Finally, notice that in our two-way interaction model (without including `ad`) Minute Maid's elasticity of −3.3 was roughly the same as Dominick's—it behaved like a budget product where its consumers are focused on value. However, in Table 1.2, you can see that Minute Maid and Tropicana have nearly identical elasticities and that both are different from Dominick's. Minute Maid is looking more similar now to the other national brand product. What happened?

	Dominick's	Minute Maid	Tropicana
Not featured	−2.8	−2.0	−2.0
Featured	−3.2	−3.6	−3.5

TABLE 1.2 Brand and ad dependent elasticities. Test that you can recover these numbers from the R output.

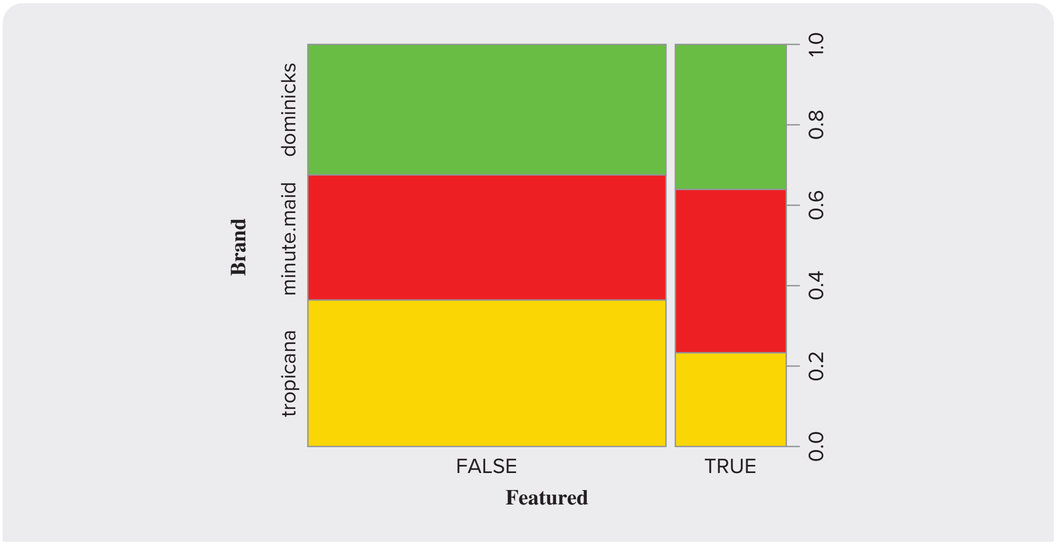


FIGURE 1.8 A mosaic plot of the amount of advertisement by brand. In a mosaic plot, the size of the boxes is proportional to the amount of data contained in that category. For example, the plot indicates that most sales are not accompanied by advertising (the featured=FALSE column is wider than for featured=TRUE) and that Minute Maid is featured (i.e., `ad=1`) more often than Tropicana.

The answer is that the simpler model in Equation (1.16) led to a *confounding* between advertisement and brand effects. Figure 1.8 shows that Minute Maid was featured more often than Tropicana. Since being featured leads to more price sensitivity, this made Minute Maid artificially appear more price sensitive when you don't account for the ad's effect. The model in Equation (1.18) corrects this by including `ad` in the regression. This phenomenon, where variable effects can get confounded if you don't *control* for them correctly (i.e., include those effects in your regression model), will play an important role in our later discussions of causal inference.

1.1.2 Prediction with `glm`

Once you have decided on a fitted model, using it for prediction is easy in `R`. The `predict` function takes the model you want to use for prediction and a data frame containing the new data you want to predict with.

Example 1.4 **Orange Juice Sales: Predicting Sales** We can use our fitted model, `fit3way`, to make predictions of sales of Orange Juice. Suppose you want to predict sales for all three brands when orange juice is featured at a price of $2.00 per carton. The first step is to create a data frame containing the observations to predict from. Be sure to specify a value for each predictor in the model.

```
> newdata <- data.frame(price=rep(2,3),
+     brand=factor(c("tropicana","minute.maid","dominicks"),
+             levels=levels(oj$brand)),
+     ad=rep(1,3))
> newdata #our data frame of 3 new observations
  price       brand    ad
1     2   tropicana     1
2     2 minute.maid     1
3     2   dominicks     1
```

Once you have your data frame specifying values for each variable in the model, simply feed it into the `predict` function and specify the model `glm` should use for prediction.

```
> predict(fit3way, newdata=newdata)
        1         2         3
 10.571588 10.245901  9.251922
```

Of course, there is uncertainty about these predictions (and about all of our coefficient estimates above). For now we are just fitting lines, but in Chapter 2 we will detail how you go about uncertainty quantification.

Note that you can exponentiate these values to translate from log sales to raw sales:

```
> exp(predict(fit3way, newdata=newdata))
       1        2        3
39010.56 28166.85 10424.59
```

However, these raw sales predictions are actually *biased* estimates of the raw expected sales. Due to the nonlinearity introduced through log transformation, exponentiating the expected log sales will give you a different answer than the expected raw sales (the exponentiated expected log sales will tend to be lower than the expected sales). We will discuss this bias further in the next chapter and introduce a technique for bias correction.

1.2 Residuals

When we fit a linear regression model, we have estimated an expected value for each observation in our dataset. These are often called the *fitted values,* and they are written as $\hat{y}_i = \mathbf{x}_i'\hat{\boldsymbol{\beta}}$ using our usual convention of putting hats on estimated values. Unless you have data with zero noise or as many input variables as observations (in either case you have no business fitting a regression with `glm`), then the fitted values will not be equal to the observed values.

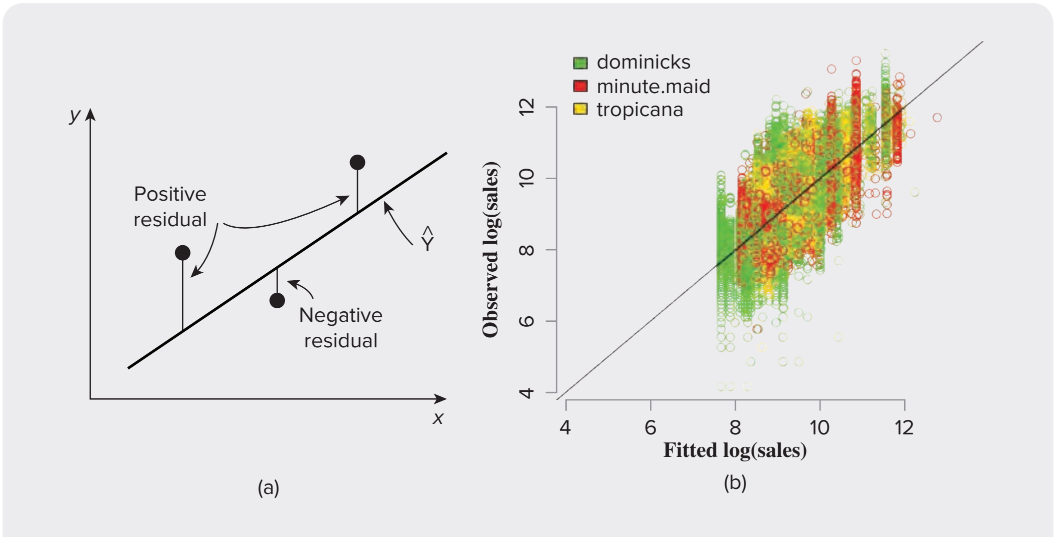


FIGURE 1.9 Panel (a) shows residuals in a simple linear regression, and panel (b) shows the fitted response $\hat{y}$ vs. observed response y for the Dominick's OJ example along with a line along $\hat{y} = y$. In both plots, the residuals are the vertical distance between each point and the fitted line.

The difference between them is called the *residual.* We will usually denote the residual as e_i, such that

$$e_i = y_i - \hat{y}_i = y_i - \mathbf{x}_i'\hat{\boldsymbol{\beta}} \tag{1.19}$$

Residuals play a central role in how linear regression works. They are our estimates of the error terms, ε_i, and they represent the variability in response that is not explained by the model.

Figure 1.9a illustrates residuals for a single-input regression. Points above the line have a positive residual and points below have a negative residual. Figure 1.9b shows the observed versus fitted y for our OJ example. The residuals are the vertical distance between each point and the fitted line. For observed y that are higher than the predicted response, $\hat{y}$, the residual is positive and for observed y that are lower, the residual is negative, that is, observed y that are higher than the predicted response $\hat{y}$.

The residuals tell you about your fit of the linear regression model. Recall our full model is $\log(\texttt{sales}_i) = \mathbf{x}_i'\boldsymbol{\beta} + \varepsilon_i$ where $\varepsilon_i \sim \mathrm{N}(0, \sigma^2)$. The residuals are your estimates for the errors ε_i, and you can use them to evaluate the model $\varepsilon_i \sim \mathrm{N}(0, \sigma^2)$. For example, one important consideration is whether we are correct to assume a *constant error variance*: the fact that σ^2 is the same for every ε_i. If you look at Figure 1.9b, you can see that this is probably not true. On the bottom side of the plot, there is a collection of large negative residuals for Dominick's. The model appears to have a floor on expected log sales. If you look at the results, the maximum price ever charged for Dominick's is \$2.69 and this leads to a floor on expected log sales of $\hat{y} = 7.66$ (when $\texttt{ad} = 0$). However, our residuals show that sometimes Dominick's sells far less than this floor. This is possibly driven by stock-outs, where the supply of Dominick's orange juice can't keep up with demand. Although the problem here appears isolated to a small number of observations, we could likely improve our model for the Dominick's OJ sales-price elasticity if we were able to remove observations where the store ran out of OJ. In Chapter 2 we will discuss strategies for dealing with this type of nonconstant error variance.

Error Variance

Another important use of the residuals is to estimate the error variance, σ^2. Notwithstanding the minor issue of the handful of large Dominick's errors just described, we can do this by looking at the variability of the residuals. When you call `summary` on the fitted `glm` object, `R` calls the `summary.glm` function that prints a bunch of information about the model estimates. Don't worry about all of this information; we will work through most of it in the coming two chapters. But near the bottom it prints out an estimate for the "`dispersion parameter for gaussian family`." This is what `glm` calls its estimate for the error variance, $\hat{\sigma}^2$.

```
> summary(fit3way)
...
(Dispersion parameter for gaussian family taken to be 0.4829706)
    Null deviance: 30079  on 28946  degrees of freedom
Residual deviance: 13975  on 28935  degrees of freedom
...
```

In the case of our three-way interaction model for OJ log sales, the estimated error variance is $\hat{\sigma}^2 = 0.4829706$. To understand how `glm` came up with this estimate, we need to dive deeper into the concepts in the bottom two lines shown here: deviance and degrees of freedom.

1.2.1 Deviance and Least Squares Regression

Deviance is the distance between your fitted model and the data. We will look at the specifics of deviance later, in the context of both linear and logistic regression. But for now you just need to know that deviance for linear regression is the sum of squared errors. The *null deviance* is calculated for the "null model," i.e. a model where none of the regression inputs have an impact on y. This is just the model $y_i \sim \mathrm{N}(\mu, \sigma^2)$. Estimating μ with the sample mean response, $\bar{y}$, we can calculate this null deviance as $\Sigma_i(y_i - \bar{y})^2$. For our OJ regression, this produces the 30079 value from the `summary` output.

```
> ( SST <- sum( (log(oj$sales) - mean(log(oj$sales)))^2 ) )
[1] 30078.71
```

The null deviance for linear regression is known as the *sum squared total* error, or SST. It measures how much variation you have in your response before fitting the regression.

The *residual deviance*, or more commonly *fitted deviance* or simply *deviance*, is calculated for your fitted regression model. It measures the amount of variation you have after fitting the regression. Given residuals $e_i = y_i - \hat{y}_i$, the residual deviance is just the sum of squared residuals $\Sigma_i e_i^2$. For our OJ regression, this gives us a residual deviance of 13975.

```
> ( SSE <- sum( ( log(oj$sales) - fit3way$fitted )^2 ) )
[1] 13974.76
```

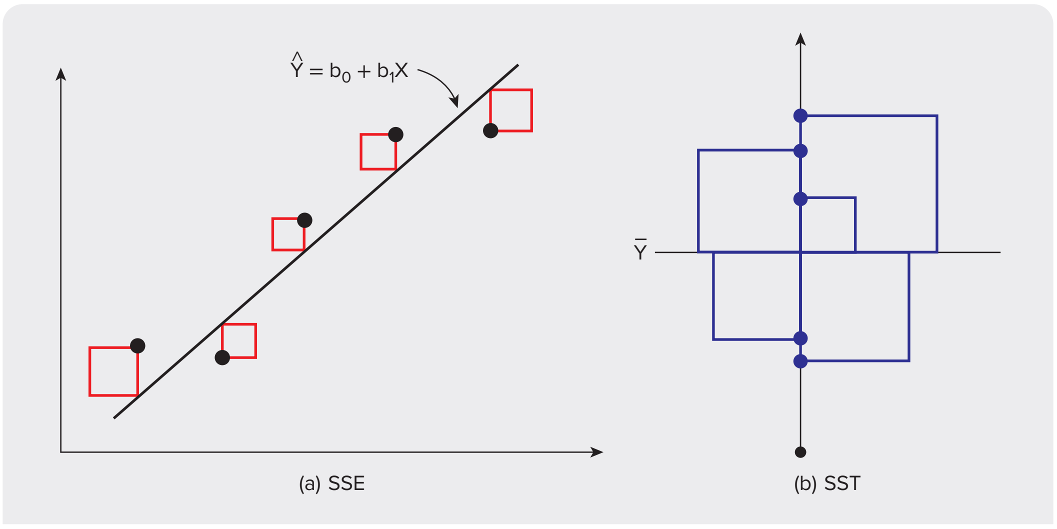


FIGURE 1.10 Figure 1.10a shows the squared residual errors. These are the components of the SSE, the quantity that linear regression minimizes and is output in `glm` as `Residual deviance`. Figure 1.10b shows the squared vertical distance for each observation to the overall mean, $\bar{y}$. The sum of these squared areas is the sum square total (SST) and is output as `Null deviance`.

This residual deviance for linear regression is known as the *sum squared residual error*, or SSE. It measures the tightness of your model fit. For example, a common metric for model fit takes the SSE and scales it by the number of observations to get mean squared error: $MSE = SSE/n$, where n is the sample size.

Proportion of Deviance Explained

The calculations behind SSE and SST are illustrated for simple linear regression in Figure 1.10. Comparison between these two deviances tells you how the variability has been *reduced* due to the information in your regression inputs. A common and useful statistic, one that we will use throughout the book, is the R^2 equal to one minus the residual deviance over the null deviance. In linear regression this is

$$R^2 = 1 - \frac{SSE}{SST} \tag{1.20}$$

The R^2 is the *proportion of variability explained by the regression.* It is the proportional reduction in squared errors due to your regression inputs. The name R^2 is derived from the fact that, for linear regression only, it is equal to the square of the correlation (usually denoted r) between fitted $\hat{y}_i$ and observed values y_i.

Example 1.5 **Orange Juice Sales: *R*^{2}** We can calculate the R^2 a couple of different ways for our three-way interaction OJ regression.

```
# using the glm object attributes
> 1-fit3way$deviance/fit3way$null.deviance
```

```
[1] 0.5353939
# using the SSE and SST calculated above
> 1 - SSE/SST
[1] 0.5353939
# correlation squared
> cor(fit3way$fitted,log(oj$sales))^2
[1] 0.5353939
```

However you calculate it, the regression model explains around 54% of the variability in log orange juice sales. The interpretation of R^2 as squared correlation can help you get a sense of what this means: if $R^2 = 1$, a plot of fitted vs. observed values should lie along the perfectly straight line $\hat{y} = y$. As R^2 decreases the scatter around this line increases.

The residual, or "fitted," deviance plays a crucial role in how models are fit. The concept of deviance minimization is crucial to all model estimation and machine learning. In the case of linear regression, you are minimizing the sum of squared residual errors. This gives linear regression its common name: *Ordinary Least Squares,* or OLS (the "ordinary" is in contrast to "weighted least squares" in which some observations are given more weight than others). Our readers coming from an economics or social sciences background might be more familiar with this terminology. We will use the terms OLS and linear regression interchangeably.

1.2.2 Degrees of Freedom

Reprinting the relevant summary output, we have one final concept to decipher.

```
> summary(fit3way)
...
(Dispersion parameter for gaussian family taken to be 0.4829706)
    Null deviance: 30079  on 28946  degrees of freedom
Residual deviance: 13975  on 28935  degrees of freedom
...
```

The *degrees of freedom* are crucial for mapping from your deviance to the estimated dispersion parameter, $\hat{\sigma}^2$. Unfortunately, the way that `summary.glm` uses this term is confusing because it doesn't differentiate between two different types of degrees of freedom: those used in the model fit, and those left for calculating the error variance. These concepts are important in statistics and machine learning, so we'll take the time to pull them apart.

To understand degrees of freedom, take a step back from regression and consider one of the most basic estimation problems in statistics: estimating the variance of a random variable.

Say you have a sample $\{z_1 \ldots z_n\}$ drawn independently from the probability distribution p(z). Recall your usual formula for estimating the variance of this distribution:

$$\mathrm{var}(z) \approx \sum_{i=1}^{n} \frac{(z_i - \bar{z})^2}{n-1} \tag{1.21}$$

where $\bar{z} = (1/n)\sum_{i=1}^{n} z_i$ is the sample mean. Why are we dividing by (n – 1) instead of n? If we divide by n our estimate of the variance will be *biased low*—it will tend to underestimate var(z). To get the intuition behind this, consider an n =1 sample that consists of a single draw: $\bar{z} = z_1$, and thus $z_1 - \bar{z} = 0$ by construction. Since you are estimating the mean from your sample, you have the flexibility to fit perfectly a single observation. In other words, when n =1 you have zero opportunities to view any actual variation around the mean. Extending to a larger sample of size n, you have only n – 1 opportunities to observe variation around $\bar{z}$.

To use the language of statistics, "opportunities to observe variation" are called degrees of freedom. In our simple variance example, we used one *model degree of freedom* to estimate $\mathbb{E}[z]$, and that leaves us with n – 1 *residual degrees of freedom* to observe error variance. More generally:

- The **model degrees of freedom** are the number of random observations your model could fit perfectly. In regression models, this is the number of coefficients. For example, given a model with two coefficients (an intercept and slope) you can fit a line directly through two points.
- The **residual degrees of freedom** are equal to the number of opportunities that you have to observe variation around the fitted model mean. This is the sample size, n, minus the model degrees of freedom.

The model degrees of freedom are used for fitting the model and the residual degrees of freedom are what is left over for calculating variability after fitting the model. Throughout the rest of the book, we will follow the convention of using *degrees of freedom* (or df) to refer to the model degrees of freedom unless stated otherwise. Somewhat confusingly, the `summary.glm` output uses `degrees of freedom` to refer to the residual degrees of freedom, or $n - df$ in our notation.

Once we have the terminology straight, we can now complete our original mission to understand how `glm` has calculated $\hat{\sigma}^2$. In fitting the linear regression, the number of model degrees of freedom used is equal to the number of parameters in the regression line. For our model in `fit3way`, there are a total of 12 parameters in the model (use `length(coef(fit3way))` to verify). So we would say $df = 12$ for this model. And since there are 28,947 observations in the OJ dataset, the residual degrees of freedom for this model are 28,947 − 12 = 28,935. This is the number that `glm` outputs next to the residual deviance. It is the number of opportunities that we have to view variation around the regression line. So, to estimate the residual variance, we take the sum of the squared residuals (the SSE) and divide by 28,935.

```
> SSE/fit3way$df.residual
[1] 0.4829706
```

This gives you $\hat{\sigma}^2$, or what `summary.glm` calls the "dispersion." The summary output also provides a `degrees of freedom` for the null deviance. Since the null model fits only a single mean value, $\mathbb{E}[y] = \bar{y}$, this is equal to n – 1, the denominator in our simple variance equation (1.21). For the OJ example this is 28,946.

1.3 Logistic Regression

Linear regression is just one instance of the general linear modeling framework. Another extremely useful GLM is *logistic regression.* This is the GLM that you will want to use for modeling a *binary* response: a y that is either 1 or 0 (e.g., true or false). While linear regression is probably the most commonly used technique in business analytics, logistic regression would come a close second in popularity. In machine learning, logistic regression and extensions of the framework are the dominant tools for prediction and classification.

Binary responses arise from a number of prediction targets:

- Will this person pay their bills or default?
- Is this a thumbs-up or thumbs-down review?
- Will the customer take advantage of the offer?
- Is the writer a Republican or Democrat?

Even when the response of interest is not binary (e.g., revenue), it may be that your decision-relevant information is binary (e.g., profit versus loss) and it is simplest to think in these terms. Logistic regression is also the stepping stone to more complex classification methodologies, which we will dive into in Chapter 4.

As you read through this section, it is important to keep in mind that logistic regression works very similarly to linear regression. It is easy to get wrapped up in the differences between logistic and linear regression, but the basics are exactly the same. In each case you are fitting a linear function of the input features and you are estimating the model by minimizing the deviance. The only difference is the choice of link function in your generalized linear model.

1.3.1 Logit Link Function

Recall our basic GLM specification of Equation (1.1) for expressing the expected value of response y given inputs $\mathbf{x}$: $\mathbb{E}[y|\mathbf{x}] = f(\mathbf{x}'\boldsymbol{\beta})$. The *link function*, f, is used to map from a linear function to a response that takes a few specific values. When the response y is 0 or 1, the conditional mean becomes

$$\mathbb{E}[y|\mathbf{x}] = \mathrm{p}(y = 1|\mathbf{x}) \times 1 + \mathrm{p}(y = 0|\mathbf{x}) \times 0 = \mathrm{p}(y = 1|\mathbf{x})$$

Therefore, the expectation you're modeling is a *probability.* This implies that you need to choose the link function $f(\cdot)$ to give values between zero and one.

Using the shorthand of $p = \mathrm{p}(y = 1|\mathbf{x})$, you need to choose a link function such that it makes sense to write

$$p = \mathrm{p}(y = 1|\mathbf{x}) = f(\beta_0 + \beta_1 x_1 \ldots + \beta_k x_k)$$

Logistic regression addresses this by using a *logit* link function, $f(z) = e^z/(1 + e^z)$. This function, which is also often called the "sigmoidal function," is plotted in Figure 1.11. Notice that the function asymptotes (approaches but does not cross) zero at large negative z values, and one at large positive z values.

To see how this link works, consider extreme values for z. At large negative values, say as $z \to -\infty$, then $f(z) \to 0/(1 + 0) = 0$ and the event $y = 1$ approaches zero probability. At large positive values, say as $z \to \infty$, then $f(z) \to \infty/(\infty + 1) = 1$ and $y = 1$ becomes guaranteed. Thus, the logit link maps from the "real line" of numbers to the [0,1] space of probabilities.

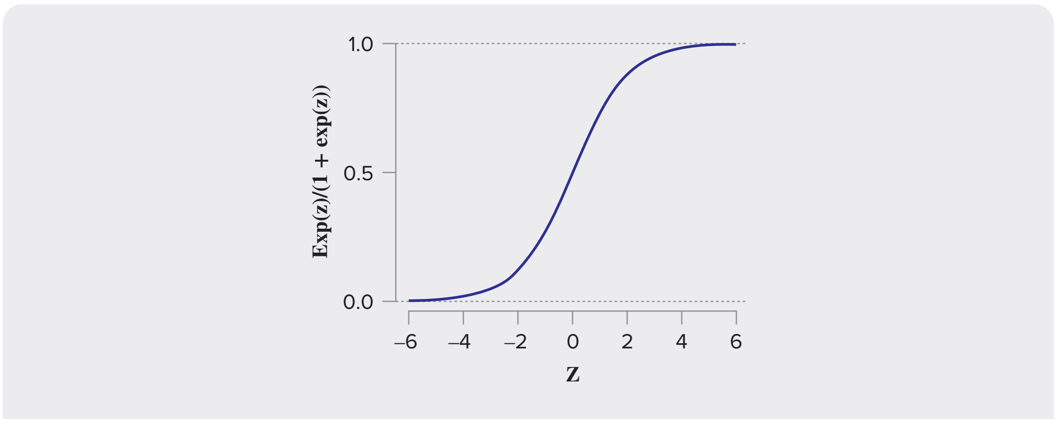


FIGURE 1.11 The logit link function, $f(z) = e^z/(1 + e^z)$.

Using a logit link, the GLM equation for $\mathbb{E}[y|\mathbf{x}]$ is defined as

$$\mathbb{E}[y|\mathbf{x}] = \mathrm{p}(y = 1|\mathbf{x}) = \frac{e^{\mathbf{x}'\boldsymbol{\beta}}}{1 + e^{\mathbf{x}'\boldsymbol{\beta}}} = \frac{e^{\beta_0+\beta_1 x_1 \cdots +\beta_k x_k}}{1 + e^{\beta_0+\beta_1 x_1 \cdots +\beta_k x_k}} \tag{1.22}$$

A common alternate way to write (1.22) results from dividing the numerator and denominator by $e^{\mathbf{x}'\boldsymbol{\beta}}$:

$$\mathbb{E}[y|\mathbf{x}] = \frac{e^{\mathbf{x}'\boldsymbol{\beta}}}{1 + e^{\mathbf{x}'\boldsymbol{\beta}}} = \frac{\frac{e^{\mathbf{x}'\boldsymbol{\beta}}}{e^{\mathbf{x}'\boldsymbol{\beta}}}}{\frac{1}{e^{\mathbf{x}'\boldsymbol{\beta}}} + \frac{e^{\mathbf{x}'\boldsymbol{\beta}}}{e^{\mathbf{x}'\boldsymbol{\beta}}}} = \frac{1}{e^{-\mathbf{x}'\boldsymbol{\beta}} + 1} \tag{1.23}$$

How do we interpret the β coefficients in this model? We need to start with the relationship between probability and *odds*. The odds of an event are defined as the probability that it happens over the probability that it doesn't.

$$\text{odds} = \frac{p}{1 - p} \tag{1.24}$$

For example, if an event has a 0.25 probability of happening, then its odds are 0.25/0.75, or 1/3. If an event has a probability of 0.9 of happening, the odds of its happening are 0.9/0.1 = 9. Odds transform from probabilities, which take values between zero and one, to the space of all positive values from zero to infinity.

Looking at (1.22), we can do some algebra and then take the log to derive an interpretation for the β_j coefficients. Using the shorthand $p = \mathrm{p}(y = 1|\mathbf{x})$, we have

$$\begin{aligned} p &= \frac{e^{\mathbf{x}'\boldsymbol{\beta}}}{1 + e^{\mathbf{x}'\boldsymbol{\beta}}} \\ &\Rightarrow p + pe^{\mathbf{x}'\boldsymbol{\beta}} = e^{\mathbf{x}'\boldsymbol{\beta}} \\ &\Rightarrow \frac{p}{1 - p} = e^{\mathbf{x}'\boldsymbol{\beta}} \\ &\Rightarrow \log\left(\frac{p}{1 - p}\right) = \beta_0 + \beta_1 x_1 \cdots + \beta_k x_k \end{aligned}$$

Thus, logistic regression is a *linear model for log odds*. Using what we know about logs and exponentiation, you can interpret e^{β_j} as the *multiplicative* effect for a unit increase in x_j on the odds for the event $y = 1$. For example, consider a logistic regression model with a single predictor x, such that $\text{odds}(x) = \exp[\beta_0 + \beta_1 x]$.

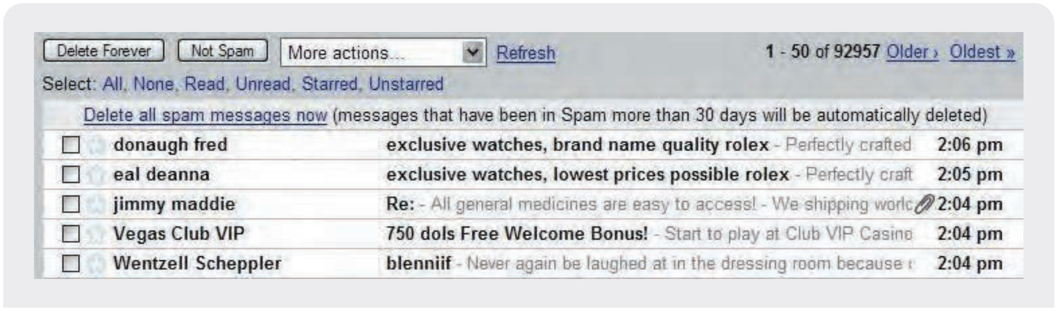
1.3.2 Fitting logistic regression in R

You can use `glm` to fit logistic regressions in `R`. The syntax is exactly the same as for linear regression, you just add the argument `family="binomial"`. Recall that the binomial distribution is the distribution for random trials with a binary outcome. The classic binomial distribution is a coin toss. Telling `glm` that you are working with a binomial distribution implies that you will be working with a binary response and want to estimate probabilities. The logit link is how `glm` fits probabilities. The response variable can take a number of forms including numeric 0 or 1, logical `TRUE` or `FALSE`, or a two level factor such as `win` vs. `lose`.

Example 1.6 Logistic Regression: Detecting Spam For our first logistic regression example, we'll build a filter for email spam—junk mail that can be ignored. Every time an email arrives, your email client performs a binary classification: is this *spam* or *not spam*? The email that is classified as spam gets automatically moved to a spam folder (like that in Figure 1.12), keeping your inbox free for important messages. We'll train our own spam filter by fitting logistic regression to previous emails.

Our training data `spam.csv` has 4601 emails, 1813 of which are spam. It contains 57 email features including indicators for the presence of 54 keywords or characters (e.g., `free` or `!`), counts for capitalized letters (total number and longest continuous block length), and a numeric `spam` variable for whether each email has been tagged as spam by a human reader (`spam` is one for true spam, zero for important emails). We read this data into `R` as a data frame named `spammy`.

```
> spammy<- read.csv("spam.csv")
> spammy[c(1,4000), c(16,56,58)]
     word_free capital_run_length_longest spam
1            1                         61    1
4000         0                         26    0
```



Delete Forever | Not Spam | More actions... | Refresh — 1 - 50 of 92957 Older › Oldest »

Select: All, None, Read, Unread, Starred, Unstarred

Delete all spam messages now (messages that have been in Spam more than 30 days will be automatically deleted)

☐	donaugh fred	**exclusive watches, brand name quality rolex** - Perfectly crafted	2:06 pm
☐	eal deanna	**exclusive watches, lowest prices possible rolex** - Perfectly craft	2:05 pm
☐	jimmy maddie	**Re:** - All general medicines are easy to access! - We shipping worlc	2:04 pm
☐	Vegas Club VIP	**750 dols Free Welcome Bonus!** - Start to play at Club VIP Casino	2:04 pm
☐	Wentzell Scheppler	**blenniif** - Never again be laughed at in the dressing room because	2:04 pm

FIGURE 1.12 An email folder filled with spam.

Notice that the first email, which contained the word `free` and had a block of 61 capitalized letters, was tagged as spam. Email 4000, with its more modest sequence of 26 capital letters, is not spam.

Our logistic regression will use all of the features in `spammy` as inputs. The `R` formula "`y ~ .`" tells `glm` to regress onto all variables in the data frame except for the response.

```
> spamFit <- glm(spam ~ ., data=spammy, family='binomial')
Warning message:
glm.fit: fitted probabilities numerically 0 or 1 occurred
```

The warning message you get when you run this regression, `fitted probabilities numerically 0 or 1 occurred`, means the regression is able to fit some data points *exactly*. For example, a spam email is modeled as having a 100% probability of being spam. This situation is called *perfect separation*; it can lead to strange estimates for some coefficients and their standard errors. It is a symptom of *overfit*, and in Chapter 3 we show how to avoid it via regularization techniques.

The fitted object, `spamFit`, is a `glm` object that contains all the same attributes that we were able to access when doing linear regression. For example, you can use the `summary` function to get statistics about your coefficients and model fit.

```
> summary(spamFit)
...
 Coefficients:
                Estimate Std. Error  z value  Pr(>|z|)
(Intercept)    -1.9682470  0.1465703  -13.429   < 2e-16 ***
word_make      -0.5529572  0.2356753   -2.346  0.018963 *
word_address   -0.1338696  0.2217334   -0.604  0.546016
word_all       -0.4946420  0.1775333   -2.786  0.005333 **
word_3d         0.8301668  0.8244961    1.007  0.313994
...
(Dispersion parameter for binomial family taken to be 1)
    Null deviance: 6170.2  on 4600  degrees of freedom
Residual deviance: 1548.7  on 4543  degrees of freedom
...
```

The output looks basically the same as what you get for a linear regression. Note that in logistic regression there is no σ^2 to estimate as the "dispersion parameter" because there is no error term like the ε of linear regression. Instead, `glm` outputs `Dispersion parameter for binomial family taken to be 1`. If you don't see this, then you might have forgotten to put "`type=binomial`".

Interpreting Coefficients

Take a look at one of the large positive coefficients in your fit:

```
> coef(spamFit)["word_free"]
word_free
 1.542706
> exp(1.542706)
[1] 4.67723
```

The `word_free` variable is either one if the email contains the word `free`, and zero if it doesn't. Thus, the *odds* that an email is spam are about five times higher for emails that contain the word `free` than for those that do not. On the other hand, you can see next that if the email contains the word `george`, the odds of its being spam *decrease*.

```
> coef(spamFit)["word_george"]
word_george
  -5.779841
> exp(-5.779841)
[1] 0.003089207
> 1/exp(-5.779841)
[1] 323.7077
```

The odds of the email being spam when the word `george` is present are 0.003 of the odds of its being spam if it does not contain the word `george`. Or, taking the reciprocal, the odds of its being spam when the word `george` is *absent* are about 324 times higher than if the word `george` is *present*. This is an old dataset collected from the inbox of a guy named George. Spammers were not very sophisticated in the 1990s, so emails containing your name were most likely not spam.

Predicting Spam Probabilities

As with linear regression, prediction for logistic regression is easy after you've fit the model with `glm`. You call `predict` on your fitted `glm` object and provide some `newdata`, with the same variable names as the training data, at the locations where you'd like to predict. The output will be $\mathbf{x}'\hat{\boldsymbol{\beta}}$ for each $\mathbf{x}$ row of `mynewdata`.

```
> predict(spamFit, newdata=spammy[c(1,4000),])
        1      4000
2.029963 -1.726788
```

Of course, these are *not* probabilities. To get those, you need to transform to $f(\mathbf{x}'\hat{\boldsymbol{\beta}})$ through the logit link as $e^{\mathbf{x}'\hat{\beta}}/(1 + e^{\mathbf{x}'\hat{\beta}})$, as in Eqn 1.22. The `predict()` function lets you add the `type="response"` argument to make this transformation and get predictions on the scale of the response (i.e., in [0,1] probability space).

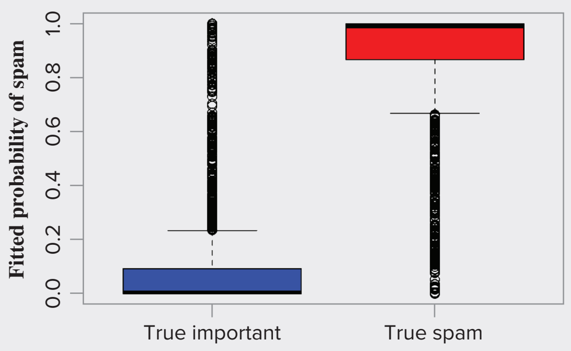


FIGURE 1.13 Fit plot of $\hat{y}$ versus y for the spam logistic regression. Since the true y is binary for spam, you get a boxplot rather than a scatterplot. As a test of your intuition, imagine what a *perfect* fit (i.e., $\hat{y} = y$) would look like for this regression.

```
> predict(spamFit,newdata=spammy[c(1,4000),],type="response")
        1      4000
0.8839073 0.1509989
```

The first email (true spam) has an 88% chance of being spam, while email 4000 (not spam) has a 15% chance of being spam—in other words, an 85% chance of being important email that George wants to read. Figure 1.13 shows predicted probabilities of spam by actual spam status for every email in the dataset. Note the long tails of small spam probabilities for true spam and of large spam probabilities for truly important mail: any spam classifier that you construct based on this model will occasionally make a mistake on how it treats the email. See Chapter 4 for material on designing and evaluating classification rules.

1.4 Likelihood and Deviance

Earlier in this chapter, you learned that deviance is the distance between the *model* and the *data.* In the case of linear regression, the deviance is the sum of squared errors. Logistic regression also has a deviance, and this is the metric that `glm` minimizes to fit the model. But what is the deviance for logistic regression? It is not a sum of squared errors. Instead, the logistic regression deviance is derived from the assumed binomial distribution for the response.

How this works relies on two complementary concepts: the likelihood and the deviance. These concepts are a bit abstract, but they play a key role in the statistical learning algorithms that we will be working with throughout this book.

- *Likelihood* is the probability of your data given the estimated model. When you maximize the likelihood, you are fitting the parameters to "make the data look most likely."
- *Deviance* is a measure of the distance between the data and the estimated model. When you minimize the deviance, you are fitting the parameters to make the model and data look as close as possible.

Likelihood

To unravel these concepts we'll start with the likelihood function. Consider a dataset, say Z, with probability p(Z|Θ). This probability is a function of both the data Z and the parameters Θ. The likelihood function takes a given dataset as fixed and represents how this probability changes as a function of the parameters. Thus we write the likelihood as lhd(Θ; Z), or sometimes just lhd(Θ) for short, to indicate that it is a function of the parameters Θ. But there is nothing complicated going on: the likelihood is just a probability. In particular, lhd(Θ) = p(Z|Θ) in our imaginary setup here.

Consider a simple binomial example. You have a weighted coin with probability p of coming up heads. You have flipped the coin ten times: it has come up heads eight times and tails twice. We could write our dataset as $Z = \{\text{heads} = 8, \text{tails} = 2\}$. The probability of this dataset, and the likelihood, is written

$$p(Z|p) = \binom{10}{8} p^8 (1-p)^2 = \text{lhd}(p) \tag{1.25}$$

We can evaluate and plot this likelihood in R (see Figure 1.14a).

```
> p <- seq(0,1,length=100)
> plot(p, dbinom(8, size=10, prob=p), type="l", ylab="Likelihood")
```

Every time glm fits a model, it is choosing the parameters to maximize the likelihood. This is a very common estimation strategy with many great properties. Although we will look at other techniques in the next chapter, in particular adding *penalties* on parameter size during estimation, everything will still be built around the foundation of likelihood maximization. In our coin-flipping example, the maximum likelihood estimate (the MLE) is $\hat{p} = 0.8$. This is marked with a vertical line in Figure 1.14a, and it corresponds to the highest point on the likelihood curve.

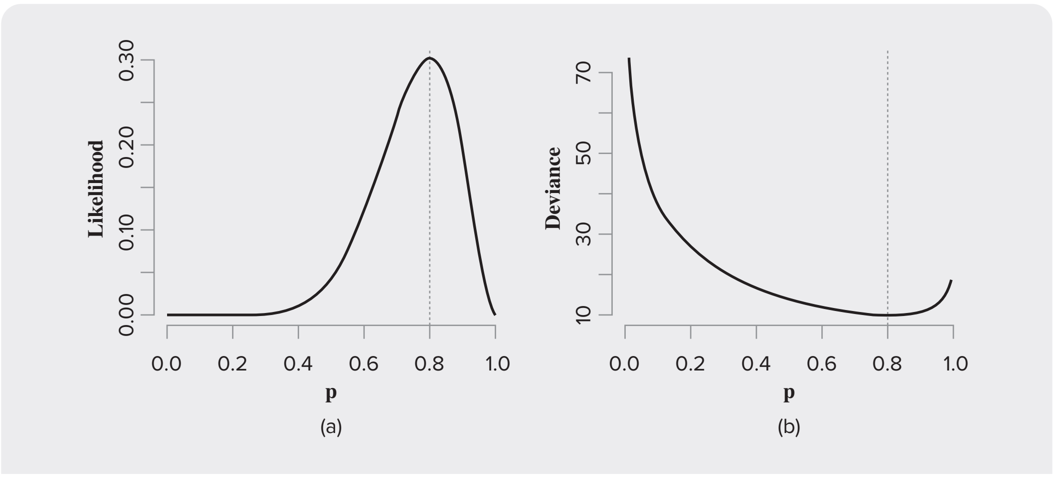


FIGURE 1.14 The likelihood (a) and deviance (b) for the probability of success, p, in a binomial trial with eight successes and two failures.

Deviance

The deviance—the distance between your model and the data—is a simple transformation of the likelihood. In particular,

$$\text{Deviance} = -2\log[\text{Likelihood}] + C \tag{1.26}$$

Here, C is a constant that you can ignore. The precise definition for deviance is -2 times the difference between log likelihoods for your fitted model and for a "fully saturated" model where you have as many parameters as observations. The term corresponding to this fully saturated model gets wrapped into the constant, C, but again you can ignore this in most situations. In practice, we will often use the $\propto$, or proportional to, symbol when working with the deviance and only keep track of the parts that change as a function of the parameters. For example, in our coin tossing example, the deviance is

$$\text{dev}(p) \propto -2\log(p^8(1-p)^2) = -16\log(p) - 4\log(1-p) \tag{1.27}$$

This is plotted in Figure 1.14b, with the deviance minimizing solution marked at $\hat{p} = 0.8$. Deviance minimization is the mirror image of likelihood maximization. With `glm` you have been fitting models by minimizing the deviance, just the same as you have been fitting models to maximize the likelihood.

Example 1.7 **Gaussian Deviance** Let's work through an example with linear regression and Gaussian (normal) errors. The probability model is $y \sim \text{N}(\mathbf{x}'\beta, \sigma^2)$, where the Gaussian probability density function is

$$\text{N}(\mathbf{x}'\boldsymbol{\beta}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left[-\frac{(y-\mathbf{x}'\boldsymbol{\beta})^2}{2\sigma^2}\right] \tag{1.28}$$

Recall that independent random variables have the property that $\text{p}(y_1, \ldots, y_n) = \text{p}(y_1) \times \text{p}(y_2) \times \ldots \text{p}(y_n)$. Given n independent observations, the likelihood (i.e., the probability density of the data) is

$$\prod_{i=1}^{n}\text{p}(y_i|\mathbf{x}_i) = \prod_{i=1}^{n}\text{N}(y_i;\, \mathbf{x}_i'\boldsymbol{\beta}, \sigma^2) = (2\pi\sigma^2)^{-\frac{n}{2}}\exp\left[-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mathbf{x}_i'\boldsymbol{\beta})^2\right] \tag{1.29}$$

Taking a log and multiplying by -2 (and removing terms that don't involve $\boldsymbol{\beta}$), you get

$$\text{dev}(\boldsymbol{\beta}) = \frac{1}{\sigma^2}\sum_{i=1}^{n}(y_i - \mathbf{x}_i'\boldsymbol{\beta})^2 + C \propto \sum_{i=1}^{n}(y_i - \mathbf{x}_i'\boldsymbol{\beta})^2 \tag{1.30}$$

Thus, for linear regression with Gaussian errors, the deviance is proportional to the sum of squared errors (the SSE). We stated this fact earlier in the chapter, but now you can derive it for yourself. This is why linear regression is also "least-squares" regression: deviance minimization is the same thing as minimizing the SSE.

Example 1.8 **Logistic Deviance** We can do a similar derivation for logistic regression. For binary response with probabilities $p_i = \text{p}(y_i = 1)$, the likelihood is

$$\prod_{i=1}^{n}P(y_i|\mathbf{x}_i) = \prod_{i=1}^{n}p_i^{yi}(1-p_i)^{1-y_i} \tag{1.31}$$

Using your logistic regression equation for p_i, this becomes

$$\text{lhd}(\boldsymbol{\beta}) = \prod_{i=1}^{n} \left(\frac{\exp(\mathbf{x}_i'\boldsymbol{\beta})}{1+\exp(\mathbf{x}_i'\boldsymbol{\beta})}\right)^{y_i} \left(\frac{1}{1+\exp(\mathbf{x}_i'\boldsymbol{\beta})}\right)^{1-y_i} \tag{1.32}$$

Taking log and multiplying by -2 gives you the logistic regression deviance:

$$\begin{aligned}\text{dev}(\boldsymbol{\beta}) &= -2\sum_{i=1}^{n}[y_i \log(p_i) + (1-y_i)\log(1-p_i)] \\ &\propto \sum_{i=1}^{n}[\log(1+\exp^{\mathbf{x}_i'\boldsymbol{\beta}}) - y_i\mathbf{x}_i'\boldsymbol{\beta}]\end{aligned} \tag{1.33}$$

This is the function that `glm` minimizes for logistic regression.

Deviance in `summary.glm`

Returning to our output from `summary.glm` (which is the function that is called when you apply `summary` to a fitted `glm` object), we have deviances for each of the OJ and spam regressions. For the three-way interaction OJ linear regression:

```
> summary(fit3way)
...
    Null deviance: 30079  on 28946  degrees of freedom
Residual deviance: 13975  on 28935  degrees of freedom
...
```

And for the spam filter logistic regression:

```
> summary(spamFit)
...
    Null deviance: 6170.2  on 4600  degrees of freedom
Residual deviance: 1548.7  on 4543  degrees of freedom
...
```

From Equations (1.30) and (1.33), we now know how to calculate these residual deviance values. The null deviances come from the same models, and so have the same functional form, but they replace the regression fitted values for y with simple sample averages. With D_0 as the symbol for null deviance, we have

- $D_0 = \Sigma(y_i - \bar{y})^2$ in linear regression
- $D_0 = -2\Sigma[y_i \log(\bar{y}) + (1 - y_i)\log(1 - \bar{y})]$ in logistic regression

While some statistics texts restrict the concept of R^2 to linear regression, we find it useful to generalize it as *the proportion of deviance that is reduced due to the regression model.* Using the symbol D to denote the residual deviance, our R^2 formula is

$$R^2 = 1 - \frac{D}{D_0} \tag{1.34}$$

This R^2 formula is often called McFadden's Pseudo R^2 when it is used outside of linear regression.

In the case of our OJ example, we previously calculated an R^2 of 0.54. For the spam regression, we can apply (1.34) to calculate that $R^2 = 1 - 1549/6170 = 0.75$ such that around three-quarters of the variability in spam occurrence is explained by our logistic regression. We introduced R^2 in the context of linear regression, as a function of the SST and SSE, but expressing it in terms of deviance means that it applies to any model that we fit.

1.5 Time Series

We close this chapter with an introduction to working with *dependent* data. The models we have looked at so far all assume that you have *independent* observations. However, events that occur one after the other in time, or say geographically near to each other, can be correlated. For example, lawn furniture sales are always higher in spring and summer, the weather today gives you information about what the weather will be tomorrow, or when a popular restaurant has a busy night its neighboring restaurants also gain traffic from those who couldn't get a table. In this section we'll figure out how to work with data that occurs in *time,* and in the next section we consider data that occurs in *space.*

Fortunately, the main tools for dealing with dependence all fit within a standard regression framework. For the most part, you simply include the variables that cause dependence in your set of inputs. By engineering the right input features, you can control for underlying trends (e.g., monthly trends or regional effects) and for *autoregression,* which is the dependence between neighboring outcomes. In this section we will focus on *time series* dependence. This is the sort of dependence that you get for data that are observed over time, and it is common in business analysis settings. The tools you learn for time series extend to other dependence settings, and we give some pointers on this at the end of the section.

The traditional statistics approach to time series emphasizes careful testing for different forms of time series structure. Through the regularization and machine learning material from later chapters in this book, we can avoid a lot of this manual feature selection. Although it won't work for all types of time series dependence, a powerful modeling strategy is to simply include a large set of time series features and rely on the data to tell you what works best. Thus, this section will focus on helping you understand how to construct the features that are useful for modeling time series data rather than on techniques for testing for time series dependence. If you have a good intuition about the ingredients of a time series model, you will be in good shape to use these features in your applied analysis work.

1.5.1 Regression for Time Series Data

A time series dataset contains observations of a response variable, and input features, taken over time. Typical time intervals are daily, weekly, monthly, quarterly, or yearly. In business settings, the response of interest is typically sales numbers, revenue, profit, active users, or prices. The response variables are almost always correlated over time.

Example 1.9 Airline Passenger Data: Regression for Time Series As an introductory example, consider a series of monthly total international airline passengers between the years 1949 and 1960.

```
> air<-read.csv("airline.csv")
> air[c(1,70,144),]
    Year Month Passengers
1     49     1        112
70    54    10        229
144   60    12        432
```

To work with time series data in `R`, your first step is to create a time variable. We can use the `as.Date` function to build a `Date` class vector. Note that if you are working on a finer time scale, `R` has the `POSIXct` class that can be used to represent dates and times down to fractions of a second. To create a `Date` variable in our air travel example, we need to translate from the `Year` and `Month` variables in our data frame. The first step is to paste these two variables into a single year-month string for each observation, and we then call `as.Date` to tell `R` that this is date information. The default format to read in dates is `year-month-day`, and that is what we will use here. We set the day to the first of each month for convenience; these are monthly counts so it doesn't matter what day we use.

```
> air$date<-paste("19",air$Year,"-",air$Month,"-01", sep="")
> air$date[c(1,70,144)]
[1] "1949-1-01" "1954-10-01" "1960-12-01"
> air$date<-as.Date(air$date)
> air$date[c(1,70,144)]
[1] "1949-01-01" "1954-10-01" "1960-12-01"
> class(air$date)
[1] "Date"
```

We now have the `date` variable, which `R` knows to treat as a calendar date. These dates are represented internally as days relative to January 1, 1970. You can convert them to `numeric` to see how `R` tracks the date.

```
> as.numeric(air$date[c(1,70,144)])
[1] -7670 -5571 -3318
```

```
> as.numeric(as.Date("1970-01-01"))
[1] 0
```

The two lines of code below produce plots of this data as in Figure 1.15.

```
> plot(Passengers ~ date, data=air, type="l")
> plot(log(Passengers) ~ date, data=air, type="l")
```

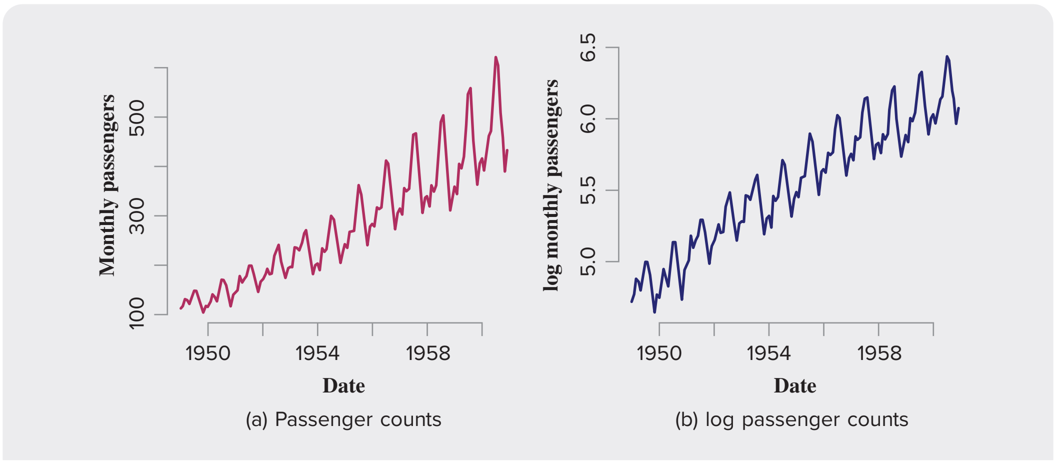


FIGURE 1.15 Time series data for 12 years of the monthly total count of international air passengers, 1949 through 1960.

Unlike our usual approach of plotting scatters of data points, here we have drawn a line plot (using the `type="l"` argument) to indicate dependence over time. In Figure 1.15a, you see an overall trend of an increasing number of passengers with time. In addition to this upward trend, you also see a repeated annual oscillation around the annual average. It is evident from Figure 1.15a that the oscillations around this upward trend are getting larger with time. This is a hallmark of a time series that is changing on a percentage scale with each observation. Recalling our work with sales data earlier in this chapter, that is an indication that you will want to be building a linear model on the log scale. Figure 1.15b shows the log monthly passenger volume. You can see that the log transformation yields consistently sized annual oscillations around a roughly linear-looking trend.

Linear Time Trend

We could fit a simple linear regression model to this data, say

$$\log(y_t) = \alpha + \beta t + \varepsilon_t$$

If you use the `date` variable as the input to `glm` to fit this regression, then from our `as.numeric` representations above you can see that the time trend will be counted in terms of days. This means that the impact of β on the monthly change will be a function of the number of the days of the month. This might be desirable in some applications, but to keep things simple here we will instead regress onto a simple index variable t that tracks the counts of months since the beginning of the dataset.

```
> air$t <- 1:nrow(air)
> fitAirSLR <- glm(log(Passengers)~t, data=air)
> coef(fitAirSLR)
```

```
(Intercept)          t
 4.81366828  0.01004838
> exp(coef(fitAirSLR)["t"])
       t
1.010099
```

The expected count of passengers increases by about 1% every month.

Seasonal Effects

This model just fits a line through the data in Figure 1.15b. It doesn't capture any of the oscillation around this line. This oscillating trend appears to be regular, trending up in summer months and down in winter. This is a classic *seasonal* pattern and we should account for it in the model. To do so, we can add *monthly* factor effects that encode the fact that, for example, people travel more in July than they do in November. The model is then

$$\log(y_t) = \alpha + \beta t + \gamma_{m_t} + \varepsilon_t \tag{1.35}$$

where γ_{m_t} denotes a separate seasonal effect for each month m_t (we are mixing in new greek letters beyond α and β here because the indexing gets complicated). To fit this in `R`, you need to encode `Month` as a factor and add it to your regression.

```
> air$Month <- factor(air$Month)
> levels(air$Month)
 [1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12"
> fitAirMonth <- glm(log(Passengers) ~ t + Month, data=air)
> round(coef(fitAirMonth),2)
(Intercept)       t     Month2     Month3      Month4      Month5
       4.73    0.01      -0.02       0.11        0.08        0.07
     Month6  Month7     Month8     Month9     Month10     Month11
       0.20    0.30       0.29       0.15        0.01       -0.14
    Month12
      -0.02
```

Noting that January is the reference month, absorbed into the intercept, we see that the highest months for travel are in the summer (June through August) and the lowest months are November through February. The fitted values for this regression are plotted in Figure 1.16 alongside the original data. The model appears to be doing a nice job of summarizing the passenger traffic (on log scale). To illustrate what each component is doing, Figure 1.17 shows the decomposition of this time series model into its two components: the fitted linear trend and the annual seasonal oscillations.

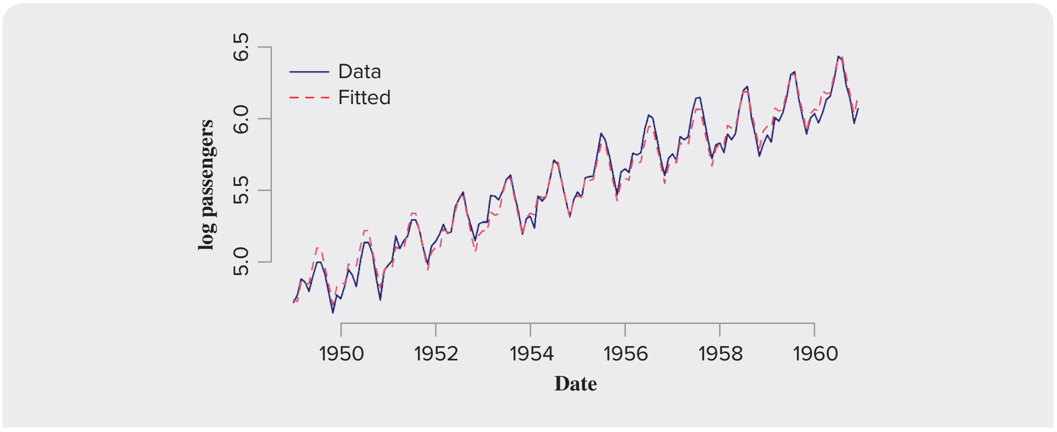


FIGURE 1.16 Airline passenger regression modeling using linear and monthly trends.

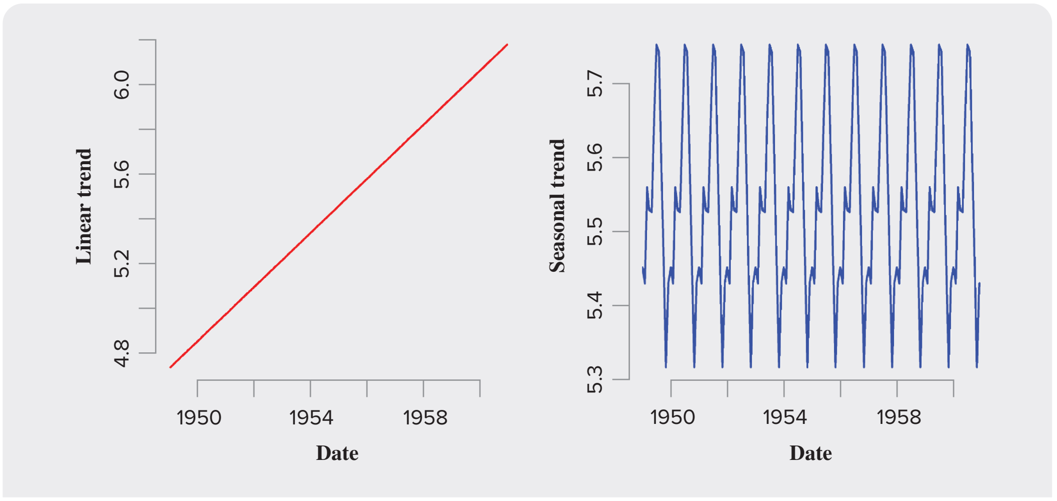


FIGURE 1.17 Decomposition of the fitted time series model for air travel.

Following the recipe we've just worked through, modeling the trends in time series data is easy. If your data include dates, then you should create indicator variables for, say, each year, month, and day. A best practice is to *proceed hierarchically:* if you are going to include an effect for `May-1955`, then you should also include broader effects for `May` and for `1955`. This allows the model to use the broad effects as baselines, and the `May-1955` effect will only summarize deviations from this base. In the language of Chapter 3, `May-1955` is *shrunk* toward generic levels for `May` and `1955`. The same logic applies for space: if you condition on counties, then you should also include state and region effects. This hierarchical approach is not strictly necessary when you are fitting OLS regressions via `glm`, but it will be crucial once we start to use the model regularization and selection techniques of Chapter 3.

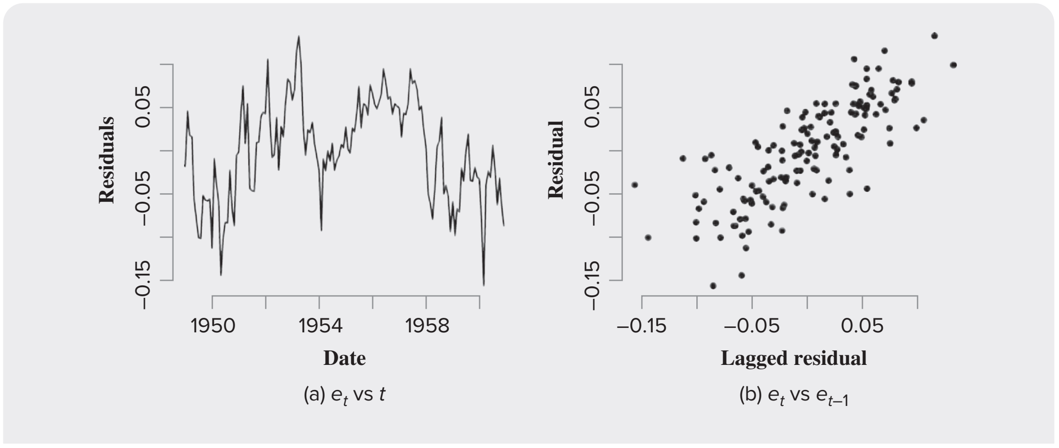


FIGURE 1.18 Residuals from the airline passenger regression including a linear time trend and monthly effects, plotted against time in (a) and against the lagged residuals in (b).

1.5.2 Autoregressive Models

Consider the air travel regression residuals shown in Figure 1.18a. That is a common approach to see what or how much is left unexplained by the model. When you look at this time series of residuals, you are looking for any patterns in how the residuals move over time. In this case, even though the combination of a linear trend and monthly effects did a decent job of predicting log passenger counts, there appears to be a stickiness in the residual time series: when they are high one month, they tend to be high the next month. More precisely, they appear correlated in time, such that $e_t = (y_t - \hat{y}_t)$ is dependent upon e_{t-1}. Diving deeper, Figure 1.18b plots the residuals e_t against their *lagged values,* e_{t-1}. There is a clear relationship between the current value and the one previous. This means that y_{t-1} can be used to predict y_t. It also implies that our regression residuals are now correlated, which violates the basic linear regression assumption of independence between residual errors.

This phenomenon is called *autocorrelation:* correlation between periods in a time series. Time series data is simply a collection of observations gathered over time. For example, suppose $y_1 \dots y_T$ are weekly sales, daily temperatures, or five-minute stock returns. In each case, you might expect what happens at time t to be correlated with time $t - 1$. For example, suppose you measure temperatures daily for several years. Which would work better as an estimate for today's temperature:

- The average of the temperatures from the previous year?
- The temperature on the previous day?

In most cases, yesterday's temperature is most informative. That means you view the *local* dependence as more important than the broad annual pattern.

Autocorrelation Function

You can summarize dependence between subsequent observations with an autocorrelation function (ACF) that tracks 'lag-l' correlations.

$$\text{acf}(l) = \text{cor}(\varepsilon_t, \varepsilon_{t-l}) \qquad \textbf{(1.36)}$$

Figure 1.19 shows the ACF for our airline regression residuals. You can produce this with the command below.

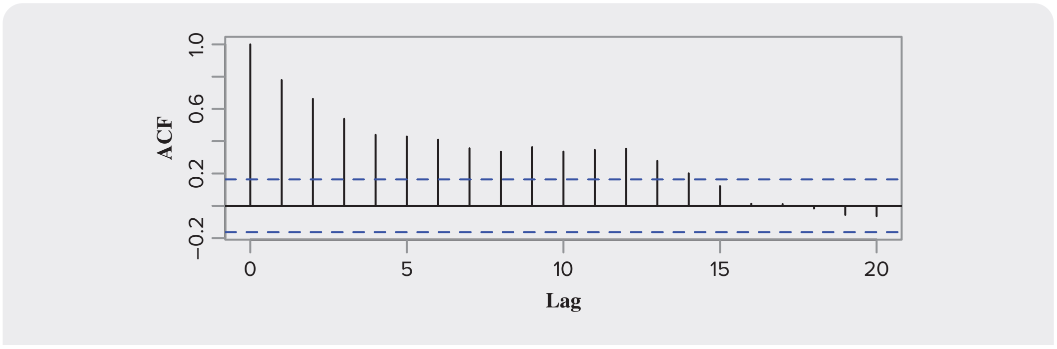


FIGURE 1.19 ACF for the residual time series shown in Figure 1.18. Note that acf(0) = 1 because this is the correlation between y_t and itself. The dashed horizontal line marks a rough calculation on the threshold for "significant" autocorrelations.

```
> plot(acf(fitAirMonth$residuals))
```

The plot confirms our visual inspection of the residual plots: there is significant correlation in the residuals. The correlation between y_t and y_{t-1} is around 0.8, which is pretty high. It indicates that 64% of the variation in e_t could be explained through a simple linear regression onto e_{t-1} (from 0.8^2 using our R^2 formula for linear regression).

How do you model this type of data? Consider a simple cumulative error process, where each ε_t is random with mean zero:

$$\begin{aligned} y_1 &= \varepsilon_1, \\ y_2 &= \varepsilon_1 + \varepsilon_2, \\ &\vdots \\ y_t &= \varepsilon_1 + \varepsilon_2 + \ldots + e_t \end{aligned}$$

Each y_t is a function of every previous observation all the way back to the first observation. This implies

$$y_t = \sum_{s=1}^{t} \varepsilon_s = y_{t-1} + \varepsilon_t \tag{1.37}$$

such that you can define y_t in terms of y_{t-1} and ε_t. This means that $\mathbb{E}[y_t|y_{t-1}] = y_{t-1}$ and all you need to know to predict t is what happened at $t - 1$. The model in (1.37) is called a *random walk*. It is defined by the fact that the expectation of what will happen next is always what happened most recently.

Random walks are one type of a general class of *autoregressive* (AR) models. In an autoregressive model of order one, you have

$$AR(1) : y_t = \beta_0 + \beta_1 y_{t-1} + \varepsilon_t \tag{1.38}$$

This is just a simple linear regression model, where y_t is the response and lagged y_{t-1} is the input. The random walk of (1.37) corresponds to $\beta_1 = 1$, and in a random walk any nonzero β_0 is referred to as *drift*. But β_1 can take all sorts of values, and you can complicate (1.38) by adding in whatever covariates that are also useful to predict y_t. Or, to think about it another way, you can add an AR(1) term to any regression where you suspect correlation between residuals.

Example 1.10 Airline Passenger Counts: Accounting for Autocorrelation To fit an AR(1) model, all you need to do is to create lagged values of your response and then include them in the regressions. To do this with the airline passenger counts, you can create a new column called `lag1` and fill it with the previous month's passenger counts.

```
> air$lag1 <- c(NA, air$Passengers[-nrow(air)])
> air[1:3,]
  Year Month Passengers       date t lag1
1   49     1        112 1949-01-01 1   NA
2   49     2        118 1949-02-01 2  112
3   49     3        132 1949-03-01 3  118
```

Notice that `lag1` for the first observation is empty (set to `NA`) because we don't know the passenger counts for December 1948. When we run our regression with an AR(1) term, we will want to exclude this first observation from the training data; `glm` actually does this automatically, because it skips observations containing `NA` values. Fitting the model is straightforward: you just add this lagged variable to your `R` formula (on log scale, to match the response).

```
> fitAirAR1 <- glm(log(Passengers) ~ log(lag1) + t + Month,
data=air)
> coef(fitAirAR1)["log(lag1)"]
log(lag1)
0.7930716
```

The resulting coefficient on the AR(1) term is 0.79. That means that each month's log counts are expected to be about 80% of the previous month's, before you add the linear and monthly trend effects. Figure 1.20 shows the resulting residuals and their ACF plot. Now, the residuals appear

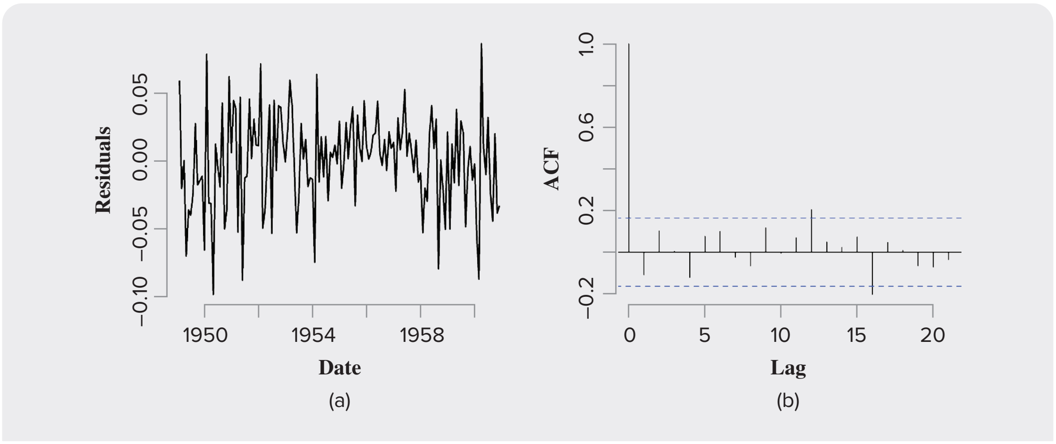


FIGURE 1.20 Residuals for the airline passenger regression that includes an AR(1) term and linear and monthly trends, showing them against time in (a) and their ACF in (b).

to be completely random from one month to the next (contrast with the residual time series in Figure 1.18). It appears the single lag term solved the bulk of our autocorrelation problems. For example, in Figure 1.20b there are no autocorrelations larger than 0.2 at any of the time lags.

Properties of AR Models

The AR(1) model is simple but hugely powerful. If you have any suspicion of autocorrelation, it is a good move to include lagged response as an input. The coefficient on this lag gives you important information about the time-series properties.

- If $|\beta_1| = 1$, you have a random walk.
- If $|\beta_1| > 1$, the series diverges and will move to very large or small values.
- If $|\beta_1| < 1$, the values are mean reverting.

Random Walk

In a random walk, the series just wanders around, and the autocorrelation stays high for a long time. See Figure 1.21. More precisely, the series is *nonstationary:* it has no average level that it wants to be near but rather diverges off into space. For example, consider the daily Dow Jones Average (DJA) composite index from 2000 to 2007, shown in Figure 1.22a. The DJA appears as though it is just wandering around. Sure enough, if you fit a regression model it looks like a random walk.

```
> dja <- read.csv("dja.csv")[,1]
> n<-length(dja)
> coef(ARdj <- glm(dja[-1] ~ dja[-n]))
(Intercept)     dja[-n]
   7.054185     0.997643
```

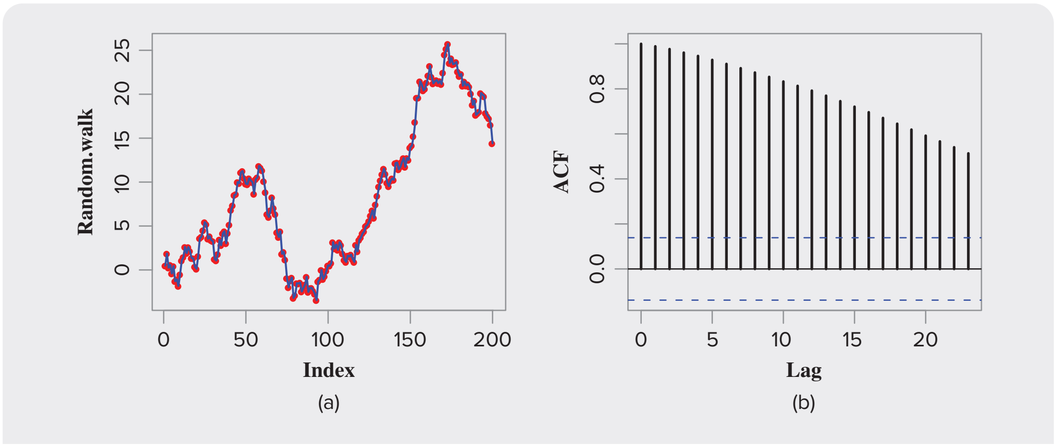


FIGURE 1.21 A simulated random walk and its ACF.

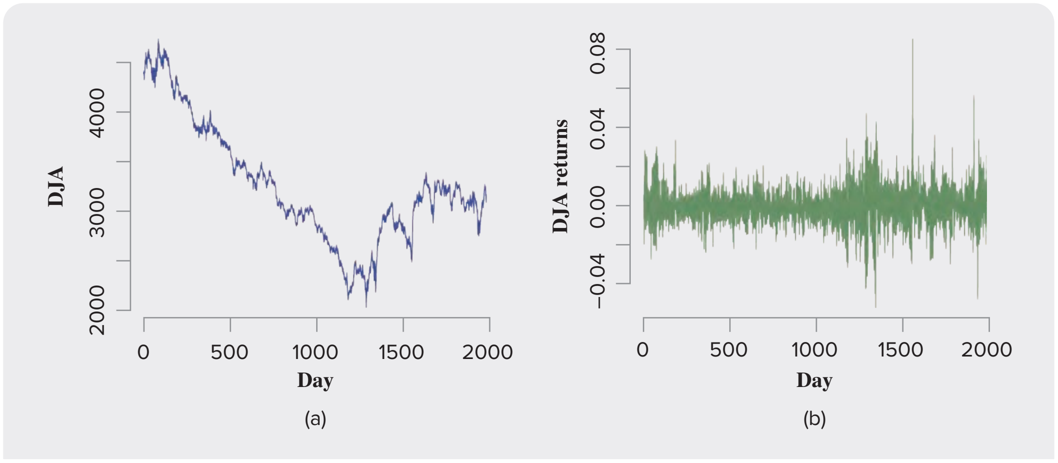


FIGURE 1.22 Dow Jones Average daily value (a) and returns (b) from 2000 to 2007.

The AR(1) term has a coefficient very near to one.

However, when we switch from prices to returns, $(y_t - y_{t-1})/y_{t-1}$, we get data that looks more like pure noise as shown in Figure 1.22b. Rerunning the regression on returns, we find that the AR(1) term is now very close to zero.

```
> returns <- (dja[-1]-dja[-n])/dja[-n]
> coef(glm(returns[-1] ~ returns[-(n-1)]))
    (Intercept) returns[-(n - 1)]
  -0.0001138386     -0.0144411430
```

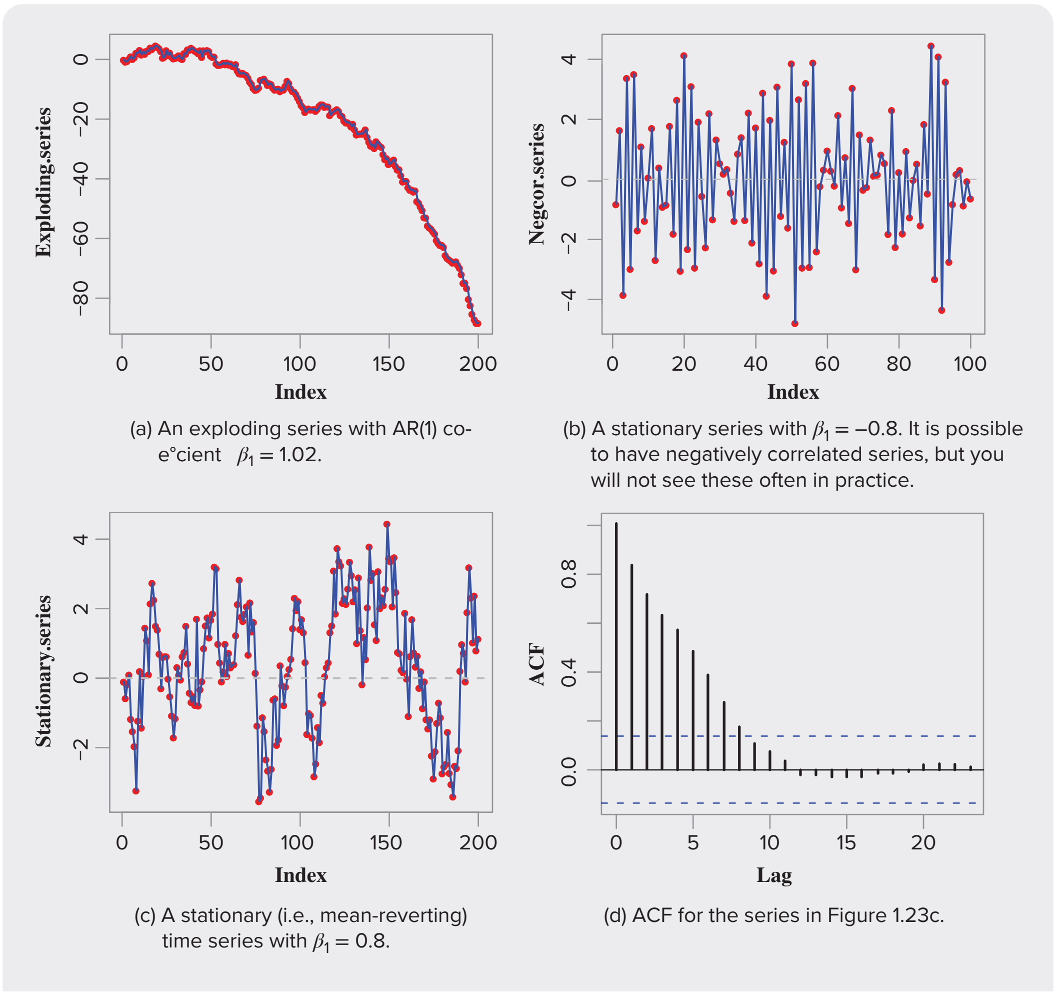
This property is implied by the series being a random walk: the *differences* between y_t and y_{t-1} are independent. If you have a random walk, you should perform this "returns" transformation to obtain something that is easier to model. For example, it is standard to model asset price series in terms of returns rather than raw prices.

Diverging Series

For AR(1) terms larger than one, life is more complicated. This case results in what is called a *diverging* series because the y_t values move exponentially far from y_1. For example, Figure 1.23a shows how quickly the observations diverge even for $\beta_1 = 1.02$, very close to one. Since these series explode, they are useless for modeling and prediction. If you run a regression and find such an AR(1) term, you are likely missing a trend variable that needs to be included in your regression.

Mean Reverting Series

Finally, the most interesting series have AR(1) terms between –1 and 1. These series are called *stationary* because y_t is always pulled back toward the mean. These are the most common, and most useful, type of AR series. The past matters in a stationary series, but with limited horizon and autocorrelation drops off rapidly.



(a) An exploding series with AR(1) co-e°cient $\beta_1 = 1.02$.

(b) A stationary series with $\beta_1 = -0.8$. It is possible to have negatively correlated series, but you will not see these often in practice.

(c) A stationary (i.e., mean-reverting) time series with $\beta_1 = 0.8$.

(d) ACF for the series in Figure 1.23c.

FIGURE 1.23 Various AR(1) time series examples.

An important property of stationary series is mean reversion. Think about shifting both y_t and y_{t-1} by their mean μ. A simple AR(1) model holds that

$$y_t - \mu = \beta_1(y_{t-1} - \mu) + \varepsilon_t$$

Since $|\beta_1| < 1$, y_t is expected to be closer to the μ than y_{t-1}. That is, each subsequent observation is expected to be closer to the mean than the previous one. Mean reversion is common and if you find an AR(1) coefficient between -1 and 1 it should give you some confidence that you have included the right trend variables and are modeling the right version of the response. The AR(1) component of our regression for log passenger counts was mean reverting, with each y_t expected to be 0.79 times the response for the previous month.

It is also possible to expand the AR idea to higher lags.

$$AR(p) : y_t = \beta_0 + \beta_1 y_{t-1} + \ldots \beta_p y_{t-p} + \varepsilon_t$$

The model selection and regularization methods of Chapter 3 make it straightforward to let the data choose the appropriate lags. Using those tools, you can feel free to consider bigger p in your $AR(p)$. The only problem is that the simple stationary versus nonstationary interpretations for β_1 no longer apply if you include higher lags. In addition, the need for higher lags sometimes indicates that you are missing a more persistent trend or seasonality in the data.

Before moving to the next section, it is worth emphasizing how we have been dealing with all sorts of time series dependence: we just engineer features to explain that dependence and include them in our regression models. It really is that easy. Some techniques based on data sampling, such as the bootstrapping or cross validation of next chapters, need to be adapted for dependent data. But with regression you have a great tool set for dealing with time dependence.

1.5.3 Panel Data

A common scenario for analysis has multiple time series together in a single dataset. You might have sales data over time for a number of different stores. Or, you might have a *longitudinal* survey where you ask a set of customers the same questions at a regular interval to see how their opinions change over time. This type of data—a stack of time series for multiple observation units—is called *panel data*. You have N units (e.g., stores or individuals) and a time series of length T_i for each unit. Your total number of observations is $n = \sum_{i=1}^{N} T_i$. When all of the time series are the same length, such that $Ti = T$ for all units, it is called a balanced panel; otherwise it is an unbalanced panel.

Panel data is especially common in economics applications, and econometricians have an extensive toolset for estimating models on this type of data. The literature in this area is pretty dense and jargon heavy, but the `plm` package for `R` (Croissant and Millo (2008)) is an extensively documented library of tools from econometrics for panel data analysis. Fortunately, with modern computing techniques (especially the regularization and sparse matrix tools from Chapter 3), you can do the state of the art of panel data analysis using standard regression models. As we've stated before in this chapter, time series analysis is just regression analysis where you have engineered and included some special features. The same holds true for panel data.

Example 1.11 Hass Avocado Panel Data We will introduce panel data analysis through an analysis of weekly sales by U.S. region for Hass avocados. Avocados have become enormously popular in recent years and the Hass variety is dominating the market. The Hass Avocado Board (HAB) was formed in 2002 to maintain and expand demand for avocados in the United States. The dataset `hass.csv` contains data from HAB. These data represent weekly retail sales of Hass avocados measured directly via cash register transactions. The data include sales from supercenters, club stores, national chains, regional chains, independent grocers, and the military. They do not reflect sales from farm stands, drug stores, or convenience outlets.

The data include unit sales and the average sales price per unit (ASP) aggregated by week and by region.

```
> hass<-read.csv("hass.csv",stringsAsFactors=TRUE)
> head(hass,3)
  region       date  asp     units
```

```
1 Albany 12/27/2015 1.33  64236.62
2 Albany 12/20/2015 1.35  54876.98
3 Albany 12/13/2015 0.93 118220.22
```

The first things we will do are to convert the date into a `Date` variable, being careful to check the date formatting, and then *order* the data frame by region and date:

```
> hass$date <- as.Date(hass$date,format='%m/%d/%Y')
> hass<-hass[order(hass$region,hass$date),]
> head(hass,3)
   region       date  asp    units
52 Albany 2015-01-04 1.22 40873.28
51 Albany 2015-01-11 1.24 41195.08
50 Albany 2015-01-18 1.17 44511.28
```

This step of ordering by unit and date is a good practice with panel data, since it helps you stay organized during data manipulation. It will be essential for our calculation of lagged variables below.

We will investigate the sales-price elasticity of avocados by regressing log units sold onto log ASP. The most basic panel data model doesn't include any real time series modeling: you just regress your response of interest (log(`units`)) on the explanatory variables interest (log(`asp`)). (Recall our earlier discussion of Log-Log models and price elasticity around Equation 1.8.) The key step is that you need to account for the fact that your data are *grouped* according to the panel units—in our case grouped by region. You will expect different average weekly avocado sales in Boise than you would get in Los Angeles. You deal with this by including the regions as factor effects in the regression. The model is then

$$\log(\texttt{units}_{it}) = \alpha_i + \beta \log(\texttt{asp}_{it}) + \varepsilon_{it} \tag{1.39}$$

The subscript indexing here is important: x_{it} indicates the value at the t^{th} time series observation for the i^{th} region. This double indexing references a unique row of the `hass` data frame. However, the region effects α_i are only indexed by region, such that every time series observation from the same region is fit as having this region-specific mean. These α_i are called *fixed effects* in a panel data setting. Including them is equivalent to removing the regional mean from your sales data before fitting the regression. With panel data it is important that you include these fixed effects; otherwise, the difference in baseline sales between, say, Boise and Los Angeles will be included in your estimate of β.

We'll fit this model with `glm`.

```
> fitHass <- glm(log(units) ~ log(asp) + region, data=hass)
> coef(fitHass)["log(asp)"]
  log(asp)
-0.7283443
```

This estimated elasticity implies that expected unit sales drop by 0.73% for every 1% increase in ASP. This is a fine result in the context of the model we have specified, but it is suspicious if you want to interpret β in terms of what grocers experience in terms of lost or gained sales when they change prices. In particular, elasticities greater than -1 indicate what economists call inelastic products. For such products, the grocers could increase *revenue* (i.e., total sales before costs) by increasing prices. Finding $\beta = -0.73$ as the sales-price elasticity for avocados, which suggests that they are an inelastic good, is surprising. If you are interested to learn more, see the nearby box on this topic. Regardless, we will find soon that if we control for additional influences on sales, then avocados no longer appear inelastic.

Pricing, Elasticities, and the Limits of Log-Log Regression

First, we note that the economics literature on pricing and sales-price elasticities is deep and complex. There is no single framework that describes to any degree of fidelity how firms set prices, and the prices you experience as a consumer are driven by a massive variety of influences. Simple log-log regression models like those we use throughout this chapter measure the *short-term* elasticity to *local* changes in price for a *single* product. This means that a number of price-demand effects are not included without further modeling.

- The long-term elasticity to prices can be different from the short term. If your store increases prices consumers may still buy the items out of convenience, but eventually they will shift their entire shopping trip to a store that offers better value across a broad range of items. Your log-log regression doesn't capture this long-term view, and a firm that optimizes too much for short-term profit risks losing customers in the long term.
- Our regression models measure the immediate customer sales variation as a function of relatively small price changes around the average price (i.e., within the range of observed price variation). If the average price increases by a large amount (e.g., if avocados suddenly doubled in average price) then the measured elasticities would likely change. You should view your measured elasticities from a simple log-log model as applicable only within the observed range of current prices.
- Stores sell many products. Some may be heavily discounted to get people into the store, or because they are "basket builders" that encourage people to buy other (more profitable) products. In a more sophisticated demand analysis, it is common to incorporate *cross-price* elasticities that measure how changes in price on one product (e.g., pasta) influence the sales on complementary products (e.g., pasta sauce).

Other issues include the impact of supply (if you have too many avocados you might discount them to avoid spoilage), temporal substitution (if you discount one day, then customers can stock up and buy less in the future), and product substitution (price changes on one product can cause people to switch to or from another item).

Despite these limitations, the log-log regressions of this chapter are a common and useful tool for understanding pricing and demand. To see why we say that -0.73 is a suspicious elasticity for avocados, consider that a sales-price elasticity greater than -1 implies that a grocer could raise total revenue by raising prices. To see this, take Equation (1.39) and do a bit of basic algebra to write

$$\begin{aligned} &\log(\texttt{units}_{it}) = \alpha_i + \beta \log(\texttt{asp}_{it}) + \log(\texttt{asp}_{it}) - \log(\texttt{asp}_{it}) + \varepsilon_{it} \\ &\Rightarrow \log(\texttt{units}_{it} \times \texttt{asp}_{it}) = \alpha_i + (\beta + 1)\log(\texttt{asp}_{it}) + \varepsilon_{it} \end{aligned} \tag{1.40}$$

Now, `units × asp` is your total revenue (units sold times price). And if $\beta > -1$, then (1.40) leads to $\beta + 1 > 0$ and you have a *positive* revenue-price elasticity. If we apply this elasticity to set prices at each store, it implies that the grocers can increase *revenue* by increasing prices.

Grocers are not attempting to maximize short-term *profit* (revenue minus costs) on every item: they need to worry about long-term customer retention and the cross-price elasticities. However, it is a bit unusual to see average prices so low that they could be raised without negatively impacting revenue. Unless cheap avocados play an outsized role in basket building or attracting people to shop, then the inelastic demand implied by $\hat{\beta} = -0.73$ suggests that average prices might drift upward until consumers start to become more price sensitive. More likely, however, is that our elasticity here is "polluted" by effects on sales separate from price.

To learn more about pricing and demand, look to Chapter 6 where we discuss a higher dimensional log-log elasticity regression in the context of beer pricing.

Two-Way Fixed Effects

The likely issue here is that we are missing an underlying variable that is correlated with both prices and sales. Such "omitted variables" can make it difficult to interpret the estimated relationships in your regression (like the one between price and sales). In panel data, you can mitigate this issue by including *fixed time effects*. If we include fixed effects for each week then we will be controlling for events or seasonal effects that impact both the price and sales of avocados. For example, events such as the Super Bowl cause Americans to eat a lot of avocados (guacamole!) and grocers will want to increase prices on their limited supply, leading to a positive relationship between price and sales. If we include a fixed effect for the Super Bowl week, this positive relationship will be explained by that effect rather than being incorporated into our price elasticity estimate.

We have now talked about having a region fixed effect for each time series in our panel and having a week fixed effect for each time point across all series. The term "fixed effect" might be a bit confusing if you haven't seen it before. There is nothing complicated going on here: we are simply including additional factor variables into our regression (a factor for region and a factor for week). The "fixed" label is used in panel data analysis to differentiate from so-called random effects, where you allow for correlations between the error terms in your analysis. We consider this type of dependence between errors in the next chapter as part of our uncertainty quantification. However, this is not a replacement for including the proper fixed effects in your regression specification. These fixed effects play a crucial role in accounting for the influence of unobserved factors, such as different tastes or budgets across regions or the Super Bowl effect described above.

Getting back to our avocado sales analysis, an alternative model with week fixed effects is

$$\log(\texttt{units}_{it}) = \alpha_i + \delta_t + \beta \log(\texttt{asp}_{it}) + \varepsilon_{it} \tag{1.41}$$

Here, δ_t is the fixed effect for week t.

To fit this in R, you create a factor variable for *week* and add it to the R formula.

```
> hass$week <- factor(hass$date)
> fitHassDF <- glm(log(units) ~ log(asp) + region + week,
data=hass)
> coef(fitHassDF)[“log(asp)”]
  log(asp)
-0.9465598
```

Sure enough, the elasticity is now close enough to -1 to seem plausible. At a sales-price elasticity of -1, the grocers can't make more revenue by raising prices.

Note that the model in Equation (1.41) is often referred to as the *two-way fixed effects model.* It is very common in economic analysis, since it allows you to control for unobserved influences in both time (t) and region (i). Since it is such a common model, we caution you to remember that it is not magic and you can still get bad estimates if, for example, the elasticities are different for each region or if there is an unmodeled time trend in each region (e.g., if sales are increasing at different linear rates in each region). As we've said before: panel data analysis is just applied regression, so you need to think about the process you are trying to model and not assume that you can just fit a common model and interpret the fitted values the way you want.

Adding AR Terms

We can further improve the model by including lagged variables. First, we can include lagged log unit sales to account for the autoregressive correlation across weeks in the same region. Just like the weather, this week's avocado sales in your city are almost certainly correlated with last week's sales (even after controlling for price and the major regional or weekly trends). But we can also include the *lagged log price effect.* Such lagged price effects are often a good idea to include because of *pull forward in demand.* If you have a deal on avocados, then customers will stock up, and the next week they won't buy avocados because their pantry is already full.

The full regression model is then

$$\log(\texttt{units}_{it}) = \alpha_i + \delta_t + \beta_1 \log(\texttt{units}_{i,t-1}) + \beta_2 \log(\texttt{asp}_{it}) + \beta_3 \log(\texttt{asp}_{i,t-1}) + \varepsilon_{it} \qquad \textbf{(1.42)}$$

Calculating the lagged variables for panel data takes a bit of care. We need to group the data into its region-level time series and calculate the lags for each individual time series. We will use the `tapply` function to do this. `tapply` takes any vector as its first argument, splits the vector up according to the factor levels in its second argument, and then applies to the splits whatever function you give as its third argument. Hence, we can use it to calculate things like the maximum for each subgroup.

```
> tapply(c(1:5), c(“a”,“b”,“a”,“b”,“c”), function(x) max(x))
a b c
3 4 5
```

Since we *ordered* the `hass` data frame in the beginning of this example (which is crucial for this to work), we can use `tapply` to split `asp` and `units` into their region-level time series and create lagged variables within each region (same as we did to create lagged passengers for the airline example).

```
> hass$lag.asp <- unlist(tapply(hass$asp, hass$region,
+                     function(x) c(NA,x[-length(x)])))
> hass$lag.units <- unlist(tapply(hass$units, hass$region,
+                     function(x) c(NA,x[-length(x)])))
> head(hass,3)
   region       date  asp    units lag.asp lag.units
52 Albany 2015-01-04 1.22 40873.28      NA        NA
51 Albany 2015-01-11 1.24 41195.08    1.22  40873.28
50 Albany 2015-01-18 1.17 44511.28    1.24  41195.08
```

The `unlist` command is necessary because `tapply` returns here a list of vectors, one for each lagged time series, and you want to collapse them into a single vector. You can confirm by inspection that we have things lined up properly. For example, the avocado ASP in Albany for the week of January 4 is $1.22, and this is also the lagged avocado ASP for the week of January 11.

Fitting the model with `glm`, you can see that the estimate of price sensitivity has increased:

```
> fitHassLags <- glm(log(units) ~ log(lag.units) + log(asp)
+                      + log(lag.asp) + region, data=hass)
> coef(fitHassLags)[2:4]
log(lag.units)      log(asp)   log(lag.asp)
     0.7734225     -1.4025468      1.2631973
```

The model estimate says that Hass unit sales decrease by about 1.4% per every 1% price increase. Notice that the AR(1) term on lagged units is 0.77, indicating a stationary and mean reverting autocorrelation process. Finally, the effect of lagged log ASP is positive. Interpreting this through the lens of pull forward in demand, expected unit sales in a given week *drop* by 1.3% per every 1% price *decrease* in the week prior. That is the effect of customers stocking up on those cheap avocados.

1.6 Spatial Data

We've shown that you can model all sorts of time series dependence using basic regression tools. This same lesson applies to spatial dependence: space is just like time but with another dimension. The same as you can include monthly or weekly factors in your regressions, you can include geographic factors like region or city. This approach of adding *spatial fixed effects* will be your main tool in managing dependence in spatial data. To see how this works in practice, consider the example of our next chapter: we will be modeling the listing price for used cars, and always include the `city` where a car is listed as a spatial fixed effect.

Modeling autocorrelation in spatial data is a bit more complicated than it is for time series. Whereas time series are *ordered,* spatial data is not. In AR models the current observation is regressed onto the previous observation, but with spatial data there is no simple notion of "previous." There do exist spatial autoregressive (SAR) models, where each observation is regressed onto the observations of its "neighbors." For example, when processing image data you can regress the value at one pixel onto the average of the neighboring pixels. These SAR models work fine for spatial data that is observed on a regular *grid,* such that each observation is evenly spaced from its neighbors. Unfortunately, most examples of spatial data that we encounter in practice do not live on a nice grid. Instead, you need models for autocorrelation that allow for the observations to be unevenly spaced from each other.

1.6.1 Gaussian Process Modeling

Gaussian processes (GPs) are the dominant modeling framework for spatially dependent data. GPs are models that smooth predictions across observations according to distances between their locations. They are a relatively simple example of the sort of *stochastic process* models that are commonly used for such purposes. However, to describe GPs we need to talk about multivariate distributions that describe the *joint* distribution for multiple observations.

Suppose that you have two response observations y_i and y_j observed at two different locations with coordinates $\mathbf{s}_i$ and $\mathbf{s}_j$. These can be latitude and longitude coordinates, or they can correspond to any other spatial coordinate system that makes sense for your data. A Gaussian process models the responses at two locations as draws from a Gaussian distribution:

$$\begin{bmatrix} y_i \\ y_j \end{bmatrix} \sim \mathrm{N}\left(\begin{bmatrix} \mu_i \\ \mu_j \end{bmatrix}, \sigma^2 \begin{bmatrix} 1+\delta & \kappa(\mathbf{s}_i, \mathbf{s}_j) \\ \kappa(\mathbf{s}_i, \mathbf{s}_j) & 1+\delta \end{bmatrix} \right) \qquad \textbf{(1.43)}$$

Equation (1.43) is a *multivariate* distribution. It describes the expectation (μ_i and μ_j) and variance ($\sigma^2(1 + \delta)$) for each variable, as well as the covariance between the responses. The expectations can be functions of covariates, say $\mu_i = \mathbf{x}_i'\boldsymbol{\beta}$, or a simple mean such that $\mu_i = \mu_j = \mu$. The variance is determined by σ^2 and the *nugget* δ. This nugget term measures how much variance you will have for repeated observations *at the same location*. If you have two observations at the same spatial location, then δ represents the amount that they tend to differ from each other.

The covariance between observations is determined by $\kappa(\mathbf{s}_i, \mathbf{s}_j)$, the *kernel function.* It defines the correlation between the corresponding responses, such that

$$\mathrm{cor}(y_t, y_s) = \frac{\kappa(\mathbf{s}_i, \mathbf{s}_j)}{1+\delta} \qquad \textbf{(1.44)}$$

and the *covariance* between these responses is $\sigma^2\kappa,(\mathbf{s}_i, \mathbf{s}_j)$. The form for this kernel function dictates how the GP models spatial correlation. For example, the common *exponential* kernel function is

$$\kappa(\mathbf{s}_i, \mathbf{s}_j) = \frac{1}{\exp\left[\left(\frac{(s_{i1}-s_{j1})^2}{\rho_1} + \frac{(s_{i2}-s_{j2})^2}{\rho_2}\right)\right]} \qquad \textbf{(1.45)}$$

Here, correlation decreases with the exponentiated distance between locations. The *range* parameters, ρ_1 and ρ_2, allow for different units of distance in your two coordinates (say lat and long). The kernel from (1.45) results in smoothly decaying dependence between responses, y_i and y_j, as a function of distance between inputs, $\mathbf{s}_i$ and $\mathbf{s}_j$. Note that this $\kappa(\cdot,\cdot)$ produces values between zero and one, since $\kappa(\mathbf{s}_i, \mathbf{s}_j) = 1$ if $\mathbf{s}_i = \mathbf{s}_j$ and it approaches zero as the locations get further apart.

GP Predictions

Predictions from a GP combine information in the mean function with the information from observations at nearby locations. Consider prediction (forecasting) at a new location $\mathbf{s}_f$ given a single observation y_i at location $\mathbf{s}_i$. The conditional expectation for the response at the new location is

$$\mathbb{E}[y_f | y_i] = \mu_f + \frac{\kappa(\mathbf{s}_f, \mathbf{s}_i)}{1+\delta}(y_i - \mu_i) \tag{1.46}$$

This shows the prediction will be a combination of the mean for the new observation, μ_f, and the correlation between y_f and y_i multiplied by the residual error $y_i - \mu_i$. When the residual error is positive, then the residual at y_f is also expected to be positive. The strength of this relationship is determined by the kernel function and the distance between locations. When you have many observations, the analogue to (1.46) has that the prediction for y_f is a function of the *matrix* of correlations between y_f and those observations and the *vector* of residual errors.

Estimating and predicting from GP models is not a simple exercise. You need to consider the correlations between all pairs of locations in your data in order to estimate the range, variance, and mean parameters. And you need to do the matrix-vector calculations to calculate predictions that depend upon the correlations with residuals errors—that is, to calculate the n-observation analogue to (1.46). This gets computationally expensive when you have many observations (many locations) in your dataset. For big datasets, an efficient technique is to consider a subsample of neighboring observations for each location where you want to predict the response. That is, if you want to predict at new location $\mathbf{s}_k$, then you fit a *local* GP that only uses a fixed number of observations at locations near to $\mathbf{s}_k$.

Fitting GPs with `laGP`

To fit such local GPs in `R`, you can make use of the `laGP` package (Gramacy, 2015). It contains the `aGP` function which takes as arguments your observed locations and responses, and a set of new locations where you want to get predictions. The algorithm goes through each location where you want to predict, and fits a local GP model to the set of nearby locations. The argument `end` sets the number of observations used to estimate each local GP; if you set `end` bigger, the algorithm takes longer to run but will consider a wider set of neighbors for each location. The `aGP` function uses the efficient search algorithm described in Gramacy and Apley (2015) to choose the most useful neighbors for use in each local GP fit. Detailed examples are provided in the vignette of Gramacy (2015) and in the package documentation.

Example 1.12 California Census Data: Gaussian Processes To illustrate GP estimation, we will look at some (old) census data from California. In the `CalCensus.csv` dataset you have the longitude and latitude for the center of each census tract in the state, along with some statistics for that census tract. We will be focusing on the median values for income and home value within each census tract, and investigating the relationship between them. Note that each of these metrics has been *trimmed* to replace very large values with a threshold: income is thresholded at \$150k and home value is thresholded at \$500k.

```
> ca <- read.csv("CalCensus.csv")
> ca[1,]
  longitude latitude housingMedianAge population households
1   -122.23    37.88               41        322        126
  medianIncome medianHouseValue AveBedrms AveRooms AveOccupancy
1        83252           452600   1.02381 6.984127     2.555556
```

To understand the relationship between house prices and income, we can calculate the *elasticity* between them using a log-log regression.

```
> linc <- log(ca[,"medianIncome"])
> lhval <- log(ca[,"medianHouseValue"])
> summary(glm(lhval ~ linc))
...
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 3.569235   0.065110   54.82   <2e-16 ***
linc        0.814520   0.006221  130.92   <2e-16 ***
```

Using what we know about interpreting coefficients in log-log regression, the estimated coefficient on `linc` says that median home prices should increase by about 0.8% per every 1% increase in income.

We might want to be skeptical about that result, however. There is a likely dependence in the home values across neighboring census tracts: people can easily commute from their home in one census tract to their work in another one. Indeed, Figure 1.24 shows a strong pattern of spatial correlation in both incomes and home values across California (it also shows that the highest home prices are tightly concentrated in the SF Bay Area and along the coast—the color maps here are calculated on percentiles of the distributions—while incomes are more evenly distributed). Thus the estimated relationship between income and home values here might be *polluted* by the incomes and home values in neighboring census tracts. This is the same as how you can get polluted estimates for regression coefficients in time series if you do not control for autocorrelation.

To control for this spatial dependence, you can use a GP to model the error structure for log home value conditional upon log income. However, the `laGP` package doesn't allow for regression functions to specify the response mean; it applies a single fixed mean, such that $\mu_i = \mu_j = \mu$ in our notation of Equation (1.43). In a preview of Chapter 6, however, we can proceed by using GP predictions to first *residualize* both the income and home values against their predicted value given the spatial correlation between observations. That is, we'll use a GP model to obtain fitted values for income and home value, calculate the residuals (the

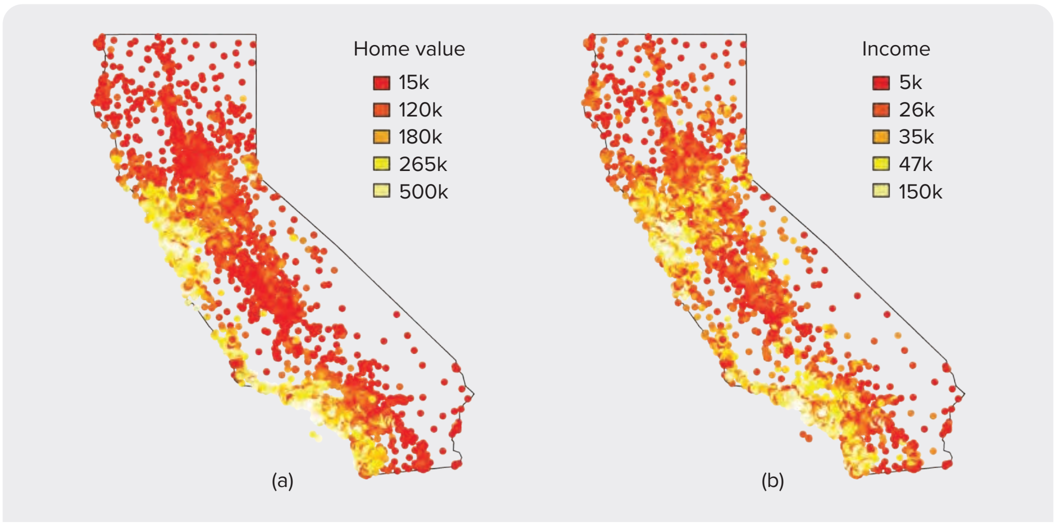


FIGURE 1.24 Median home value and income by census tract.

difference in the fitted and observed values), and then regress the residuals for home value onto the residuals for income. The regression relationship between these residuals then tells you the elasticity for home values on incomes *after* having controlled for spatial dependence across census tracts.

Say that y denotes log median home values (`lhval`) and d denotes the log median incomes (`linc`), and the full samples for each are $\mathbf{y}$ and $\mathbf{d}$. Then we will use GP models to estimate $\hat{y}_i = \mathbb{E}[y_i|\mathbf{y}]$ and $\hat{d}_i = \mathbb{E}[d_i|\mathbf{d}]$. The residuals are calculated as $\tilde{y}_i = y_i - \hat{y}_i$ and $\tilde{d}_i = d_i - \hat{d}_i$ for each census tract i. The final regression for $\tilde{y}_i$ onto $\tilde{d}_i$ gives us an unpolluted estimate of the elasticity between incomes and home prices. Note that if you find this residualization confusing, you can ignore the motivation here and just think about fitting two GPs: one to predict income and the other to predict home values.

To fit the GPs with `aGP`, you pull out `longitude` and `latitude` as the spatial coordinates both for the current observations and the locations where you want to predict. We use `end=20` here so that each fitted GP will use 20 local census tracts.

```
> library(laGP)
> s <- ca[,1:2] # long and lat
> gpinc <- aGP(s, linc, XX=s, end=20)
> gphval <- aGP(s, lhval, XX=s, end=20)
```

When you run this, it will take several minutes and print a lot of information. Recall that `aGP` is fitting a unique GP for each of the locations where you want to predict—in this case, 20640 census tracts. The `laGP` package is actually very sophisticated under the hood, and if you "compile" (i.e., turn the code into an executable computer program) correctly it can execute each local GP in parallel across the many processors on your computer. However, this will be tricky if you are not familiar with these tools (it is not as easy as loading the `parallel`

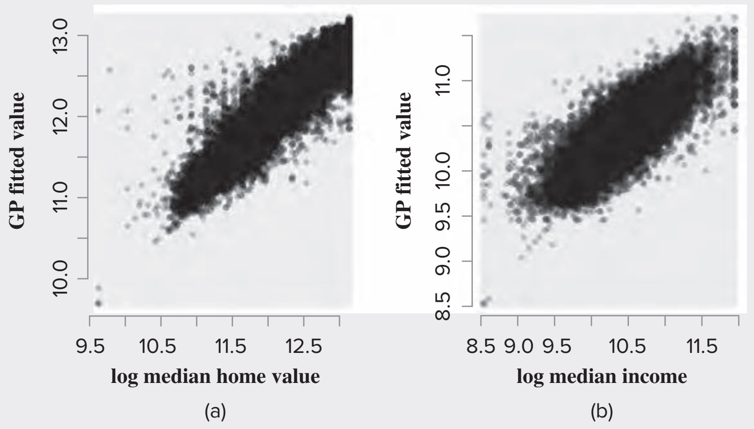


FIGURE 1.25 Log median home value and income, fitted values from `laGP` plotted against the observed values.

library). You can run the code here without fancy parallel tricks and make yourself some coffee or tea while it runs.

The resulting predictions at `XX` are in the `mean` entry of the fitted `aGP` objects. The fitted values are plotted in Figure 1.25. Remember: these predictions are based on only the dependence between nearby census tracts. We use these fitted values to calculate residuals for each of *y* and *d* and regress the residuals onto each other. Note that you will get slightly different output since aGP is taking a random sample from the posterior distribution (refer to the Bayes section in the next chapter).

```
> rinc <- linc - gpinc$mean
> rhval <- lhval - gphval$mean
> summary(glm(rhval ~ rinc))
...
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.001351   0.001427  -0.947    0.344
rinc         0.353034   0.004942  71.438   <2e-16 ***
```

The estimated elasticity is less than half of our previous estimate. Controlling for spatial dependence through the GP-based residualization leads us to conclude that home values rise by about 0.4% per every 1% increase in income. It is intuitive that controlling for spatial dependence reduces the elasticity. If incomes increase for jobs in one census tract then the influence on home price will be spread over a large regional area (people can commute and move) rather than concentrated in a single census tract. There will also be correlated spatial effects of, say, school districts and neighborhood appeal.

The models we've fit here are the simplest form of GPs: they have a single fixed mean parameter μ and a shared homoskedastic error structure (same σ^2 and δ for every observation). The `tgp` package implements a much wider array of GP models, including those that specify linear regression functions as the mean and even models that use regression trees to split the input space and fit a different GP within each partition (these are the Treed Gaussian Processes of Gramacy and Lee 2008). Although the code and ideas are beyond the scope of this book, you can look to Gramacy (2007) and Gramacy and Taddy (2010) for lengthy vignettes illustrating the capabilities of `tgp` and use that as a stepping stone for more complex analyses of spatial data.

QUICK REFERENCE

This chapter moves rapidly through the methods of linear and logistic regression, and explains how these methods are both applications of maximum likelihood estimation. The connection between likelihood maximization and deviance minimization will be important for future chapters where we consider the deviance as a part of more complex "loss functions" that are estimated as part of machine learning. We also introduce the basic concepts of time series analysis, with the main point being that regression techniques can be applied to analyze data that is correlated across time.

Key Practical Concepts

- To fit a linear regression in `R`, where your data frame `data` contains response `y` and inputs `1`, `x2`, etc, you use `glm`.

  ```
  fit <- glm( y ~ x1 + x2, data=data)
  ```

  Other formula options are in Table 1.2.
- To fit a logistic regression, for when your `y` is binary, logical, or a two-level factor, you just add `family="binomial"`.

  ```
  fit <- glm( y ~ x1 + x2, data=data, family="binomial")
  ```

- Calling `summary(fit)` returns a summary of the model and coefficient estimation, and `coef(fit)` just returns the regression coefficients.
- For prediction, with new data in `newdata` with the same variable names as the `data` used to fit your regression, use the `predict` function.

  ```
  predict(fit, newdata=newdata)
  ```

  This returns the linear equation predictions $\mathbf{x}'\hat{\boldsymbol{\beta}}$. If you have fit logistic regression and you instead want predicted probabilities (after logit transformation), then you need to add the argument `type="response"`.
- The `glm` object includes the residual `deviance` and null model `null.deviance`. The R^2 is then available as

  ```
  1-fit$deviance/fit$null.deviance
  ```

- To deal with time dependence, you can
  1. Create factor variables like `month` to represent time fixed effects.
  2. Add numeric variables like `day` to allow for time trends.
  3. Create lagged variables y_{t-1} as inputs to allow for autoregressive errors.

- Panel data has time series for many units together in the same dataset. You typically want to include fixed effects both for each unit and each time period. You can use `tapply` to calculate different lags for each time series.
- Use the `laGP` package to fit a Gaussian Process (GP) that models spatial dependence in data. A useful strategy is to use the residuals from a GP as data for a downstream analysis—these residuals will have no spatial dependence on each other so you can use standard regression methods for analysis.

3 REGULARIZATION AND SELECTION

This chapter lays out the modern approach to building models from data: use regularization to obtain paths of candidate models, and use estimates of *out-of-sample* predictive performance to choose the best model.

● **Section 3.1 Out-of-Sample Performance:** Understand the concepts of overfitting and underfitting a model and how to avoid both situations. Run *cross-validation* experiments to test out-of-sample predictive performance of candidate models.

■ **Section 3.2 Building Candidate Models:** Estimate models to minimize a penalized deviance, introducing *regularization* that helps you avoid overfit by putting a price on model complexity. Build paths of models estimated under a sequence using Lasso penalties.

■ **Section 3.3 Model Selection:** Use cross-validation experiments and information criteria such as AICc to select the best model from a Lasso path.

◆ **Section 3.4 Uncertainty Quantification for the Lasso:** Adapt bootstrap algorithms to estimate sampling distributions for estimated parameters in models selected from a Lasso path.

When you are estimating a model from data, your goal is always to use as much of the true *signal* as possible to estimate your parameters. In doing this, you need to guard against the *noise* in the data introducing noise in your estimates. Noise here can be due to error terms or (in regression) the presence of spurious covariates that are not useful for predicting the response. You need to use methods that pull apart the signal from the noise.

This is especially important in the high-dimensional (many parameter) settings that you will encounter in data science. Today's companies—whether in services, technology, manufacturing, or any other sector—have the opportunity to collect data on a huge number of signals related to process improvement and product performance. Analyzing these signals requires the use of high-dimensional models. These models have the flexibility to recognize complex patterns, but they expose you to the risk of *overfit:* tuning the model to predict random errors in your current sample rather than to predict the signal that will persist in new samples.

This chapter introduces the tools of *regularization* and *model selection* that allow you to separate signal from noise and find the best model among many high-dimensional possibilities. The basic principle is that you want to build models that predict well *out-of-sample:* when you apply them to predict what will happen in the future, they give results that are accurate and robust. More specifically, using the language of Chapter 1, you want models that will have a low deviance when tested on data beyond your training sample. This chapter will give you the recipes for constructing such models.

3.1 Out-of-Sample Performance

To understand the need to separate signal from noise, consider the issues of *overfit* and *underfit* for regression on some simple simulated data. Figure 3.1 shows three different models fit to data generated from a quadratic model with Gaussian errors ($y = \beta_0 + \beta_1 x + \beta_2 x^2 + \varepsilon$). A simple linear model fit to this data is underfit: it misses the curvature in the underlying true mean

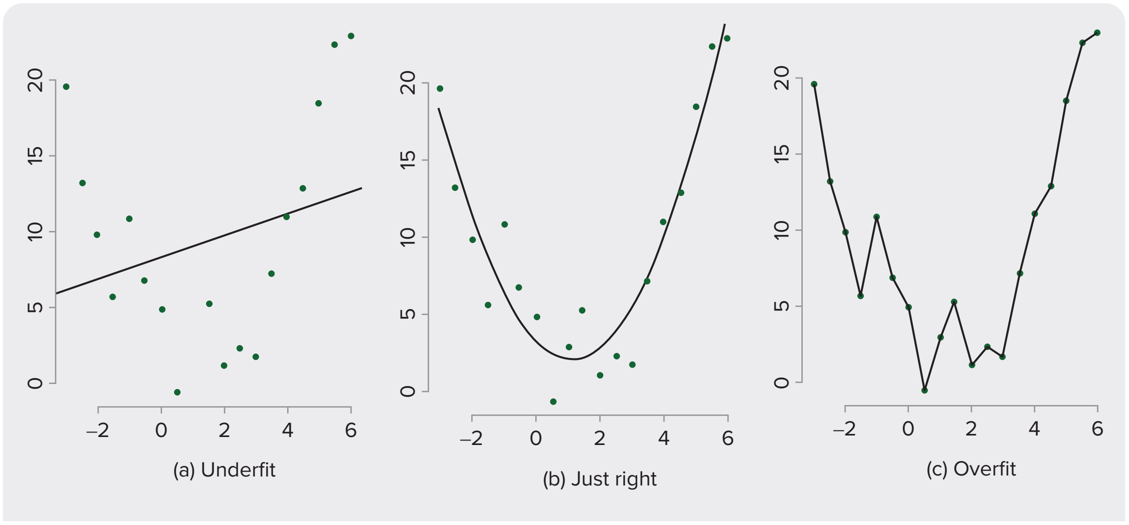


FIGURE 3.1 Data were generated from a quadratic model $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \varepsilon$ where ε is the random noise (error) term generated from a Gaussian distribution. Panel (a) demonstrates *underfit* by fitting the simple linear regression $\mathbb{E}[y] = \beta_0 + \beta_1 x$, panel (b) shows the fit for the true quadratic model, and panel (c) demonstrates *overfit* when simply connecting the dots.

function. The model in Figure 3.1c interpolates each point exactly (i.e., connects the dots) and confuses the sample error (ε) for true signal. This model does a worse job predicting new observations than a simpler model.

In Chapter 1, we introduced *deviance* as a measure of how tightly your model fits your sample of data. The data that has been used to estimate the model is called the *training* sample, and the deviance you calculate on the training data is called the *in-sample* (IS) deviance. If we calculate the in-sample deviance for the model in Figure 3.1c, it will be zero: this interpolator fits the data perfectly. But when we want to use a model for prediction on *new data* (i.e., to predict the y for new x) the model fit will be no longer be perfect. The noise in new observations will be different from the noise in your training sample, and the jagged fitted surface in Figure 3.1c will lead to predictions that are far from the new observations. The underfit model in Figure 3.1a and the "just right" model in Figure 3.1b will also not have predictions that line up perfectly with new observations. When deciding which model is "best," we want to evaluate which model will have the lowest *out-of-sample* (OOS) deviance: the lowest deviance when evaluated against new data.

In-sample vs. Out-of-Sample Deviance

Recall from Chapter 1 that we defined the regression R^2 as one minus the ratio of the deviance for your fitted model over the deviance for a *null model* (i.e., the model with no input variables such that you are estimating $\mathbb{E}[y] = \beta_0$). This is our measure of "the proportion of deviance explained by your fitted model." The same as how we have both in-sample deviance (that calculated on the training data) and out-of-sample deviance (that calculated on new, or *test,* data), you have both in- and out-of-sample R^2. And the only R^2 you ever really care about in practice is the OOS R^2.

Suppose that you have data $[\mathbf{x}_1, y_1] \dots [\mathbf{x}_n, y_n]$ and you use this data to fit $\hat{\boldsymbol{\beta}}$ in a linear regression model $\mathbb{E}[y] = \mathbf{x}'\boldsymbol{\beta}$. The in-sample deviance is then

$$\text{dev}_{IS}(\hat{\boldsymbol{\beta}}) = \sum_{i=1}^{n} (y_i - \mathbf{x}_i'\hat{\boldsymbol{\beta}})^2 \tag{3.1}$$

For out-of-sample deviance, $\hat{\boldsymbol{\beta}}$ is the same (still fit with observations $1 \dots n$), but the deviance is now calculated over m new observations (say $n + 1, \dots, n + m$):

$$\text{dev}_{OOS}(\hat{\boldsymbol{\beta}}) = \sum_{i=n+1}^{n+m} (y_i - \mathbf{x}_i'\hat{\boldsymbol{\beta}})^2$$

Similarly, when calculating the null deviance for R^2 you will look at $(y_i - \bar{y})^2$ where $\bar{y}$ is the response mean in the training data; for IS null deviance you sum these errors over $i = 1, \dots, n$ and for OOS null deviance you sum over $n + 1, \dots, n + m$. And for other models you just swap out squared errors for the appropriate deviance function (e.g., the logistic deviance of Equation (1.33) when doing logistic regression). The distinction between IS and OOS deviance (and R^2) is massively important. When you have many input variables it is easy to *overfit* the training data so that your model is being driven by noise that will not be replicated in new observations. That adds errors to your predictions, and it is possible that the overfit model becomes worse than no model at all.

Example 3.1 Semiconductor Manufacturing: OOS Validation As a real data example, let's consider quality-control data from a semiconductor manufacturing process. This industrial setting involves many complicated operations with little margin for error. There are hundreds of diagnostic sensors along the production line, measuring various inputs and outputs in the process. The goal is to build a model that maps from this sensor data to a prediction for chip failure. On the basis of this model, chips at risk of failure can be flagged for further (expensive, human) inspection.

For training data we have 1500 observations of a length-200 vector **x** of diagnostic signals, along with binary data on whether the chip was a failure. Note that the x_j inputs here are actually independent from each other (i.e., orthogonal): they are the first 200 Principal Component directions from a bigger set (see Chapter 8 on factorization). The response `fail` is binary (0 or 1). We define a logistic regression model to predict failure probability from the diagnostics:

$$\text{p}(\texttt{fail}|\mathbf{x}) = \frac{e^{\mathbf{x}'\boldsymbol{\beta}}}{1 + e^{\mathbf{x}'\boldsymbol{\beta}}} = \frac{e^{\beta_0 + \beta_1 x_1 \ldots + \beta_k x_k}}{1 + e^{\beta_0 + \beta_1 x_1 \ldots + \beta_k x_k}} \tag{3.2}$$

You can fit this in `R` using `glm`.

```
> SC <- read.csv("semiconductor.csv")
> full <- glm(fail ~ ., data=SC, family="binomial")
Warning message:
glm.fit: fitted probabilities numerically 0 or 1 occurred
```

Note that you get the same perfect fit warning we had in the logistic regression of Example 1.6 in Chapter 1. This is symptomatic of overfit: it indicates that for some observations your fitted regression assigns probabilities $\hat{y} = y$ (1 or 0), similar to how our 18 degree polynomial interpolates points perfectly in Figure 3.1c.

The IS-deviances are available in the fitted `glm` object.

```
> full$deviance
[1] 320.3321
> full$null.deviance
[1] 731.5909
> 1 - full$deviance/full$null.deviance
[1] 0.5621432
```

Since this is logistic regression, these metrics are based upon the binomial deviance from Equation (1.33). We see that this regression has an R^2 of 56%—more than half of the variation in failure versus success is explained by the 200 diagnostic signals.

We can pull out the 200 p-values from `summary.glm`.

```
> pvals <- summary(full)$coef[-1,4] # -1 drops intercept
```

Figure 3.2a shows the distribution of the 200 p-values for tests of null hypothesis $\beta_k = 0$ in this regression. Recall from our FDR discussion in Chapter 2 that p-values from the null distribution have a uniform distribution; in contrast, here we see a spike near zero (indicating useful diagnostic signals), while the remainder sprawl out toward one (most likely useless signals for predicting failure). You can use the Benjamini-Hochberg algorithm to obtain a smaller model

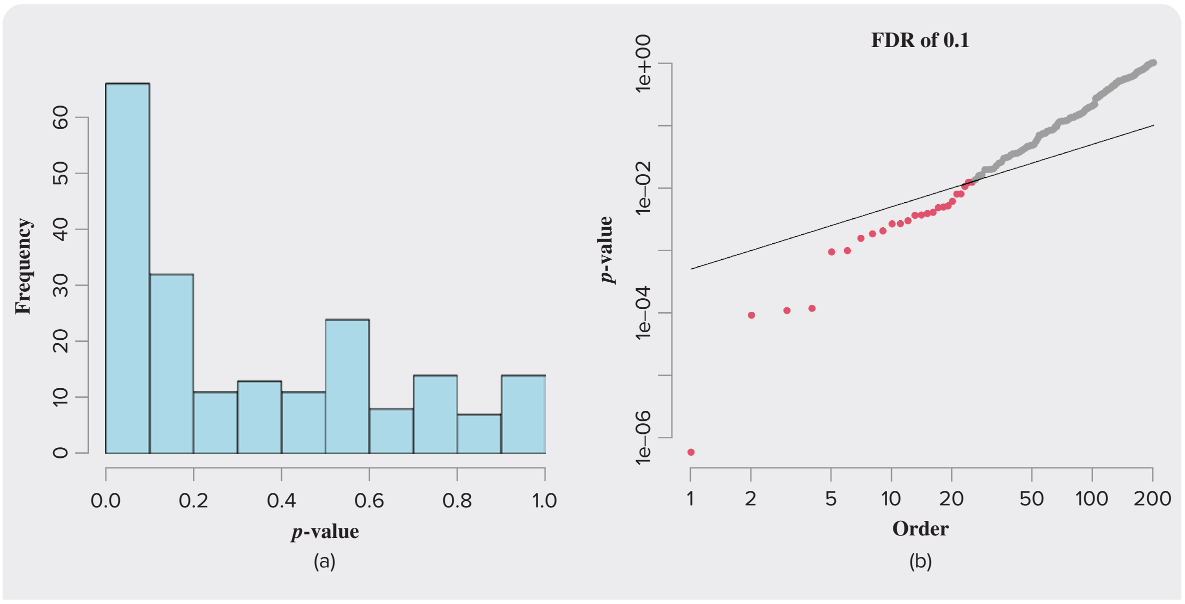


FIGURE 3.2 The distribution of p-values for the 200 coefficients from the semiconductor regression. Panel (a) shows the histogram of p-values, and (b) shows them ranked in increasing size. Red points in (b) are the 25 $\hat{\beta}_k$ that are significant using the BH algorithm for an FDR of 10%.

with controlled false discovery rate. Figure 3.2b illustrates the procedure for an FDR of 10% (`q=0.1`), as executed using the code below.

```
> pvals <- sort(pvals[!is.na(pvals)])
> J <- length(pvals)
> k <- rank(pvals, ties.method="min")
> q=0.1
> ( alpha <- max(pvals[ pvals<= (q*k/(J+1)) ]) )
[1] 0.01217043
> sum(pvals<=alpha)
[1] 25
```

This yields an $\alpha = 0.0122$ p-value rejection cutoff and implies 25 significant regression coefficients (of which you expect 22 to 23 are true signals). This is illustrated by the code fdr_cut(pVals).

We can identify these 25 significant signals and rerun `glm` on only those variables, yielding a much more parsimonious model.

```
> signif <- which(pvals < 0.0122)
> cut <- glm(fail ~ ., data=SC[,c("fail", names(signif))],
+            family="binomial")
> 1 - cut$deviance/cut$null.deviance # new in-sample R2
[1] 0.1811822
```

Notice that the *cut* model, using only 25 signals, has IS $R^2_{cut} = 0.18$. This is much smaller than the full model's $R^2_{full} = 0.56$. In general, the *IS R^2 always increases with more covariates.* This in-sample R^2 is exactly what the maximum likelihood estimate (MLE), $\hat{\boldsymbol{\beta}}$, is fit to maximize. If you give `glm` more knobs to turn (more β_k's), then it will be able to get you a tighter fit. This is exactly why we don't really care about IS R^2—it can be made to look arbitrarily good just by adding more variables to the design. The real question is, how well does each model predict *new* data?

Out-of-Sample Experiments

Of course, you can't know about performance on unseen data because you don't have it. However, you can mimic the experience of predicting on unseen data by performing an out-of-sample experiment to evaluate your models on data that was not used for training. You do this by breaking your data into several *folds* and then repeatedly training your model on all data except one fold and recording the deviance on the left-out fold.

We perform this OOS experiment for both the *full* and *cut* regression models. First, we randomly sample a fold ID for each observation in the semiconductor data set (we set a seed here so you can repeat the same experiment).

```
> n <- nrow(SC) # the number of observations
> K <- 10 # the number of 'folds'
> # create a vector of fold memberships (random order)
> set.seed(1)
> foldid <- rep(1:K,each=ceiling(n/K))[sample(1:n)]
> foldid[1:20]
 [1] 7 5 1 7 4 3 2 9 9 5 9 3 1 8 5 6 4 6 3 7
```

We set $K = 10$ folds, and now `foldid` contains the allocated random fold for each observation. The OOS experiment is then run using a `for`-loop.

```
> fulldev <- cutdev <- nulldev <- rep(NA,K)
> for(k in 1:K){
+     train <- which(foldid!=k) # train on all but fold 'k'
+
+     ## fit the two regressions
+     cuts <- c("fail",names(signif))
+     rfull <- glm(fail~., data=SC, subset=train,
+                  family="binomial")
+     rcut <- glm(fail~., data=SC[,cuts], subset=train,
+                 family="binomial")
+
+     ## predict (type=response for probabilities)
+     pfull <- predict(rfull, newdata=SC[-train,],
+                       type="response")
+     pcut <- predict(rcut, newdata=SC[-train,],
+                      type="response")
+
+     ## calculate OOS deviances
+     y <- SC$fail[-train]
+     ybar <- mean(y)
+     fulldev[k] <- -2*sum(y*log(pfull)+(1-y)*log(1-pfull))
+     cutdev[k] <- -2*sum(y*log(pcut)+(1-y)*log(1-pcut))
+     nulldev[k] <- -2*sum(y*log(ybar)+(1-y)*log(1-ybar))
+
+     ## print progress
+     cat(k, " ")
+ }
1  2  3  4  5  6  7  8  9  10
```

There is a lot going on here, but if you go through each step you'll find that we are simply (a) fitting each regression on the subset that excludes fold `k` via the `subset` argument, and (b) predicting and calculating deviances for observations in fold `k`.

Looking at the resulting OOS deviances, you will see that the cut model has much lower (better) deviance than the full model.

```
> round(fulldev)
 [1] 1838  306  284  198  455  221  263  301  822  158
> round(cutdev)
 [1] 98 50 40 81 68 59 67 57 87 53
> R2 <- data.frame(
+     full = 1 - fulldev/nulldev,
+     cut = 1 - cutdev/nulldev )
> colMeans(R2)
       full         cut
-5.24145031  0.08020355
```

The resulting R^2 values are especially striking: the full model has a *negative* R^2, indicating that its predictions are further from new observations than you get using the training sample mean as your predictor. In this case, the average OOS R^2 are -5.2 for the full model (or -520%) and a positive 0.08 for the cut model. So, while the cut-model's OOS R^2 is lower than its IS R^2, it still manages to do 8% better than the null.

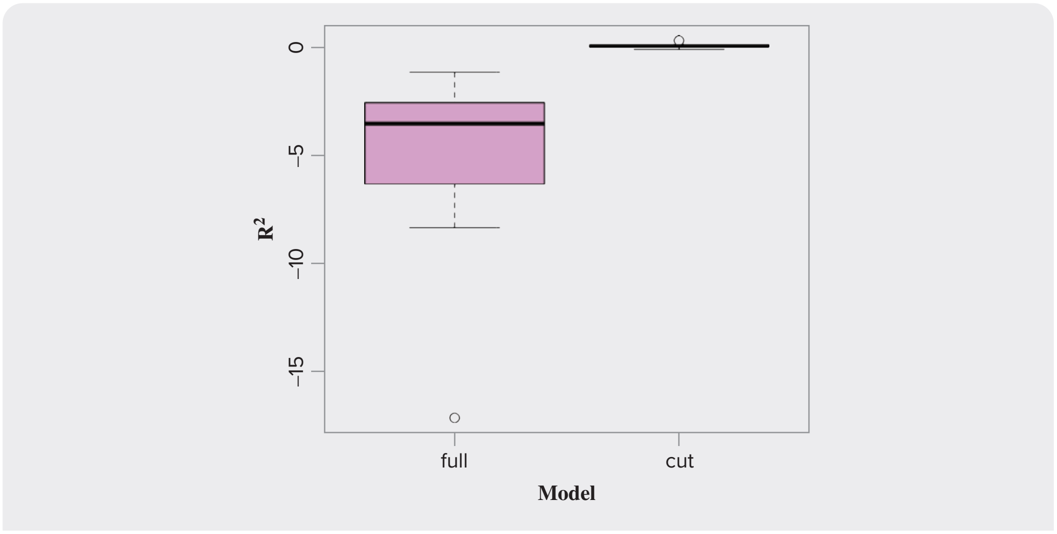


FIGURE 3.3 OOS R^2 for both full (200 signal) and cut (25 signal) semiconductor regressions.

Figure 3.3 shows the distribution of R^2 values across the folds: the full model has a negative R^2 for every fold of the OOS experiment. How can this happen? Look at the R^2 formula: $1 - \text{dev}(\hat{\beta})/\text{dev}(\beta = 0)$. The R^2 will be negative if your fitted model performs worse than the null model, that is, if your $\hat{y}$ estimates are further from the truth than the overall average, $\bar{y}$. Since $\bar{y} \approx 1/15$ here, you are better off simply auditing every 15th chip instead of using a quality control process based upon the overfit full model.

You may have never seen a negative R^2 before. If so, it is likely because you have been looking only at in-sample performance. Out-of-sample, negative R^2 are unfortunately more common than you might expect. Example 3.1 is a dramatic demonstration of the basic principle: *all that matters is out-of-sample R^2*. You don't care about in-sample R^2, because you can get better numbers simply by adding junk variables and inducing overfit.

3.1.1 Cross-Validation

The routine we introduced in Example 3.1—using OOS experiments to select the best model—is called *cross-validation.* The generic procedure is outlined in Algorithm 3.1. Note that we are folding the data into nonoverlapping subsets. Folding your data in this way guarantees that each observation is left out once for validation—each data point is given a chance to yield a large error in a prediction exercise. Doing this, rather than sampling overlapping subsets, reduces the variance of CV model selection.

Cross-validation will play a big role in this text, since using OOS performance in selection of "the best" model is at the core of practical data science. Note that, in the way we ran the OOS experiment in Example 3.1, we were actually violating one of the key rules of running a CV experiment. In our analysis, the full sample was used to choose the 25 variables that are in the cut model. A true OOS experiment would have done FDR control *inside* the for loop, such that the OOS results are a validation of the end-to-end selection procedure. Anything you do to the data, do it without the left-out fold if you want an accurate assessment of OOS performance.

As another general point, you want the CV scheme to mirror how you will actually be applying the model in practice. For example, if you are going to be predicting time series data, then you might want to use only past training data to predict future left-out folds (e.g., you might build a CV routine to predict each month after training on a set number of previous months of data).

Algorithm 3.1 *K*-fold Cross-Validation

Given a dataset of n observations, $\{[\mathbf{x}_i, y_i]\}_{i=1}^{n}$, and M candidate models (or algorithms),

- Split the data into K roughly evenly sized nonoverlapping random subsets (*folds*).
- For $k = 1 \ldots K$:

  Fit the parameters $\hat{\boldsymbol{\beta}}^m$ for each candidate model/algorithm using all but the kth fold of data.

  Record deviance (or, equivalently R^2) on the left-out kth fold based on predictions from each model.

This will yield a set of K OOS deviances for each of your candidate models. This sample is an estimate of the distribution of each model's predictive performance on new data, and you can select the model with the best OOS performance.

3.2 Building Candidate Models

We will revisit cross-validation and other model selection tools in Section 3.3. However, before we get there, we need to develop algorithms for constructing good *sets of candidate* models. The bulk of modern statistical analysis proceeds in two steps. First, you build a set of plausible candidate models. Second, you use a tool like cross-validation to choose among these candidates. This chapter is split into these two steps: building a set of candidate models, and then later selecting amongst the candidates. In this section we will introduce the twin ideas of regularization and model paths as the tools for constructing candidate sets of regression models.

Stepwise Selection

How do you build sets of candidate models? With any reasonable input dimension, it is impossible to simply catalog all possible models. For example, if you have a regression setting with J potential covariates, there are 2^J different possible models depending upon whether each covariate is included. With just 20 covariates, this implies already more than 1 million candidate models. In the previous section we used p-values on the most complex model as the basis for selecting a simpler model. However, this is generally a *bad idea* for a couple of reasons:

- When you have *multicollinearity*—correlation between inputs—the p-values for all of these variables will be large (they will look insignificant) even if any one of the variables provides a useful signal on the response. You will end up including none because you don't know which one of them should be included.
- The p-values are based on a likely overfit model, and this leads to an *unstable* foundation for model construction. You are choosing candidate models on the basis of a noisy regression fit, and small changes in the data can lead to big changes in the candidate models this implies. More dramatically, when you have more covariates than observations there is no full model because `glm` will fail to converge (it will give you an error or a warning and default some coefficient estimates to zero).

This general approach—estimating the most complex model and then using metrics like "significance" to cut it down to size—is sometimes called backward stepwise regression. It should be avoided.

A better solution is to proceed in the opposite direction, building from simplicity to complexity in a *forward stepwise regression.* The procedure is simple: you start by estimating all single input models (one for each dimension of $\mathbf{x}$) and select the one that has the lowest in-sample deviance. You then estimate all two input models that include this first best covariate (the one that was best among all single input models), and then select the best two input model. This process repeats until you get to some maximum model dimension that you are willing to consider.

Example 3.2 Semiconductor Manufacturing: Forward Stepwise Regression The `step` function can be used to execute this stepwise routine. You give a starting point, called the `null` model, and a biggest possible model, called the `scope`. We can use forward stepwise regression on the semiconductor manufacturing data from Example 3.1. The null model is the model with only the intercept, denoted by 1 in the `glm` formulation. The code below takes a few minutes to run (notice we are timing it with the `system.time` function).

```
> null <- glm(fail~1, data=SC)
> system.time(
+ fwd <- step(null, scope=formula(full), dir="forward") )
...
Step:  AIC=92.59
FAIL ~ SIG2
...
   user  system elapsed
  82.55   16.75  128.93
> length(coef(fwd))
[1] 69
```

This procedure enumerated 69 models, ranging from a univariate model including only `SIG2` up to a model with 68 input signals (plus the intercept). The algorithm stopped at 68 because the AIC (Akaike Information Criterion) score for that model was lower (better) than any of the AIC scores for models with 69 inputs. AIC is a "model selection criteria" that attempts to predict how well the model will perform in OOS prediction (without actually running an OOS experiment). We will detail AIC and other information criteria in Section 3.3.1. The `step` function stopped when it thought it had found the best model, making the assumption that since the aic is not getting any better when moving from 67 to 68 inputs it will not improve with 70+ inputs. From this perspective, `step` has determined that the 68 input regression is "best" overall and this is the model that you should use for prediction.

Problems with Subset Selection

In general, forward stepwise regression has a lot of flaws. It can be improved upon dramatically using the regularization ideas that we will introduce in this section. But the general approach of proceeding forward in your search, from simplicity to complexity, is a common and useful approach to building sets—or paths—of candidate models. This is an example of a *greedy* search strategy. In a greedy search you proceed myopically, at each point adding the next iteration of complexity that seems most useful given the current search state. Despite not optimizing for *global* properties of the search path (i.e., each decision does not consider implications for future decisions), greedy algorithms are a useful way to reduce the complexity of your model search and they play a prominent role in many ML strategies.

The problems with forward stepwise selection are that it is *slow* (`step` took approximately 129 elapsed seconds) and *unstable*. These are going to be issues for any model selection procedure that is based on *subset selection*: choosing sets of inputs, and then using maximum likelihood estimation to fit a regression to each set of inputs. The slowness is because you need to estimate from scratch every model (every subset of inputs) that you want to consider, and because there will be a massive set of possible input subsets to consider (this is true even if you use a greedy search to reduce the number of candidate models). If you run the `step` function you will see the massive number of `glm` models that it fits during its greedy search. This is a waste of time.

The alternative to subset selection (and to slow tools like `step`) is to introduce the idea of regularization: replacing deviance minimization with *penalized* deviance minimization, where you are incorporating a "cost of complexity" in your model estimation. Then, instead of selecting subsets of inputs, you select the "price of complexity" as a tuning parameter. We will see that this yields fast construction of useful sets of candidate models.

3.2.1 Penalized Deviance Estimation

The key to modern statistics is *regularization:* penalizing complexity so as to depart from optimality and stabilize your set of candidate models. Incorporating the cost of complexity in your estimations, and considering different prices on complexity in this cost function, allow you to enumerate a list of promising candidate models that range from simple to complex.

Recall from Section 1.4 in the regression chapter that, in classical maximum likelihood estimation, you are fitting $\hat{\boldsymbol{\beta}}$ to minimize the in-sample deviance (which is just -2 times the likelihood). You are choosing the MLE $\hat{\boldsymbol{\beta}}$ to minimize dev(β), for example, to minimize $\Sigma_i(y_i - \mathbf{x}_i'\boldsymbol{\beta})^2$ in linear regression (i.e., in OLS estimation). When you use `glm` to fit a regression model you are minimizing the deviance. A regularization strategy will instead involve minimizing *penalized* deviance. You are fitting $\hat{\boldsymbol{\beta}}$ to minimize

$$\frac{1}{n}\text{dev}(\boldsymbol{\beta}) + \lambda \sum_j c(\beta_j) \tag{3.3}$$

where λ is your penalty and $c(\beta_j)$ is the *cost function.* The cost function determines how different magnitudes of β_j translate to a cost on complexity. For example, we will be largely working with the absolute value cost function, $c(\beta) = |\beta|$, which is the basis for the common and useful *Lasso* estimation framework.

Notice that we are dividing the deviance by n, the sample size, so that (3.3) is technically the *penalized average deviance.* We do this so that λ is on the scale of the average deviance and doesn't need to increase with n to yield similar results. To help with your intuition, consider the setting of linear regression where the deviance is the sum of squared errors. Equation (3.3) then becomes

$$\frac{1}{n}\mathrm{MSE}(\boldsymbol{\beta}) + \lambda \sum_j c(\beta_j) = \frac{\sum_{i=1}^n (y_i - \mathbf{x}_i'\boldsymbol{\beta})^2}{n} + \lambda \sum_j c(\beta_j) \tag{3.4}$$

This implies that the complexity penalty is on the same scale as your mean squared error, which should be roughly similar to the variance in additive errors (σ^2).

Putting a Cost on Complexity

Equation (3.3) adds a penalty—the $\lambda \Sigma_j c(\beta_j)$ term—to the deviance function that we were minimizing in our regression chapter. This penalty puts a *cost* on the magnitude of each β_j. That penalizes *complexity,* because the β_j coefficients are what allow your predicted $\hat{y}$ values to move around with different input $\mathbf{x}$ values. If you force all the $\hat{\beta}_j$ to be close to zero, then your $\hat{y}$ values will be *shrunk* toward $\bar{y}$ and when you jitter the data your predictions will not change as much as they would if you did not include a penalty term during estimation.

Another way to think about Equation 3.3 is through the lens of decision theory—a framework built around the idea that choices have costs. If you consider the decision-making process in classical statistics—focused on a two-stage process of estimation and hypothesis testing—what are the costs?

- *Estimation cost:* Deviance, i.e. the cost of distance between data and the model, and it is what you minimize to obtain the MLE.
- *Testing cost:* There is a fixed price placed on $\hat{\beta}_j \neq 0$. This is implicit in the hypothesis testing procedure, where you set $\hat{\beta}_j = 0$ unless you have *significant evidence otherwise.* The null is *safe* and you need to "pay" to decide otherwise.

Thus, in classical statistics, the cost of $\hat{\beta}$ is deviance plus a penalty for being away from zero. However, the cost of moving away from zero is hidden away inside the hypothesis testing procedure. Equation 3.3 makes both costs explicit.

Cost Functions

What should the penalty function look like? First, $\lambda > 0$ is the penalty weight that determines the "price" of complexity. It is a tuning parameter that needs to be selected in some data-dependent matter, and the later parts of this chapter are focused on how to do this. For now we will take λ as given. The rest of the penalty is determined by the shape of the cost function. In all cases, $c(\beta)$ will be lowest at $\beta = 0$, and you pay more for $|\beta| > 0$; that is, the penalization *shrinks* the coefficients toward zero. Otherwise, the variety of options is wide; Figure 3.4 shows a few.

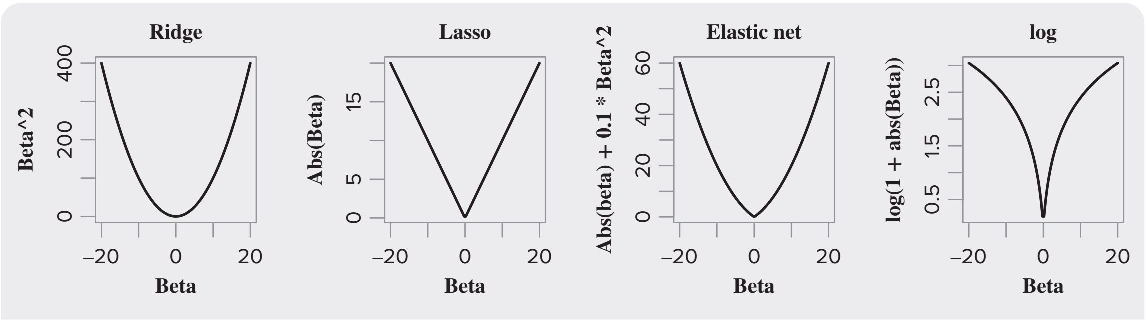


FIGURE 3.4 Common penalty functions: ridge β^2, Lasso $|\beta|$, elastic net $\alpha\beta^2 + |\beta|$, and a "nonconvex" penalty $\log(1 + |\beta|)$.

Each of these leads to different estimation results. The ridge penalty, β^2, places little penalty on small values of β but a rapidly increasing penalty on large values. This will be appropriate for scenarios where you believe each covariate has a small effect, with no big coefficients dominating the model. The Lasso's absolute value penalty, $|\beta|$, places a constant penalty on incremental deviations from zero. Moving β from 1 to 2 costs the same as a move from 101 to 102. And the "elastic net" is an elaborate name for the combination of ridge and Lasso penalties.

Penalties like the log penalty on the far right are special because they have *diminishing bias:* they place extreme cost on the move from zero to small values of β, but for large values the rate of penalty change is small. These penalties encourage lots of zeros in your fit while allowing large signals to be estimated without any bias (i.e., without shrinking large $\hat{\beta}_j$ toward zero). Such "nonconvex" penalties are sometimes favored by theoretical statisticians, but they need to be treated with care in practice because they introduce many of the instability and computational issues that you observe with subset selection. Indeed, you can interpret forward stepwise regression as solving for a penalized deviance under the extreme version of this where $c(\beta) = \mathbb{1}_{[\beta \neq 0]}$ such that $c(0) = 0$ and $c(\beta) = 1$ for any $\beta \neq 0$. The problems of subset selection—needing to refit a completely different model every time you add a variable—are extreme versions of the problems associated with any nonconvex penalty scheme.

An advantage of the Lasso is that it gives the least possible amount of bias on large signals while still retaining the stability of a convex penalty like the ridge (convex means that the penalty doesn't flatten out for large values). Another massive advantage of the Lasso, and of all the three right "spiky" penalties in Equation 3.4, is that it will yield *automatic variable screening*. That is, some of the solved $\hat{\beta}_j$ values will be exactly equal to zero—not close to zero, but zero as in "they are not in the model, so you don't need to store or think about them." The reason that this happens is illustrated in Figure 3.5: the deviance is smooth while the absolute value function is pointy, and the minimum of their sum can be at zero if the penalty dominates. Any penalty that involves a $|\beta|$ term will do this—for example, all but ridge in Figure 3.4.

In summary, there are *many* penalty options. As you will see, the Lasso is a common default. You can think of it as a baseline and consider others only if you have a strong reason to do so. There are certain settings where in theory you might prefer the elastic net (if the true regression relationship has many small coefficients) or a nonconvex log penalty (e.g., when model compression—having the fewest coefficients as possible—is the goal) but it is rare in practice that you can do much better than the Lasso.

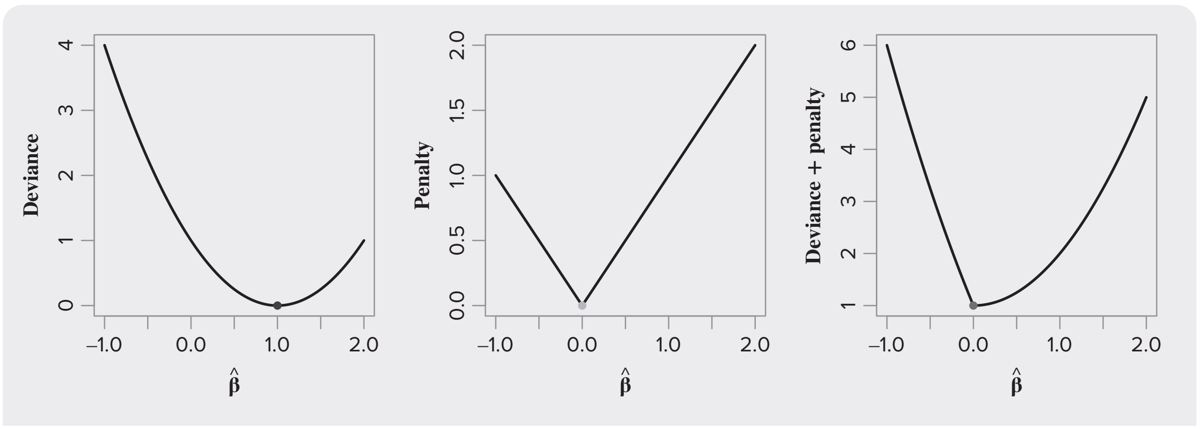


FIGURE 3.5 Illustration of penalized deviance minimization leading to $\hat{\beta} = 0$.

3.2.2 Regularization Paths

The Lasso alone does not select models. Rather, it provides a mechanism to *enumerate* a set of candidate models to choose among. A Lasso regularization path minimizes, for a *sequence* of penalties $\lambda_1 > \lambda_2 \ldots > \lambda_T$, the penalized deviance

$$\frac{1}{n}\text{dev}(\boldsymbol{\beta}) + \lambda_t \sum_j |\beta_j| \qquad \textbf{(3.5)}$$

This yields a sequence of estimated regressions with coefficients $\hat{\boldsymbol{\beta}}_1 \ldots \hat{\boldsymbol{\beta}}_T$. Given this sequence, model selection tools (e.g., cross-validation) are used to choose the best $\hat{\lambda}_t$ and hence the best $\hat{\boldsymbol{\beta}}_t$.

Algorithm 3.2 outlines this recipe. You start with λ_1 just big enough that $\hat{\boldsymbol{\beta}}_1 = 0$. This is always possible: from Equation 3.5, you can set λ so that the cost of moving any $\hat{\beta}_j$ slightly away from zero is equal to the corresponding decrease in the deviance, and so the optimization keeps $\hat{\beta}_j = 0$. Most software will find this starting point automatically. You then iteratively shrink λ while updating the estimated $\hat{\boldsymbol{\beta}}$.

A crucial detail here is that the coefficient updates are smooth in λ; that is,

$$\hat{\beta}_t \approx \hat{\beta}_{t-1} \text{ for } \lambda_t \approx \lambda_{t-1}. \qquad \textbf{(3.6)}$$

This leads to both *speed* and *stability* for the Lasso algorithm. The speed comes from the fact that each update $\hat{\beta}_{t-1} \rightarrow \hat{\beta}_t$ is small and hence fast. Selection stability is the mirror image of this property: even if the "best" λ changes across data samples, it will remain in a local neighborhood and the selected $\hat{\beta}$ will thus also stay in a small neighborhood.

Algorithm 3.2 Lasso Regularization Path

```
Begin with $\lambda_1 = \min\{\lambda : \hat{\beta}_\lambda = 0\}$.
    For $t = 1 \ldots T$:
    • Set $\lambda_t = \delta\lambda_{t-1}$ for $\delta \in (0, 1)$.
    • Then find $\hat{\boldsymbol{\beta}}_t$ to optimize Equation 3.5 under penalty $\lambda_t$.
```

Path Plots

The whole enterprise is easiest to understand visually. The *path plot* in Figure 3.6 illustrates Algorithm 3.2. The algorithm moves right to left with decreasing values of λ. The vertical axis here is $\hat{\beta}$, with each colored line a different $\hat{\beta}_j$ as a function of λ_t. Each vertical slice of the plot represents a candidate model. From right to left, the models become increasingly complex as the $\hat{\beta}_t$ include more and larger nonzero $\hat{\beta}_j$ values (the plot header marks the number of nonzero $\hat{\beta}_j$ at certain segments).

This picture and the underlying path estimation were executed using the `gamlr` package for `R` (Taddy, 2017). We will use this package heavily. It provides fast and reliable Lasso paths. The popular `glmnet` package (Friedman et al., 2010) is also an excellent option for Lasso estimation. Both `gamlr` and `glmnet` use similar syntax and use similar optimization routines (coordinate descent). The difference is in what they can do beyond the Lasso: `gamlr` offers diminishing bias penalization, like the log penalty in Figure 3.4, while `glmnet` provides for the elastic net of that same figure. We use the `gamlr` software because it was written to provide for

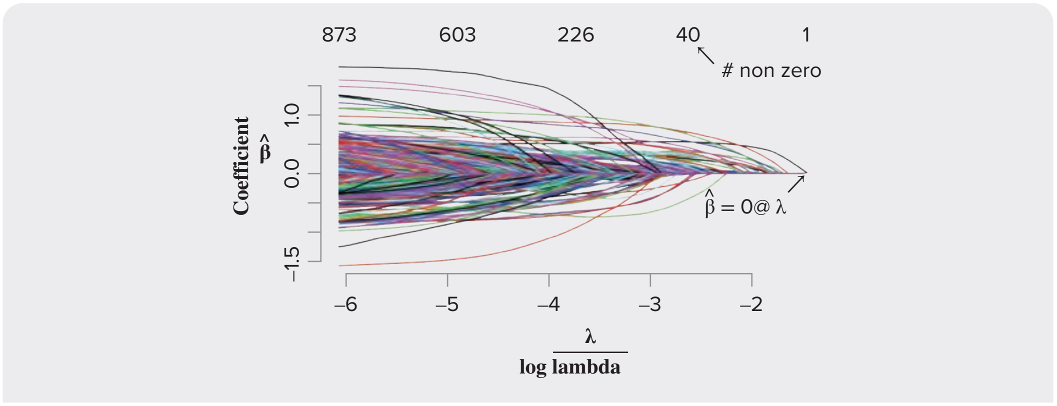


FIGURE 3.6 A Lasso regularization path plot from `gamlr`. Algorithm 3.2 proceeds from right to left, with decreasing λ_t.

useful features covered in this book (e.g., AICc selection, double ML for causal inference, and distributed multinomial regression).

Running a Lasso in `gamlr` is fairly straightforward, but there are some particularities, which we'll outline in the next section. The main difference from `glm` is that that you need to create the numeric model matrix yourself. There are some tricks to creating model matrices for Lasso regressions, and we will spend some time outlining those details before starting to fit our Lasso paths.

3.2.3 Sparse Model Matrices

To use `gamlr` you need to feed it the data in the correct format. Recall from Chapter 1 that converting a data frame to a numeric model matrix is one of the first steps done inside `glm`. The numeric model matrix contains a column for each continuous (`numeric`) variable and columns for *levels* of the categorical (`factor`) variables. In creating the model matrix, `R` will create a column for all but one level of a categorical variable. The level that is omitted is the reference level. When you fit `glm`, the coefficients for the other factor levels can each be interpreted as a comparison to the baseline expected response for this reference level.

In Chapter 1, we worked with data on orange juice sales. The `oj` data frame contains a continuous covariate `price` and a factor variable `brand` with three levels: `dominicks`, `minute.maid`, and `tropicana`. To create the model matrix that is used by `glm` for regression estimation, you pass a regression formula and the data to `model.matrix` the same way as you would to `glm` but without including the response variable. We do that here and print a row for each brand.

```
> oj<-read.csv("oj.csv",strings=T)
> modMat<-model.matrix(~log(price)+brand,data=oj)
> modMat[c(100,200,300),] #look at one row for each brand
    (Intercept) log(price) brandminute.maid brandtropicana
100           1  1.1600209                0              1
200           1  1.0260416                1              0
300           1  0.3293037                0              0
```

Notice the column of 1s for the intercept, the single column for the quantitative predictor, and a binary column for each of `minute.maid` and `tropicana`. The reference level for factor `brand` is `dominicks` and this reference level is subsumed into the intercept. Notice that the third printed row has zero for both the categorical columns: this observation is from `dominicks`, so it gets a zero for each of `brandminute.maid` and `brandtropicana` columns.

This is not the model matrix that you want to use with `gamlr` (or `glmnet`) for Lasso regression. You will want to make three changes:

1. Create a column for all factor levels (including `dominicks`).
2. Delete the column of 1s for the intercept (`gamlr` adds its own intercept).
3. Convert to a *sparse matrix* to reduce storage and increase efficiency.

We will work through these changes in turn.

Including All Factor Levels in the Model Matrix

For MLE regression with `glm`, it doesn't matter which brand of OJ is the baseline level for the `brand` factor. Even though one of them will get subsumed into the intercept, you end up with the same predicted $\hat{y}$ values. But when you start penalizing coefficients, *factor reference levels now matter*. Since the penalty rewards $\hat{\beta}_j$ values closer to (or at) zero, you are shrinking factor coefficients toward the intercept—toward the reference level. And it makes a difference which level is the baseline (i.e., whether you shrink Minute Maid toward Tropicana instead of Dominicks).

The solution is to simply get rid of the reference level. Once you add a penalty to the deviance, there is no reason to have only $K - 1$ coefficients for a K-level factor. If every category level is given its own dummy variable, then every factor level effect is shrunk toward a shared intercept. You are shrinking toward a shared mean, with only significantly distinct effects getting nonzero $\hat{\beta}_k$.

You can force `R` to create separate dummies for each level by creating an *extra* factor level. The `gamlr` package includes the utility function `naref` which makes `NA` ("not available," `R`'s code for "missing") the reference level for every factor. Conveniently, `naref` has the extra advantage of providing a framework for missing data; we will introduce this functionality later in this chapter. Here we apply `naref` to create the new data frame `ojdf`.

```
> library(gamlr)
> ojdf <-naref(oj)
> ojdf[c(100,200,300),"brand"]
[1] tropicana   minute.maid dominicks
Levels: <NA> dominicks minute.maid tropicana
```

If you apply `naref` to your data frame before creating the model matrix, then it will lead to every factor level having its own column.

```
> modMatAllLevs<-model.matrix(~log(price)+brand,data=ojdf)[,-1]
> modMatAllLevs[c(100,200,300),]
    log(price) branddominicks brandminute.maid brandtropicana
100  1.1600209              0                0              1
200  1.0260416              0                1              0
300  0.3293037              1                0              0
```

Notice that we appended `[,-1]` when we created the model matrix here. This removed the intercept column, as desired because `gamlr` adds its own intercept. You can also add a `-1` term in the regression formula to get the same result.

Sparse Matrices

The model matrix we just created is in *dense* format: `R` stores the matrix as a rectangle of data. You can provide dense matrices to `gamlr`, however `gamlr` (along with `glmnet` and many other `R` packages) is able to take advantage of the `Matrix` library representation for *sparse matrices.* A sparse matrix is one with many zero entries, which is a common scenario in modern data analysis. For example, many interacting categorical variables will—when represented as 0/1 indicator variables—lead to sparse designs. It is then efficient to ignore zero elements in the matrix whenever you can. Packages like `gamlr` use sparse matrix structures for lower storage costs and faster computation. This will be essential for big data.

One common sparse representation is a simple triplet matrix (STM) with three key elements: the row `i`, column `j`, and entry value `x`. Everything else in the matrix is assumed zero. Here's an example:

$$\begin{bmatrix} -4 & 0 \\ 0 & 10 \\ 5 & 0 \end{bmatrix} \quad \text{is stored as} \quad \left\{ \begin{array}{rl} \mathrm{i} = & 1,3,\ 2 \\ \mathrm{j} = & 1,1,\ 2 \\ \mathrm{x} = & -4,5,10 \end{array} \right\}$$

The `Matrix` library provides tools for creating and working with sparse matrices. After loading this library we can use `sparse.model.matrix` to create the same model matrix as before, but in efficient simple triplet format.

```
> xOJ<-sparse.model.matrix(~log(price)+brand,data=ojdf)[,-1]
> xOJ[c(100,200,300),]
3 x 4 sparse Matrix of class “dgCMatrix”
    log(price) branddominicks brandminute.maid brandtropicana
100  1.1600209              .                .              1
200  1.0260416              .                1              .
300  0.3293037              1                .              .
```

The zeros have been replaced by “.” when we print the matrix. This is a hint that this is a sparse matrix. Under the hood, the `Matrix` library has created a special sparse matrix structure.

```
> class(xOJ)
[1] “dgCMatrix”
attr(,“package”)
[1] “Matrix”
```

This specific matrix is of the `dgCMatrix` (compressed, sparse, column-oriented) format. The `Matrix` library has a variety of different structures that it uses for representing sparse matrices, and packages that are compatible with `Matrix`, like `gamlr`, are able to recognize and process all the different formats. You can ignore the details and just benefit from lower memory usage and faster optimizations. However, you will often need to remove the sparse matrix formatting and convert back to a dense representation. This happens, for example, if you are trying to use the output `gamlr` in another function that doesn't recognize sparse matrices. One way to do this is to apply the `as.matrix` function.

```
> as.matrix(xOJ[c(100,200,300),])
    log(price) branddominicks  brandminute.maid  brandtropicana
100   1.1600209              0                 0               1
200   1.0260416              0                 1               0
300   0.3293037              1                 0               0
```

Alternatively, if you just want to pull a single vector out from the matrix (row or column) it will automatically be converted back to dense format.

```
> xOJ[100,]
      log(price) branddominicks brandminute.maid  brandtropicana
        1.160021       0.000000         0.000000        1.000000
```

3.2.4 Path Estimation with `gamlr`

Once you know how to build your sparse model matrix, running `gamlr` is easy. You simply supply the model matrix as `x` and the response as `y` and off you go. For example, we can regress log sales onto `xoj`.

```
> fitOJ <- gamlr(x=xOJ, y=log(ojdf$sales))
> plot(fitOJ)
```

Calling `plot` on the fitted `gamlr` object produces the path plot in Figure 3.7a. You can also add `family="binomial"` to fit a Lasso logistic regression, as we do here for the semiconductor data (note that the first column of `SC` is `fail`, the response).

```
> fitSC <- gamlr(x=SC[,-1], y=SC[,1], family="binomial")
> plot(fitSC)
```

The resulting path plot is shown in Figure 3.7b. We didn't bother to create a sparse model matrix for the semiconductor Lasso because the original data is all numeric and dense (it doesn't contain a bunch of zeros). In that case you can just give `gamlr` the raw data as `x`.

The default behavior of `gamlr` solves for a sequence of 100 penalties λ_t, ranging from the initial λ_1 (set just big enough so that $\hat{\boldsymbol{\beta}}_1 = \mathbf{0}$) down to $\lambda_T = 0.01\lambda_1$. You can change this default behavior by specifying `nlambda` to set the sequence length T, or `lmr` to set the minimum λ_T

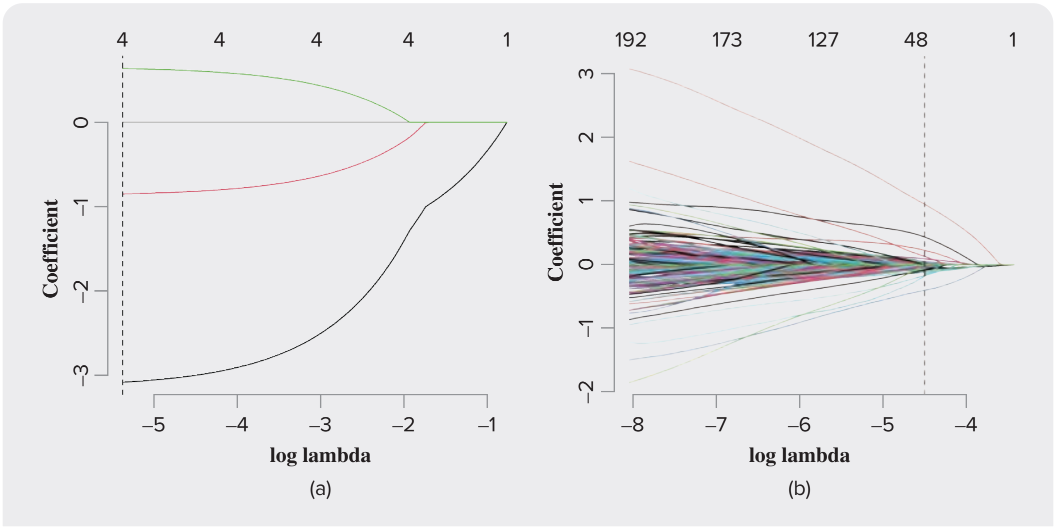


FIGURE 3.7 Lasso path plots for the orange juice (a) and semiconductor manufacturing (b) regression examples.

penalty size via the ratio `lmr` $= \lambda_T/\lambda_1$. This `lmr` argument is one you will use often; it is common that you will want to fit more complex models than you get for the default `lmr=0.01` and then you need to set a smaller `lmr`.

Example 3.3 **Orange Juice Sales: Lasso Paths** You can print the fitted `gamlr` object to get a quick statement on what we fit above.

```
> fitOJ
gaussian gamlr with 4 inputs and 100 segments.
```

And you can call `summary` on it to get information about each path *segment* (the estimation at each λ_t).

```
> summary(fitOJ)
gaussian gamlr with 4 inputs and 100 segments.
            lambda par df         r2        aicc
seg1   0.465057217   1  1 0.00000000   1112.15134
seg2   0.443919649   2  2 0.01849065    573.89239
...
seg99  0.004872013   4  4 0.39390922 -13376.34842
seg100 0.004650572   4  4 0.39392573  -13377.13705
```

The information here includes the number of nonzero parameters (`par`) and degrees of freedom (`df`, which will be the same as `par` for the Lasso) and two measures of fit: in-sample R^2 and a corrected version of the AIC (we will work with this in the next section).

The `gamlr` object contains a number of attributes.

```
> names(fitOJ)
 [1] "lambda"   "gamma"    "nobs"     "family"   "alpha"    "beta"
 [7] "df"       "deviance" "iter"     "free"     "call"
 > dim(fitOJ$beta)
[1]   4 100
```

For example, `lambda` is the sequence of fitted λ_t penalties, `alpha` is the sequence of fitted intercept terms, and `beta` is the sequence of fitted regression coefficients. The intercept sequence is a vector of length `nlambda` (100 by default) and the coefficient sequence `beta` is an `nlambda`-column matrix with number of rows equal to the number of input variables (the number of columns in your model matrix). Each column of `beta` contains a segment of the path of estimated regression coefficients. Here we show the first two and last two segments.

```
> fitOJ$beta[,c(1:2,99:100)]
4 x 4 sparse Matrix of class "dgCMatrix"
                      seg1        seg2       seg99     seg100
log(price)               .  -0.07278185  -3.0761907  -3.0790315
branddominicks           .            .  -0.8468649  -0.8479243
brandminute.maid         .            .           .           .
brandtropicana           .            .   0.6376020   0.6386095
```

At λ_1 all of the coefficients are set equal to zero. As the penalty decreases to λ_2, the coefficient on log price enters the model with a nonzero elasticity. At the end of the path, at λ_{100}, all of the coefficients are nonzero except for the `brandminute.maid` coefficient. Sensibly, `gamlr` has determined that Minute Maid is the baseline OJ and that the economy Dominick's and luxury Tropicana can be represented through deviations from this baseline. Note that the intercept is *always unpenalized* in `gamlr`, so you have a nonzero intercept at every path segment.

```
> fitOJ$alpha[c(1:2,99:100)]
     seg1      seg2     seg99    seg100
 9.167864  9.224931 11.649609 11.651854
```

At the end of the Lasso path, the penalty is very small and our model fit is approaching that which you would get by using OLS to estimate this regression model. We can compare to the OLS results after re-leveling `brand` to have `minute.maid` as the reference level.

```
> oj$brand <- relevel(ojdf$brand, "minute.maid")
> glm(log(sales) ~ log(price) + brand, data=oj)
Coefficients:
   (Intercept)      log(price)  branddominicks  brandtropicana
       11.6990         -3.1387         -0.8702          0.6598
```

Another `gamlr` argument that we will encounter frequently is the `free` argument. You can pass `free` column indices (or names) for your design matrix and `gamlr` will leave the corresponding coefficients *unpenalized* in the regression fit. This is useful if there are some variables that you *know* need to be in the regression model, and you want their coefficients to be estimated without any bias (without any shrinkage toward zero). For example, we could have log price enter without a coefficient penalty in our simple OJ regression.

```
> fitOJfree <- gamlr(x=xOJ, y=log(ojdf$sales),free="log(price)")
> fitOJfree$beta[,c(1:2,99:100)]
4 x 4 sparse Matrix of class "dgCMatrix"
                      seg1        seg2      seg99     seg100
log(price)       -1.601307 -1.64724906 -3.1199089 -3.1207643
branddominicks           . -0.04531381 -0.8612653 -0.8616711
brandminute.maid         .           .          .          .
brandtropicana           .           .  0.6508045  0.6512121
```

We now have a nonzero coefficient on `log(price)` even at λ_1. Of course, even though it is unpenalized, the coefficient on log price changes along the path as it adjusts to the influence of the estimated `brand` effects.

You may be confused at this point about what to do with this path of estimates. What use is a set of 100 model estimates if you don't know which one to use? Indeed, this path estimation is just the first step in model construction: once you have a path, you will need to use the selection tools of Section 3.3 to determine which model to deploy in practice.

Scaling the Penalty by Standard Deviations

A key thing to be aware of is that for Lasso regression the size of the covariates matters. Since the β_k values are all penalized by the same λ, you need to make sure they are on comparable scales. For example, $x\beta$ has the same effect as $(2x)\beta/2$, but $|\beta|$ is twice as much penalty cost as $|\beta/2|$. The common solution to this is to multiply β_j by $\mathrm{sd}(x_j)$ in the cost function to standardize across scales. That is, instead of Equation 3.5, you minimize

$$\frac{1}{n}\mathrm{dev}(\boldsymbol{\beta}) + \lambda \sum_j \mathrm{sd}(x_j)\,|\beta_j|. \tag{3.7}$$

This implies that each β_j's penalty is now measured on the scale of 1 standard deviation change in x_j and, for example, switching from meters to feet or Fahrenheit to Celsius won't change your model fit.

This standardization scaling is the default in `gamlr` (and in `glmnet`) via the argument `standardize=TRUE`. There are some occasions where you instead want `standardize=FALSE`. Most commonly, you might want `standardize=FALSE` if you have all indicator variables indicating category membership (such as brand or geographic region). In this case, the standardization would put *more* penalty on common categories (since $\mathrm{sd}(x_j)$ will be higher) and less penalty on rare categories, which might be undesirable. However, unless you have clear reason to do otherwise, you should stick with the default `standardize=TRUE`.

In the remainder of this section, we will introduce two regression examples—one linear and one logistic—and work through them to illustrate `gamlr` Lasso regression.

Example 3.4 Ames Housing Data: Lasso Linear Regression The data in `amesHousing.csv` consist of information that the local government in Ames, Iowa, uses to assess home values. These data were compiled from 2006 to 2010 by De Cock (2011). They contain 2930 observations on 79 variables describing properties in Ames and their observed sale price.

```
> ames <- read.csv("AmesHousing.csv", strings=T)
> dim(ames)
[1] 2930   79
> ames[1:3,c(1:5,79)]
  MS.Zoning Lot.Frontage Lot.Area  Street Alley SalePrice
1        RL          141    31770    Pave  <NA>    215000
2        RH           80    11622    Pave  <NA>    105000
3        RL           81    14267    Pave  <NA>    172000
```

Our prediction target here will be the log of `SalePrice`. We are working on log scale because, following the discussions in Chapter 1, prices tend to change with product characteristics in a multiplicative fashion. The sale price distribution is shown in Figure 3.8, both as a histogram and in a log-log scatterplot against lot size.

We will do some light processing on the raw data.

```
> ames$Yr.Sold <- factor(ames$Yr.Sold)
> ames$Mo.Sold <- factor(ames$Mo.Sold)
> ames$Lot.Area <- log(ames$Lot.Area)
```

Here we have converted the lot area to log scale, and converted the year and month of sale to factors. This is a rich data set and there is a ton of additional feature engineering that you can do to improve predictive performance. For example, you might want to use additional log

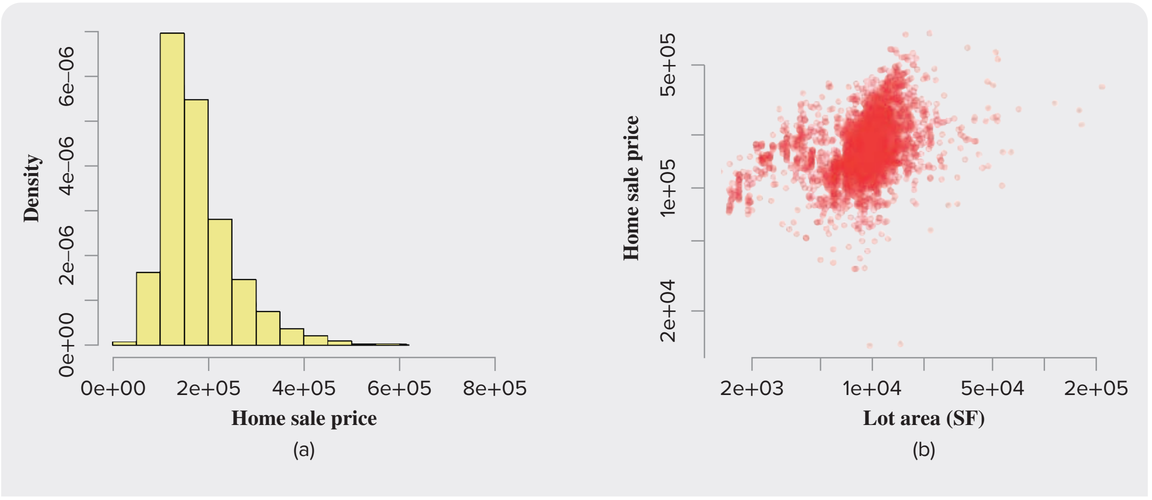


FIGURE 3.8 Home sale prices in Ames, Iowa. Panel (a) shows the marginal distribution of sale prices and (b) shows prices against the property lot size in square feet (note that this plot uses log scaling on the axes).

transforms or include factor representations of additional variables. There are also interesting interaction variables to consider (e.g., many of the variables are related to house condition and these can be interacted with the variables related to house size). For this illustration we will keep things simple, but we encourage you to explore the data further and try to engineer features that improve predictive performance. The regularization and selection tools of this chapter will allow you to build a really sophisticated prediction model.

Missing Data

Before we create our model matrix, we need to deal with a common issue: *missing data*. If you look at the data frame of observations, 13960 of the entries are NA (this is R's code for "not available").

```
> sum(is.na(ames))
[1] 13960
```

This indicates that these entries are missing in the assessor's data. Looking at some individual variables, we see examples of missing entries in Pool.QC (a quality category for the swimming pool) and Lot.Frontage.

```
> summary(ames$Pool.QC)
  Ex   Fa   Gd   TA NA's
   4    2    4    3 2917
> ames$Lot.Frontage[11:15]
[1] 75 NA 63 85 NA
```

Presumably, Pool.QC is missing because most properties do not include a swimming pool. The Lot.Frontage variable—the feet of street connected to the property—could be missing for a variety of reasons. Perhaps it is because the property has zero frontage (although some of these missing values correspond to houses with large property areas) or perhaps it is just a variable that is not always recorded.

Dealing with missing data is straightforward with the naref function provided as part of the gamlr library. We already used naref to set NA as the reference level for factor variables (so that sparse.model.matrix creates a model matrix column for every other factor level). When you have missing data, the missing observations on this factor are assigned to this reference level. You can also call the naref function with the argument impute=TRUE to *impute* a value for the missing observations in numeric variables. See the box "Dealing with Missing Data" for details on how naref works to impute missing values with the mean for that variable or, if the variable is sparse (mostly zero), with zero.

We apply naref to the ames data, with impute=TRUE, to obtain amesImputed as a data frame that contains no missing values.

```
> library(gamlr)
> amesImputed <- naref(ames, impute=TRUE)
> sum(is.na(amesImputed))
[1] 0
```

The factors now all have `<NA>` as their reference level, and numeric variables with missing values have been replaced by `var.x` which has no missing values (these have been imputed) and `var.miss` that is 1 if the entry was imputed and 0 otherwise.

```
> summary(amesImputed$Pool.QC)
<NA>    Ex    Fa    Gd    TA
2917     4     2     4     3
> amesImputed$Lot.Frontage.x[11:15]
[1] 75.00000 69.22459 63.00000 85.00000 69.22459
> amesImputed$Lot.Frontage.miss[11:15]
[1] 0 1 0 0 1
```

Again, refer to the “Missing Data” box for detail on how we deal with missing data.

Dealing with Missing Data

Missing data is an issue that will occur repeatedly in practice. Incomplete observations occur for a variety of reasons. In some cases, data will be missing because the variable doesn’t make sense for that observation (e.g., pool quality for a house without a pool). In survey data, you can have variables that are missing because people don’t answer all of your questions. And in any telemetry data (e.g., for anything from tracking industrial processes to tracking online customer behavior) information is often missed or dropped in processing.

The `naref` function from `gamlr` is useful to prepare data with missingness for use in regression analyses. We will describe here how it deals with each of categorical (`factor`) and numeric (`numeric` or `integer`) variables.

For factor variables, you simply treat the missing observations as a separate category. As described earlier in this chapter, adding `NA` as a new reference level for each factor variable forces `R` to have a separate coefficient for each observed factor level. Recall that in Example 3.4 the original data frame is `ames` and `amesImputed` is the output from calling `naref` on this original data.

```
> summary(ames$Pool.QC)
  Ex    Fa    Gd    TA  NA's
   4     2     4     3  2917
> summary(amesImputed$Pool.QC)
<NA>    Ex    Fa    Gd    TA
2917     4     2     4     3
```

In the original data, `Pool.QC` had mostly missing values. In `amesImputed`, the output from `naref`, these missing values are allocated to a new reference factor level called `<NA>`. When you build a model matrix using `amesImputed`, the impact on expected response from these missing values will be subsumed into the intercept (i.e., such that the `Pool.QC` factor level effects will be interpretable as being relative to a property with no swimming pool).

For numeric variables, you need to replace the missing values with a numeric value. This is referred to as *data imputation.* There are a bunch of different ways that you can do this—missing data imputation, or guessing what the missing values *would* have been, is an interesting regression problem in its own right. Two simple approaches that work well for most problems are *zero* and *mean* imputation. In the former, which we recommend for sparse variables (those that are mostly zero), you replace missing values with zero. In the latter, you replace the missing values with the mean of the nonmissing entries. Mean imputation has better theoretical properties because observations close to the mean have less impact on your fitted regression coefficients (they have low *leverage*). But zero imputation can be preferable if you have sparse data and you don't want to lose that convenient sparsity by imputing a bunch of close-but-not-quite-zero values (when variable is mostly zero, the mean tends to be near zero).

The `naref` function has the argument `pzero` which is used to decide between mean and zero imputation. If the proportion of zeros in the nonmissing entries is greater than `pzero`, then it does zero imputation. Otherwise it does mean imputation. The default is `pzero=0.5` so that you will impute zeros if more than half of the values are zero. If you want to force zero imputation, then you specify `pzero=0`.

Looking at our Ames housing data, the feet of property frontage is a numeric variable with missing values. After running `naref` with `impute=TRUE`, the missing entries have been replaced with the mean of the nonmissing values.

```
> ames$Lot.Frontage[11:15]
[1] 75 NA 63 85 NA
> amesImputed$Lot.Frontage.x[11:15]
[1] 75.00000 69.22459 63.00000 85.00000 69.22459
> amesImputed$Lot.Frontage.miss[11:15]
[1] 0 1 0 0 1
```

The single variable `Lot.Frontage` has been replaced with two columns, `Lot.Frontage.x` which contains the numeric values after imputation, and `Lot.Frontage.miss` which is a binary variable with 1 for those observations that were missing and are now imputed. Notice that the imputed value is around 69, indicating that the average feet of frontage is around 69 for properties where this is not missing.

For an example of zero imputation, consider the `Bsmt.Full.Bath` variable that counts the number of full bathrooms in the house basement. This variable is 58% zeros and has two `NA` values. Since this proportion of zeros is higher than the default `pzero=0.5`, `naref` has replaced the missing values with zero.

```
> mean(ames$Bsmt.Full.Bath==0, na.rm=TRUE)
[1] 0.5829918
> ames$Bsmt.Full.Bath[1341:1344]
[1]  1 NA  0  0
> amesImputed$Bsmt.Full.Bath.x[1341:1344]
[1] 1 0 0 0
> amesImputed$Bsmt.Full.Bath.miss[1341:1344]
[1] 0 1 0 0
```

Again, `naref` outputs both the `.x` imputed numeric variable and a `.miss` indicator for whether or not the original value was missing.

You should always include the missingness indicator (e.g., `Bsmt.Full.Bath.miss` and `Lot.Frontage.miss`) in your analysis because the fact that data were missing might itself be useful information. For example, the missing `Lot.Frontage` values might occur because those houses have no yard. The only situation where you can ignore the fact that data was missing is if it was missing completely at random: if the probability of data being missing is independent from the other characteristics of the observation. This is unlikely to be true in practice. And even if it is true, including the missingness indicator will make your analysis more robust to the accuracy of your missing data imputation procedure.

Finally, note that none of the observations of `SalePrice` were missing in the original `ames` data frame. When you are doing data imputation, you never want to impute missing values in your response variable. An observation with a missing response should simply be dropped from your analysis (although if you have a large number of missing responses you need to understand why that happened and how it impacts your interpretation of what you are predicting).

Building the Model Matrix

Once you have constructed the data frame `amesImputed`, which has imputed missing numeric values and created a `<NA>` reference level for all factors, you can proceed with Lasso estimation. The first step is to extract the response variable, *y,* as the log of the property sale price.

```
> yAmes <- log(ames$SalePrice)
> head(yAmes)
[1] 12.27839 11.56172 12.05525 12.40492 12.15425 12.18332
```

The next step is to create the numeric model matrix. We do that here using `sparse.model.matrix` and a formula that specifies regression onto every variable except the `SalePrice` column that we used to create our log sale price response.

```
> ycol <- which(names(amesImputed)=="SalePrice")
> xAmes <- sparse.model.matrix( ~ ., data=amesImputed[,-ycol])
[,-1]
> dim(xAmes)
[1] 2930  339
```

The result is a 339 column model matrix (note that we used `[,-1]` to remove the intercept). As mentioned earlier, you can create much more complex regression models here by combining and interacting variables. The “main effects only” model is a starting point but you can expand it yourself by changing the regression formula that defines `x`.

Fitting the Lasso Path

Now that we have `x` and `y`, we can apply `gamlr` to fit the Lasso regularization paths.

```
> fitAmes <- gamlr(xAmes, yAmes, lmr=1e-4)
```

Note that we have specified `lmr=1e-4` here to set a smaller than default (`lmr=1e-2`) ratio for the smallest λ_T relative to the starting λ_1. This tells `gamlr` to run the Lasso path down to a smaller level of penalization. The fitted path is shown in Figure 3.9b. For comparison, the path plot corresponding to a default specification with `lmr=1e-2` is shown in Figure 3.9a. While the default path stops at $\lambda_T \approx e^{-6}$, our specified path goes down to $\lambda_T < e^{-10}$. This allows for many more nonzero estimated coefficients at the end of the path (from the plot headers, 301 for `lmr=1e-4` vs 115 for `lmr=1e-2`) and these estimated coefficients are allowed to move further from zero.

Note that the vertical dashed lines on the plots in Figure 3.9 correspond to the “best” model as selected by the AICc selection rule introduced in the next section. By this rule, the “best” model is at the edge of the λ_t values evaluated using `lmr=1e-2`. Whenever your best model is at the lowest-penalty edge of the space of models you are considering (i.e., if your selection rule chooses the model at λ_T) you should consider re-running the path algorithm to consider smaller penalization levels.

The Fitted `gamlr` Object

The fitted `gamlr` object, which we’ve names `fitAmes`, contains the data behind the path plots shown in Figure 3.9. For example, we can see that `gamlr` fit 100 λ_t segments and that $\lambda_{100}/\lambda_1 = 1/10000$.

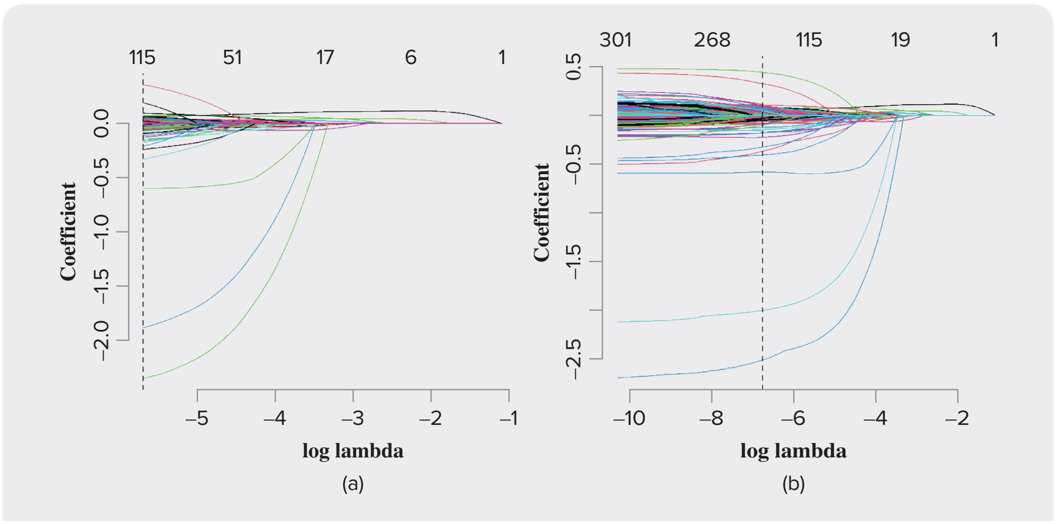


FIGURE 3.9 The Lasso regularization paths for Ames house sale price using `gamlr` with default `lmr=1e-2` in (a), and with `lmr=1e-4` in (b). In our analysis we use the specification in (b) such that $\lambda_T/\lambda_1 = 1/10000$.

```
> length(fitAmes$lambda)
[1] 100
> fitAmes$lambda[100]/fitAmes$lambda[1]
seg100
 1e-04
```

The fitted coefficients for each segment, $\hat{\boldsymbol{\beta}}_t$, are columns of the `beta` attribute.

```
> dim(fitAmes$beta)
[1] 339 100
> sum(fitAmes$beta[,100]!=0)
[1] 300
```

At the end of the path there are 300 nonzero $\hat{\beta}_{100,j}$ coefficients. This plus the intercept gives the 301 nonzero parameters marked on the left of the header in Figure 3.9b. We can pull out some coefficients from the first and last two segments.

```
> fitAmes$beta[
+ c("Overall.Qual","Lot.Area","Lot.Frontage.x","Lot.Frontage.
miss"),
+ c(1:2,99:100)]
4 x 4 sparse Matrix of class "dgCMatrix"
                     seg1      seg2        seg99       seg100
Overall.Qual            . 0.0211872 0.0497281614 0.0497218631
Lot.Area                .         . 0.0864251705 0.0864501758
Lot.Frontage.x          .         . 0.0001757198 0.0001758566
Lot.Frontage.miss       .         . 0.0049547699 0.0049655247
```

The first variable to enter the path with a nonzero coefficient is `Overall.Qual`, which rates the overall finish and construction of the house from 1 (very poor) to 10 (very excellent). Clearly the condition of your house has a major impact on its sale price. The lot size and frontage variables do not get nonzero coefficients until later in the path. Notice that the missingness indicator `Lot.Frontage.miss` has a positive estimated coefficient. The estimation has determined that the missing frontage entries tend to occur for more expensive properties.

Again, we emphasize that path estimation is just the first step of model construction. You don't know which path segment is most useful for prediction until you apply the model selection techniques of later sections.

Example 3.5 Telemarketing Data: Lasso Logistic Regression To illustrate Lasso logistic regression for a binary response, we will use the Telemarketing data from Moro et al. (2011). The `telemarketing.csv` data set consists of observations from the marketing campaign of a

Portuguese bank that was trying to get customers to subscribe to a term deposit product. Term deposits are basically loans to the bank: you deposit money that you will not be able to access for an agreed length of time, and in exchange the bank pays you a higher interest rate on this deposit than you would get on money that you can access at any time.

We read the data in as the `tlmrk` data frame.

```
> tlmrk <- read.csv("telemarketing.csv", strings=T)
> dim(tlmrk)
[1] 4521   15
> tlmrk[1,]
  age        job marital education default balance housing loan
1  30 unemployed married   primary      no    1787      no   no
   contact  campaign   durmin previous poutcome pweek subscribe
1 cellular        1 1.316667        0  unknown     0         0
```

The variables here include demographic information and finances such as their account balance and whether they have an existing `housing` or personal `loan`. We also have data about the marketing campaign for this customer, including

- `campaign`: number of contacts performed during this marketing campaign for this client.
- `contact`: the format of the most recent contact (land-line or cell phone).
- `durmin`: the length of time in minutes for the most recent phone conversation with the customer.
- `previous`: the number of previous contacts with this customer before the current marketing campaign.
- `poutcome`: the outcome of the previous marketing campaign for this customer.
- `pweek`: the weeks that have passed since the customer was last contacted during the previous campaign.

All of this data is important for designing marketing campaigns. People do not usually like getting calls from telemarketers. When designing a campaign, you want your sales agents calling only people who are likely to subscribe and not bothering those who are not. In addition, variables like `durmin` tell you how long it takes your agents to close the deal and when they are spending too much time on the phone. It is a waste of time and money for everyone to spend too long talking to a customer who will not be convinced to subscribe.

Our response of interest is the binary variable `subscribe` that indicates whether or not the customer ended up subscribing to a term deposit. We pull it out as the variable `yTD`.

```
> yTD <- tlmrk$subscribe
> mean(yTD)
[1] 0.11524
```

Roughly 11.5% of the customers end up subscribing to the term deposits.

There are no `NA` entries in this data, so we don't need to worry about missing data imputation. However, we will still run `naref` to create `<NA>` as the reference level for factor variables. Since the data contain the level `unknown` for many factors, we could have also gone through and set that as the reference level. However, it is convenient to simply call `naref` and you get the result that all factor effects will be interpretable as variations around an overall average intercept.

```
> sum(is.na(tlmrk))
[1] 0
> library(gamlr)
Loading required package: Matrix
> tlmrkX <- naref(tlmrk[,-15])
> levels(tlmrkX$job)
 [1] NA           "admin."     "blue-collar" "entrepreneur"
 [5] "housemaid" "management" "retired"     "self-employed"
 [9] "services"  "student"    "technician"  "unemployed"
[13] "unknown"
```

Notice that we removed column 15 when creating the `tlmrkX` data frame using `naref`. This was the column containing our response, `subscribe`, so that now `tlmrkX` contains only the input variables.

Building the Model Matrix

Now we are ready to build our numeric model matrix. This time, we will include all of the input variables interacted with each other and create a new variable as `durmin^2` to allow the probability of success to change quadratically (as a curve) during the phone call. We use `I(durmin^2)` in the regression formula to tell `R` to create a new variable named `I(durmin^2)` representing the square of `durmin`.

```
> xTD <- sparse.model.matrix(~.^2 + I(durmin^2), data=tlmrkX)
> dim(xTD)
[1] 4521  656
```

The resulting model matrix has $n = 4521$ observations on $J = 656$ columns.

Fitting the Logistic Lasso Regression Path

You can fit a logistic Lasso path with `gamlr` by specifying `family="binomial"`.

```
> fitTD <- gamlr(xTD, yTD, family="binomial")
```

The resulting Lasso path is shown in Figure 3.10a. We kept the default `lmr=1e-2` here such $\lambda_{100}/\lambda_1 = 1/100$. As before, the fitted `gamlr` object contains all of the estimated $\hat{\boldsymbol{\beta}}_t$ coefficients for each segment. You can pull out any specific segment, and Figure 3.10b shows the fitted $\hat{y}$

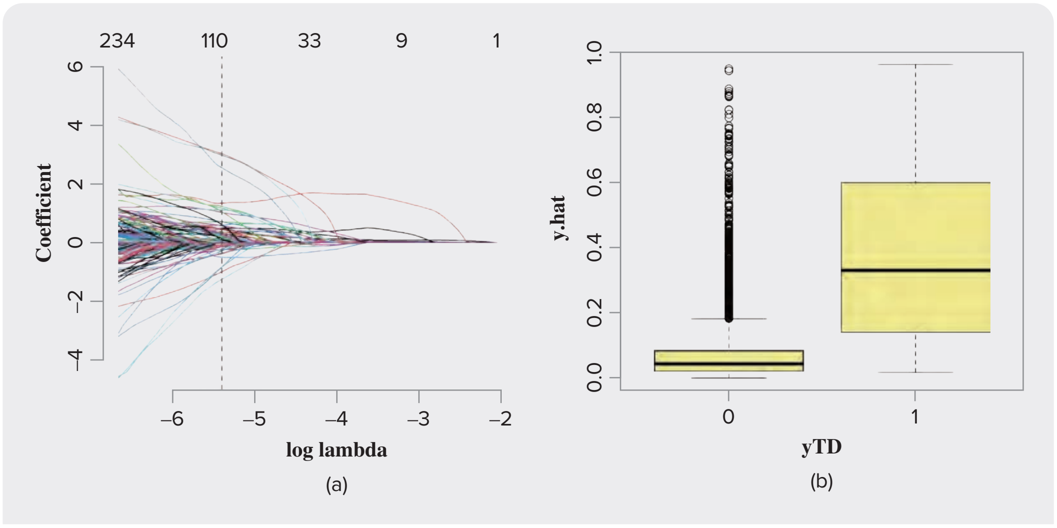


FIGURE 3.10 Panel (a) shows the Lasso regularization path for prediction of telemarketing call success, and (b) shows the fitted values corresponding to the AICc selected segment of coefficients at the dashed line in (a).

probabilities of success corresponding to the segment at the dashed line in Figure 3.10a (this is the AICc selected segment; see the next section for details). In the code below we pull out the first and last two segment coefficients on `durmin` and `I(durmin^2)`, the minutes and squared minutes spent on the phone call.

```
> fitTD$beta[c("durmin","I(durmin^2)"),c(1:2,99:100)]
2 x 4 sparse Matrix of class "dgCMatrix"
             seg1       seg2         seg99       seg100
durmin          .  0.01244651   0.338559757  0.340290880
I(durmin^2)     .          .  -0.009229655 -0.009359627
```

The minute length `durmin` is the first variable to enter the path with a nonzero coefficient, indicating that this variable has strong predictive signal on the outcome of the phone call.

3.3 Model Selection

The Lasso is used to obtain paths of promising candidate variables. Running `gamlr` with the default `nlambda=100` gives us 100 vectors of fitted coefficients $\hat{\boldsymbol{\beta}}_t$ based on 100 different penalties λ_t. Once you have this path, you need to do *model selection* to choose the best vector of coefficients to use for prediction. There are a number of different ways to do this, but all can be thought of as having the same motivation: select the model that does the best job predicting out of sample. The term "best job" will have different meanings in different applications. In some cases, you will simply want to get the best (lowest) average OOS deviance. In other cases, you might be motivated to trade some average performance for a more robust model—a simpler model that may have slightly higher OOS deviance but has less probability of producing the occasional really bad prediction. Regardless, you are always using some concept of predictive performance as the foundation for your model selection.

The benefit of the Lasso is that you have *indexed* your potential models with a single parameter: λ. This penalty weight is a *signal-to-noise filter.* It works like the squelch on a VHF radio (or, to be a bit more contemporary, the noise canceling level on a cellular phone). If you turn it all the way up, you don't hear anything. If you turn it all the way down, you hear only static. The key to being able to communicate on a radio is finding the sweet spot in the middle where you hear the other person's voice and none of the background noise. It is the same for good statistical prediction: you need to find the λ that gives you good signal with little noise. Looking at the Lasso path plots, "all the way up" is the far right where all coefficients are zero, and "all the way down" is the far left where most coefficients are nonzero.

When you do model selection on Lasso paths, you should think about what you are doing as *selecting the best* λ_t. In contrast to subset selection, where you need to consider all possible combinations of input variables (or a greedy search through possible combinations), with the Lasso you are considering selection only for this tuning parameter: the penalty weight that acts as your signal-to-noise filter. In this section we will introduce two different frameworks for selecting λ. The first framework is built on *information criteria* that combine in-sample deviance with the model degrees of freedom to estimate OOS predictive performance. The second framework is built around the sort of cross-validation experiments that we saw earlier in this chapter, where you split the data into folds and use OOS predictive performance across folds to get an estimate of future OOS performance.

3.3.1 Information Criteria

Information criteria (IC) are theoretical approximations to what OOS deviance you can expect when using your model to predict new data. Compared to running cross-validation experiments, IC estimates of OOS performance have the advantage of being fast (you need to fit the Lasso path only once, to your original dataset) and deterministic (there is no Monte Carlo variation due to random sampling). We use information criteria a lot in this book, especially the AICc that will be introduced below, and our practical experience is that this is a convenient and robust foundation for model selection.

There are many information criteria out there. We will look at Akaike's AIC, its corrected version AICc, and the Bayesian BIC. All of these IC attempt to approximate the distance between a fitted model and draws from the "true model" using different analytic approximations. That is, the IC are all attempting to approximate the OOS deviance (the distance between new data and the fitted model). Since the IC measure a distance, you can apply them in model selection by choosing the model with minimum IC.

The information criteria all take the form

$$\mathrm{IC}(\hat{\boldsymbol{\beta}}) = \mathrm{dev}(\hat{\boldsymbol{\beta}}) + \kappa \cdot df \qquad \textbf{(3.8)}$$

where $\mathrm{dev}(\hat{\boldsymbol{\beta}})$ is the *in-sample* deviance and df is the *model degrees of freedom*: the number of observations that the procedure you used to estimate $\hat{\boldsymbol{\beta}}$ should be able to fit exactly (refer back to Chapter 1 for more discussion on the model degrees of freedom). As we will describe below, df for the Lasso is equal to the number of nonzero estimated parameters.

Note that the deviance calculations used for Eqution (3.8) require you to use the full deviance formula that expands the constant C from Equation (1.26). This constant includes a bunch of negative log likelihood terms that don't change with $\hat{\boldsymbol{\beta}}$. However, this will be taken care of by the `R` functions you use to calculate ICs.

The term κ in (3.8) is the IC *complexity penalty:* it is the price you pay for adding additional degrees of freedom to your model. Equation (3.8) looks similar to the penalized deviance equations, like (3), that we used to define our Lasso regression estimator. And it is similar! You are combining an in-sample deviance with a price on complexity. You apply the IC equation to evaluate models fit to optimize a penalized deviance, and the value of κ that you want to use is derived from theory rather than being a tuning parameter that you estimate from the data.

Recall the simulated data example that opened this chapter. We aplied models with 2, 3, and 19 parameters (linear, quadratic, and 18-degree polynomial regression) fit to a cloud of points. The model selection goal is to select the "just right" model in Figure 3.1 that fits the persistent pattern but doesn't overfit to noise. In this example, which used OLS to estimate the coefficients, the model degrees of freedom is equal to the number of parameters in each model (2, 3, and 19). The correct value of κ in our IC of Equation (3.8) should lead to the IC being lowest for the 3-parameter quadratic model. Even though the 18-degree polynomial fits the data perfectly (and has zero in-sample deviance), you should have that the complexity penalty $\kappa \cdot 19$ is large enough to compensate for this low in-sample deviance.

The AIC

A common IC is *Akaike's* information criterion, the AIC. The AIC is an attempt to approximate the average OOS deviance on new data. Through a lot of theoretical statistics work, Akaike (1973) determined that the "right" complexity penalty to approximate the OOS deviance is simply $\kappa = 2$.

$$\text{AIC}(\hat{\boldsymbol{\beta}}) = \text{dev}(\hat{\boldsymbol{\beta}}) + 2 \cdot df \qquad \textbf{(3.9)}$$

This AIC score is an output of many standard statistical software routines. For example, in Example 3.1, in the semiconductor dataset, we used `glm` to fit the 25-input `cut` logistic regression model. The printed information from calling `summary` on the `cut` object includes the AIC.

```
> summary(cut)
...
    Null deviance: 731.59  on 1476  degrees of freedom
Residual deviance: 599.04  on 1451  degrees of freedom
AIC: 651.04
> 599.04 + 2*26
[1] 651.04
```

The AIC here is equal to the in-sample deviance, 599.04, plus two times the number of parameters in the model (26 for the 25 inputs plus the intercept). Recall that, confusingly, what `R` calls degrees of freedom here is actually the residual degrees of freedom: the number of opportunities to observe error variability around the fitted model, or $n - df$ in our notation. There are $n = 1477$ observations here, and $1477 - 26 = 1451$ as in the `R` output (note that the residual degrees of freedom for the null model is $n - 1$ since it just fits a single mean response parameter).

In maximum likelihood estimation, as applied inside `glm`, the df is simply the number of parameters in the model. More generally, the df measures the in-sample correlation between $\hat{y}$

and *y*—how much flexibility you have to make the model fit look like the observed data. For important theoretical reasons (Zou et al., 2007) the Lasso, like for MLE fitted models, has *df* simply equal to the number of nonzero $\hat{\beta}_j$ at a given λ. This is *not* true for any other penalization cost function. For example, if you use a ridge penalty in your penalized deviance minimization then all coefficients will be nonzero, but *df* will be less than the full model dimensions because the coefficients are shrunk toward zero.

It is a massive advantage for the Lasso that you have a simple measure of the number of degrees of freedom (the number of nonzero estimated parameters) at a given λ penalty weight. This fact lets you apply ICs to choose the best Lasso model. For example, `R` has the `AIC` function that you can apply to a fitted `gamlr` object. Here we apply it to the telemarketing example Lasso path from Example 3.5.

```
> AIC(fitTD)
    seg1     seg2     seg3     seg4
3233.000 3173.971 3123.801 3081.680
...
   seg97    seg98    seg99   seg100
2293.355 2297.469 2295.603 2296.141
> which.min(AIC(fitTD))
seg92
   92
> sum(fitTD$beta[,92]!=0)
[1] 185
> fitTD$lambda[92]
      seg92
0.001858245
```

The minimum AIC score occurs for the 92nd segment, with $\lambda_{92} \approx 0.00186$ and 185 nonzero coefficients in $\hat{\boldsymbol{\beta}}_{92}$.

The Corrected AICc

The AIC is the most commonly applied IC, but *you should not use the AIC*. We introduced the AIC only as a stepping stone to the superior corrected AIC that we now describe. When you have a lot of potential parameters in your model, the AIC will tend to overfit. The reason why this happens is worth understanding because it motivates the IC that you *should* use: the corrected AICc. The AIC overfits because the actual κ that Akaike derives as optimal for linear regression is

$$\kappa = 2\mathbb{E}\left[\frac{\sigma^2}{\hat{\sigma}^2}\right] \tag{3.10}$$

where σ^2 is the true error variance and $\hat{\sigma}^2$ is the variance of your fitted residuals. Akaike made the simplifying assumption that the variance of the residuals is a good approximation to the true error variance, such that $\hat{\sigma}^2 \approx \sigma^2$ and you can simplify Equation (3.10) to say $\kappa \approx 2$. However, an overfit model (e.g., our 18-degree polynomial from the chapter opening) will have very small residuals because the model is fitting the noise. That implies that σ^2 will be much bigger than $\hat{\sigma}^2$ and $\kappa = 2$ will be too small a complexity penalty.

It turns out that you can actually predict the ratio of variances in (3.10) as

$$\mathbb{E}\left[\frac{\sigma^2}{\hat{\sigma}^2}\right] \approx \frac{n}{n - df - 1} \tag{3.11}$$

This approximation comes from the theoretical definition of degrees of freedom. It leads to the *corrected AIC* (Hurvich and Tsai, 1989) with $\kappa = 2n/(n - df - 1)$,

$$\text{AICc}(\hat{\boldsymbol{\beta}}) = \text{dev}(\hat{\boldsymbol{\beta}}) + \frac{n}{n - df - 1} 2 \cdot df \tag{3.12}$$

You *should* use the AICc for IC-based model selection. Although motivated using a ratio of error variances in linear regression, it also works for logistic regression or any other generalized linear model (fit via likelihood maximization or with Lasso estimation). Notice that for big *n*/*df,* the AICc becomes similar to the AIC as the ratio $n/(n - df - 1)$ gets closer to one. Hence, the AIC will work well for large *n*/*df* (which is the classical statistics setting where it was developed) while the corrected AICc works for any *n*/*df* you encounter.

The `gamlr` package uses AICc for selection by default. The AICc selected segment is marked on the path plot with a vertical line (see Figures 3.9 or 3.10a, for example), and if you call `predict` or `coef` on a fitted `gamlr` object it will give you results corresponding to the AICc minimizing path segment.

The BIC

Before diving into AICc selection for our Lasso examples, we note one additional IC that you will sometimes encounter. The BIC, where B stands for Bayes, is motivated from the Bayesian inference ideas we outlined in Chapter 2. Instead of attempting to predict the average OOS deviance for candidate models, as the AIC and AICc do, the BIC is attempting to approximate the *posterior probability* that each model is best. This subtle difference leads it to select more simple (fewer parameter) models than you get from the AICc. A more complex model might have lower expected OOS deviance, but higher variation in OOS deviance around this expectation. For example, you have higher probability of getting a bad model fit from an unlucky training sample when you are working with more model complexity.

Schwarz et al. (1978) developed the BIC around the same time as Akaike's work on the AIC. The Schwarz complexity penalty is $\kappa = \log(n)$, such that

$$\text{BIC}(\hat{\boldsymbol{\beta}}) = \text{dev}(\hat{\boldsymbol{\beta}}) + \log(n) \cdot df \tag{3.13}$$

Although the BIC can be useful for model selection in small sample settings, for most applications the $\log(n)$ complexity penalty tends to be too large. That is, the BIC will tend to underfit and choose overly simple models. You can treat it as giving a lower bound on the amount of useful model complexity.

Example 3.6 Ames Housing Data: IC Model Selection Returning to the house price prediction of Example 3.2, we have `fitAmes` as our `gamlr` object containing the fitted Lasso path. Since `gamlr` uses the AICc as its default model selection rule, if you call `coef` on this object you will get the coefficients corresponding to the AICc-minimizing path segment.

```
> bAmes <- coef(fitAmes) ## coefficients selected under AICc
> head(bAmes)
6 x 1 sparse Matrix of class "dgCMatrix"
                        seg62
intercept          4.5352809
MS.ZoningA (agr)  -0.3664627
MS.ZoningC (all)  -0.1779210
MS.ZoningFV        .
MS.ZoningI (all)   .
MS.ZoningRH        .
```

The output is in a `Matrix` library sparse format. We will drop the intercept and convert it to a dense vector before exploring the selected $\hat{\boldsymbol{\beta}}$ coefficients.

```
> bAmes <- bAmes[-1,]
> sum(bAmes!=0)
[1] 195
> tail(sort(bAmes),3) ## big increaser
Kitchen.QualPo Exterior.1stPreCast NeighborhoodGrnHill
     0.1132968           0.3267350           0.4425938
> bAmes[c("Lot.Area","Lot.Frontage.x","Lot.Frontage.miss")]
     Lot.Area    Lot.Frontage.x Lot.Frontage.miss
 7.525206e-02      9.585679e-05      0.000000e+00
```

The AICc selects a model with 195 nonzero coefficients. The largest positive coefficient is the effect of the property being in the Green Hill neighborhood, and we notice that in the selected model the missingness indicator for `Lot.Frontage` has a zero coefficient: the AICc has decided that the fact that this variable is missing is not a useful predictor of house price. You can call the `AICc` function on `fitAmes` to see the AICc values underlying this selection.

```
> which.min(AICc(fitAmes))
seg62
   62
> fitAmes$lambda[62]
      seg62
0.001154232
```

We find that the AICc has selected $\lambda_{62} \approx 0.00115$ as the best penalty weight.

The `predict` function also uses AICc selection by default. For example, if we call `predict` on `fitAmes` for the 1st and 11th observations we get (after exponentiating the predicted log price) expected sale price values of around $210k and $170k.

```
> ( yhat <- predict(fitAmes, xAmes[c(1,11),]) )
2 x 1 Matrix of class "dgeMatrix"
      seg62
1  12.25147
11 12.05597
> drop(yhat)
       1       11
12.25147  12.05597
> exp(drop(yhat))
       1       11
209289.1  172123.5
```

Notice that we applied `drop` to `yhat` to convert the predictions from special sparse `Matrix` format to a simple vector. You will often want to take this step when working with predictions from `gamlr`.

To use the other IC for model selection, you apply the appropriate function to the fitted `gamlr` object to get the set of IC scores and then find the segment with the minimum score. This segment can then be passed to `coef` (and to `predict`) with the `select` argument to get results for that specified path segment.

```
> (bicsel <- which.min(BIC(fitAmes)))
seg48
   48
> bAmesBIC <- coef(fitAmes, select=bicsel)[-1,] ## and BIC
> sum(bAmesBIC!=0)
[1] 95
```

We see here that the BIC selects λ_{48} which corresponds to 95 nonzero coefficients (about half of what the AICc selected). The AIC ends up selecting the same segment as the AICc: that with 195 coefficients at λ_{62}.

```
> (aicsel <- which.min(AIC(fitAmes)))
seg62
   62
```

The AIC and AICc give the same results here because n is much larger than the full potential df (we have 2930 observations on 339 input dimensions), such that the correction ratio is close to one: $2930/(2930 - 339 - 1) \approx 1.13$. All of the IC surfaces are plotted in Figure 3.11a. The AIC and AICc scores are very similar, again due to the high n/df ratio.

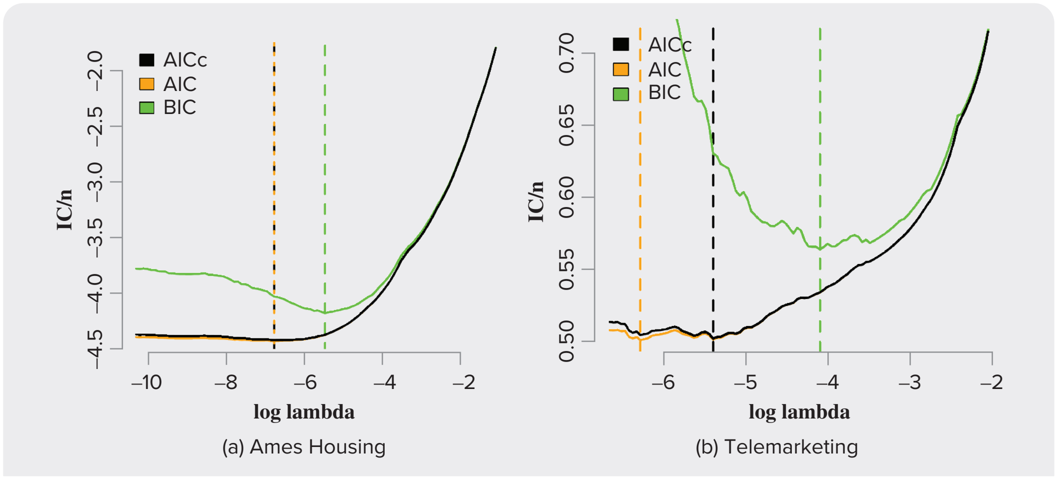


FIGURE 3.11 Information criteria scores as a function of the λ_t penalty weights for each of our Ames Housing (a) and Telemarketing (b) examples.

We are presenting all of the options here so that you can understand what to expect when you encounter AIC, AICc, or BIC model selection in practice. However, you should use the AICc as your default IC for model selection.

Example 3.7 Telemarketing Data: IC Model Selection Turning to logistic regression and our telemarketing example, recall that we have the fitted `gamlr` object `fitTD` for prediction of call success (getting the customer to subscribe to a term deposit) as a function of customer characteristics and campaign information.

The default AICc selection chooses a model with 90 nonzero coefficients.

```
> bTD <- coef(fitTD)[-1,] ## coefficients selected under AICc
> sum(bTD!=0)
[1] 90
```

Notice that we appended `[-1,]` on the exctracted coefficients to remove the intercept and drop the `Matrix` formatting. We can pull out the coefficients on the length of the phone call.

```
> bTD[c("durmin","I(durmin^2)")]
      durmin   I(durmin^2)
 0.276468974 -0.004644089
```

The AICc selected model has a positive coefficient on `durmin` and a negative coefficient on `durmin`^{2}, such that the odds of success will increase with the beginning of the phone call but eventually start to decrease as time progresses.

The BIC selects a much simpler model with only 20 nonzero coefficients and the AIC selects a much more complex model with 185 nonzero coefficients.

```
> bTDbic <- coef(fitTD, select=which.min(BIC(fitTD)))[-1,]
> sum(bTDbic!=0)
[1] 20
> bTDaic <- coef(fitTD, select=which.min(AIC(fitTD)))[-1,]
> sum(bTDaic!=0)
[1] 185
```

All three IC scores are shown as a function of λ in Figure 3.11b. The AIC and AICc match up at large λ, where the correction $n/(n - df - 1) \approx 1$, but then start to separate from each other at larger values. Both AIC and AICc surfaces have a very different shape from the BIC.

3.3.2 Cross-Validation for Lasso Paths

An alternative to IC model selection is to apply the sort of cross-validation (CV) experiment we described in Algorithm 3.1. For Lasso paths, you want to design a CV experiment to evaluate the OOS predictive performance of different λ penalty values. This is in contrast to CV for subset selection, where you evaluated different pre-set subsets of variables inside the CV experiment. To execute Algorithm 3.1 for Lasso paths, you need to

- Fit the Lasso path for the full dataset to get a grid of candidate λ_t penalties.
- Run a CV experiment where you split your data into K folds and apply these λ_t penalties in Lasso estimation on the training data excluding each fold. Record OOS deviances for prediction on each left-out fold.
- Select the λ_t with "best" OOS performance. Your selected model is defined by the corresponding $\hat{\boldsymbol{\beta}}_t$ coefficients that were obtained through Lasso estimation on the full dataset with penalty λ_t.

How Many Folds?

A common question around CV is *How do I choose K?* The short answer is that more is better (it reduces the Monte Carlo variation due to random fold assignment) but only up to a point. Using too many folds gets computationally very expensive. Moreover, using too many folds (anything approaching $K = n$) gives bad results if there is even a tiny amount of dependence between your observations. Smaller values of K lead to CV that is more robust to this type of mis-specification.

To figure out how many folds is "enough," note that the variance on your CV estimate of *average* OOS deviance is the variance of the K OOS deviances on each left-out fold divided by $\sqrt{K}$ (recall from Chapter 2 that the variance on a mean is the variance of the observations divided by the square root of the number of observations). If you run your CV experiment and the uncertainty around average OOS deviance is larger than you want, then you can re-run the experiment with more folds. However, if adding a small number of folds doesn't significantly reduce the uncertainty then you are probably better off using the AICc for model selection.

CV Path Plots

Once again, this is all easiest to understand visually. The `gamlr` library provides the `cv.gamlr` function to run CV experiments for Lasso paths. This function uses the exact same syntax as

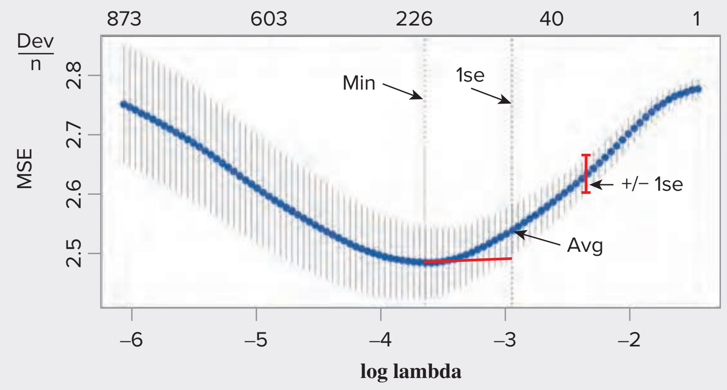


FIGURE 3.12 A cross-validated Lasso path plot from `cv.gamlr`. The blue dots are mean OOS deviances (here MSE, as this is from a linear regression) and the error bars mark ±1 standard error around the estimated average OOS deviance. The CV-min and CV-1se selection rules are marked with vertical dashed lines.

the standard `gamlr` function, and you can pass it any arguments that you would pass to `gamlr`; these arguments are used for the full sample path and for the reduced-sample fits inside the CV experiment. Figure 3.12 shows the results of `cv.gamlr` on a linear regression example. Just like the path plot, the CV plot has λ on the x axis and the degrees of freedom (number of nonzero coefficients) on the top. The average OOS deviances are marked with blue dots, and error bars are extended one standard error on each side of these estimates of the expected OOS deviance. From our discussion above, the standard errors are calculated as the variance of the K OOS deviances at each λ_t value, divided by $\sqrt{K}$. The default K for `cv.gamlr` is `nfold=5`. If this results in error bounds that are too large for you to decide which λ is best, then you can simply increase the number of folds to reduce the standard error on your expected OOS deviance estimates.

Given the results from `cv.gamlr`, there are two common options for how you select the optimal λ_t (and hence select your coefficients $\hat{\beta}_t$). The CV-min rule, shown as the leftmost dashed line in Figure 3.12, simply selects the λ_t corresponding to the smallest *average* OOS deviance. The CV-1se rule, shown as the rightmost dashed line in Figure 3.12, selects the *biggest* λ_t with average OOS deviance no more than one standard error away from the minimum. For most applications, we recommend using the CV-min rule. This is the best choice if you are focused on OOS predictive performance. The `1se` rule is more *conservative:* it hedges toward a simpler model. This can be used if you have a heightened worry about accidentally including useless coefficients in your model. The CV-1se rule is the default in `cv.gamlr` (and in `cv.glmnet`) but we will often specify that we want to use CV-min selection instead.

Example 3.8 Ames Housing Data: CV Lasso Selection To run a CV Lasso for the Ames house sale price regression, you can use the same model matrix `xAmes` and response `yAmes` that you previously used as input to `gamlr`. In Example 3.4 we specified `lmr=1e-4` to get smaller λ_t values than the default, and we will pass that same argument to `cv.gamlr`.

```
> ### cross validation
> set.seed(0)
> cvfitAmes <- cv.gamlr(xAmes, yAmes, verb=TRUE, lmr=1e-4)
fold 1,2,3,4,5,done.
```

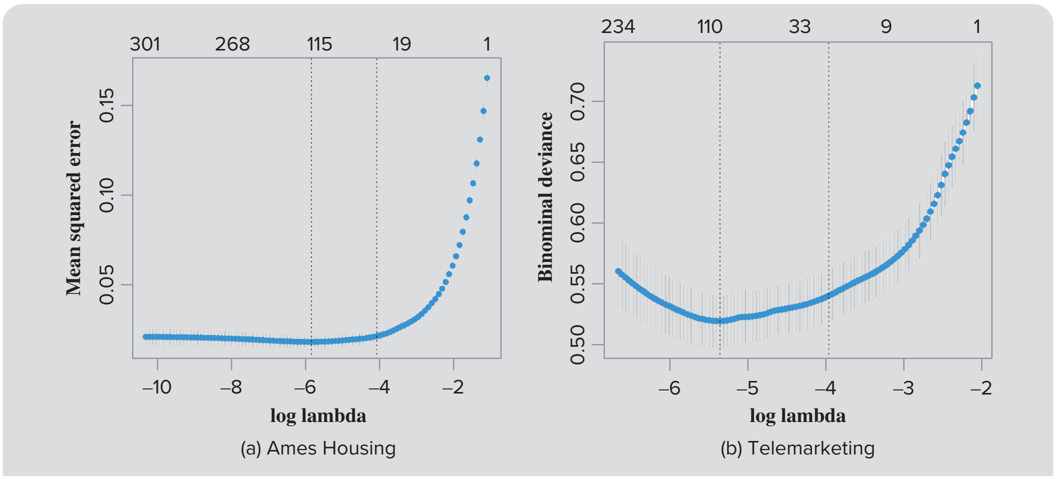


FIGURE 3.13 Cross-validated OOS deviance as a function of the λ_t penalty weights for each of our Ames Housing (a) and Telemarketing (b) examples.

The `verb=TRUE` argument leads `cv.gamlr` to output a progress report as it progresses through the folds. Notice that we set a random seed here so that if you run this code you should get the exact same results. The CV experiment results are plotted in Figure 3.13a. We can work through the attributes of the `cvfitAmes` object output from `cv.gamlr` to understand all of the pieces here.

```
> attributes(cvfitAmes)
$names
 [1] "gamlr"   "family"    "nfold"     "foldid"     "cvm"
 [6] "cvs"     "seg.min"   "seg.1se"   "lambda.min" "lambda.1se"
> cvfitAmes$gamlr
gaussian gamlr with 339 inputs and 100 segments.
$class
[1] "cv.gamlr"
> cvfitAmes$nfold
[1] 5
```

There is a `gamlr` object contained within `cvfitAmes`. This is the fitted Lasso path for the full data sample, and it is exactly the same as the `fitAmes` object that we estimated in Example 3.4. The `nfold` attribute here is *K*, the number of folds, and it is set to 5 by default.

The `cvm` attribute contains mean OOS deviances and `cvs` contains their standard errors. These are vectors with one value for each of the `nlambda` λ_t values used in the full sample path.

```
> cvfitAmes$cvm
  [1] 0.16532834 0.14703130 0.13104589 0.11777469
...
 [97] 0.02118427 0.02121335 0.02122546 0.02123655
> cvfitAmes$cvs
  [1] 0.004943823 0.004566041 0.003893101 0.003353137
...
 [97] 0.004669086 0.004688207 0.004690687 0.004694711
```

The `cvm` vector contains the blue dots marked on Figure 3.13a and `cvs` contains the half-length of the gray error bars. In this case the standard errors are small relative to the range of `cvm` deviance estimates, and you need to squint at the plot to see them. You can also use the `cvm` OOS deviance estimates to get estimates for the OOS R^2. We know that the null model corresponds to the first Lasso path segment where $\hat{\boldsymbol{\beta}}_1 = 0$ by design. Thus an estimate of the OOS R^2 at any λ_t is available as one minus the corresponding mean OOS deviance over the first element of `cvm`.

```
> 1 - cvfitAmes$cvm[100]/cvfitAmes$cvm[1]
[1] 0.8715493
```

The OOS R^2 at the end of our Lasso path (at λ_{100}) is around 87%.

The λ penalties selected by CV-min and CV-1se rules (corresponding to the two vertical dashed lines in Figure 3.13a) are in attributes `lambda.min` and `lambda.1se` respectively. These correspond to the segment indices in `seg.min` and `seg.1se`.

```
> cvfitAmes$seg.min
[1] 52
> log(cvfitAmes$lambda.min)
[1] -5.833983
> cvfitAmes$seg.1se
[1] 33
> log(cvfitAmes$lambda.1se)
[1] -4.066342
```

The CV-min rule selects the 52nd segment with $\log(\lambda_{52}) \approx -5.8$, and the CV-1se rule selects the 33rd segment with $\log(\lambda_{33}) \approx -4.1$. You can pass `select=“min”` or `select=“1se”` to `coef` and `predict` functions to access coefficients and predictions corresponding to the models selected under each rule.

```
> bAmesCVmin <- coef(cvfitAmes, select=“min”)[-1,]
> sum(bAmesCVmin!=0)
[1] 127
> bAmesCV1se <- coef(cvfitAmes, select=“1se”)[-1,]
> sum(bAmesCV1se!=0)
[1] 35
> cbind(bAmesCV1se,bAmesCVmin)[c(“Lot.Area”,“Lot.Frontage.x”),]
               bAmesCV1se bAmesCVmin
Lot.Area        0.0661714 0.07424132
Lot.Frontage.x  0.0000000 0.00000000
```

We see that the CV-min rule selects a model with 127 nonzero coefficients and the CV-1se model selects a model with 35 nonzero coefficients. Both selection rules lead to a nonzero coefficient on lot area and a zero coefficient on frontage. Note that the default for `cv.gamlr` is to use CV-1se so that if you don't specify `select` you will get the same results as for `select="1se"`.

Example 3.9 Telemarketing Data: CV Lasso Selection For the telemarketing logistic regression example, we apply `cv.gamlr` to our `xTD` model matrix and `yTD` binary response with `family="binomial"`. To illustrate the parallel computing capability of `cv.gamlr`, we also used the `parallel` library to create a parallel cluster and passed it as the `cl` argument. This will allow `cv.gamlr` to run each CV fold iteration in parallel across the processors on your computer (which is a handy speed-up if each Lasso path takes a while to fit).

```
> library(parallel)
> cl <- makeCluster(detectCores())
> set.seed(0)
> cvfitTD <- cv.gamlr(xTD, yTD, family="binomial", cl=cl)
```

The results of this CV experiment are plotted in Figure 3.13b. This time it is easier to see the error bars (from `cvfitTD$cvs`) around the mean OOS deviance values (from `cvfitTD$cvm`). To access select coefficients, we call the `coef` function and set the different CV selection rules via the `select` argument.

```
> betamin <- coef(cvfitTD, select="min")[-1,]
> sum(betamin!=0)
[1] 88
> beta1se <- coef(cvfitTD, select="1se")[-1,]
> sum(beta1se!=0)
[1] 18
```

The CV-min rule returns a model with 88 nonzero coefficients and the CV-1se rule returns a model with 18 nonzero coefficients.

To illustrate prediction, we will apply the `predict` function using each CV selection rule on the 1st and 100th observations. Note that, as was also necessary for `glm` fitted logistic regressions, we specify `type="response"` to get predicted probabilities rather than $\mathbf{x}'\hat{\boldsymbol{\beta}}_t$. This outputs predicted probabilities that the campaign resulted in a successful subscription for each customer.

```
> yTD[c(1,100)]
[1] 0 1
>
drop(predict(cvfitTD,xTD[c(1,100),],select="min",type="response"))
          1         100
```

```
0.04862848 0.42464789
> drop(predict(cvfitTD,xTD[c(1,100),],select="1se",type="response"))
         1         100
0.05605473 0.36512179
```

Both methods give a probability less than 0.06 for $\hat{y}_1$ (true $y_1 = 0$) and above 0.36 for $\hat{y}_{100}$ (true $y_{100} = 1$). The CV-min selected predictions fit these observations a bit more tightly than the CV-1se selected predictions (CV-min $\hat{y}$ is lower for the true failure and higher for the true success).

The techniques of this chapter give you a diversity of tools for selecting from a path of candidate models. Figure 3.14 shows the segments selected under all of these different selection rules for both the Ames housing and telemarketing examples. For most applications, we recommend using either the AICc or CV-min selection rules. The results for the Telemarketing example, in Figure 3.14b, are typical in that the AICc and CV-min rules select very similar λ values. Figure 3.14a shows that the rules give more varied selections in the Ames housing example, where the AICc selects $\log(\lambda) \approx -6.8$ and CV-min selects $\log(\lambda) \approx -5.8$. However, looking at the Ames Housing CV plot in Figure 3.13a you can see that the OOS deviance is nearly flat between these two λ values and thus both are expected to yield similar average OOS deviance. In the case where the AICc selects a model that your CV experiment predicts will do poorly OOS, then it could indicate that something weird is going on and that neither is giving a good result (e.g., this can happen when you have dependence between observations that is not incorporated into your model).

The BIC and CV-1se rules can be used if you have a strong preference for simpler models, e.g., if you want your model to be portable enough to provide predictions on slightly different data generating processes than that which produced your training sample. You should never use the AIC when you could instead use the AICc.

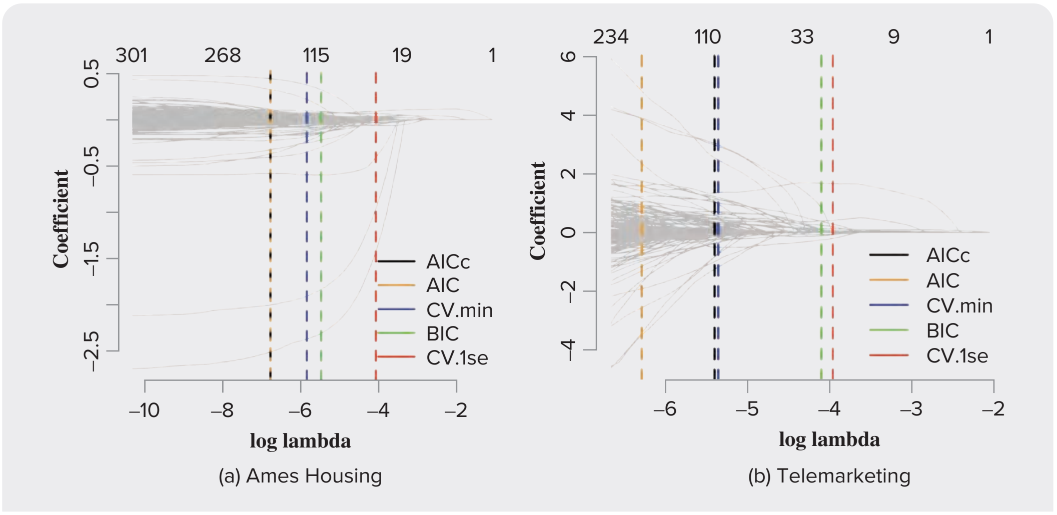


FIGURE 3.14 Path plots and the selected segments under all of our IC and CV selection rules, for each of the (a) Ames Housing and (b) Telemarketing examples.

3.4 Uncertainty Quantification for the Lasso

As a final topic for this chapter, we will touch on techniques for quantifying uncertainty *after* you have used model selection to choose a Lasso estimated regression. At the outset, we note that if you really care about the uncertainty for a set of parameters in your regression you should look to the purpose-built tools of Chapter 6. In that chapter, we introduce the techniques of "double ML" and "cross-fitting" that can be used to quantify uncertainty about treatment effects in the presence of high-dimensional controls—that is, to get uncertainty distributions for coefficients on specific input variables conditional on a larger set of other predictors.

That said, *if* you want to quantify uncertainty for functions or parameters of a fitted Lasso, then you can make use of some of the bootstrapping techniques that we introduced in Chapter 2. Unfortunately, the standard nonparametric bootstrap of Algorithm 2.2 does not work well for quantifying uncertainty of a Lasso path selection procedure. The issue with bootstrapping IC selection is that having repeated with-replacement samples of the same observations makes your dataset seem less noisy (easier to predict) than it actually is, and the theory behind the various IC selection rules implies that they will tend to select more complex models on the bootstrap resamples than they would on a true new sample from the original data generating process. For CV selection the issue is similar: if you naively bootstrap, then you can have the same observation repeated both in the kth CV training sample and in the left-out fold. This will make the OOS prediction seem easier than it actually is, and you will tend to select a more complex model inside the bootstrap than you would want for a true OOS prediction exercise.

You can adapt the bootstrap to work in this setting by replacing the random with-replacement sampling with a random re-weighting of the original observations. This is referred to as the *Bayesian bootstrap* (Rubin, 1981). In each bootstrap iteration, you generate n independent random weights from a *standard exponential* distribution. Draws from this distribution have the probability density function $\mathrm{p}(w) = e^{-w}$, with $\mathbb{E}[w] = 1$ and $\mathrm{var}(w) = 1$, and they are restricted to be positive ($w > 0$). The "Bayesian" moniker comes from how this is derived as the "right" distribution to use for the random weights under a specific prior model. Under this model, the standard exponential describes the posterior distribution for the prevalence of similar observations in new samples. However, you can think of it as simply a continuous extension of the standard bootstrap that is implicitly assigning discrete weights to the observations (0,1,2, etc; the number of times an observation occurs in each bootstrap sample).

Because the AICc (and other IC) functions are not set up to work with weights on the observations, applying this Bayesian bootstrap for IC selection would require you to write a bunch of custom `R` code. However, for linear regression only, `cv.gamlr` accepts the `obsweight` argument and these weights are applied at every step of the CV experiment (both in training and in evaluating OOS deviance). We will illustrate this below in quantifying uncertainty about predicted property prices in the Ames housing example. You can also use `cv.glmnet` with the `weights` argument to apply the Bayesian bootstrap for nonlinear models like logistic regression.

Another alternative is to use the parametric bootstrap of Section 2.4.2 where each bootstrap iteration involves simulating *new* observations from a model fit to the original sample. This is usually a bad idea for linear regression, because you need to make strong assumptions about the distribution of the random errors (e.g., Gaussianity and a constant error variance). However, the parametric bootstrap can be a decent option for logistic regression where the response distribution is a simple binomial. We will illustrate this approach for estimating the effect of phone call length on odds of success in the telemarketing example.

Example 3.10 Ames Housing Data: Bayesian Bootstrap We will consider uncertainty quantification for the expected sale price for two of the homes in our dataset. We are going to be studying the distribution for the CV-min selected sale price prediction.

First, calculate the original sample fitted predictions for sale price for these two properties.

```
> xnew <- xAmes[c(1,11),]
> yhat0 <- drop( predict(cvfitAmes, xnew, select="min") )
> exp(yhat0)
       1        11
204531.5 173117.9
```

Next, set the number of bootstrap estimates `B` and create a 2 by `B` matrix `yhatB` to fill with these estimates.

```
> B <- 100
> yhatB <- matrix(nrow=2, ncol=B)
```

We will also use the `parallel` library to create a parallel cluster `cl` that can be used by `cv.gamlr` to speed up each bootstrap iteration.

```
> library(parallel)
> cl <- makeCluster(detectCores())
```

We will now use a `for` loop to run through the bootstrap iterations. Note that the random weights are drawn in the first line inside this loop with a call to `rexp`. The other lines run `cv.gamlr` using these observation weights and then store the CV-min selected predictions in `yhatB` (and print a progress report, so you have something to watch while you wait).

```
> for(b in 1:100){
+     wb <- rexp(nrow(xAmes))
+     fitb <- cv.gamlr(xAmes, yAmes, obsweight=wb, lmr=1e-4,
cl=cl)
+     yhatB[,b] <- drop(predict(fitb, xnew, select="min"))
+     cat(b, " ")
+ }
1  2  3  4  5  ... 95  96  97  98  99  100
```

At the end of this loop, each column of `yhatB` contains a single bootstrap estimate of the expected log sale price for our two properties.

You can calculate a 95% probability interval for each house's sales price by exponentiating and looking at the 2.5th and 97.5th percentiles of the bootstrap sampled prices. We do this for each row of `yhatB` via the `apply` function.

```
apply(exp(yhatB), 1, quantile, probs=c(.025,.975))
          [,1]     [,2]
2.5%  190347.5 169952.7
97.5% 211883.9 175364.4
```

The expected sale price range for the first house is \$190k–\$212k and for the second house is \$170k–\$175k.

Alternatively, since we are exponentiating the predictions and introducing bias, you can apply the bias-corrected bootstrap of Algorithm 2.3 by taking percentiles on the distribution of bias-corrected sales prices, $e^{\hat{y}} - \left(e^{\hat{y}_b} - e^{\hat{y}}\right) = 2\,e^{\hat{y}} - e^{\hat{y}_b}$ where $\hat{y}$ is the original sample prediction and $\hat{y}_b$ is a bootstrap estimate.

```
> apply(2*exp(yhat0)-exp(yhatB),1,quantile,probs=c(.025,.975))
          [,1]     [,2]
2.5%  197179.1 170871.5
97.5% 218715.5 176283.1
```

The first house's expected sale price interval is now \$197k–\$219k and the second house's is \$171k–\$176k. Note that because of the randomness in our bootstrap procedure, your results will differ slightly when you replicate these procedures.

Example 3.11 Telemarketing Data: Parametric Bootstrap For our telemarketing example, we will consider uncertainty quantification for the effect of call duration in minutes (`durmin`) on the odds of success. Recall that we included both `durmin` and `durmin`^{2} as inputs in our regression. We will use the parametric bootstrap to obtain a sampling distribution for the AICc selected estimates for these parameters.

To run a parametric bootstrap, you need to build a function that simulates data from a fitted model. You want to use a model that is low bias (corresponds to a small λ) for this simulator even if it is potentially overfit. We will use the fitted probabilities from our smallest penalty fit, at λ_{100}, as the basis for simulating new realizations of success and failure for the marketing campaign.

```
> p0 <- drop( predict(fitTD ,xTD, type="response", select=100) )
```

These probabilities, `p0`, are then used in the `getBoot` simulator function to draw a new random response vector `yb` from a binomial distribution with `prob=p0`. The rest of the `getBoot` function fits a `gamlr` path for the simulated responses and returns the AICc selected coefficients on `durmin` and `durmin`^{2}. Note that the argument `b` to `getboot` doesn't do anything, it is just there for our convenience when calling this function inside `parSapply`.

```
> getBoot <- function(b){
+     yb <- rbinom(nrow(xTD),size=1,prob=p0)
+     fitTDb <- gamlr(xTD, yb, family="binomial")
+     coef(fitTDb)[c("durmin","I(durmin^2)"),]
+ }
> getBoot(1)
      durmin  I(durmin^2)
 0.154883844 -0.003303299
 > bTD[c("durmin","I(durmin^2)")]
      durmin  I(durmin^2)
 0.276468974 -0.004644089
```

The last two lines show a random draw of the coefficients from `getBoot` and the original sample AICc selected estimates for these same coefficients.

We will use `parSapply` and the `parallel` library to distribute 100 runs of `getBoot` across multiple processors on our computer. Note that, to use `parSapply`, we need to call the `clusterExport` function that copies a list of objects we will need inside `getBoot` to each processor in the cluster.

```
> ## run the bootstrap
> library(parallel)
> cl <- makeCluster(detectCores())
> clusterExport(cl, c("gamlr", "xTD", "p0"))
> betaB <- parSapply(cl, 1:100, getBoot)
> plot(t(betaB))
```

The output `betaB` is 2 by 100 matrix containing a parametric bootstrap sample of 100 realizations from the sampling distribution for these two coefficients (as estimated via an AICc Lasso). The bootstrap sample is plotted in Figure 3.15a. This is a *joint* distribution: the two coefficients are correlated with each other in their sampling distribution. For a given call length, the implied impact on the log odds of call success is available as the first coefficient times `durmin` plus the second times `durmin`^{2}. We create a grid of `durmin` values and evaluate and plot this function for each bootstrap estimate.

```
> grid <- seq(0,max(tlmrk$durmin),length=200)
> dmy <- apply(betaB, 2, function(b){ b[1]*grid+b[2]*grid^2 } )
> matplot(grid, dmy, col=8, type="l")
```

The bootstrap sample of effect curves is shown in Figure 3.15b along with the original sample estimated curve. This is the additive effect on the log odds of success (getting a customer to subscribe to a term deposit). We see that the call success probability is increasing until around 1/2 hour, and then after that it becomes decreasingly likely that the customer will subscribe. However, there is considerable variability around the effect of `durmin` for long phone calls (in the right half of Figure 3.15b) since we have few observations of calls that take longer than 30 minutes.

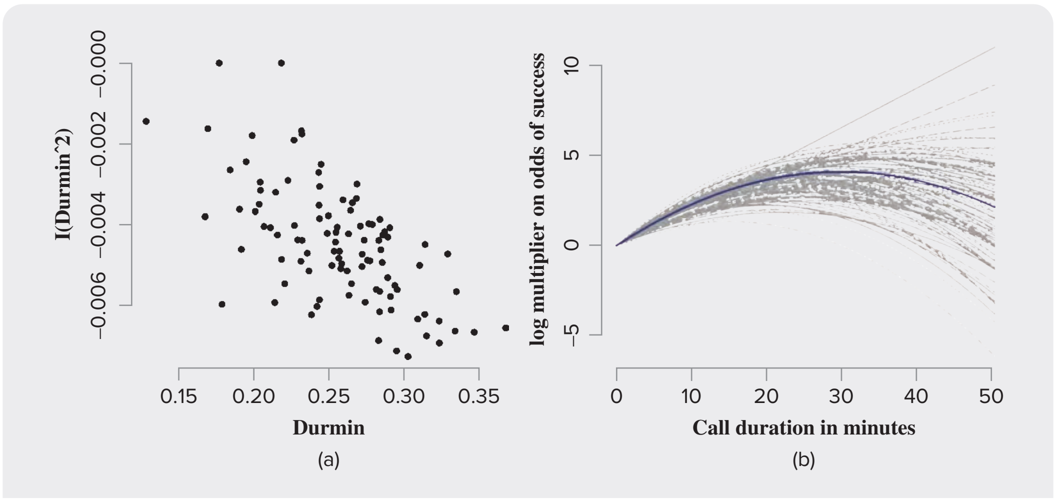


FIGURE 3.15 Parametric bootstrap sampling distribution for the coefficients on `durmin` and `durmin`^{2}, shown as a scatterplot in (a) and in terms of the implied effect on the log odds of success in (b). The original sample estimate is shown in (b) as the dark blue curve.

QUICK REFERENCE

This chapter presented many options for estimating paths of candidate models and for selecting among those candidates. We also introduced some key material for practical computational analysis, including sparse model matrices and dealing with missing data. There is a lot of content here, however we emphasize that the overall procedure is really simple:

- Fit a Lasso path to obtain multiple model estimates corresponding to different penalty values.
- Use CV or AICc to select the best model estimate from this path.

Everything else is context to help you adapt to practical difficulties and have a solid understanding of what is happening when you apply these techniques.

Key Practical Concepts

- Lasso regression models with coefficients β are estimated to minimize the penalized deviance

  $$\frac{1}{n}\text{dev}(\boldsymbol{\beta}) + \lambda_t \sum_j |\beta_j|$$

  You fit the regression for a sequence of λ_t penalties and use model selection tools to choose the best.
- To build Lasso regression models, you need to create your own numeric model matrix for input. You can do this using the `sparse.model.matrix` function from the `Matrix` library to use efficient sparse matrix storage.

  ```
  x <- sparse.model.matrix( y ~ . , data=naref(data) )[,-1]
  ```

  The result will be a model matrix for regressing `y` on all the variables in `data`. We removed the intercept with `[,-1]`. You can use other formulas to add interactions or include specific variables.
- When fitting a Lasso regression, you typically want to include in your model matrix a separate binary indicator for each level of the factor variables in your `data`. In the above call to `sparse.model.matrix`, we applied the `naref` function from `gamlr` to set `NA` as the reference level for each factor so that all other levels are represented in the model matrix.
- When you have missing data, you can call `naref(data, impute=TRUE)` to impute missing values. For numeric variables `var` that include `NA`s, they will be replaced by `var.x` with no missing values and `var.miss` indicating which entries have been imputed.

- To run a path of Lasso regressions along a sequence of penalties, with input matrix `x` and response `y`, you call

  ```
  fit <- gamlr(x, y)
  ```

  and you can add `family="binomial"` for logistic regression.
- Add the argument `lmr=1e-4` (or another small number) to run the path to smaller λ than the default $\lambda_{100}/\lambda_1 = 0.01$. You should run the path to small enough penalties such that when you look at the output of `plot(fit)` the AICc selection is not at the left edge of the figure.
- You can call `coef` and `predict` on the `gamlr` object `fit` exactly as you would for a fitted `glm` object, although you will need to apply the drop function to the output if you want to transform it to a simple array of predictions (rather than a sparse matrix). The resulting predictions and coefficients will correspond to the AICc-selected segment of the Lasso path.
- To run a CV experiment to select the optimal Lasso penalization, run

  ```
  cvfit <- cv.gamlr(x, y)
  ```

  and again you can add `family="binomial"` for logistic regression. When calling `coef` and `predict` on `cvfit` you can specify either `select="min"` for CV-min selection (choose the λ_t corresponding to lowest mean OOS deviance) of `select="1se"` for CV-1se selection (choose the largest λ_t with mean OOS deviance no more than 1 standard error larger than the minimum).
- To run CV experiments in parallel, create a `parallel` cluster and pass this to `cv.gamlr` as the `cl` argument.

  ```
  cl=makceCluster(detectCores())
  ```