
Gareth James • Daniela Witten • Trevor Hastie
Robert Tibshirani • Jonathan Taylor

An Introduction to Statistical Learning

with Applications in Python

First Printing: July 5, 2023

To our parents:

Alison and Michael James

Chiara Nappi and Edward Witten

Valerie and Patrick Hastie

Vera and Sami Tibshirani

John and Brenda Taylor

and to our families:

Michael, Daniel, and Catherine

Tessa, Theo, Otto, and Ari

Samantha, Timothy, and Lynda

Charlie, Ryan, Julie, and Cheryl

Lee-Ann and Isobel

Preface

Statistical learning refers to a set of tools for *making sense of complex datasets*. In recent years, we have seen a staggering increase in the scale and scope of data collection across virtually all areas of science and industry. As a result, statistical learning has become a critical toolkit for anyone who wishes to understand data — and as more and more of today’s jobs involve data, this means that statistical learning is fast becoming a critical toolkit for *everyone*.

One of the first books on statistical learning — *The Elements of Statistical Learning* (ESL, by Hastie, Tibshirani, and Friedman) — was published in 2001, with a second edition in 2009. ESL has become a popular text not only in statistics but also in related fields. One of the reasons for ESL’s popularity is its relatively accessible style. But ESL is best-suited for individuals with advanced training in the mathematical sciences.

An Introduction to Statistical Learning, With Applications in R (ISLR) — first published in 2013, with a second edition in 2021 — arose from the clear need for a broader and less technical treatment of the key topics in statistical learning. In addition to a review of linear regression, ISLR covers many of today’s most important statistical and machine learning approaches, including resampling, sparse methods for classification and regression, generalized additive models, tree-based methods, support vector machines, deep learning, survival analysis, clustering, and multiple testing.

Since it was published in 2013, ISLR has become a mainstay of undergraduate and graduate classrooms worldwide, as well as an important reference book for data scientists. One of the keys to its success has been that, beginning with Chapter 2, each chapter contains an **R** lab illustrating how to implement the statistical learning methods seen in that chapter, providing the reader with valuable hands-on experience.

However, in recent years **Python** has become an increasingly popular language for data science, and there has been increasing demand for a **Python**-

based alternative to ISLR. Hence, this book, *An Introduction to Statistical Learning, With Applications in Python* (ISLP), covers the same materials as ISLR but with labs implemented in **Python** — a feat accomplished by the addition of a new co-author, Jonathan Taylor. Several of the labs make use of the **ISLP Python** package, which we have written to facilitate carrying out the statistical learning methods covered in each chapter in **Python**. These labs will be useful both for **Python** novices, as well as experienced users.

The intention behind ISLP (and ISLR) is to concentrate more on the applications of the methods and less on the mathematical details, so it is appropriate for advanced undergraduates or master's students in statistics or related quantitative fields, or for individuals in other disciplines who wish to use statistical learning tools to analyze their data. It can be used as a textbook for a course spanning two semesters.

We are grateful to these readers for providing valuable comments on the first edition of ISLR: Pallavi Basu, Alexandra Chouldechova, Patrick Danner, Will Fithian, Luella Fu, Sam Gross, Max Grazier G'Sell, Courtney Paulson, Xinghao Qiao, Elisa Sheng, Noah Simon, Kean Ming Tan, Xin Lu Tan. We thank these readers for helpful input on the second edition of ISLR: Alan Agresti, Iain Carmichael, Yiqun Chen, Erin Craig, Daisy Ding, Lucy Gao, Ismael Lemhadri, Bryan Martin, Anna Neufeld, Geoff Tims, Carsten Voelkmann, Steve Yadlowsky, and James Zou. We are immensely grateful to Balasubramanian "Naras" Narasimhan for his assistance on both ISLR and ISLP.

It has been an honor and a privilege for us to see the considerable impact that ISLR has had on the way in which statistical learning is practiced, both in and out of the academic setting. We hope that this new **Python** edition will continue to give today's and tomorrow's applied statisticians and data scientists the tools they need for success in a data-driven world.

It's tough to make predictions, especially about the future.

-Yogi Berra

Contents

Preface	vii
1 Introduction	1
2 Statistical Learning	15
2.1 What Is Statistical Learning?	15
2.1.1 Why Estimate f ?	17
2.1.2 How Do We Estimate f ?	20
2.1.3 The Trade-Off Between Prediction Accuracy and Model Interpretability	23
2.1.4 Supervised Versus Unsupervised Learning	25
2.1.5 Regression Versus Classification Problems	27
2.2 Assessing Model Accuracy	27
2.2.1 Measuring the Quality of Fit	28
2.2.2 The Bias-Variance Trade-Off	31
2.2.3 The Classification Setting	34
2.3 Lab: Introduction to Python	40
2.3.1 Getting Started	40
2.3.2 Basic Commands	40
2.3.3 Introduction to Numerical Python	42
2.3.4 Graphics	48
2.3.5 Sequences and Slice Notation	51
2.3.6 Indexing Data	51
2.3.7 Loading Data	55
2.3.8 For Loops	59
2.3.9 Additional Graphical and Numerical Summaries .	61
2.4 Exercises	63
3 Linear Regression	69
3.1 Simple Linear Regression	70
3.1.1 Estimating the Coefficients	71
3.1.2 Assessing the Accuracy of the Coefficient Estimates	72
3.1.3 Assessing the Accuracy of the Model	77
3.2 Multiple Linear Regression	80
3.2.1 Estimating the Regression Coefficients	81

3.2.2	Some Important Questions	83
3.3	Other Considerations in the Regression Model	91
3.3.1	Qualitative Predictors	91
3.3.2	Extensions of the Linear Model	94
3.3.3	Potential Problems	100
3.4	The Marketing Plan	109
3.5	Comparison of Linear Regression with K -Nearest Neighbors	111
3.6	Lab: Linear Regression	116
3.6.1	Importing packages	116
3.6.2	Simple Linear Regression	117
3.6.3	Multiple Linear Regression	122
3.6.4	Multivariate Goodness of Fit	123
3.6.5	Interaction Terms	124
3.6.6	Non-linear Transformations of the Predictors	125
3.6.7	Qualitative Predictors	126
3.7	Exercises	127
4	Classification	135
4.1	An Overview of Classification	135
4.2	Why Not Linear Regression?	136
4.3	Logistic Regression	138
4.3.1	The Logistic Model	139
4.3.2	Estimating the Regression Coefficients	140
4.3.3	Making Predictions	141
4.3.4	Multiple Logistic Regression	142
4.3.5	Multinomial Logistic Regression	144
4.4	Generative Models for Classification	146
4.4.1	Linear Discriminant Analysis for $p = 1$	147
4.4.2	Linear Discriminant Analysis for $p > 1$	150
4.4.3	Quadratic Discriminant Analysis	156
4.4.4	Naive Bayes	158
4.5	A Comparison of Classification Methods	161
4.5.1	An Analytical Comparison	161
4.5.2	An Empirical Comparison	164
4.6	Generalized Linear Models	167
4.6.1	Linear Regression on the Bikeshare Data	167
4.6.2	Poisson Regression on the Bikeshare Data	169
4.6.3	Generalized Linear Models in Greater Generality .	172
4.7	Lab: Logistic Regression, LDA, QDA, and KNN	173
4.7.1	The Stock Market Data	173
4.7.2	Logistic Regression	174
4.7.3	Linear Discriminant Analysis	179
4.7.4	Quadratic Discriminant Analysis	181
4.7.5	Naive Bayes	182
4.7.6	K-Nearest Neighbors	183
4.7.7	Linear and Poisson Regression on the Bikeshare Data	188
4.8	Exercises	193

5 Resampling Methods	201
5.1 Cross-Validation	202
5.1.1 The Validation Set Approach	202
5.1.2 Leave-One-Out Cross-Validation	204
5.1.3 k -Fold Cross-Validation	206
5.1.4 Bias-Variance Trade-Off for k -Fold Cross-Validation	208
5.1.5 Cross-Validation on Classification Problems	209
5.2 The Bootstrap	212
5.3 Lab: Cross-Validation and the Bootstrap	215
5.3.1 The Validation Set Approach	216
5.3.2 Cross-Validation	217
5.3.3 The Bootstrap	220
5.4 Exercises	224
6 Linear Model Selection and Regularization	229
6.1 Subset Selection	231
6.1.1 Best Subset Selection	231
6.1.2 Stepwise Selection	233
6.1.3 Choosing the Optimal Model	235
6.2 Shrinkage Methods	240
6.2.1 Ridge Regression	240
6.2.2 The Lasso	244
6.2.3 Selecting the Tuning Parameter	252
6.3 Dimension Reduction Methods	253
6.3.1 Principal Components Regression	254
6.3.2 Partial Least Squares	260
6.4 Considerations in High Dimensions	262
6.4.1 High-Dimensional Data	262
6.4.2 What Goes Wrong in High Dimensions?	263
6.4.3 Regression in High Dimensions	265
6.4.4 Interpreting Results in High Dimensions	266
6.5 Lab: Linear Models and Regularization Methods	267
6.5.1 Subset Selection Methods	268
6.5.2 Ridge Regression and the Lasso	273
6.5.3 PCR and PLS Regression	280
6.6 Exercises	283
7 Moving Beyond Linearity	289
7.1 Polynomial Regression	290
7.2 Step Functions	292
7.3 Basis Functions	293
7.4 Regression Splines	294
7.4.1 Piecewise Polynomials	294
7.4.2 Constraints and Splines	296
7.4.3 The Spline Basis Representation	296
7.4.4 Choosing the Number and Locations of the Knots	297
7.4.5 Comparison to Polynomial Regression	299

7.5	Smoothing Splines	300
7.5.1	An Overview of Smoothing Splines	300
7.5.2	Choosing the Smoothing Parameter λ	301
7.6	Local Regression	303
7.7	Generalized Additive Models	305
7.7.1	GAMs for Regression Problems	306
7.7.2	GAMs for Classification Problems	308
7.8	Lab: Non-Linear Modeling	309
7.8.1	Polynomial Regression and Step Functions	310
7.8.2	Splines	315
7.8.3	Smoothing Splines and GAMs	317
7.8.4	Local Regression	324
7.9	Exercises	325
8	Tree-Based Methods	331
8.1	The Basics of Decision Trees	331
8.1.1	Regression Trees	331
8.1.2	Classification Trees	337
8.1.3	Trees Versus Linear Models	341
8.1.4	Advantages and Disadvantages of Trees	341
8.2	Bagging, Random Forests, Boosting, and Bayesian Additive Regression Trees	343
8.2.1	Bagging	343
8.2.2	Random Forests	346
8.2.3	Boosting	347
8.2.4	Bayesian Additive Regression Trees	350
8.2.5	Summary of Tree Ensemble Methods	353
8.3	Lab: Tree-Based Methods	354
8.3.1	Fitting Classification Trees	355
8.3.2	Fitting Regression Trees	358
8.3.3	Bagging and Random Forests	360
8.3.4	Boosting	361
8.3.5	Bayesian Additive Regression Trees	362
8.4	Exercises	363
9	Support Vector Machines	367
9.1	Maximal Margin Classifier	367
9.1.1	What Is a Hyperplane?	368
9.1.2	Classification Using a Separating Hyperplane	368
9.1.3	The Maximal Margin Classifier	370
9.1.4	Construction of the Maximal Margin Classifier	372
9.1.5	The Non-separable Case	372
9.2	Support Vector Classifiers	373
9.2.1	Overview of the Support Vector Classifier	373
9.2.2	Details of the Support Vector Classifier	374
9.3	Support Vector Machines	377
9.3.1	Classification with Non-Linear Decision Boundaries	378
9.3.2	The Support Vector Machine	379

9.3.3	An Application to the Heart Disease Data	382
9.4	SVMs with More than Two Classes	383
9.4.1	One-Versus-One Classification	384
9.4.2	One-Versus-All Classification	384
9.5	Relationship to Logistic Regression	384
9.6	Lab: Support Vector Machines	387
9.6.1	Support Vector Classifier	387
9.6.2	Support Vector Machine	390
9.6.3	ROC Curves	392
9.6.4	SVM with Multiple Classes	393
9.6.5	Application to Gene Expression Data	394
9.7	Exercises	395
10	Deep Learning	399
10.1	Single Layer Neural Networks	400
10.2	Multilayer Neural Networks	402
10.3	Convolutional Neural Networks	406
10.3.1	Convolution Layers	407
10.3.2	Pooling Layers	410
10.3.3	Architecture of a Convolutional Neural Network .	410
10.3.4	Data Augmentation	411
10.3.5	Results Using a Pretrained Classifier	412
10.4	Document Classification	413
10.5	Recurrent Neural Networks	416
10.5.1	Sequential Models for Document Classification .	418
10.5.2	Time Series Forecasting	420
10.5.3	Summary of RNNs	424
10.6	When to Use Deep Learning	425
10.7	Fitting a Neural Network	427
10.7.1	Backpropagation	428
10.7.2	Regularization and Stochastic Gradient Descent .	429
10.7.3	Dropout Learning	431
10.7.4	Network Tuning	431
10.8	Interpolation and Double Descent	432
10.9	Lab: Deep Learning	435
10.9.1	Single Layer Network on Hitters Data	437
10.9.2	Multilayer Network on the MNIST Digit Data .	444
10.9.3	Convolutional Neural Networks	448
10.9.4	Using Pretrained CNN Models	452
10.9.5	IMDB Document Classification	454
10.9.6	Recurrent Neural Networks	458
10.10	Exercises	465
11	Survival Analysis and Censored Data	469
11.1	Survival and Censoring Times	470
11.2	A Closer Look at Censoring	470
11.3	The Kaplan–Meier Survival Curve	472
11.4	The Log-Rank Test	474
11.5	Regression Models With a Survival Response	476



The linear regression model discussed in Chapter 3 assumes that the response variable Y is quantitative. But in many situations, the response variable is instead *qualitative*. For example, eye color is qualitative. Often qualitative variables are referred to as *categorical*; we will use these terms interchangeably. In this chapter, we study approaches for predicting qualitative responses, a process that is known as *classification*. Predicting a qualitative response for an observation can be referred to as *classifying* that observation, since it involves assigning the observation to a category, or class. On the other hand, often the methods used for classification first predict the probability that the observation belongs to each of the categories of a qualitative variable, as the basis for making the classification. In this sense they also behave like regression methods.

There are many possible classification techniques, or *classifiers*, that one might use to predict a qualitative response. We touched on some of these in Sections 2.1.5 and 2.2.3. In this chapter we discuss some widely-used classifiers: *logistic regression*, *linear discriminant analysis*, *quadratic discriminant analysis*, *naive Bayes*, and *K-nearest neighbors*. The discussion of logistic regression is used as a jumping-off point for a discussion of *generalized linear models*, and in particular, *Poisson regression*. We discuss more computer-intensive classification methods in later chapters: these include generalized additive models (Chapter 7); trees, random forests, and boosting (Chapter 8); and support vector machines (Chapter 9).

qualitative
classification
classifier
logistic regression
linear discriminant analysis
quadratic discriminant analysis
naive Bayes
K-nearest neighbors
generalized linear models
Poisson regression

4.1 An Overview of Classification

Classification problems occur often, perhaps even more so than regression problems. Some examples include:

1. A person arrives at the emergency room with a set of symptoms that could possibly be attributed to one of three medical conditions. Which of the three conditions does the individual have?
2. An online banking service must be able to determine whether or not a transaction being performed on the site is fraudulent, on the basis of the user's IP address, past transaction history, and so forth.
3. On the basis of DNA sequence data for a number of patients with and without a given disease, a biologist would like to figure out which DNA mutations are deleterious (disease-causing) and which are not.

Just as in the regression setting, in the classification setting we have a set of training observations $(x_1, y_1), \dots, (x_n, y_n)$ that we can use to build a classifier. We want our classifier to perform well not only on the training data, but also on test observations that were not used to train the classifier.

In this chapter, we will illustrate the concept of classification using the simulated `Default` data set. We are interested in predicting whether an individual will default on his or her credit card payment, on the basis of annual income and monthly credit card balance. The data set is displayed in Figure 4.1. In the left-hand panel of Figure 4.1, we have plotted annual `income` and monthly credit card `balance` for a subset of 10,000 individuals. The individuals who defaulted in a given month are shown in orange, and those who did not in blue. (The overall default rate is about 3%, so we have plotted only a fraction of the individuals who did not default.) It appears that individuals who defaulted tended to have higher credit card balances than those who did not. In the center and right-hand panels of Figure 4.1, two pairs of boxplots are shown. The first shows the distribution of `balance` split by the binary `default` variable; the second is a similar plot for `income`. In this chapter, we learn how to build a model to predict `default` (Y) for any given value of `balance` (X_1) and `income` (X_2). Since Y is not quantitative, the simple linear regression model of Chapter 3 is not a good choice: we will elaborate on this further in Section 4.2.

It is worth noting that Figure 4.1 displays a very pronounced relationship between the predictor `balance` and the response `default`. In most real applications, the relationship between the predictor and the response will not be nearly so strong. However, for the sake of illustrating the classification procedures discussed in this chapter, we use an example in which the relationship between the predictor and the response is somewhat exaggerated.

4.2 Why Not Linear Regression?

We have stated that linear regression is not appropriate in the case of a qualitative response. Why not?

Suppose that we are trying to predict the medical condition of a patient in the emergency room on the basis of her symptoms. In this simplified example, there are three possible diagnoses: `stroke`, `drug overdose`, and

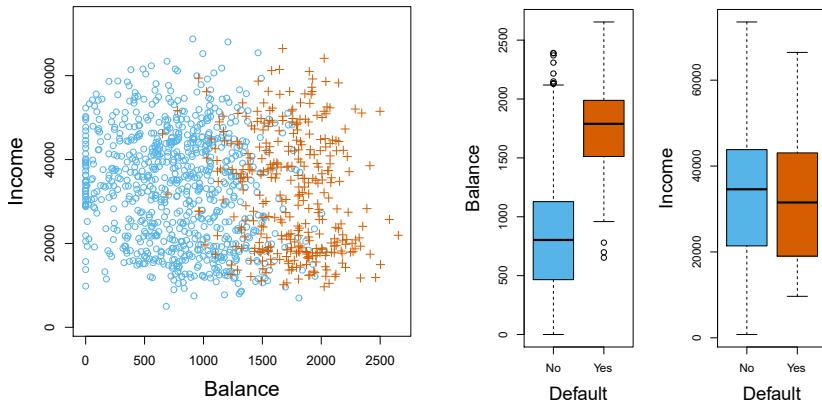


FIGURE 4.1. The `Default` data set. Left: The annual incomes and monthly credit card balances of a number of individuals. The individuals who defaulted on their credit card payments are shown in orange, and those who did not are shown in blue. Center: Boxplots of `balance` as a function of `default` status. Right: Boxplots of `income` as a function of `default` status.

epileptic seizure. We could consider encoding these values as a quantitative response variable, Y , as follows:

$$Y = \begin{cases} 1 & \text{if } \text{stroke}; \\ 2 & \text{if } \text{drug overdose}; \\ 3 & \text{if } \text{epileptic seizure}. \end{cases}$$

Using this coding, least squares could be used to fit a linear regression model to predict Y on the basis of a set of predictors X_1, \dots, X_p . Unfortunately, this coding implies an ordering on the outcomes, putting `drug overdose` in between `stroke` and `epileptic seizure`, and insisting that the difference between `stroke` and `drug overdose` is the same as the difference between `drug overdose` and `epileptic seizure`. In practice there is no particular reason that this needs to be the case. For instance, one could choose an equally reasonable coding,

$$Y = \begin{cases} 1 & \text{if } \text{epileptic seizure}; \\ 2 & \text{if } \text{stroke}; \\ 3 & \text{if } \text{drug overdose}, \end{cases}$$

which would imply a totally different relationship among the three conditions. Each of these codings would produce fundamentally different linear models that would ultimately lead to different sets of predictions on test observations.

If the response variable's values did take on a natural ordering, such as *mild*, *moderate*, and *severe*, and we felt the gap between *mild* and *moderate* was similar to the gap between *moderate* and *severe*, then a 1, 2, 3 coding would be reasonable. Unfortunately, in general there is no natural way to

convert a qualitative response variable with more than two levels into a quantitative response that is ready for linear regression.

For a *binary* (two level) qualitative response, the situation is better. For instance, perhaps there are only two possibilities for the patient's medical condition: `stroke` and `drug overdose`. We could then potentially use the *dummy variable* approach from Section 3.3.1 to code the response as follows:

$$Y = \begin{cases} 0 & \text{if } \text{stroke}; \\ 1 & \text{if } \text{drug overdose}. \end{cases}$$

We could then fit a linear regression to this binary response, and predict `drug overdose` if $\hat{Y} > 0.5$ and `stroke` otherwise. In the binary case it is not hard to show that even if we flip the above coding, linear regression will produce the same final predictions.

For a binary response with a 0/1 coding as above, regression by least squares is not completely unreasonable: it can be shown that the $X\hat{\beta}$ obtained using linear regression is in fact an estimate of $\Pr(\text{drug overdose}|X)$ in this special case. However, if we use linear regression, some of our estimates might be outside the [0, 1] interval (see Figure 4.2), making them hard to interpret as probabilities! Nevertheless, the predictions provide an ordering and can be interpreted as crude probability estimates. Curiously, it turns out that the classifications that we get if we use linear regression to predict a binary response will be the same as for the linear discriminant analysis (LDA) procedure we discuss in Section 4.4.

To summarize, there are at least two reasons not to perform classification using a regression method: (a) a regression method cannot accommodate a qualitative response with more than two classes; (b) a regression method will not provide meaningful estimates of $\Pr(Y|X)$, even with just two classes. Thus, it is preferable to use a classification method that is truly suited for qualitative response values. In the next section, we present logistic regression, which is well-suited for the case of a binary qualitative response; in later sections we will cover classification methods that are appropriate when the qualitative response has two or more classes.

4.3 Logistic Regression

Consider again the `Default` data set, where the response `default` falls into one of two categories, `Yes` or `No`. Rather than modeling this response Y directly, logistic regression models the *probability* that Y belongs to a particular category.

For the `Default` data, logistic regression models the probability of default. For example, the probability of default given `balance` can be written as

$$\Pr(\text{default} = \text{Yes}|\text{balance}).$$

The values of $\Pr(\text{default} = \text{Yes}|\text{balance})$, which we abbreviate $p(\text{balance})$, will range between 0 and 1. Then for any given value of `balance`, a prediction can be made for `default`. For example, one might predict `default = Yes`

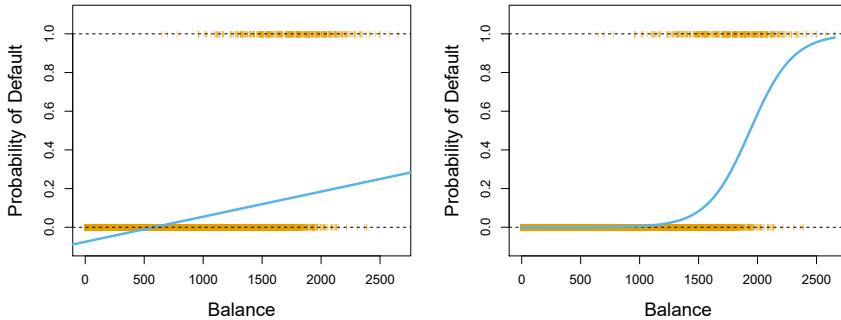


FIGURE 4.2. Classification using the `Default` data. Left: Estimated probability of `default` using linear regression. Some estimated probabilities are negative! The orange ticks indicate the 0/1 values coded for `default`(No or Yes). Right: Predicted probabilities of `default` using logistic regression. All probabilities lie between 0 and 1.

for any individual for whom $p(\text{balance}) > 0.5$. Alternatively, if a company wishes to be conservative in predicting individuals who are at risk for default, then they may choose to use a lower threshold, such as $p(\text{balance}) > 0.1$.

4.3.1 The Logistic Model

How should we model the relationship between $p(X) = \Pr(Y = 1|X)$ and X ? (For convenience we are using the generic 0/1 coding for the response.) In Section 4.2 we considered using a linear regression model to represent these probabilities:

$$p(X) = \beta_0 + \beta_1 X. \quad (4.1)$$

If we use this approach to predict `default=Yes` using `balance`, then we obtain the model shown in the left-hand panel of Figure 4.2. Here we see the problem with this approach: for balances close to zero we predict a negative probability of default; if we were to predict for very large balances, we would get values bigger than 1. These predictions are not sensible, since of course the true probability of default, regardless of credit card balance, must fall between 0 and 1. This problem is not unique to the credit default data. Any time a straight line is fit to a binary response that is coded as 0 or 1, in principle we can always predict $p(X) < 0$ for some values of X and $p(X) > 1$ for others (unless the range of X is limited).

To avoid this problem, we must model $p(X)$ using a function that gives outputs between 0 and 1 for all values of X . Many functions meet this description. In logistic regression, we use the *logistic function*,

$$p(X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}}. \quad (4.2)$$

To fit the model (4.2), we use a method called *maximum likelihood*, which we discuss in the next section. The right-hand panel of Figure 4.2 illustrates the fit of the logistic regression model to the `Default` data. Notice that for

logistic
function

maximum
likelihood

low balances we now predict the probability of default as close to, but never below, zero. Likewise, for high balances we predict a default probability close to, but never above, one. The logistic function will always produce an *S-shaped* curve of this form, and so regardless of the value of X , we will obtain a sensible prediction. We also see that the logistic model is better able to capture the range of probabilities than is the linear regression model in the left-hand plot. The average fitted probability in both cases is 0.0333 (averaged over the training data), which is the same as the overall proportion of defaulters in the data set.

After a bit of manipulation of (4.2), we find that

$$\frac{p(X)}{1 - p(X)} = e^{\beta_0 + \beta_1 X}. \quad (4.3)$$

The quantity $p(X)/[1 - p(X)]$ is called the *odds*, and can take on any value between 0 and ∞ . Values of the odds close to 0 and ∞ indicate very low and very high probabilities of default, respectively. For example, on average 1 in 5 people with an odds of 1/4 will default, since $p(X) = 0.2$ implies an odds of $\frac{0.2}{1-0.2} = 1/4$. Likewise, on average nine out of every ten people with an odds of 9 will default, since $p(X) = 0.9$ implies an odds of $\frac{0.9}{1-0.9} = 9$. Odds are traditionally used instead of probabilities in horse-racing, since they relate more naturally to the correct betting strategy.

odds

By taking the logarithm of both sides of (4.3), we arrive at

$$\log\left(\frac{p(X)}{1 - p(X)}\right) = \beta_0 + \beta_1 X. \quad (4.4)$$

The left-hand side is called the *log odds* or *logit*. We see that the logistic regression model (4.2) has a logit that is linear in X .

log odds
logit

Recall from Chapter 3 that in a linear regression model, β_1 gives the average change in Y associated with a one-unit increase in X . By contrast, in a logistic regression model, increasing X by one unit changes the log odds by β_1 (4.4). Equivalently, it multiplies the odds by e^{β_1} (4.3). However, because the relationship between $p(X)$ and X in (4.2) is not a straight line, β_1 does *not* correspond to the change in $p(X)$ associated with a one-unit increase in X . The amount that $p(X)$ changes due to a one-unit change in X depends on the current value of X . But regardless of the value of X , if β_1 is positive then increasing X will be associated with increasing $p(X)$, and if β_1 is negative then increasing X will be associated with decreasing $p(X)$. The fact that there is not a straight-line relationship between $p(X)$ and X , and the fact that the rate of change in $p(X)$ per unit change in X depends on the current value of X , can also be seen by inspection of the right-hand panel of Figure 4.2.

4.3.2 Estimating the Regression Coefficients

The coefficients β_0 and β_1 in (4.2) are unknown, and must be estimated based on the available training data. In Chapter 3, we used the least squares approach to estimate the unknown linear regression coefficients. Although we could use (non-linear) least squares to fit the model (4.4), the more general method of *maximum likelihood* is preferred, since it has better statistical properties. The basic intuition behind using maximum likelihood

to fit a logistic regression model is as follows: we seek estimates for β_0 and β_1 such that the predicted probability $\hat{p}(x_i)$ of default for each individual, using (4.2), corresponds as closely as possible to the individual's observed default status. In other words, we try to find $\hat{\beta}_0$ and $\hat{\beta}_1$ such that plugging these estimates into the model for $p(X)$, given in (4.2), yields a number close to one for all individuals who defaulted, and a number close to zero for all individuals who did not. This intuition can be formalized using a mathematical equation called a *likelihood function*:

$$\ell(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} (1 - p(x_{i'})). \quad (4.5)$$

likelihood
function

The estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ are chosen to *maximize* this likelihood function.

Maximum likelihood is a very general approach that is used to fit many of the non-linear models that we examine throughout this book. In the linear regression setting, the least squares approach is in fact a special case of maximum likelihood. The mathematical details of maximum likelihood are beyond the scope of this book. However, in general, logistic regression and other models can be easily fit using statistical software such as R, and so we do not need to concern ourselves with the details of the maximum likelihood fitting procedure.

Table 4.1 shows the coefficient estimates and related information that result from fitting a logistic regression model on the **Default** data in order to predict the probability of **default=Yes** using **balance**. We see that $\hat{\beta}_1 = 0.0055$; this indicates that an increase in **balance** is associated with an increase in the probability of **default**. To be precise, a one-unit increase in **balance** is associated with an increase in the log odds of **default** by 0.0055 units.

Many aspects of the logistic regression output shown in Table 4.1 are similar to the linear regression output of Chapter 3. For example, we can measure the accuracy of the coefficient estimates by computing their standard errors. The z -statistic in Table 4.1 plays the same role as the t -statistic in the linear regression output, for example in Table 3.1 on page 77. For instance, the z -statistic associated with β_1 is equal to $\hat{\beta}_1/\text{SE}(\hat{\beta}_1)$, and so a large (absolute) value of the z -statistic indicates evidence against the null hypothesis $H_0 : \beta_1 = 0$. This null hypothesis implies that $p(X) = \frac{e^{\beta_0}}{1+e^{\beta_0}}$: in other words, that the probability of **default** does not depend on **balance**. Since the p -value associated with **balance** in Table 4.1 is tiny, we can reject H_0 . In other words, we conclude that there is indeed an association between **balance** and probability of **default**. The estimated intercept in Table 4.1 is typically not of interest; its main purpose is to adjust the average fitted probabilities to the proportion of ones in the data (in this case, the overall default rate).

4.3.3 Making Predictions

Once the coefficients have been estimated, we can compute the probability of **default** for any given credit card balance. For example, using the coefficient estimates given in Table 4.1, we predict that the default probability

	Coefficient	Std. error	<i>z</i> -statistic	<i>p</i> -value
Intercept	-10.6513	0.3612	-29.5	<0.0001
balance	0.0055	0.0002	24.9	<0.0001

TABLE 4.1. For the `Default` data, estimated coefficients of the logistic regression model that predicts the probability of `default` using `balance`. A one-unit increase in `balance` is associated with an increase in the log odds of `default` by 0.0055 units.

	Coefficient	Std. error	<i>z</i> -statistic	<i>p</i> -value
Intercept	-3.5041	0.0707	-49.55	<0.0001
student[Yes]	0.4049	0.1150	3.52	0.0004

TABLE 4.2. For the `Default` data, estimated coefficients of the logistic regression model that predicts the probability of `default` using student status. Student status is encoded as a dummy variable, with a value of 1 for a student and a value of 0 for a non-student, and represented by the variable `student[Yes]` in the table.

for an individual with a `balance` of \$1,000 is

$$\hat{p}(X) = \frac{e^{\hat{\beta}_0 + \hat{\beta}_1 X}}{1 + e^{\hat{\beta}_0 + \hat{\beta}_1 X}} = \frac{e^{-10.6513 + 0.0055 \times 1,000}}{1 + e^{-10.6513 + 0.0055 \times 1,000}} = 0.00576,$$

which is below 1%. In contrast, the predicted probability of default for an individual with a balance of \$2,000 is much higher, and equals 0.586 or 58.6%.

One can use qualitative predictors with the logistic regression model using the dummy variable approach from Section 3.3.1. As an example, the `Default` data set contains the qualitative variable `student`. To fit a model that uses student status as a predictor variable, we simply create a dummy variable that takes on a value of 1 for students and 0 for non-students. The logistic regression model that results from predicting probability of default from student status can be seen in Table 4.2. The coefficient associated with the dummy variable is positive, and the associated *p*-value is statistically significant. This indicates that students tend to have higher default probabilities than non-students:

$$\widehat{\Pr}(\text{default=Yes} | \text{student=Yes}) = \frac{e^{-3.5041 + 0.4049 \times 1}}{1 + e^{-3.5041 + 0.4049 \times 1}} = 0.0431,$$

$$\widehat{\Pr}(\text{default=Yes} | \text{student=No}) = \frac{e^{-3.5041 + 0.4049 \times 0}}{1 + e^{-3.5041 + 0.4049 \times 0}} = 0.0292.$$

4.3.4 Multiple Logistic Regression

We now consider the problem of predicting a binary response using multiple predictors. By analogy with the extension from simple to multiple linear regression in Chapter 3, we can generalize (4.4) as follows:

$$\log \left(\frac{p(X)}{1 - p(X)} \right) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p, \quad (4.6)$$

where $X = (X_1, \dots, X_p)$ are p predictors. Equation 4.6 can be rewritten as

$$p(X) = \frac{e^{\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p}}{1 + e^{\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p}}. \quad (4.7)$$

	Coefficient	Std. error	<i>z</i> -statistic	<i>p</i> -value
Intercept	−10.8690	0.4923	−22.08	<0.0001
balance	0.0057	0.0002	24.74	<0.0001
income	0.0030	0.0082	0.37	0.7115
student [Yes]	−0.6468	0.2362	−2.74	0.0062

TABLE 4.3. For the `Default` data, estimated coefficients of the logistic regression model that predicts the probability of `default` using `balance`, `income`, and student status. Student status is encoded as a dummy variable `student[Yes]`, with a value of 1 for a student and a value of 0 for a non-student. In fitting this model, `income` was measured in thousands of dollars.

Just as in Section 4.3.2, we use the maximum likelihood method to estimate $\beta_0, \beta_1, \dots, \beta_p$.

Table 4.3 shows the coefficient estimates for a logistic regression model that uses `balance`, `income` (in thousands of dollars), and `student` status to predict probability of `default`. There is a surprising result here. The *p*-values associated with `balance` and the dummy variable for `student` status are very small, indicating that each of these variables is associated with the probability of `default`. However, the coefficient for the dummy variable is negative, indicating that students are less likely to default than non-students. In contrast, the coefficient for the dummy variable is positive in Table 4.2. How is it possible for student status to be associated with an *increase* in probability of default in Table 4.2 and a *decrease* in probability of default in Table 4.3? The left-hand panel of Figure 4.3 provides a graphical illustration of this apparent paradox. The orange and blue solid lines show the average default rates for students and non-students, respectively, as a function of credit card balance. The negative coefficient for `student` in the multiple logistic regression indicates that *for a fixed value of balance and income*, a student is less likely to default than a non-student. Indeed, we observe from the left-hand panel of Figure 4.3 that the student default rate is at or below that of the non-student default rate for every value of `balance`. But the horizontal broken lines near the base of the plot, which show the default rates for students and non-students averaged over all values of `balance` and `income`, suggest the opposite effect: the overall student default rate is higher than the non-student default rate. Consequently, there is a positive coefficient for `student` in the single variable logistic regression output shown in Table 4.2.

The right-hand panel of Figure 4.3 provides an explanation for this discrepancy. The variables `student` and `balance` are correlated. Students tend to hold higher levels of debt, which is in turn associated with higher probability of default. In other words, students are more likely to have large credit card balances, which, as we know from the left-hand panel of Figure 4.3, tend to be associated with high default rates. Thus, even though an individual student with a given credit card balance will tend to have a lower probability of default than a non-student with the same credit card balance, the fact that students on the whole tend to have higher credit card balances means that overall, students tend to default at a higher rate than non-students. This is an important distinction for a credit card company that is trying to determine to whom they should offer credit. A student is riskier than a non-student if no information about the student's credit card

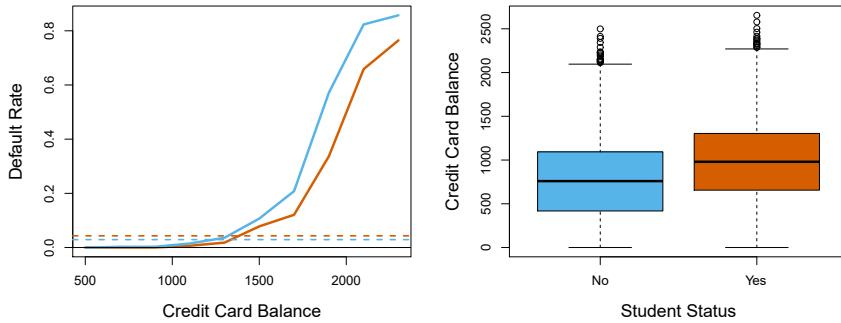


FIGURE 4.3. Confounding in the `Default` data. Left: Default rates are shown for students (orange) and non-students (blue). The solid lines display default rate as a function of `balance`, while the horizontal broken lines display the overall default rates. Right: Boxplots of `balance` for students (orange) and non-students (blue) are shown.

balance is available. However, that student is less risky than a non-student with the same credit card balance!

This simple example illustrates the dangers and subtleties associated with performing regressions involving only a single predictor when other predictors may also be relevant. As in the linear regression setting, the results obtained using one predictor may be quite different from those obtained using multiple predictors, especially when there is correlation among the predictors. In general, the phenomenon seen in Figure 4.3 is known as *confounding*.

By substituting estimates for the regression coefficients from Table 4.3 into (4.7), we can make predictions. For example, a student with a credit card balance of \$1,500 and an income of \$40,000 has an estimated probability of default of

$$\hat{p}(X) = \frac{e^{-10.869 + 0.00574 \times 1,500 + 0.003 \times 40 - 0.6468 \times 1}}{1 + e^{-10.869 + 0.00574 \times 1,500 + 0.003 \times 40 - 0.6468 \times 1}} = 0.058. \quad (4.8)$$

A non-student with the same balance and income has an estimated probability of default of

$$\hat{p}(X) = \frac{e^{-10.869 + 0.00574 \times 1,500 + 0.003 \times 40 - 0.6468 \times 0}}{1 + e^{-10.869 + 0.00574 \times 1,500 + 0.003 \times 40 - 0.6468 \times 0}} = 0.105. \quad (4.9)$$

(Here we multiply the `income` coefficient estimate from Table 4.3 by 40, rather than by 40,000, because in that table the model was fit with `income` measured in units of \$1,000.)

4.3.5 Multinomial Logistic Regression

We sometimes wish to classify a response variable that has more than two classes. For example, in Section 4.2 we had three categories of medical condition in the emergency room: `stroke`, `drug overdose`, `epileptic seizure`. However, the logistic regression approach that we have seen in this section only allows for $K = 2$ classes for the response variable.

confounding

It turns out that it is possible to extend the two-class logistic regression approach to the setting of $K > 2$ classes. This extension is sometimes known as *multinomial logistic regression*. To do this, we first select a single class to serve as the *baseline*; without loss of generality, we select the K th class for this role. Then we replace the model (4.7) with the model

$$\Pr(Y = k|X = x) = \frac{e^{\beta_{k0} + \beta_{k1}x_1 + \dots + \beta_{kp}x_p}}{1 + \sum_{l=1}^{K-1} e^{\beta_{l0} + \beta_{l1}x_1 + \dots + \beta_{lp}x_p}} \quad (4.10)$$

for $k = 1, \dots, K-1$, and

$$\Pr(Y = K|X = x) = \frac{1}{1 + \sum_{l=1}^{K-1} e^{\beta_{l0} + \beta_{l1}x_1 + \dots + \beta_{lp}x_p}}. \quad (4.11)$$

It is not hard to show that for $k = 1, \dots, K-1$,

$$\log \left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = K|X = x)} \right) = \beta_{k0} + \beta_{k1}x_1 + \dots + \beta_{kp}x_p. \quad (4.12)$$

Notice that (4.12) is quite similar to (4.6). Equation 4.12 indicates that once again, the log odds between any pair of classes is linear in the features.

It turns out that in (4.10)–(4.12), the decision to treat the K th class as the baseline is unimportant. For example, when classifying emergency room visits into **stroke**, **drug overdose**, and **epileptic seizure**, suppose that we fit two multinomial logistic regression models: one treating **stroke** as the baseline, another treating **drug overdose** as the baseline. The coefficient estimates will differ between the two fitted models due to the differing choice of baseline, but the fitted values (predictions), the log odds between any pair of classes, and the other key model outputs will remain the same.

Nonetheless, interpretation of the coefficients in a multinomial logistic regression model must be done with care, since it is tied to the choice of baseline. For example, if we set **epileptic seizure** to be the baseline, then we can interpret $\beta_{\text{stroke}0}$ as the log odds of **stroke** versus **epileptic seizure**, given that $x_1 = \dots = x_p = 0$. Furthermore, a one-unit increase in X_j is associated with a $\beta_{\text{stroke}j}$ increase in the log odds of **stroke** over **epileptic seizure**. Stated another way, if X_j increases by one unit, then

$$\frac{\Pr(Y = \text{stroke}|X = x)}{\Pr(Y = \text{epileptic seizure}|X = x)}$$

increases by $e^{\beta_{\text{stroke}j}}$.

We now briefly present an alternative coding for multinomial logistic regression, known as the *softmax* coding. The softmax coding is equivalent to the coding just described in the sense that the fitted values, log odds between any pair of classes, and other key model outputs will remain the same, regardless of coding. But the softmax coding is used extensively in some areas of the machine learning literature (and will appear again in Chapter 10), so it is worth being aware of it. In the softmax coding, rather than selecting a baseline class, we treat all K classes symmetrically, and assume that for $k = 1, \dots, K$,

$$\Pr(Y = k|X = x) = \frac{e^{\beta_{k0} + \beta_{k1}x_1 + \dots + \beta_{kp}x_p}}{\sum_{l=1}^K e^{\beta_{l0} + \beta_{l1}x_1 + \dots + \beta_{lp}x_p}}. \quad (4.13)$$

multinomial logistic regression

softmax

Thus, rather than estimating coefficients for $K - 1$ classes, we actually estimate coefficients for all K classes. It is not hard to see that as a result of (4.13), the log odds ratio between the k th and k' th classes equals

$$\log \left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = k'|X = x)} \right) = (\beta_{k0} - \beta_{k'0}) + (\beta_{k1} - \beta_{k'1})x_1 + \cdots + (\beta_{kp} - \beta_{k'p})x_p. \quad (4.14)$$

4.4 Generative Models for Classification

Logistic regression involves directly modeling $\Pr(Y = k|X = x)$ using the logistic function, given by (4.7) for the case of two response classes. In statistical jargon, we model the conditional distribution of the response Y , given the predictor(s) X . We now consider an alternative and less direct approach to estimating these probabilities. In this new approach, we model the distribution of the predictors X separately in each of the response classes (i.e. for each value of Y). We then use Bayes' theorem to flip these around into estimates for $\Pr(Y = k|X = x)$. When the distribution of X within each class is assumed to be normal, it turns out that the model is very similar in form to logistic regression.

Why do we need another method, when we have logistic regression? There are several reasons:

- When there is substantial separation between the two classes, the parameter estimates for the logistic regression model are surprisingly unstable. The methods that we consider in this section do not suffer from this problem.
- If the distribution of the predictors X is approximately normal in each of the classes and the sample size is small, then the approaches in this section may be more accurate than logistic regression.
- The methods in this section can be naturally extended to the case of more than two response classes. (In the case of more than two response classes, we can also use multinomial logistic regression from Section 4.3.5.)

Suppose that we wish to classify an observation into one of K classes, where $K \geq 2$. In other words, the qualitative response variable Y can take on K possible distinct and unordered values. Let π_k represent the overall or *prior* probability that a randomly chosen observation comes from the k th class. Let $f_k(X) \equiv \Pr(X|Y = k)$ ¹ denote the *density function* of X for an observation that comes from the k th class. In other words, $f_k(x)$ is relatively large if there is a high probability that an observation in the k th class has $X \approx x$, and $f_k(x)$ is small if it is very unlikely that an observation in the k th class has $X \approx x$. Then *Bayes' theorem* states that

prior
density
function

Bayes'
theorem

¹Technically, this definition is only correct if X is a qualitative random variable. If X is quantitative, then $f_k(x)dx$ corresponds to the probability of X falling in a small region dx around x .

$$\Pr(Y = k|X = x) = \frac{\pi_k f_k(x)}{\sum_{l=1}^K \pi_l f_l(x)}. \quad (4.15)$$

In accordance with our earlier notation, we will use the abbreviation $p_k(x) = \Pr(Y = k|X = x)$; this is the *posterior* probability that an observation $X = x$ belongs to the k th class. That is, it is the probability that the observation belongs to the k th class, *given* the predictor value for that observation.

posterior

Equation 4.15 suggests that instead of directly computing the posterior probability $p_k(x)$ as in Section 4.3.1, we can simply plug in estimates of π_k and $f_k(x)$ into (4.15). In general, estimating π_k is easy if we have a random sample from the population: we simply compute the fraction of the training observations that belong to the k th class. However, estimating the density function $f_k(x)$ is much more challenging. As we will see, to estimate $f_k(x)$, we will typically have to make some simplifying assumptions.

We know from Chapter 2 that the Bayes classifier, which classifies an observation x to the class for which $p_k(x)$ is largest, has the lowest possible error rate out of all classifiers. (Of course, this is only true if all of the terms in (4.15) are correctly specified.) Therefore, if we can find a way to estimate $f_k(x)$, then we can plug it into (4.15) in order to approximate the Bayes classifier.

In the following sections, we discuss three classifiers that use different estimates of $f_k(x)$ in (4.15) to approximate the Bayes classifier: *linear discriminant analysis*, *quadratic discriminant analysis*, and *naive Bayes*.

4.4.1 Linear Discriminant Analysis for $p = 1$

For now, assume that $p = 1$ —that is, we have only one predictor. We would like to obtain an estimate for $f_k(x)$ that we can plug into (4.15) in order to estimate $p_k(x)$. We will then classify an observation to the class for which $p_k(x)$ is greatest. To estimate $f_k(x)$, we will first make some assumptions about its form.

In particular, we assume that $f_k(x)$ is *normal* or *Gaussian*. In the one-dimensional setting, the normal density takes the form

$$f_k(x) = \frac{1}{\sqrt{2\pi}\sigma_k} \exp\left(-\frac{1}{2\sigma_k^2}(x - \mu_k)^2\right), \quad (4.16)$$

normal
Gaussian

where μ_k and σ_k^2 are the mean and variance parameters for the k th class. For now, let us further assume that $\sigma_1^2 = \dots = \sigma_K^2$: that is, there is a shared variance term across all K classes, which for simplicity we can denote by σ^2 . Plugging (4.16) into (4.15), we find that

$$p_k(x) = \frac{\pi_k \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(x - \mu_k)^2\right)}{\sum_{l=1}^K \pi_l \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(x - \mu_l)^2\right)}. \quad (4.17)$$

(Note that in (4.17), π_k denotes the prior probability that an observation belongs to the k th class, not to be confused with $\pi \approx 3.14159$, the mathematical constant.) The Bayes classifier² involves assigning an observation

²Recall that the *Bayes classifier* assigns an observation to the class for which $p_k(x)$ is largest. This is different from *Bayes' theorem* in (4.15), which allows us to manipulate conditional distributions.

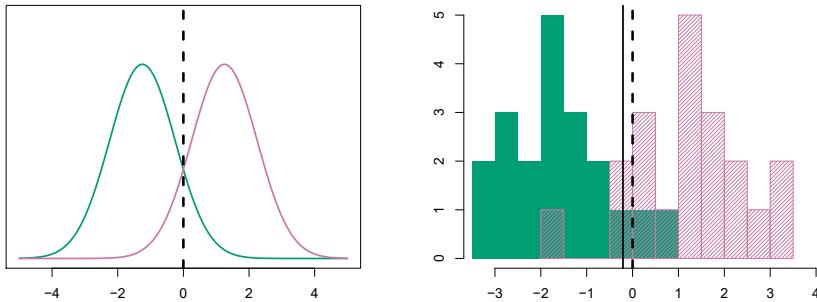


FIGURE 4.4. Left: Two one-dimensional normal density functions are shown. The dashed vertical line represents the Bayes decision boundary. Right: 20 observations were drawn from each of the two classes, and are shown as histograms. The Bayes decision boundary is again shown as a dashed vertical line. The solid vertical line represents the LDA decision boundary estimated from the training data.

$X = x$ to the class for which (4.17) is largest. Taking the log of (4.17) and rearranging the terms, it is not hard to show³ that this is equivalent to assigning the observation to the class for which

$$\delta_k(x) = x \cdot \frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2} + \log(\pi_k) \quad (4.18)$$

is largest. For instance, if $K = 2$ and $\pi_1 = \pi_2$, then the Bayes classifier assigns an observation to class 1 if $2x(\mu_1 - \mu_2) > \mu_1^2 - \mu_2^2$, and to class 2 otherwise. The Bayes decision boundary is the point for which $\delta_1(x) = \delta_2(x)$; one can show that this amounts to

$$x = \frac{\mu_1^2 - \mu_2^2}{2(\mu_1 - \mu_2)} = \frac{\mu_1 + \mu_2}{2}. \quad (4.19)$$

An example is shown in the left-hand panel of Figure 4.4. The two normal density functions that are displayed, $f_1(x)$ and $f_2(x)$, represent two distinct classes. The mean and variance parameters for the two density functions are $\mu_1 = -1.25$, $\mu_2 = 1.25$, and $\sigma_1^2 = \sigma_2^2 = 1$. The two densities overlap, and so given that $X = x$, there is some uncertainty about the class to which the observation belongs. If we assume that an observation is equally likely to come from either class—that is, $\pi_1 = \pi_2 = 0.5$ —then by inspection of (4.19), we see that the Bayes classifier assigns the observation to class 1 if $x < 0$ and class 2 otherwise. Note that in this case, we can compute the Bayes classifier because we know that X is drawn from a Gaussian distribution within each class, and we know all of the parameters involved. In a real-life situation, we are not able to calculate the Bayes classifier.

In practice, even if we are quite certain of our assumption that X is drawn from a Gaussian distribution within each class, to apply the Bayes classifier we still have to estimate the parameters μ_1, \dots, μ_K , π_1, \dots, π_K , and σ^2 . The *linear discriminant analysis* (LDA) method approximates the Bayes classifier by plugging estimates for π_k , μ_k , and σ^2 into (4.18). In

linear
discriminant
analysis

³See Exercise 2 at the end of this chapter.

particular, the following estimates are used:

$$\begin{aligned}\hat{\mu}_k &= \frac{1}{n_k} \sum_{i:y_i=k} x_i \\ \hat{\sigma}^2 &= \frac{1}{n-K} \sum_{k=1}^K \sum_{i:y_i=k} (x_i - \hat{\mu}_k)^2\end{aligned}\quad (4.20)$$

where n is the total number of training observations, and n_k is the number of training observations in the k th class. The estimate for μ_k is simply the average of all the training observations from the k th class, while $\hat{\sigma}^2$ can be seen as a weighted average of the sample variances for each of the K classes. Sometimes we have knowledge of the class membership probabilities π_1, \dots, π_K , which can be used directly. In the absence of any additional information, LDA estimates π_k using the proportion of the training observations that belong to the k th class. In other words,

$$\hat{\pi}_k = n_k/n. \quad (4.21)$$

The LDA classifier plugs the estimates given in (4.20) and (4.21) into (4.18), and assigns an observation $X = x$ to the class for which

$$\hat{\delta}_k(x) = x \cdot \frac{\hat{\mu}_k}{\hat{\sigma}^2} - \frac{\hat{\mu}_k^2}{2\hat{\sigma}^2} + \log(\hat{\pi}_k) \quad (4.22)$$

is largest. The word *linear* in the classifier's name stems from the fact that the *discriminant functions* $\hat{\delta}_k(x)$ in (4.22) are linear functions of x (as opposed to a more complex function of x).

discriminant function

The right-hand panel of Figure 4.4 displays a histogram of a random sample of 20 observations from each class. To implement LDA, we began by estimating π_k , μ_k , and σ^2 using (4.20) and (4.21). We then computed the decision boundary, shown as a black solid line, that results from assigning an observation to the class for which (4.22) is largest. All points to the left of this line will be assigned to the green class, while points to the right of this line are assigned to the purple class. In this case, since $n_1 = n_2 = 20$, we have $\hat{\pi}_1 = \hat{\pi}_2$. As a result, the decision boundary corresponds to the midpoint between the sample means for the two classes, $(\hat{\mu}_1 + \hat{\mu}_2)/2$. The figure indicates that the LDA decision boundary is slightly to the left of the optimal Bayes decision boundary, which instead equals $(\mu_1 + \mu_2)/2 = 0$. How well does the LDA classifier perform on this data? Since this is simulated data, we can generate a large number of test observations in order to compute the Bayes error rate and the LDA test error rate. These are 10.6% and 11.1%, respectively. In other words, the LDA classifier's error rate is only 0.5% above the smallest possible error rate! This indicates that LDA is performing pretty well on this data set.

To reiterate, the LDA classifier results from assuming that the observations within each class come from a normal distribution with a class-specific mean and a common variance σ^2 , and plugging estimates for these parameters into the Bayes classifier. In Section 4.4.3, we will consider a less stringent set of assumptions, by allowing the observations in the k th class to have a class-specific variance, σ_k^2 .

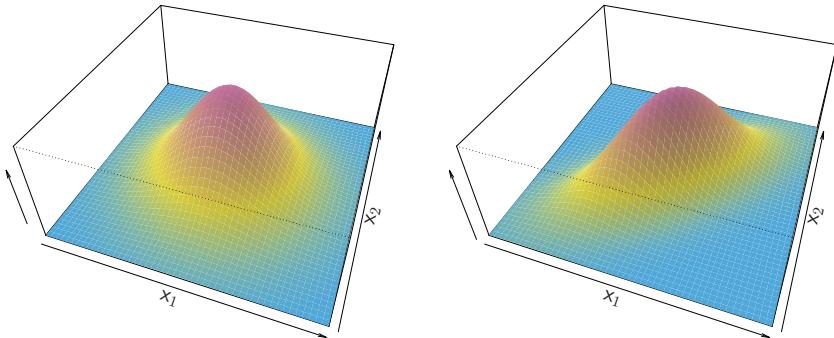


FIGURE 4.5. Two multivariate Gaussian density functions are shown, with $p = 2$. Left: The two predictors are uncorrelated. Right: The two variables have a correlation of 0.7.

4.4.2 Linear Discriminant Analysis for $p > 1$

We now extend the LDA classifier to the case of multiple predictors. To do this, we will assume that $X = (X_1, X_2, \dots, X_p)$ is drawn from a *multivariate Gaussian* (or multivariate normal) distribution, with a class-specific mean vector and a common covariance matrix. We begin with a brief review of this distribution.

multivariate Gaussian

The multivariate Gaussian distribution assumes that each individual predictor follows a one-dimensional normal distribution, as in (4.16), with some correlation between each pair of predictors. Two examples of multivariate Gaussian distributions with $p = 2$ are shown in Figure 4.5. The height of the surface at any particular point represents the probability that both X_1 and X_2 fall in a small region around that point. In either panel, if the surface is cut along the X_1 axis or along the X_2 axis, the resulting cross-section will have the shape of a one-dimensional normal distribution. The left-hand panel of Figure 4.5 illustrates an example in which $\text{Var}(X_1) = \text{Var}(X_2)$ and $\text{Cor}(X_1, X_2) = 0$; this surface has a characteristic *bell shape*. However, the bell shape will be distorted if the predictors are correlated or have unequal variances, as is illustrated in the right-hand panel of Figure 4.5. In this situation, the base of the bell will have an elliptical, rather than circular, shape. To indicate that a p -dimensional random variable X has a multivariate Gaussian distribution, we write $X \sim N(\mu, \Sigma)$. Here $E(X) = \mu$ is the mean of X (a vector with p components), and $\text{Cov}(X) = \Sigma$ is the $p \times p$ covariance matrix of X . Formally, the multivariate Gaussian density is defined as

$$f(x) = \frac{1}{(2\pi)^{p/2} |\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)\right). \quad (4.23)$$

In the case of $p > 1$ predictors, the LDA classifier assumes that the observations in the k th class are drawn from a multivariate Gaussian distribution $N(\mu_k, \Sigma)$, where μ_k is a class-specific mean vector, and Σ is a covariance matrix that is common to all K classes. Plugging the density function for the k th class, $f_k(X = x)$, into (4.15) and performing a little bit of algebra reveals that the Bayes classifier assigns an observation $X = x$

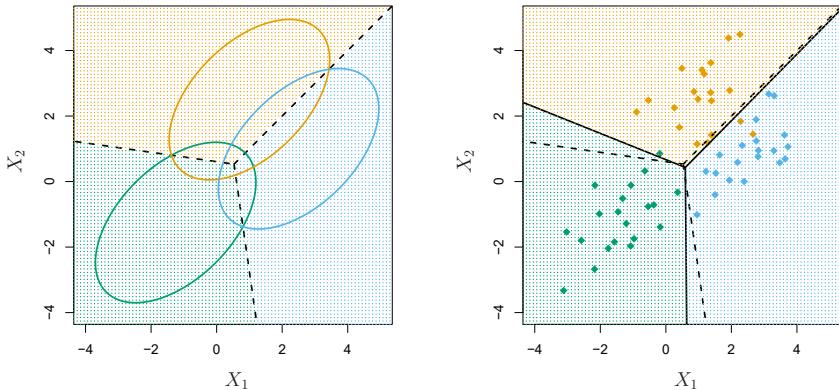


FIGURE 4.6. An example with three classes. The observations from each class are drawn from a multivariate Gaussian distribution with $p = 2$, with a class-specific mean vector and a common covariance matrix. Left: Ellipses that contain 95 % of the probability for each of the three classes are shown. The dashed lines are the Bayes decision boundaries. Right: 20 observations were generated from each class, and the corresponding LDA decision boundaries are indicated using solid black lines. The Bayes decision boundaries are once again shown as dashed lines.

to the class for which

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k + \log \pi_k \quad (4.24)$$

is largest. This is the vector/matrix version of (4.18).

An example is shown in the left-hand panel of Figure 4.6. Three equally-sized Gaussian classes are shown with class-specific mean vectors and a common covariance matrix. The three ellipses represent regions that contain 95 % of the probability for each of the three classes. The dashed lines are the Bayes decision boundaries. In other words, they represent the set of values x for which $\delta_k(x) = \delta_\ell(x)$; i.e.

$$x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k = x^T \Sigma^{-1} \mu_l - \frac{1}{2} \mu_l^T \Sigma^{-1} \mu_l \quad (4.25)$$

for $k \neq l$. (The $\log \pi_k$ term from (4.24) has disappeared because each of the three classes has the same number of training observations; i.e. π_k is the same for each class.) Note that there are three lines representing the Bayes decision boundaries because there are three *pairs of classes* among the three classes. That is, one Bayes decision boundary separates class 1 from class 2, one separates class 1 from class 3, and one separates class 2 from class 3. These three Bayes decision boundaries divide the predictor space into three regions. The Bayes classifier will classify an observation according to the region in which it is located.

Once again, we need to estimate the unknown parameters μ_1, \dots, μ_K , π_1, \dots, π_K , and Σ ; the formulas are similar to those used in the one-dimensional case, given in (4.20). To assign a new observation $X = x$, LDA plugs these estimates into (4.24) to obtain quantities $\hat{\delta}_k(x)$, and classifies to the class for which $\hat{\delta}_k(x)$ is largest. Note that in (4.24) $\delta_k(x)$ is a linear function of x ; that is, the LDA decision rule depends on x only

		True default status		
		No	Yes	Total
Predicted default status	No	9644	252	9896
	Yes	23	81	104
Total	9667	333	10000	

TABLE 4.4. A confusion matrix compares the LDA predictions to the true default statuses for the 10,000 training observations in the `Default` data set. Elements on the diagonal of the matrix represent individuals whose default statuses were correctly predicted, while off-diagonal elements represent individuals that were misclassified. LDA made incorrect predictions for 23 individuals who did not default and for 252 individuals who did default.

through a linear combination of its elements. As previously discussed, this is the reason for the word *linear* in LDA.

In the right-hand panel of Figure 4.6, 20 observations drawn from each of the three classes are displayed, and the resulting LDA decision boundaries are shown as solid black lines. Overall, the LDA decision boundaries are pretty close to the Bayes decision boundaries, shown again as dashed lines. The test error rates for the Bayes and LDA classifiers are 0.0746 and 0.0770, respectively. This indicates that LDA is performing well on this data.

We can perform LDA on the `Default` data in order to predict whether or not an individual will default on the basis of credit card balance and student status.⁴ The LDA model fit to the 10,000 training samples results in a *training* error rate of 2.75 %. This sounds like a low error rate, but two caveats must be noted.

- First of all, training error rates will usually be lower than test error rates, which are the real quantity of interest. In other words, we might expect this classifier to perform worse if we use it to predict whether or not a new set of individuals will default. The reason is that we specifically adjust the parameters of our model to do well on the training data. The higher the ratio of parameters p to number of samples n , the more we expect this *overfitting* to play a role. For these data we don't expect this to be a problem, since $p = 2$ and $n = 10,000$.
- Second, since only 3.33 % of the individuals in the training sample defaulted, a simple but useless classifier that always predicts that an individual will not default, regardless of his or her credit card balance and student status, will result in an error rate of 3.33 %. In other words, the trivial *null* classifier will achieve an error rate that is only a bit higher than the LDA training set error rate.

overfitting

null

In practice, a binary classifier such as this one can make two types of errors: it can incorrectly assign an individual who defaults to the *no default* category, or it can incorrectly assign an individual who does not default to

⁴The careful reader will notice that student status is qualitative — thus, the normality assumption made by LDA is clearly violated in this example! However, LDA is often remarkably robust to model violations, as this example shows. Naive Bayes, discussed in Section 4.4.4, provides an alternative to LDA that does not assume normally distributed predictors.

the *default* category. It is often of interest to determine which of these two types of errors are being made. A *confusion matrix*, shown for the **Default** data in Table 4.4, is a convenient way to display this information. The table reveals that LDA predicted that a total of 104 people would default. Of these people, 81 actually defaulted and 23 did not. Hence only 23 out of 9,667 of the individuals who did not default were incorrectly labeled. This looks like a pretty low error rate! However, of the 333 individuals who defaulted, 252 (or 75.7%) were missed by LDA. So while the overall error rate is low, the error rate among individuals who defaulted is very high. From the perspective of a credit card company that is trying to identify high-risk individuals, an error rate of $252/333 = 75.7\%$ among individuals who default may well be unacceptable.

Class-specific performance is also important in medicine and biology, where the terms *sensitivity* and *specificity* characterize the performance of a classifier or screening test. In this case the sensitivity is the percentage of true defaulters that are identified; it equals 24.3%. The specificity is the percentage of non-defaulters that are correctly identified; it equals $(1 - 23/9667) = 99.8\%$.

Why does LDA do such a poor job of classifying the customers who default? In other words, why does it have such low sensitivity? As we have seen, LDA is trying to approximate the Bayes classifier, which has the lowest *total* error rate out of all classifiers. That is, the Bayes classifier will yield the smallest possible total number of misclassified observations, regardless of the class from which the errors stem. Some misclassifications will result from incorrectly assigning a customer who does not default to the default class, and others will result from incorrectly assigning a customer who defaults to the non-default class. In contrast, a credit card company might particularly wish to avoid incorrectly classifying an individual who will default, whereas incorrectly classifying an individual who will not default, though still to be avoided, is less problematic. We will now see that it is possible to modify LDA in order to develop a classifier that better meets the credit card company's needs.

The Bayes classifier works by assigning an observation to the class for which the posterior probability $p_k(X)$ is greatest. In the two-class case, this amounts to assigning an observation to the *default* class if

$$\Pr(\text{default} = \text{Yes}|X = x) > 0.5. \quad (4.26)$$

Thus, the Bayes classifier, and by extension LDA, uses a threshold of 50% for the posterior probability of default in order to assign an observation to the *default* class. However, if we are concerned about incorrectly predicting the default status for individuals who default, then we can consider lowering this threshold. For instance, we might label any customer with a posterior probability of default above 20% to the *default* class. In other words, instead of assigning an observation to the *default* class if (4.26) holds, we could instead assign an observation to this class if

$$\Pr(\text{default} = \text{Yes}|X = x) > 0.2. \quad (4.27)$$

The error rates that result from taking this approach are shown in Table 4.5. Now LDA predicts that 430 individuals will default. Of the 333 individuals who default, LDA correctly predicts all but 138, or 41.4%. This is a vast

confusion
matrix

sensitivity
specificity

		<i>True default status</i>		Total
		No	Yes	
<i>Predicted default status</i>	No	9432	138	9570
	Yes	235	195	430
	Total	9667	333	10000

TABLE 4.5. A confusion matrix compares the LDA predictions to the true default statuses for the 10,000 training observations in the **Default** data set, using a modified threshold value that predicts default for any individuals whose posterior default probability exceeds 20 %.

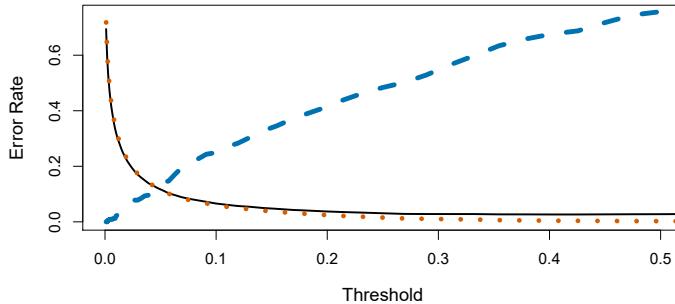


FIGURE 4.7. For the **Default** data set, error rates are shown as a function of the threshold value for the posterior probability that is used to perform the assignment. The black solid line displays the overall error rate. The blue dashed line represents the fraction of defaulting customers that are incorrectly classified, and the orange dotted line indicates the fraction of errors among the non-defaulting customers.

improvement over the error rate of 75.7 % that resulted from using the threshold of 50 %. However, this improvement comes at a cost: now 235 individuals who do not default are incorrectly classified. As a result, the overall error rate has increased slightly to 3.73 %. But a credit card company may consider this slight increase in the total error rate to be a small price to pay for more accurate identification of individuals who do indeed default.

Figure 4.7 illustrates the trade-off that results from modifying the threshold old value for the posterior probability of default. Various error rates are shown as a function of the threshold value. Using a threshold of 0.5, as in (4.26), minimizes the overall error rate, shown as a black solid line. This is to be expected, since the Bayes classifier uses a threshold of 0.5 and is known to have the lowest overall error rate. But when a threshold of 0.5 is used, the error rate among the individuals who default is quite high (blue dashed line). As the threshold is reduced, the error rate among individuals who default decreases steadily, but the error rate among the individuals who do not default increases. How can we decide which threshold value is best? Such a decision must be based on *domain knowledge*, such as detailed information about the costs associated with default.

The *ROC curve* is a popular graphic for simultaneously displaying the two types of errors for all possible thresholds. The name “ROC” is historic, and comes from communications theory. It is an acronym for *receiver operating characteristics*. Figure 4.8 displays the ROC curve for the LDA classifier on the training data. The overall performance of a classifier, sum-

ROC curve

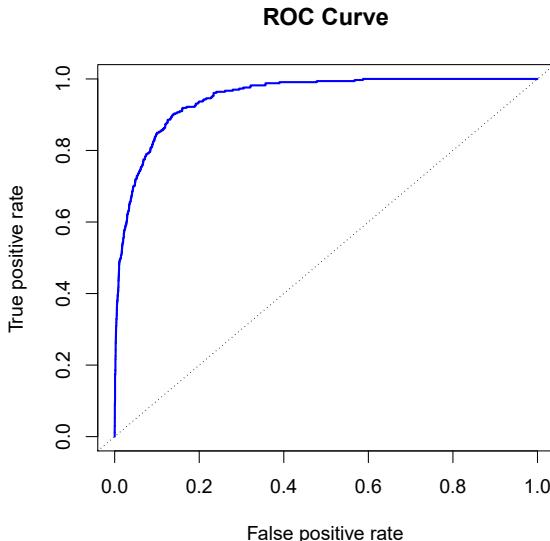


FIGURE 4.8. A ROC curve for the LDA classifier on the `Default` data. It traces out two types of error as we vary the threshold value for the posterior probability of default. The actual thresholds are not shown. The true positive rate is the sensitivity: the fraction of defaulters that are correctly identified, using a given threshold value. The false positive rate is 1-specificity: the fraction of non-defaulters that we classify incorrectly as defaulters, using that same threshold value. The ideal ROC curve hugs the top left corner, indicating a high true positive rate and a low false positive rate. The dotted line represents the “no information” classifier; this is what we would expect if student status and credit card balance are not associated with probability of default.

marized over all possible thresholds, is given by the *area under the (ROC) curve* (AUC). An ideal ROC curve will hug the top left corner, so the larger the AUC the better the classifier. For this data the AUC is 0.95, which is close to the maximum of 1.0, so would be considered very good. We expect a classifier that performs no better than chance to have an AUC of 0.5 (when evaluated on an independent test set not used in model training). ROC curves are useful for comparing different classifiers, since they take into account all possible thresholds. It turns out that the ROC curve for the logistic regression model of Section 4.3.4 fit to these data is virtually indistinguishable from this one for the LDA model, so we do not display it here.

area under
the (ROC)
curve

As we have seen above, varying the classifier threshold changes its true positive and false positive rate. These are also called the *sensitivity* and one minus the *specificity* of our classifier. Since there is an almost bewildering array of terms used in this context, we now give a summary. Table 4.6 shows the possible results when applying a classifier (or diagnostic test) to a population. To make the connection with the epidemiology literature, we think of “+” as the “disease” that we are trying to detect, and “−” as the “non-disease” state. To make the connection to the classical hypothesis testing literature, we think of “−” as the null hypothesis and “+” as the

sensitivity
specificity

		<i>True class</i>		
		– or Null	+ or Non-null	Total
<i>Predicted class</i>	– or Null	True Neg. (TN)	False Neg. (FN)	N*
	+ or Non-null	False Pos. (FP)	True Pos. (TP)	P*
Total		N	P	

TABLE 4.6. Possible results when applying a classifier or diagnostic test to a population.

Name	Definition	Synonyms
False Pos. rate	FP/N	Type I error, 1–Specificity
True Pos. rate	TP/P	1–Type II error, power, sensitivity, recall
Pos. Pred. value	TP/P*	Precision, 1–false discovery proportion
Neg. Pred. value	TN/N*	

TABLE 4.7. Important measures for classification and diagnostic testing, derived from quantities in Table 4.6.

alternative (non-null) hypothesis. In the context of the **Default** data, “+” indicates an individual who defaults, and “–” indicates one who does not.

Table 4.7 lists many of the popular performance measures that are used in this context. The denominators for the false positive and true positive rates are the actual population counts in each class. In contrast, the denominators for the positive predictive value and the negative predictive value are the total predicted counts for each class.

4.4.3 Quadratic Discriminant Analysis

As we have discussed, LDA assumes that the observations within each class are drawn from a multivariate Gaussian distribution with a class-specific mean vector and a covariance matrix that is common to all K classes. *Quadratic discriminant analysis* (QDA) provides an alternative approach. Like LDA, the QDA classifier results from assuming that the observations from each class are drawn from a Gaussian distribution, and plugging estimates for the parameters into Bayes’ theorem in order to perform prediction. However, unlike LDA, QDA assumes that each class has its own covariance matrix. That is, it assumes that an observation from the k th class is of the form $X \sim N(\mu_k, \Sigma_k)$, where Σ_k is a covariance matrix for the k th class. Under this assumption, the Bayes classifier assigns an observation $X = x$ to the class for which

$$\begin{aligned} \delta_k(x) &= -\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k) - \frac{1}{2} \log |\Sigma_k| + \log \pi_k \\ &= -\frac{1}{2}x^T \Sigma_k^{-1} x + x^T \Sigma_k^{-1} \mu_k - \frac{1}{2}\mu_k^T \Sigma_k^{-1} \mu_k - \frac{1}{2} \log |\Sigma_k| + \log \pi_k \end{aligned} \tag{4.28}$$

quadratic
discriminant
analysis

is largest. So the QDA classifier involves plugging estimates for Σ_k , μ_k , and π_k into (4.28), and then assigning an observation $X = x$ to the class for which this quantity is largest. Unlike in (4.24), the quantity x appears as a *quadratic* function in (4.28). This is where QDA gets its name.

Why does it matter whether or not we assume that the K classes share a common covariance matrix? In other words, why would one prefer LDA to

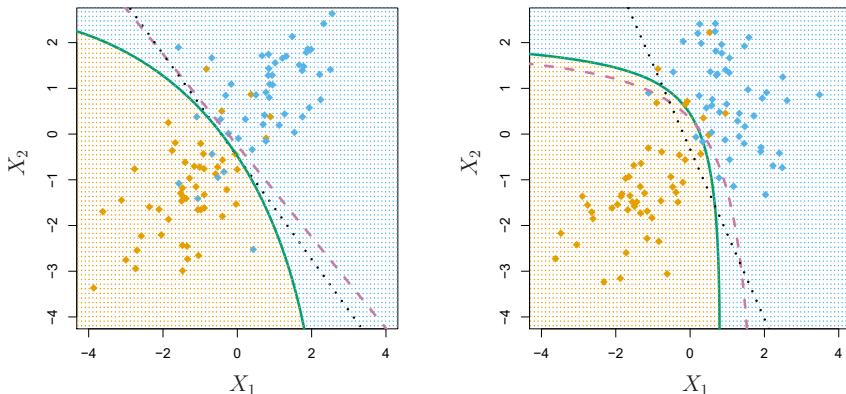


FIGURE 4.9. Left: The Bayes (purple dashed), LDA (black dotted), and QDA (green solid) decision boundaries for a two-class problem with $\Sigma_1 = \Sigma_2$. The shading indicates the QDA decision rule. Since the Bayes decision boundary is linear, it is more accurately approximated by LDA than by QDA. Right: Details are as given in the left-hand panel, except that $\Sigma_1 \neq \Sigma_2$. Since the Bayes decision boundary is non-linear, it is more accurately approximated by QDA than by LDA.

QDA, or vice-versa? The answer lies in the bias-variance trade-off. When there are p predictors, then estimating a covariance matrix requires estimating $p(p+1)/2$ parameters. QDA estimates a separate covariance matrix for each class, for a total of $Kp(p+1)/2$ parameters. With 50 predictors this is some multiple of 1,275, which is a lot of parameters. By instead assuming that the K classes share a common covariance matrix, the LDA model becomes linear in x , which means there are Kp linear coefficients to estimate. Consequently, LDA is a much less flexible classifier than QDA, and so has substantially lower variance. This can potentially lead to improved prediction performance. But there is a trade-off: if LDA's assumption that the K classes share a common covariance matrix is badly off, then LDA can suffer from high bias. Roughly speaking, LDA tends to be a better bet than QDA if there are relatively few training observations and so reducing variance is crucial. In contrast, QDA is recommended if the training set is very large, so that the variance of the classifier is not a major concern, or if the assumption of a common covariance matrix for the K classes is clearly untenable.

Figure 4.9 illustrates the performances of LDA and QDA in two scenarios. In the left-hand panel, the two Gaussian classes have a common correlation of 0.7 between X_1 and X_2 . As a result, the Bayes decision boundary is linear and is accurately approximated by the LDA decision boundary. The QDA decision boundary is inferior, because it suffers from higher variance without a corresponding decrease in bias. In contrast, the right-hand panel displays a situation in which the orange class has a correlation of 0.7 between the variables and the blue class has a correlation of -0.7. Now the Bayes decision boundary is quadratic, and so QDA more accurately approximates this boundary than does LDA.

4.4.4 Naive Bayes

In previous sections, we used Bayes' theorem (4.15) to develop the LDA and QDA classifiers. Here, we use Bayes' theorem to motivate the popular *naive Bayes* classifier.

Recall that Bayes' theorem (4.15) provides an expression for the posterior probability $p_k(x) = \Pr(Y = k|X = x)$ in terms of π_1, \dots, π_K and $f_1(x), \dots, f_K(x)$. To use (4.15) in practice, we need estimates for π_1, \dots, π_K and $f_1(x), \dots, f_K(x)$. As we saw in previous sections, estimating the prior probabilities π_1, \dots, π_K is typically straightforward: for instance, we can estimate $\hat{\pi}_k$ as the proportion of training observations belonging to the k th class, for $k = 1, \dots, K$.

naive Bayes

However, estimating $f_1(x), \dots, f_K(x)$ is more subtle. Recall that $f_k(x)$ is the p -dimensional density function for an observation in the k th class, for $k = 1, \dots, K$. In general, estimating a p -dimensional density function is challenging. In LDA, we make a very strong assumption that greatly simplifies the task: we assume that f_k is the density function for a multivariate normal random variable with class-specific mean μ_k , and shared covariance matrix Σ . By contrast, in QDA, we assume that f_k is the density function for a multivariate normal random variable with class-specific mean μ_k , and class-specific covariance matrix Σ_k . By making these very strong assumptions, we are able to replace the very challenging problem of estimating K p -dimensional density functions with the much simpler problem of estimating K p -dimensional mean vectors and one (in the case of LDA) or K (in the case of QDA) $(p \times p)$ -dimensional covariance matrices.

The naive Bayes classifier takes a different tack for estimating $f_1(x), \dots, f_K(x)$. Instead of assuming that these functions belong to a particular family of distributions (e.g. multivariate normal), we instead make a single assumption:

Within the k th class, the p predictors are independent.

Stated mathematically, this assumption means that for $k = 1, \dots, K$,

$$f_k(x) = f_{k1}(x_1) \times f_{k2}(x_2) \times \cdots \times f_{kp}(x_p), \quad (4.29)$$

where f_{kj} is the density function of the j th predictor among observations in the k th class.

Why is this assumption so powerful? Essentially, estimating a p -dimensional density function is challenging because we must consider not only the *marginal distribution* of each predictor — that is, the distribution of each predictor on its own — but also the *joint distribution* of the predictors — that is, the association between the different predictors. In the case of a multivariate normal distribution, the association between the different predictors is summarized by the off-diagonal elements of the covariance matrix. However, in general, this association can be very hard to characterize, and exceedingly challenging to estimate. But by assuming that the p covariates are independent within each class, we completely eliminate the need to worry about the association between the p predictors, because we have simply assumed that there is *no* association between the predictors!

marginal
distribution
joint
distribution

Do we really believe the naive Bayes assumption that the p covariates are independent within each class? In most settings, we do not. But even though this modeling assumption is made for convenience, it often leads to

pretty decent results, especially in settings where n is not large enough relative to p for us to effectively estimate the joint distribution of the predictors within each class. In fact, since estimating a joint distribution requires such a huge amount of data, naive Bayes is a good choice in a wide range of settings. Essentially, the naive Bayes assumption introduces some bias, but reduces variance, leading to a classifier that works quite well in practice as a result of the bias-variance trade-off.

Once we have made the naive Bayes assumption, we can plug (4.29) into (4.15) to obtain an expression for the posterior probability,

$$\Pr(Y = k|X = x) = \frac{\pi_k \times f_{k1}(x_1) \times f_{k2}(x_2) \times \cdots \times f_{kp}(x_p)}{\sum_{l=1}^K \pi_l \times f_{l1}(x_1) \times f_{l2}(x_2) \times \cdots \times f_{lp}(x_p)} \quad (4.30)$$

for $k = 1, \dots, K$.

To estimate the one-dimensional density function f_{kj} using training data x_{1j}, \dots, x_{nj} , we have a few options.

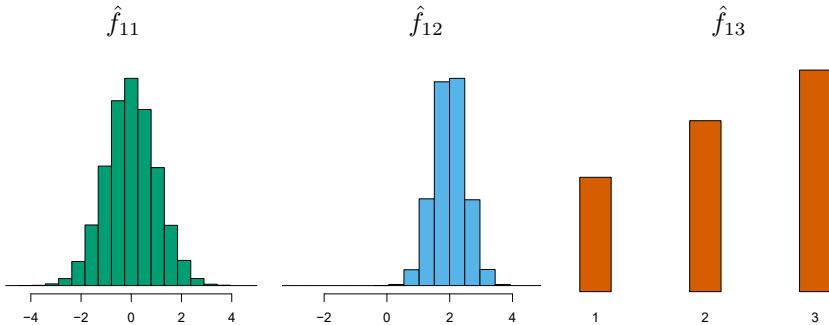
- If X_j is quantitative, then we can assume that $X_j|Y = k \sim N(\mu_{jk}, \sigma_{jk}^2)$. In other words, we assume that within each class, the j th predictor is drawn from a (univariate) normal distribution. While this may sound a bit like QDA, there is one key difference, in that here we are assuming that the predictors are independent; this amounts to QDA with an additional assumption that the class-specific covariance matrix is diagonal.
- If X_j is quantitative, then another option is to use a non-parametric estimate for f_{kj} . A very simple way to do this is by making a histogram for the observations of the j th predictor within each class. Then we can estimate $f_{kj}(x_j)$ as the fraction of the training observations in the k th class that belong to the same histogram bin as x_j . Alternatively, we can use a *kernel density estimator*, which is essentially a smoothed version of a histogram.
- If X_j is qualitative, then we can simply count the proportion of training observations for the j th predictor corresponding to each class. For instance, suppose that $X_j \in \{1, 2, 3\}$, and we have 100 observations in the k th class. Suppose that the j th predictor takes on values of 1, 2, and 3 in 32, 55, and 13 of those observations, respectively. Then we can estimate f_{kj} as

$$\hat{f}_{kj}(x_j) = \begin{cases} 0.32 & \text{if } x_j = 1 \\ 0.55 & \text{if } x_j = 2 \\ 0.13 & \text{if } x_j = 3. \end{cases}$$

kernel
density
estimator

We now consider the naive Bayes classifier in a toy example with $p = 3$ predictors and $K = 2$ classes. The first two predictors are quantitative, and the third predictor is qualitative with three levels. Suppose further that $\hat{\pi}_1 = \hat{\pi}_2 = 0.5$. The estimated density functions \hat{f}_{kj} for $k = 1, 2$ and $j = 1, 2, 3$ are displayed in Figure 4.10. Now suppose that we wish to classify a new observation, $x^* = (0.4, 1.5, 1)^T$. It turns out that in this

Density estimates for class k=1



Density estimates for class k=2

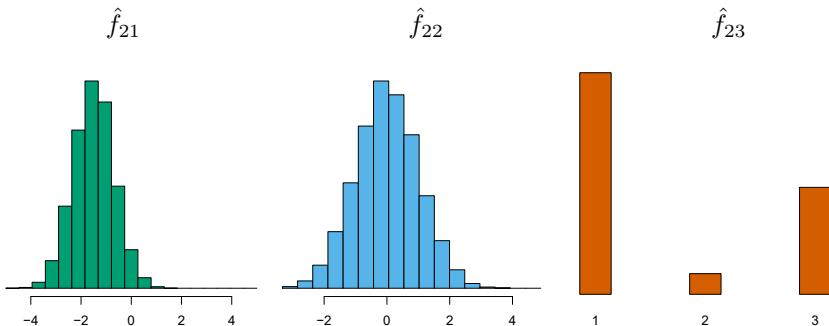


FIGURE 4.10. In the toy example in Section 4.4.4, we generate data with $p = 3$ predictors and $K = 2$ classes. The first two predictors are quantitative, and the third predictor is qualitative with three levels. In each class, the estimated density for each of the three predictors is displayed. If the prior probabilities for the two classes are equal, then the observation $x^* = (0.4, 1.5, 1)^T$ has a 94.4% posterior probability of belonging to the first class.

		True default status		Total
		No	Yes	
Predicted default status	No	9621	244	9865
	Yes	46	89	135
	Total	9667	333	10000

TABLE 4.8. Comparison of the naive Bayes predictions to the true default status for the 10,000 training observations in the `Default` data set, when we predict default for any observation for which $P(Y = \text{default}|X = x) > 0.5$.

example, $\hat{f}_{11}(0.4) = 0.368$, $\hat{f}_{12}(1.5) = 0.484$, $\hat{f}_{13}(1) = 0.226$, and $\hat{f}_{21}(0.4) = 0.030$, $\hat{f}_{22}(1.5) = 0.130$, $\hat{f}_{23}(1) = 0.616$. Plugging these estimates into (4.30) results in posterior probability estimates of $\Pr(Y = 1|X = x^*) = 0.944$ and $\Pr(Y = 2|X = x^*) = 0.056$.

Table 4.8 provides the confusion matrix resulting from applying the naive Bayes classifier to the `Default` data set, where we predict a default if the posterior probability of a default — that is, $P(Y = \text{default}|X = x)$ — exceeds 0.5. Comparing this to the results for LDA in Table 4.4, our findings are mixed. While LDA has a slightly lower overall error rate, naive Bayes

		True default status		
		No	Yes	Total
Predicted default status	No	9339	130	9469
	Yes	328	203	531
Total	9667	333	10000	

TABLE 4.9. Comparison of the naive Bayes predictions to the true default status for the 10,000 training observations in the `Default` data set, when we predict default for any observation for which $P(Y = \text{default}|X = x) > 0.2$.

correctly predicts a higher fraction of the true defaulters. In this implementation of naive Bayes, we have assumed that each quantitative predictor is drawn from a Gaussian distribution (and, of course, that within each class, each predictor is independent).

Just as with LDA, we can easily adjust the probability threshold for predicting a default. For example, Table 4.9 provides the confusion matrix resulting from predicting a default if $P(Y = \text{default}|X = x) > 0.2$. Again, the results are mixed relative to LDA with the same threshold (Table 4.5). Naive Bayes has a higher error rate, but correctly predicts almost two-thirds of the true defaults.

In this example, it should not be too surprising that naive Bayes does not convincingly outperform LDA: this data set has $n = 10,000$ and $p = 2$, and so the reduction in variance resulting from the naive Bayes assumption is not necessarily worthwhile. We expect to see a greater pay-off to using naive Bayes relative to LDA or QDA in instances where p is larger or n is smaller, so that reducing the variance is very important.

4.5 A Comparison of Classification Methods

4.5.1 An Analytical Comparison

We now perform an *analytical* (or mathematical) comparison of LDA, QDA, naive Bayes, and logistic regression. We consider these approaches in a setting with K classes, so that we assign an observation to the class that maximizes $\Pr(Y = k|X = x)$. Equivalently, we can set K as the *baseline* class and assign an observation to the class that maximizes

$$\log \left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = K|X = x)} \right) \quad (4.31)$$

for $k = 1, \dots, K$. Examining the specific form of (4.31) for each method provides a clear understanding of their similarities and differences.

First, for LDA, we can make use of Bayes' theorem (4.15) as well as the assumption that the predictors within each class are drawn from a multivariate normal density (4.23) with class-specific mean and shared co-

variance matrix in order to show that

$$\begin{aligned}
\log \left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = K|X = x)} \right) &= \log \left(\frac{\pi_k f_k(x)}{\pi_K f_K(x)} \right) \\
&= \log \left(\frac{\pi_k \exp \left(-\frac{1}{2}(x - \mu_k)^T \Sigma^{-1} (x - \mu_k) \right)}{\pi_K \exp \left(-\frac{1}{2}(x - \mu_K)^T \Sigma^{-1} (x - \mu_K) \right)} \right) \\
&= \log \left(\frac{\pi_k}{\pi_K} \right) - \frac{1}{2}(x - \mu_k)^T \Sigma^{-1} (x - \mu_k) \\
&\quad + \frac{1}{2}(x - \mu_K)^T \Sigma^{-1} (x - \mu_K) \\
&= \log \left(\frac{\pi_k}{\pi_K} \right) - \frac{1}{2}(\mu_k + \mu_K)^T \Sigma^{-1} (\mu_k - \mu_K) \\
&\quad + x^T \Sigma^{-1} (\mu_k - \mu_K) \\
&= a_k + \sum_{j=1}^p b_{kj} x_j,
\end{aligned} \tag{4.32}$$

where $a_k = \log \left(\frac{\pi_k}{\pi_K} \right) - \frac{1}{2}(\mu_k + \mu_K)^T \Sigma^{-1} (\mu_k - \mu_K)$ and b_{kj} is the j th component of $\Sigma^{-1} (\mu_k - \mu_K)$. Hence LDA, like logistic regression, assumes that the log odds of the posterior probabilities is linear in x .

Using similar calculations, in the QDA setting (4.31) becomes

$$\log \left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = K|X = x)} \right) = a_k + \sum_{j=1}^p b_{kj} x_j + \sum_{j=1}^p \sum_{l=1}^p c_{kjl} x_j x_l, \tag{4.33}$$

where a_k, b_{kj} , and c_{kjl} are functions of $\pi_k, \pi_K, \mu_k, \mu_K, \Sigma_k$ and Σ_K . Again, as the name suggests, QDA assumes that the log odds of the posterior probabilities is quadratic in x .

Finally, we examine (4.31) in the naive Bayes setting. Recall that in this setting, $f_k(x)$ is modeled as a product of p one-dimensional functions $f_{kj}(x_j)$ for $j = 1, \dots, p$. Hence,

$$\begin{aligned}
\log \left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = K|X = x)} \right) &= \log \left(\frac{\pi_k f_k(x)}{\pi_K f_K(x)} \right) \\
&= \log \left(\frac{\pi_k \prod_{j=1}^p f_{kj}(x_j)}{\pi_K \prod_{j=1}^p f_{Kj}(x_j)} \right) \\
&= \log \left(\frac{\pi_k}{\pi_K} \right) + \sum_{j=1}^p \log \left(\frac{f_{kj}(x_j)}{f_{Kj}(x_j)} \right) \\
&= a_k + \sum_{j=1}^p g_{kj}(x_j),
\end{aligned} \tag{4.34}$$

where $a_k = \log \left(\frac{\pi_k}{\pi_K} \right)$ and $g_{kj}(x_j) = \log \left(\frac{f_{kj}(x_j)}{f_{Kj}(x_j)} \right)$. Hence, the right-hand side of (4.34) takes the form of a *generalized additive model*, a topic that is discussed further in Chapter 7.

Inspection of (4.32), (4.33), and (4.34) yields the following observations about LDA, QDA, and naive Bayes:

- LDA is a special case of QDA with $c_{kjl} = 0$ for all $j = 1, \dots, p$, $l = 1, \dots, p$, and $k = 1, \dots, K$. (Of course, this is not surprising, since LDA is simply a restricted version of QDA with $\Sigma_1 = \dots = \Sigma_K = \Sigma$.)
- Any classifier with a linear decision boundary is a special case of naive Bayes with $g_{kj}(x_j) = b_{kj}x_j$. In particular, this means that LDA is a special case of naive Bayes! This is not at all obvious from the descriptions of LDA and naive Bayes earlier in this chapter, since each method makes very different assumptions: LDA assumes that the features are normally distributed with a common within-class covariance matrix, and naive Bayes instead assumes independence of the features.
- If we model $f_{kj}(x_j)$ in the naive Bayes classifier using a one-dimensional Gaussian distribution $N(\mu_{kj}, \sigma_j^2)$, then we end up with $g_{kj}(x_j) = b_{kj}x_j$ where $b_{kj} = (\mu_{kj} - \mu_{Kj})/\sigma_j^2$. In this case, naive Bayes is actually a special case of LDA with Σ restricted to be a diagonal matrix with j th diagonal element equal to σ_j^2 .
- Neither QDA nor naive Bayes is a special case of the other. Naive Bayes can produce a more flexible fit, since any choice can be made for $g_{kj}(x_j)$. However, it is restricted to a purely *additive* fit, in the sense that in (4.34), a function of x_j is *added* to a function of x_l , for $j \neq l$; however, these terms are never multiplied. By contrast, QDA includes multiplicative terms of the form $c_{kjl}x_jx_l$. Therefore, QDA has the potential to be more accurate in settings where interactions among the predictors are important in discriminating between classes.

None of these methods uniformly dominates the others: in any setting, the choice of method will depend on the true distribution of the predictors in each of the K classes, as well as other considerations, such as the values of n and p . The latter ties into the bias-variance trade-off.

How does logistic regression tie into this story? Recall from (4.12) that multinomial logistic regression takes the form

$$\log \left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = K|X = x)} \right) = \beta_{k0} + \sum_{j=1}^p \beta_{kj}x_j.$$

This is identical to the linear form of LDA (4.32): in both cases, $\log \left(\frac{\Pr(Y = k|X = x)}{\Pr(Y = K|X = x)} \right)$ is a linear function of the predictors. In LDA, the coefficients in this linear function are functions of estimates for π_k , π_K , μ_k , μ_K , and Σ obtained by assuming that X_1, \dots, X_p follow a normal distribution within each class. By contrast, in logistic regression, the coefficients are chosen to maximize the likelihood function (4.5). Thus, we expect LDA to outperform logistic regression when the normality assumption (approximately) holds, and we expect logistic regression to perform better when it does not.

We close with a brief discussion of *K-nearest neighbors* (KNN), introduced in Chapter 2. Recall that KNN takes a completely different approach from the classifiers seen in this chapter. In order to make a prediction for an observation $X = x$, the training observations that are closest to x are identified. Then X is assigned to the class to which the plurality of these observations belong. Hence KNN is a completely non-parametric approach: no assumptions are made about the shape of the decision boundary. We make the following observations about KNN:

- Because KNN is completely non-parametric, we can expect this approach to dominate LDA and logistic regression when the decision boundary is highly non-linear, provided that n is very large and p is small.
- In order to provide accurate classification, KNN requires *a lot* of observations relative to the number of predictors—that is, n much larger than p . This has to do with the fact that KNN is non-parametric, and thus tends to reduce the bias while incurring a lot of variance.
- In settings where the decision boundary is non-linear but n is only modest, or p is not very small, then QDA may be preferred to KNN. This is because QDA can provide a non-linear decision boundary while taking advantage of a parametric form, which means that it requires a smaller sample size for accurate classification, relative to KNN.
- Unlike logistic regression, KNN does not tell us which predictors are important: we don't get a table of coefficients as in Table 4.3.

4.5.2 An Empirical Comparison

We now compare the *empirical* (practical) performance of logistic regression, LDA, QDA, naive Bayes, and KNN. We generated data from six different scenarios, each of which involves a binary (two-class) classification problem. In three of the scenarios, the Bayes decision boundary is linear, and in the remaining scenarios it is non-linear. For each scenario, we produced 100 random training data sets. On each of these training sets, we fit each method to the data and computed the resulting test error rate on a large test set. Results for the linear scenarios are shown in Figure 4.11, and the results for the non-linear scenarios are in Figure 4.12. The KNN method requires selection of K , the number of neighbors (not to be confused with the number of classes in earlier sections of this chapter). We performed KNN with two values of K : $K = 1$, and a value of K that was chosen automatically using an approach called *cross-validation*, which we discuss further in Chapter 5. We applied naive Bayes assuming univariate Gaussian densities for the features within each class (and, of course — since this is the key characteristic of naive Bayes — assuming independence of the features).

In each of the six scenarios, there were $p = 2$ quantitative predictors. The scenarios were as follows:

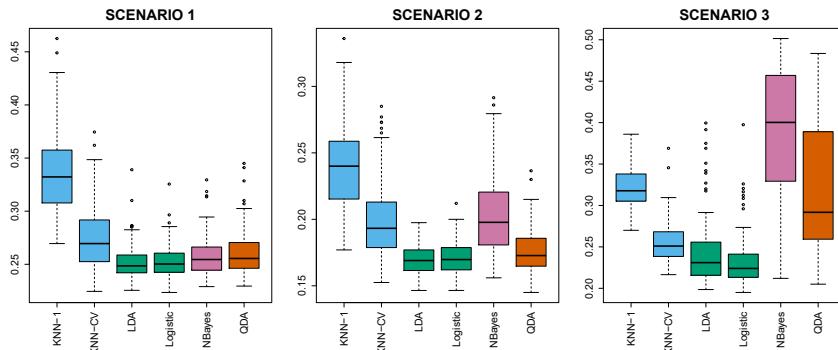


FIGURE 4.11. Boxplots of the test error rates for each of the linear scenarios described in the main text.

Scenario 1: There were 20 training observations in each of two classes. The observations within each class were uncorrelated random normal variables with a different mean in each class. The left-hand panel of Figure 4.11 shows that LDA performed well in this setting, as one would expect since this is the model assumed by LDA. Logistic regression also performed quite well, since it assumes a linear decision boundary. KNN performed poorly because it paid a price in terms of variance that was not offset by a reduction in bias. QDA also performed worse than LDA, since it fit a more flexible classifier than necessary. The performance of naive Bayes was slightly better than QDA, because the naive Bayes assumption of independent predictors is correct.

Scenario 2: Details are as in Scenario 1, except that within each class, the two predictors had a correlation of -0.5 . The center panel of Figure 4.11 indicates that the performance of most methods is similar to the previous scenario. The notable exception is naive Bayes, which performs very poorly here, since the naive Bayes assumption of independent predictors is violated.

Scenario 3: As in the previous scenario, there is substantial negative correlation between the predictors within each class. However, this time we generated X_1 and X_2 from the t -distribution, with 50 observations per class. The t -distribution has a similar shape to the normal distribution, but it has a tendency to yield more extreme points—that is, more points that are far from the mean. In this setting, the decision boundary was still linear, and so fit into the logistic regression framework. The set-up violated the assumptions of LDA, since the observations were not drawn from a normal distribution. The right-hand panel of Figure 4.11 shows that logistic regression outperformed LDA, though both methods were superior to the other approaches. In particular, the QDA results deteriorated considerably as a consequence of non-normality. Naive Bayes performed very poorly because the independence assumption is violated.

t -distribution

Scenario 4: The data were generated from a normal distribution, with a correlation of 0.5 between the predictors in the first class, and correlation of -0.5 between the predictors in the second class. This setup corresponded to the QDA assumption, and resulted in quadratic decision boundaries. The left-hand panel of Figure 4.12 shows that QDA outperformed all of the

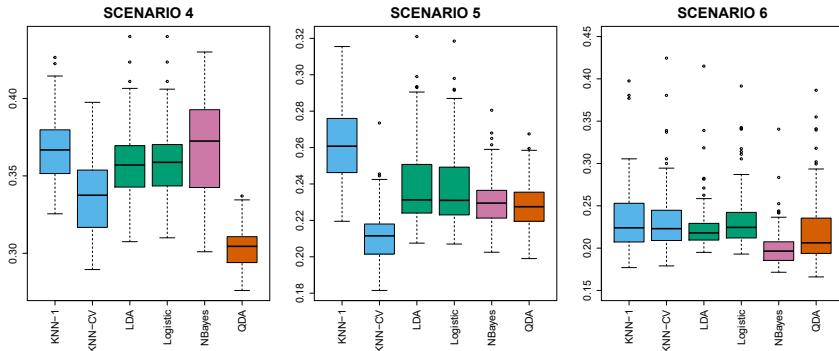


FIGURE 4.12. Boxplots of the test error rates for each of the non-linear scenarios described in the main text.

other approaches. The naive Bayes assumption of independent predictors is violated, so naive Bayes performs poorly.

Scenario 5: The data were generated from a normal distribution with uncorrelated predictors. Then the responses were sampled from the logistic function applied to a complicated non-linear function of the predictors. The center panel of Figure 4.12 shows that both QDA and naive Bayes gave slightly better results than the linear methods, while the much more flexible KNN-CV method gave the best results. But KNN with $K = 1$ gave the worst results out of all methods. This highlights the fact that even when the data exhibits a complex non-linear relationship, a non-parametric method such as KNN can still give poor results if the level of smoothness is not chosen correctly.

Scenario 6: The observations were generated from a normal distribution with a different diagonal covariance matrix for each class. However, the sample size was *very* small: just $n = 6$ in each class. Naive Bayes performed very well, because its assumptions are met. LDA and logistic regression performed poorly because the true decision boundary is non-linear, due to the unequal covariance matrices. QDA performed a bit worse than naive Bayes, because given the very small sample size, the former incurred too much variance in estimating the correlation between the predictors within each class. KNN's performance also suffered due to the very small sample size.

These six examples illustrate that no one method will dominate the others in every situation. When the true decision boundaries are linear, then the LDA and logistic regression approaches will tend to perform well. When the boundaries are moderately non-linear, QDA or naive Bayes may give better results. Finally, for much more complicated decision boundaries, a non-parametric approach such as KNN can be superior. But the level of smoothness for a non-parametric approach must be chosen carefully. In the next chapter we examine a number of approaches for choosing the correct level of smoothness and, in general, for selecting the best overall method.

Finally, recall from Chapter 3 that in the regression setting we can accommodate a non-linear relationship between the predictors and the response by performing regression using transformations of the predictors. A similar approach could be taken in the classification setting. For instance, we could

	Coefficient	Std. error	t-statistic	p-value
Intercept	73.60	5.13	14.34	0.00
workingday	1.27	1.78	0.71	0.48
temp	157.21	10.26	15.32	0.00
weathersit [cloudy/misty]	-12.89	1.96	-6.56	0.00
weathersit [light rain/snow]	-66.49	2.97	-22.43	0.00
weathersit [heavy rain/snow]	-109.75	76.67	-1.43	0.15

TABLE 4.10. Results for a least squares linear model fit to predict `bikers` in the `Bikeshare` data. The predictors `mnth` and `hr` are omitted from this table due to space constraints, and can be seen in Figure 4.13. For the qualitative variable `weathersit`, the baseline level corresponds to clear skies.

create a more flexible version of logistic regression by including X^2 , X^3 , and even X^4 as predictors. This may or may not improve logistic regression's performance, depending on whether the increase in variance due to the added flexibility is offset by a sufficiently large reduction in bias. We could do the same for LDA. If we added all possible quadratic terms and cross-products to LDA, the form of the model would be the same as the QDA model, although the parameter estimates would be different. This device allows us to move somewhere between an LDA and a QDA model.

4.6 Generalized Linear Models

In Chapter 3, we assumed that the response Y is quantitative, and explored the use of least squares linear regression to predict Y . Thus far in this chapter, we have instead assumed that Y is qualitative. However, we may sometimes be faced with situations in which Y is neither qualitative nor quantitative, and so neither linear regression from Chapter 3 nor the classification approaches covered in this chapter is applicable.

As a concrete example, we consider the `Bikeshare` data set. The response is `bikers`, the number of hourly users of a bike sharing program in Washington, DC. This response value is neither qualitative nor quantitative: instead, it takes on non-negative integer values, or *counts*. We will consider predicting `bikers` using the covariates `mnth` (month of the year), `hr` (hour of the day, from 0 to 23), `workingday` (an indicator variable that equals 1 if it is neither a weekend nor a holiday), `temp` (the normalized temperature, in Celsius), and `weathersit` (a qualitative variable that takes on one of four possible values: clear; misty or cloudy; light rain or light snow; or heavy rain or heavy snow.)

counts

In the analyses that follow, we will treat `mnth`, `hr`, and `weathersit` as qualitative variables.

4.6.1 Linear Regression on the Bikeshare Data

To begin, we consider predicting `bikers` using linear regression. The results are shown in Table 4.10.

We see, for example, that a progression of weather from clear to cloudy results in, on average, 12.89 fewer bikers per hour; however, if the weather progresses further to rain or snow, then this further results in 53.60 fewer bikers per hour. Figure 4.13 displays the coefficients associated with `mnth`

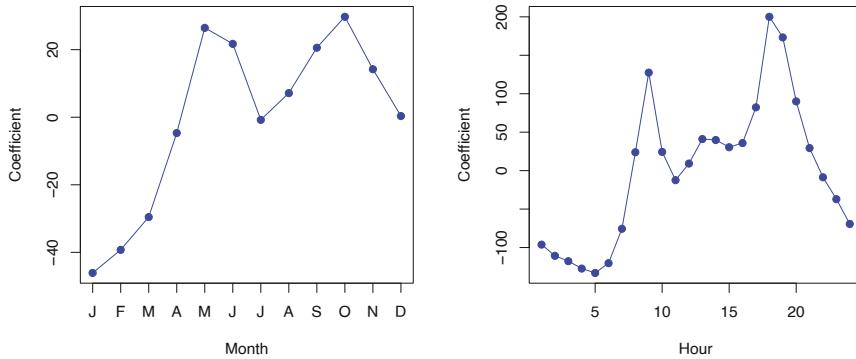


FIGURE 4.13. A least squares linear regression model was fit to predict `bikers` in the `Bikeshare` data set. Left: The coefficients associated with the month of the year. Bike usage is highest in the spring and fall, and lowest in the winter. Right: The coefficients associated with the hour of the day. Bike usage is highest during peak commute times, and lowest overnight.

and the coefficients associated with `hr`. We see that bike usage is highest in the spring and fall, and lowest during the winter months. Furthermore, bike usage is greatest around rush hour (9 AM and 6 PM), and lowest overnight. Thus, at first glance, fitting a linear regression model to the `Bikeshare` data set seems to provide reasonable and intuitive results.

But upon more careful inspection, some issues become apparent. For example, 9.6% of the fitted values in the `Bikeshare` data set are negative: that is, the linear regression model predicts a *negative* number of users during 9.6% of the hours in the data set. This calls into question our ability to perform meaningful predictions on the data, and it also raises concerns about the accuracy of the coefficient estimates, confidence intervals, and other outputs of the regression model.

Furthermore, it is reasonable to suspect that when the expected value of `bikers` is small, the variance of `bikers` should be small as well. For instance, at 2 AM during a heavy December snow storm, we expect that extremely few people will use a bike, and moreover that there should be little variance associated with the number of users during those conditions. This is borne out in the data: between 1 AM and 4 AM, in December, January, and February, when it is raining, there are 5.05 users, on average, with a standard deviation of 3.73. By contrast, between 7 AM and 10 AM, in April, May, and June, when skies are clear, there are 243.59 users, on average, with a standard deviation of 131.7. The mean-variance relationship is displayed in the left-hand panel of Figure 4.14. This is a major violation of the assumptions of a linear model, which state that $Y = \sum_{j=1}^p X_j \beta_j + \epsilon$, where ϵ is a mean-zero error term with variance σ^2 that is *constant*, and not a function of the covariates. Therefore, the heteroscedasticity of the data calls into question the suitability of a linear regression model.

Finally, the response `bikers` is integer-valued. But under a linear model, $Y = \beta_0 + \sum_{j=1}^p X_j \beta_j + \epsilon$, where ϵ is a continuous-valued error term. This means that in a linear model, the response Y is necessarily continuous-valued (quantitative). Thus, the integer nature of the response `bikers` suggests that a linear regression model is not entirely satisfactory for this data set.

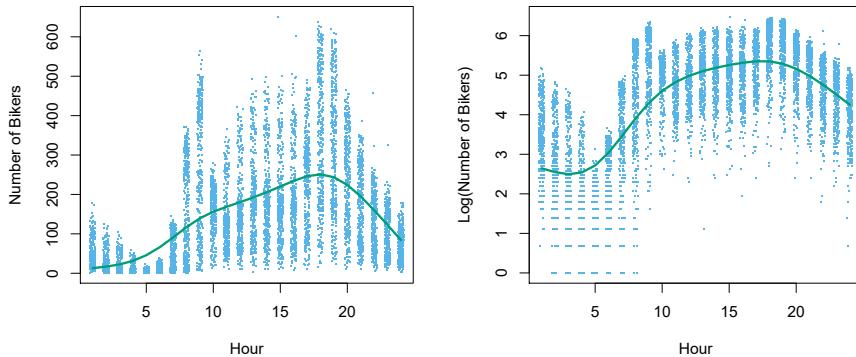


FIGURE 4.14. Left: On the `Bikeshare` dataset, the number of bikers is displayed on the y -axis, and the hour of the day is displayed on the x -axis. Jitter was applied for ease of visualization. For the most part, as the mean number of bikers increases, so does the variance in the number of bikers. A smoothing spline fit is shown in green. Right: The log of the number of bikers is now displayed on the y -axis.

Some of the problems that arise when fitting a linear regression model to the `Bikeshare` data can be overcome by transforming the response; for instance, we can fit the model

$$\log(Y) = \sum_{j=1}^p X_j \beta_j + \epsilon.$$

Transforming the response avoids the possibility of negative predictions, and it overcomes much of the heteroscedasticity in the untransformed data, as is shown in the right-hand panel of Figure 4.14. However, it is not quite a satisfactory solution, since predictions and inference are made in terms of the log of the response, rather than the response. This leads to challenges in interpretation, e.g. “*a one-unit increase in X_j is associated with an increase in the mean of the log of Y by an amount β_j* ”. Furthermore, a log transformation of the response cannot be applied in settings where the response can take on a value of 0. Thus, while fitting a linear model to a transformation of the response may be an adequate approach for some count-valued data sets, it often leaves something to be desired. We will see in the next section that a Poisson regression model provides a much more natural and elegant approach for this task.

4.6.2 Poisson Regression on the Bikeshare Data

To overcome the inadequacies of linear regression for analyzing the `Bikeshare` data set, we will make use of an alternative approach, called *Poisson regression*. Before we can talk about Poisson regression, we must first introduce the *Poisson distribution*.

Suppose that a random variable Y takes on nonnegative integer values, i.e. $Y \in \{0, 1, 2, \dots\}$. If Y follows the Poisson distribution, then

$$\Pr(Y = k) = \frac{e^{-\lambda} \lambda^k}{k!} \quad \text{for } k = 0, 1, 2, \dots \quad (4.35)$$



Poisson regression
Poisson distribution

Here, $\lambda > 0$ is the expected value of Y , i.e. $E(Y)$. It turns out that λ also equals the variance of Y , i.e. $\lambda = E(Y) = \text{Var}(Y)$. This means that if Y follows the Poisson distribution, then the larger the mean of Y , the larger its variance. (In (4.35), the notation $k!$, pronounced “ k factorial”, is defined as $k! = k \times (k - 1) \times (k - 2) \times \dots \times 3 \times 2 \times 1$.)

The Poisson distribution is typically used to model *counts*; this is a natural choice for a number of reasons, including the fact that counts, like the Poisson distribution, take on nonnegative integer values. To see how we might use the Poisson distribution in practice, let Y denote the number of users of the bike sharing program during a particular hour of the day, under a particular set of weather conditions, and during a particular month of the year. We might model Y as a Poisson distribution with mean $E(Y) = \lambda = 5$. This means that the probability of no users during this particular hour is $\Pr(Y = 0) = \frac{e^{-5}5^0}{0!} = e^{-5} = 0.0067$ (where $0! = 1$ by convention). The probability that there is exactly one user is $\Pr(Y = 1) = \frac{e^{-5}5^1}{1!} = 5e^{-5} = 0.034$, the probability of two users is $\Pr(Y = 2) = \frac{e^{-5}5^2}{2!} = 0.084$, and so on.

Of course, in reality, we expect the mean number of users of the bike sharing program, $\lambda = E(Y)$, to vary as a function of the hour of the day, the month of the year, the weather conditions, and so forth. So rather than modeling the number of bikers, Y , as a Poisson distribution with a fixed mean value like $\lambda = 5$, we would like to allow the mean to vary as a function of the covariates. In particular, we consider the following model for the mean $\lambda = E(Y)$, which we now write as $\lambda(X_1, \dots, X_p)$ to emphasize that it is a function of the covariates X_1, \dots, X_p :

$$\log(\lambda(X_1, \dots, X_p)) = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p \quad (4.36)$$

or equivalently

$$\lambda(X_1, \dots, X_p) = e^{\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p}. \quad (4.37)$$

Here, $\beta_0, \beta_1, \dots, \beta_p$ are parameters to be estimated. Together, (4.35) and (4.36) define the Poisson regression model. Notice that in (4.36), we take the *log* of $\lambda(X_1, \dots, X_p)$ to be linear in X_1, \dots, X_p , rather than having $\lambda(X_1, \dots, X_p)$ itself be linear in X_1, \dots, X_p ; this ensures that $\lambda(X_1, \dots, X_p)$ takes on nonnegative values for all values of the covariates.

To estimate the coefficients $\beta_0, \beta_1, \dots, \beta_p$, we use the same maximum likelihood approach that we adopted for logistic regression in Section 4.3.2. Specifically, given n independent observations from the Poisson regression model, the likelihood takes the form

$$\ell(\beta_0, \beta_1, \dots, \beta_p) = \prod_{i=1}^n \frac{e^{-\lambda(x_i)} \lambda(x_i)^{y_i}}{y_i!}, \quad (4.38)$$

where $\lambda(x_i) = e^{\beta_0 + \beta_1 x_{i1} + \dots + \beta_p x_{ip}}$, due to (4.37). We estimate the coefficients that maximize the likelihood $\ell(\beta_0, \beta_1, \dots, \beta_p)$, i.e. that make the observed data as likely as possible.

We now fit a Poisson regression model to the **Bikeshare** data set. The results are shown in Table 4.11 and Figure 4.15. Qualitatively, the results are similar to those from linear regression in Section 4.6.1. We again see that bike usage is highest in the spring and fall and during rush hour,

	Coefficient	Std. error	z-statistic	p-value
Intercept	4.12	0.01	683.96	0.00
workingday	0.01	0.00	7.5	0.00
temp	0.79	0.01	68.43	0.00
weathersit[cloudy/misty]	-0.08	0.00	-34.53	0.00
weathersit[light rain/snow]	-0.58	0.00	-141.91	0.00
weathersit[heavy rain/snow]	-0.93	0.17	-5.55	0.00

TABLE 4.11. Results for a Poisson regression model fit to predict `bikers` in the `Bikeshare` data. The predictors `mnth` and `hr` are omitted from this table due to space constraints, and can be seen in Figure 4.15. For the qualitative variable `weathersit`, the baseline corresponds to clear skies.

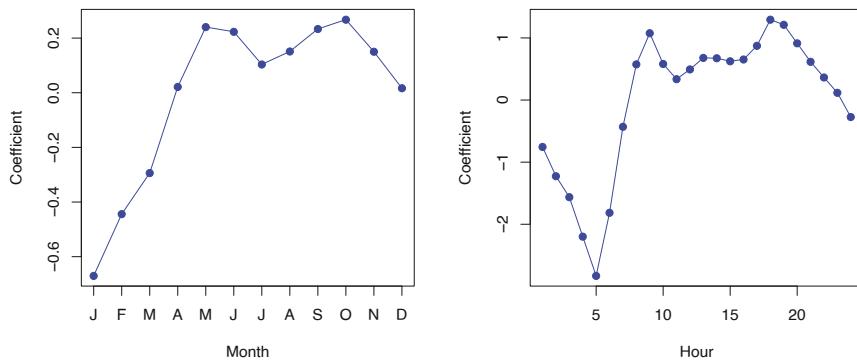


FIGURE 4.15. A Poisson regression model was fit to predict `bikers` in the `Bikeshare` data set. Left: The coefficients associated with the month of the year. Bike usage is highest in the spring and fall, and lowest in the winter. Right: The coefficients associated with the hour of the day. Bike usage is highest during peak commute times, and lowest overnight.

and lowest during the winter and in the early morning hours. Moreover, bike usage increases as the temperature increases, and decreases as the weather worsens. Interestingly, the coefficient associated with `workingday` is statistically significant under the Poisson regression model, but not under the linear regression model.

Some important distinctions between the Poisson regression model and the linear regression model are as follows:

- *Interpretation:* To interpret the coefficients in the Poisson regression model, we must pay close attention to (4.37), which states that an increase in X_j by one unit is associated with a change in $E(Y) = \lambda$ by a factor of $\exp(\beta_j)$. For example, a change in weather from clear to cloudy skies is associated with a change in mean bike usage by a factor of $\exp(-0.08) = 0.923$, i.e. on average, only 92.3% as many people will use bikes when it is cloudy relative to when it is clear. If the weather worsens further and it begins to rain, then the mean bike usage will further change by a factor of $\exp(-0.5) = 0.607$, i.e. on average only 60.7% as many people will use bikes when it is rainy relative to when it is cloudy.

- *Mean-variance relationship:* As mentioned earlier, under the Poisson model, $\lambda = E(Y) = \text{Var}(Y)$. Thus, by modeling bike usage with a Poisson regression, we implicitly assume that mean bike usage in a given hour equals the variance of bike usage during that hour. By contrast, under a linear regression model, the variance of bike usage always takes on a constant value. Recall from Figure 4.14 that in the **Bikeshare** data, when biking conditions are favorable, both the mean and the variance in bike usage are much higher than when conditions are unfavorable. Thus, the Poisson regression model is able to handle the mean-variance relationship seen in the **Bikeshare** data in a way that the linear regression model is not.⁵

overdispersion

- *nonnegative fitted values:* There are no negative predictions using the Poisson regression model. This is because the Poisson model itself only allows for nonnegative values; see (4.35). By contrast, when we fit a linear regression model to the **Bikeshare** data set, almost 10% of the predictions were negative.

4.6.3 Generalized Linear Models in Greater Generality

We have now discussed three types of regression models: linear, logistic and Poisson. These approaches share some common characteristics:



1. Each approach uses predictors X_1, \dots, X_p to predict a response Y . We assume that, conditional on X_1, \dots, X_p , Y belongs to a certain family of distributions. For linear regression, we typically assume that Y follows a Gaussian or normal distribution. For logistic regression, we assume that Y follows a Bernoulli distribution. Finally, for Poisson regression, we assume that Y follows a Poisson distribution.
2. Each approach models the mean of Y as a function of the predictors. In linear regression, the mean of Y takes the form

$$E(Y|X_1, \dots, X_p) = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p, \quad (4.39)$$

i.e. it is a linear function of the predictors. For logistic regression, the mean instead takes the form

$$\begin{aligned} E(Y|X_1, \dots, X_p) &= \Pr(Y = 1|X_1, \dots, X_p) \\ &= \frac{e^{\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p}}{1 + e^{\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p}}, \end{aligned} \quad (4.40)$$

while for Poisson regression it takes the form

$$E(Y|X_1, \dots, X_p) = \lambda(X_1, \dots, X_p) = e^{\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p}. \quad (4.41)$$

Equations (4.39)–(4.41) can be expressed using a *link function*, η , which

link function

⁵In fact, the variance in the **Bikeshare** data appears to be much higher than the mean, a situation referred to as *overdispersion*. This causes the Z-values to be inflated in Table 4.11. A more careful analysis should account for this overdispersion to obtain more accurate Z-values, and there are a variety of methods for doing this. But they are beyond the scope of this book.

applies a transformation to $E(Y|X_1, \dots, X_p)$ so that the transformed mean is a linear function of the predictors. That is,

$$\eta(E(Y|X_1, \dots, X_p)) = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p. \quad (4.42)$$

The link functions for linear, logistic and Poisson regression are $\eta(\mu) = \mu$, $\eta(\mu) = \log(\mu/(1 - \mu))$, and $\eta(\mu) = \log(\mu)$, respectively.

The Gaussian, Bernoulli and Poisson distributions are all members of a wider class of distributions, known as the *exponential family*. Other well-known members of this family are the *exponential* distribution, the *Gamma* distribution, and the *negative binomial* distribution. In general, we can perform a regression by modeling the response Y as coming from a particular member of the exponential family, and then transforming the mean of the response so that the transformed mean is a linear function of the predictors via (4.42). Any regression approach that follows this very general recipe is known as a *generalized linear model* (GLM). Thus, linear regression, logistic regression, and Poisson regression are three examples of GLMs. Other examples not covered here include *Gamma regression* and *negative binomial regression*.

exponential family
exponential
Gamma
negative binomial
generalized linear model

4.7 Lab: Logistic Regression, LDA, QDA, and KNN

4.7.1 The Stock Market Data

In this lab we will examine the `Smarket` data, which is part of the `ISLP` library. This data set consists of percentage returns for the S&P 500 stock index over 1,250 days, from the beginning of 2001 until the end of 2005. For each date, we have recorded the percentage returns for each of the five previous trading days, `Lag1` through `Lag5`. We have also recorded `Volume` (the number of shares traded on the previous day, in billions), `Today` (the percentage return on the date in question) and `Direction` (whether the market was `Up` or `Down` on this date).

We start by importing our libraries at this top level; these are all imports we have seen in previous labs.

```
In [1]: import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
import statsmodels.api as sm
from ISLP import load_data
from ISLP.models import (ModelSpec as MS,
                         summarize)
```

We also collect together the new imports needed for this lab.

```
In [2]: from ISLP import confusion_table
from ISLP.models import contrast
from sklearn.discriminant_analysis import \
    (LinearDiscriminantAnalysis as LDA,
     QuadraticDiscriminantAnalysis as QDA)
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
```

6



Linear Model Selection and Regularization

In the regression setting, the standard linear model

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p + \epsilon \quad (6.1)$$

is commonly used to describe the relationship between a response Y and a set of variables X_1, X_2, \dots, X_p . We have seen in Chapter 3 that one typically fits this model using least squares.

In the chapters that follow, we consider some approaches for extending the linear model framework. In Chapter 7 we generalize (6.1) in order to accommodate non-linear, but still additive, relationships, while in Chapters 8 and 10 we consider even more general non-linear models. However, the linear model has distinct advantages in terms of inference and, on real-world problems, is often surprisingly competitive in relation to non-linear methods. Hence, before moving to the non-linear world, we discuss in this chapter some ways in which the simple linear model can be improved, by replacing plain least squares fitting with some alternative fitting procedures.

Why might we want to use another fitting procedure instead of least squares? As we will see, alternative fitting procedures can yield better *prediction accuracy* and *model interpretability*.

- *Prediction Accuracy:* Provided that the true relationship between the response and the predictors is approximately linear, the least squares estimates will have low bias. If $n \gg p$ —that is, if n , the number of observations, is much larger than p , the number of variables—then the least squares estimates tend to also have low variance, and hence will perform well on test observations. However, if n is not much larger than p , then there can be a lot of variability in the least squares fit, resulting in overfitting and consequently poor predictions on future observations not used in model training. And if $p > n$, then there is no longer a unique least squares coefficient estimate: there are infinitely

many solutions. Each of these least squares solutions gives zero error on the training data, but typically very poor test set performance due to extremely high variance.¹ By *constraining* or *shrinking* the estimated coefficients, we can often substantially reduce the variance at the cost of a negligible increase in bias. This can lead to substantial improvements in the accuracy with which we can predict the response for observations not used in model training.

- *Model Interpretability:* It is often the case that some or many of the variables used in a multiple regression model are in fact not associated with the response. Including such *irrelevant* variables leads to unnecessary complexity in the resulting model. By removing these variables—that is, by setting the corresponding coefficient estimates to zero—we can obtain a model that is more easily interpreted. Now least squares is extremely unlikely to yield any coefficient estimates that are exactly zero. In this chapter, we see some approaches for automatically performing *feature selection* or *variable selection*—that is, for excluding irrelevant variables from a multiple regression model.

feature
selection
variable
selection

There are many alternatives, both classical and modern, to using least squares to fit (6.1). In this chapter, we discuss three important classes of methods.

- *Subset Selection.* This approach involves identifying a subset of the p predictors that we believe to be related to the response. We then fit a model using least squares on the reduced set of variables.
- *Shrinkage.* This approach involves fitting a model involving all p predictors. However, the estimated coefficients are shrunk towards zero relative to the least squares estimates. This shrinkage (also known as *regularization*) has the effect of reducing variance. Depending on what type of shrinkage is performed, some of the coefficients may be estimated to be exactly zero. Hence, shrinkage methods can also perform variable selection.
- *Dimension Reduction.* This approach involves *projecting* the p predictors into an M -dimensional subspace, where $M < p$. This is achieved by computing M different *linear combinations*, or *projections*, of the variables. Then these M projections are used as predictors to fit a linear regression model by least squares.

In the following sections we describe each of these approaches in greater detail, along with their advantages and disadvantages. Although this chapter describes extensions and modifications to the linear model for regression seen in Chapter 3, the same concepts apply to other methods, such as the classification models seen in Chapter 4.

¹When $p \gg n$, the least squares solution that has the smallest sum of squared coefficients can sometimes perform quite well. See Section 10.8 for a more detailed discussion.

6.1 Subset Selection

In this section we consider some methods for selecting subsets of predictors. These include best subset and stepwise model selection procedures.

6.1.1 Best Subset Selection

To perform *best subset selection*, we fit a separate least squares regression for each possible combination of the p predictors. That is, we fit all p models that contain exactly one predictor, all $\binom{p}{2} = p(p-1)/2$ models that contain exactly two predictors, and so forth. We then look at all of the resulting models, with the goal of identifying the one that is *best*.

best subset selection

The problem of selecting the *best model* from among the 2^p possibilities considered by best subset selection is not trivial. This is usually broken up into two stages, as described in Algorithm 6.1.

Algorithm 6.1 Best subset selection

1. Let \mathcal{M}_0 denote the *null model*, which contains no predictors. This model simply predicts the sample mean for each observation.
 2. For $k = 1, 2, \dots, p$:
 - (a) Fit all $\binom{p}{k}$ models that contain exactly k predictors.
 - (b) Pick the best among these $\binom{p}{k}$ models, and call it \mathcal{M}_k . Here *best* is defined as having the smallest RSS, or equivalently largest R^2 .
 3. Select a single best model from among $\mathcal{M}_0, \dots, \mathcal{M}_p$ using the prediction error on a validation set, C_p (AIC), BIC, or adjusted R^2 . Or use the cross-validation method.
-

In Algorithm 6.1, Step 2 identifies the best model (on the training data) for each subset size, in order to reduce the problem from one of 2^p possible models to one of $p + 1$ possible models. In Figure 6.1, these models form the lower frontier depicted in red.

Now in order to select a single best model, we must simply choose among these $p + 1$ options. This task must be performed with care, because the RSS of these $p + 1$ models decreases monotonically, and the R^2 increases monotonically, as the number of features included in the models increases. Therefore, if we use these statistics to select the best model, then we will always end up with a model involving all of the variables. The problem is that a low RSS or a high R^2 indicates a model with a low *training* error, whereas we wish to choose a model that has a low *test* error. (As shown in Chapter 2 in Figures 2.9–2.11, training error tends to be quite a bit smaller than test error, and a low training error by no means guarantees a low test error.) Therefore, in Step 3, we use the error on a validation set, C_p , BIC, or adjusted R^2 in order to select among $\mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_p$. If cross-validation is used to select the best model, then Step 2 is repeated on each training fold, and the validation errors are averaged to select the best value of k .

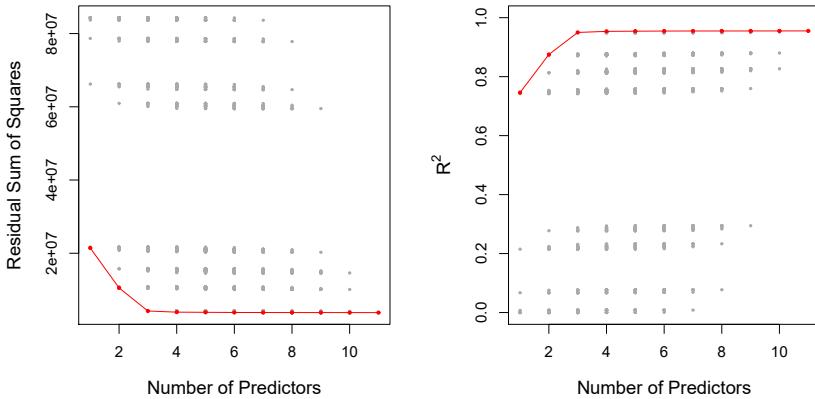


FIGURE 6.1. For each possible model containing a subset of the ten predictors in the `Credit` data set, the RSS and R^2 are displayed. The red frontier tracks the best model for a given number of predictors, according to RSS and R^2 . Though the data set contains only ten predictors, the x-axis ranges from 1 to 11, since one of the variables is categorical and takes on three values, leading to the creation of two dummy variables.

Then the model \mathcal{M}_k fit on the full training set is delivered for the chosen k . These approaches are discussed in Section 6.1.3.

An application of best subset selection is shown in Figure 6.1. Each plotted point corresponds to a least squares regression model fit using a different subset of the 10 predictors in the `Credit` data set, discussed in Chapter 3. Here the variable `region` is a three-level qualitative variable, and so is represented by two dummy variables, which are selected separately in this case. Hence, there are a total of 11 possible variables which can be included in the model. We have plotted the RSS and R^2 statistics for each model, as a function of the number of variables. The red curves connect the best models for each model size, according to RSS or R^2 . The figure shows that, as expected, these quantities improve as the number of variables increases; however, from the three-variable model on, there is little improvement in RSS and R^2 as a result of including additional predictors.

Although we have presented best subset selection here for least squares regression, the same ideas apply to other types of models, such as logistic regression. In the case of logistic regression, instead of ordering models by RSS in Step 2 of Algorithm 6.1, we instead use the *deviance*, a measure that plays the role of RSS for a broader class of models. The deviance is negative two times the maximized log-likelihood; the smaller the deviance, the better the fit.

While best subset selection is a simple and conceptually appealing approach, it suffers from computational limitations. The number of possible models that must be considered grows rapidly as p increases. In general, there are 2^p models that involve subsets of p predictors. So if $p = 10$, then there are approximately 1,000 possible models to be considered, and if $p = 20$, then there are over one million possibilities! Consequently, best subset selection becomes computationally infeasible for values of p greater than

deviance

Algorithm 6.2 *Forward stepwise selection*

1. Let \mathcal{M}_0 denote the *null* model, which contains no predictors.
 2. For $k = 0, \dots, p - 1$:
 - (a) Consider all $p - k$ models that augment the predictors in \mathcal{M}_k with one additional predictor.
 - (b) Choose the *best* among these $p - k$ models, and call it \mathcal{M}_{k+1} . Here *best* is defined as having smallest RSS or highest R^2 .
 3. Select a single best model from among $\mathcal{M}_0, \dots, \mathcal{M}_p$ using the prediction error on a validation set, C_p (AIC), BIC, or adjusted R^2 . Or use the cross-validation method.
-

around 40, even with extremely fast modern computers. There are computational shortcuts—so called branch-and-bound techniques—for eliminating some choices, but these have their limitations as p gets large. They also only work for least squares linear regression. We present computationally efficient alternatives to best subset selection next.

6.1.2 Stepwise Selection

For computational reasons, best subset selection cannot be applied with very large p . Best subset selection may also suffer from statistical problems when p is large. The larger the search space, the higher the chance of finding models that look good on the training data, even though they might not have any predictive power on future data. Thus an enormous search space can lead to overfitting and high variance of the coefficient estimates.

For both of these reasons, *stepwise* methods, which explore a far more restricted set of models, are attractive alternatives to best subset selection.

Forward Stepwise Selection

Forward stepwise selection is a computationally efficient alternative to best subset selection. While the best subset selection procedure considers all 2^p possible models containing subsets of the p predictors, forward stepwise considers a much smaller set of models. Forward stepwise selection begins with a model containing no predictors, and then adds predictors to the model, one-at-a-time, until all of the predictors are in the model. In particular, at each step the variable that gives the greatest *additional* improvement to the fit is added to the model. More formally, the forward stepwise selection procedure is given in Algorithm 6.2.

forward
stepwise
selection

Unlike best subset selection, which involved fitting 2^p models, forward stepwise selection involves fitting one null model, along with $p - k$ models in the k th iteration, for $k = 0, \dots, p - 1$. This amounts to a total of $1 + \sum_{k=0}^{p-1} (p - k) = 1 + p(p + 1)/2$ models. This is a substantial difference: when

# Variables	Best subset	Forward stepwise
One	<code>rating</code>	<code>rating</code>
Two	<code>rating, income</code>	<code>rating, income</code>
Three	<code>rating, income, student</code>	<code>rating, income, student</code>
Four	<code>cards, income</code> <code>student, limit</code>	<code>rating, income,</code> <code>student, limit</code>

TABLE 6.1. The first four selected models for best subset selection and forward stepwise selection on the `Credit` data set. The first three models are identical but the fourth models differ.

$p = 20$, best subset selection requires fitting 1,048,576 models, whereas forward stepwise selection requires fitting only 211 models.²

In Step 2(b) of Algorithm 6.2, we must identify the *best* model from among those $p - k$ that augment \mathcal{M}_k with one additional predictor. We can do this by simply choosing the model with the lowest RSS or the highest R^2 . However, in Step 3, we must identify the best model among a set of models with different numbers of variables. This is more challenging, and is discussed in Section 6.1.3.

Forward stepwise selection's computational advantage over best subset selection is clear. Though forward stepwise tends to do well in practice, it is not guaranteed to find the best possible model out of all 2^p models containing subsets of the p predictors. For instance, suppose that in a given data set with $p = 3$ predictors, the best possible one-variable model contains X_1 , and the best possible two-variable model instead contains X_2 and X_3 . Then forward stepwise selection will fail to select the best possible two-variable model, because \mathcal{M}_1 will contain X_1 , so \mathcal{M}_2 must also contain X_1 together with one additional variable.

Table 6.1, which shows the first four selected models for best subset and forward stepwise selection on the `Credit` data set, illustrates this phenomenon. Both best subset selection and forward stepwise selection choose `rating` for the best one-variable model and then include `income` and `student` for the two- and three-variable models. However, best subset selection replaces `rating` by `cards` in the four-variable model, while forward stepwise selection must maintain `rating` in its four-variable model. In this example, Figure 6.1 indicates that there is not much difference between the three- and four-variable models in terms of RSS, so either of the four-variable models will likely be adequate.

Forward stepwise selection can be applied even in the high-dimensional setting where $n < p$; however, in this case, it is possible to construct submodels $\mathcal{M}_0, \dots, \mathcal{M}_{n-1}$ only, since each submodel is fit using least squares, which will not yield a unique solution if $p \geq n$.

Backward Stepwise Selection

Like forward stepwise selection, *backward stepwise selection* provides an

backward
stepwise
selection

²Though forward stepwise selection considers $p(p + 1)/2 + 1$ models, it performs a *guided* search over model space, and so the *effective* model space considered contains substantially more than $p(p + 1)/2 + 1$ models.

efficient alternative to best subset selection. However, unlike forward stepwise selection, it begins with the full least squares model containing all p predictors, and then iteratively removes the least useful predictor, one-at-a-time. Details are given in Algorithm 6.3.

Algorithm 6.3 Backward stepwise selection

1. Let \mathcal{M}_p denote the *full* model, which contains all p predictors.
 2. For $k = p, p - 1, \dots, 1$:
 - (a) Consider all k models that contain all but one of the predictors in \mathcal{M}_k , for a total of $k - 1$ predictors.
 - (b) Choose the *best* among these k models, and call it \mathcal{M}_{k-1} . Here *best* is defined as having smallest RSS or highest R^2 .
 3. Select a single best model from among $\mathcal{M}_0, \dots, \mathcal{M}_p$ using the prediction error on a validation set, C_p (AIC), BIC, or adjusted R^2 . Or use the cross-validation method.
-

Like forward stepwise selection, the backward selection approach searches through only $1 + p(p+1)/2$ models, and so can be applied in settings where p is too large to apply best subset selection.³ Also like forward stepwise selection, backward stepwise selection is not guaranteed to yield the *best* model containing a subset of the p predictors.

Backward selection requires that the number of samples n is larger than the number of variables p (so that the full model can be fit). In contrast, forward stepwise can be used even when $n < p$, and so is the only viable subset method when p is very large.

Hybrid Approaches

The best subset, forward stepwise, and backward stepwise selection approaches generally give similar but not identical models. As another alternative, hybrid versions of forward and backward stepwise selection are available, in which variables are added to the model sequentially, in analogy to forward selection. However, after adding each new variable, the method may also remove any variables that no longer provide an improvement in the model fit. Such an approach attempts to more closely mimic best subset selection while retaining the computational advantages of forward and backward stepwise selection.

6.1.3 Choosing the Optimal Model

Best subset selection, forward selection, and backward selection result in the creation of a set of models, each of which contains a subset of the p

³Like forward stepwise selection, backward stepwise selection performs a *guided* search over model space, and so effectively considers substantially more than $1 + p(p+1)/2$ models.

predictors. To apply these methods, we need a way to determine which of these models is *best*. As we discussed in Section 6.1.1, the model containing all of the predictors will always have the smallest RSS and the largest R^2 , since these quantities are related to the training error. Instead, we wish to choose a model with a low test error. As is evident here, and as we show in Chapter 2, the training error can be a poor estimate of the test error. Therefore, RSS and R^2 are not suitable for selecting the best model among a collection of models with different numbers of predictors.

In order to select the best model with respect to test error, we need to estimate this test error. There are two common approaches:

1. We can indirectly estimate test error by making an *adjustment* to the training error to account for the bias due to overfitting.
2. We can *directly* estimate the test error, using either a validation set approach or a cross-validation approach, as discussed in Chapter 5.

We consider both of these approaches below.

C_p , AIC, BIC, and Adjusted R^2

We show in Chapter 2 that the training set MSE is generally an underestimate of the test MSE. (Recall that $\text{MSE} = \text{RSS}/n$.) This is because when we fit a model to the training data using least squares, we specifically estimate the regression coefficients such that the training RSS (but not the test RSS) is as small as possible. In particular, the training error will decrease as more variables are included in the model, but the test error may not. Therefore, training set RSS and training set R^2 cannot be used to select from among a set of models with different numbers of variables.

However, a number of techniques for *adjusting* the training error for the model size are available. These approaches can be used to select among a set of models with different numbers of variables. We now consider four such approaches: C_p , Akaike information criterion (AIC), Bayesian information criterion (BIC), and adjusted R^2 . Figure 6.2 displays C_p , BIC, and adjusted R^2 for the best model of each size produced by best subset selection on the **Credit** data set.

For a fitted least squares model containing d predictors, the C_p estimate of test MSE is computed using the equation

$$C_p = \frac{1}{n} (\text{RSS} + 2d\hat{\sigma}^2), \quad (6.2)$$

C_p	Akaike information criterion
BIC	Bayesian information criterion
adjusted R^2	

where $\hat{\sigma}^2$ is an estimate of the variance of the error ϵ associated with each response measurement in (6.1).⁴ Typically $\hat{\sigma}^2$ is estimated using the full model containing all predictors. Essentially, the C_p statistic adds a penalty of $2d\hat{\sigma}^2$ to the training RSS in order to adjust for the fact that the training error tends to underestimate the test error. Clearly, the penalty increases as the number of predictors in the model increases; this is intended to adjust

⁴Mallow's C_p is sometimes defined as $C'_p = \text{RSS}/\hat{\sigma}^2 + 2d - n$. This is equivalent to the definition given above in the sense that $C_p = \frac{1}{n}\hat{\sigma}^2(C'_p + n)$, and so the model with smallest C_p also has smallest C'_p .

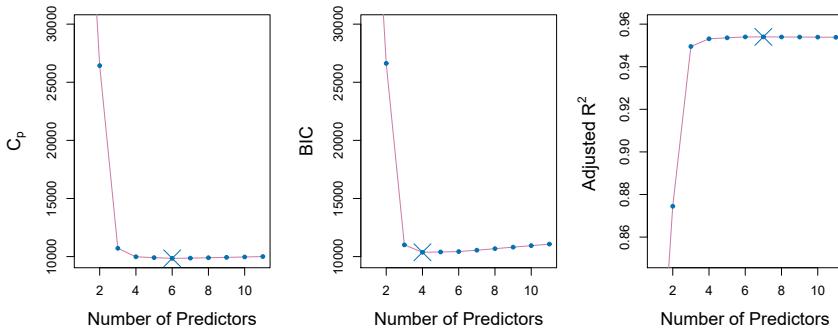


FIGURE 6.2. C_p , BIC, and adjusted R^2 are shown for the best models of each size for the **Credit** data set (the lower frontier in Figure 6.1). C_p and BIC are estimates of test MSE. In the middle plot we see that the BIC estimate of test error shows an increase after four variables are selected. The other two plots are rather flat after four variables are included.

for the corresponding decrease in training RSS. Though it is beyond the scope of this book, one can show that if $\hat{\sigma}^2$ is an unbiased estimate of σ^2 in (6.2), then C_p is an unbiased estimate of test MSE. As a consequence, the C_p statistic tends to take on a small value for models with a low test error, so when determining which of a set of models is best, we choose the model with the lowest C_p value. In Figure 6.2, C_p selects the six-variable model containing the predictors `income`, `limit`, `rating`, `cards`, `age` and `student`.

The AIC criterion is defined for a large class of models fit by maximum likelihood. In the case of the model (6.1) with Gaussian errors, maximum likelihood and least squares are the same thing. In this case AIC is given by

$$\text{AIC} = \frac{1}{n} (\text{RSS} + 2d\hat{\sigma}^2),$$

where, for simplicity, we have omitted irrelevant constants.⁵ Hence for least squares models, C_p and AIC are proportional to each other, and so only C_p is displayed in Figure 6.2.

BIC is derived from a Bayesian point of view, but ends up looking similar to C_p (and AIC) as well. For the least squares model with d predictors, the BIC is, up to irrelevant constants, given by

$$\text{BIC} = \frac{1}{n} (\text{RSS} + \log(n)d\hat{\sigma}^2). \quad (6.3)$$

Like C_p , the BIC will tend to take on a small value for a model with a low test error, and so generally we select the model that has the lowest BIC value. Notice that BIC replaces the $2d\hat{\sigma}^2$ used by C_p with a $\log(n)d\hat{\sigma}^2$ term, where n is the number of observations. Since $\log n > 2$ for any $n > 7$,

⁵There are two formulas for AIC for least squares regression. The formula that we provide here requires an expression for σ^2 , which we obtain using the full model containing all predictors. The second formula is appropriate when σ^2 is unknown and we do not want to explicitly estimate it; that formula has a $\log(\text{RSS})$ term instead of an RSS term. Detailed derivations of these two formulas are outside of the scope of this book.

the BIC statistic generally places a heavier penalty on models with many variables, and hence results in the selection of smaller models than C_p . In Figure 6.2, we see that this is indeed the case for the `Credit` data set; BIC chooses a model that contains only the four predictors `income`, `limit`, `cards`, and `student`. In this case the curves are very flat and so there does not appear to be much difference in accuracy between the four-variable and six-variable models.

The adjusted R^2 statistic is another popular approach for selecting among a set of models that contain different numbers of variables. Recall from Chapter 3 that the usual R^2 is defined as $1 - \text{RSS}/\text{TSS}$, where $\text{TSS} = \sum(y_i - \bar{y})^2$ is the *total sum of squares* for the response. Since RSS always decreases as more variables are added to the model, the R^2 always increases as more variables are added. For a least squares model with d variables, the adjusted R^2 statistic is calculated as

$$\text{Adjusted } R^2 = 1 - \frac{\text{RSS}/(n - d - 1)}{\text{TSS}/(n - 1)}. \quad (6.4)$$

Unlike C_p , AIC, and BIC, for which a *small* value indicates a model with a low test error, a *large* value of adjusted R^2 indicates a model with a small test error. Maximizing the adjusted R^2 is equivalent to minimizing $\frac{\text{RSS}}{n-d-1}$. While RSS always decreases as the number of variables in the model increases, $\frac{\text{RSS}}{n-d-1}$ may increase or decrease, due to the presence of d in the denominator.

The intuition behind the adjusted R^2 is that once all of the correct variables have been included in the model, adding additional *noise* variables will lead to only a very small decrease in RSS. Since adding noise variables leads to an increase in d , such variables will lead to an increase in $\frac{\text{RSS}}{n-d-1}$, and consequently a decrease in the adjusted R^2 . Therefore, in theory, the model with the largest adjusted R^2 will have only correct variables and no noise variables. Unlike the R^2 statistic, the adjusted R^2 statistic *pays a price* for the inclusion of unnecessary variables in the model. Figure 6.2 displays the adjusted R^2 for the `Credit` data set. Using this statistic results in the selection of a model that contains seven variables, adding `own` to the model selected by C_p and AIC.

C_p , AIC, and BIC all have rigorous theoretical justifications that are beyond the scope of this book. These justifications rely on asymptotic arguments (scenarios where the sample size n is very large). Despite its popularity, and even though it is quite intuitive, the adjusted R^2 is not as well motivated in statistical theory as AIC, BIC, and C_p . All of these measures are simple to use and compute. Here we have presented their formulas in the case of a linear model fit using least squares; however, AIC and BIC can also be defined for more general types of models.

Validation and Cross-Validation

As an alternative to the approaches just discussed, we can directly estimate the test error using the validation set and cross-validation methods discussed in Chapter 5. We can compute the validation set error or the cross-validation error for each model under consideration, and then select

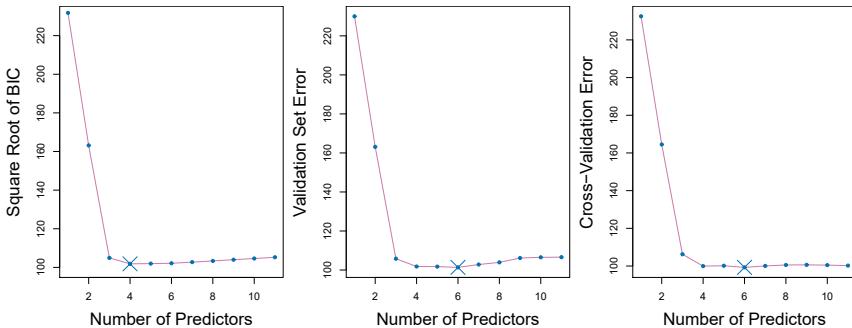


FIGURE 6.3. For the `Credit` data set, three quantities are displayed for the best model containing d predictors, for d ranging from 1 to 11. The overall best model, based on each of these quantities, is shown as a blue cross. Left: Square root of BIC. Center: Validation set errors. Right: Cross-validation errors.

the model for which the resulting estimated test error is smallest. This procedure has an advantage relative to AIC, BIC, C_p , and adjusted R^2 , in that it provides a direct estimate of the test error, and makes fewer assumptions about the true underlying model. It can also be used in a wider range of model selection tasks, even in cases where it is hard to pinpoint the model degrees of freedom (e.g. the number of predictors in the model) or hard to estimate the error variance σ^2 . Note that when cross-validation is used, the sequence of models \mathcal{M}_k in Algorithms 6.1–6.3 is determined separately for each training fold, and the validation errors are averaged over all folds for each model size k . This means, for example with best-subset regression, that \mathcal{M}_k , the best subset of size k , can differ across the folds. Once the best size k is chosen, we find the best model of that size on the full data set.

In the past, performing cross-validation was computationally prohibitive for many problems with large p and/or large n , and so AIC, BIC, C_p , and adjusted R^2 were more attractive approaches for choosing among a set of models. However, nowadays with fast computers, the computations required to perform cross-validation are hardly ever an issue. Thus, cross-validation is a very attractive approach for selecting from among a number of models under consideration.

Figure 6.3 displays, as a function of d , the BIC, validation set errors, and cross-validation errors on the `Credit` data, for the best d -variable model. The validation errors were calculated by randomly selecting three-quarters of the observations as the training set, and the remainder as the validation set. The cross-validation errors were computed using $k = 10$ folds. In this case, the validation and cross-validation methods both result in a six-variable model. However, all three approaches suggest that the four-, five-, and six-variable models are roughly equivalent in terms of their test errors.

In fact, the estimated test error curves displayed in the center and right-hand panels of Figure 6.3 are quite flat. While a three-variable model clearly has lower estimated test error than a two-variable model, the estimated test errors of the 3- to 11-variable models are quite similar. Furthermore, if we

repeated the validation set approach using a different split of the data into a training set and a validation set, or if we repeated cross-validation using a different set of cross-validation folds, then the precise model with the lowest estimated test error would surely change. In this setting, we can select a model using the *one-standard-error rule*. We first calculate the standard error of the estimated test MSE for each model size, and then select the smallest model for which the estimated test error is within one standard error of the lowest point on the curve. The rationale here is that if a set of models appear to be more or less equally good, then we might as well choose the simplest model—that is, the model with the smallest number of predictors. In this case, applying the one-standard-error rule to the validation set or cross-validation approach leads to selection of the three-variable model.

one-
standard-
error
rule

6.2 Shrinkage Methods

The subset selection methods described in Section 6.1 involve using least squares to fit a linear model that contains a subset of the predictors. As an alternative, we can fit a model containing all p predictors using a technique that *constrains* or *regularizes* the coefficient estimates, or equivalently, that *shrinks* the coefficient estimates towards zero. It may not be immediately obvious why such a constraint should improve the fit, but it turns out that shrinking the coefficient estimates can significantly reduce their variance. The two best-known techniques for shrinking the regression coefficients towards zero are *ridge regression* and the *lasso*.

6.2.1 Ridge Regression

Recall from Chapter 3 that the least squares fitting procedure estimates $\beta_0, \beta_1, \dots, \beta_p$ using the values that minimize

$$\text{RSS} = \sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2.$$

Ridge regression is very similar to least squares, except that the coefficients are estimated by minimizing a slightly different quantity. In particular, the ridge regression coefficient estimates $\hat{\beta}^R$ are the values that minimize

$$\sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^p \beta_j^2 = \text{RSS} + \lambda \sum_{j=1}^p \beta_j^2, \quad (6.5)$$

where $\lambda \geq 0$ is a *tuning parameter*, to be determined separately. Equation 6.5 trades off two different criteria. As with least squares, ridge regression seeks coefficient estimates that fit the data well, by making the RSS small. However, the second term, $\lambda \sum_j \beta_j^2$, called a *shrinkage penalty*, is small when β_1, \dots, β_p are close to zero, and so it has the effect of *shrinking* the estimates of β_j towards zero. The tuning parameter λ serves to control

ridge
regression

tuning
parameter

shrinkage
penalty

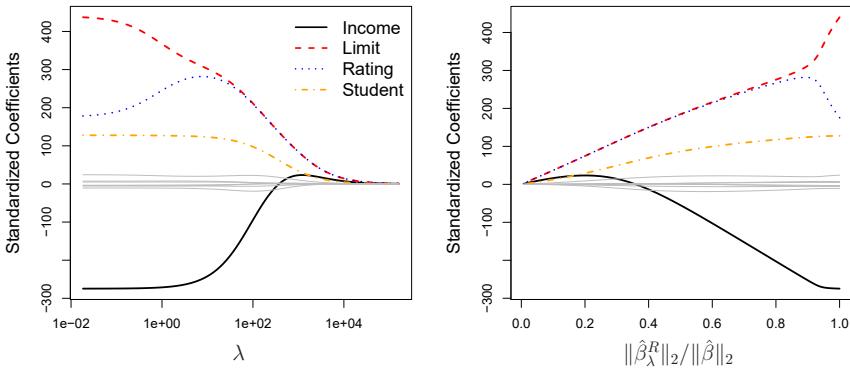


FIGURE 6.4. The standardized ridge regression coefficients are displayed for the `Credit` data set, as a function of λ and $\|\hat{\beta}_\lambda^R\|_2 / \|\hat{\beta}\|_2$.

the relative impact of these two terms on the regression coefficient estimates. When $\lambda = 0$, the penalty term has no effect, and ridge regression will produce the least squares estimates. However, as $\lambda \rightarrow \infty$, the impact of the shrinkage penalty grows, and the ridge regression coefficient estimates will approach zero. Unlike least squares, which generates only one set of coefficient estimates, ridge regression will produce a different set of coefficient estimates, $\hat{\beta}_\lambda^R$, for each value of λ . Selecting a good value for λ is critical; we defer this discussion to Section 6.2.3, where we use cross-validation.

Note that in (6.5), the shrinkage penalty is applied to β_1, \dots, β_p , but not to the intercept β_0 . We want to shrink the estimated association of each variable with the response; however, we do not want to shrink the intercept, which is simply a measure of the mean value of the response when $x_{i1} = x_{i2} = \dots = x_{ip} = 0$. If we assume that the variables—that is, the columns of the data matrix \mathbf{X} —have been centered to have mean zero before ridge regression is performed, then the estimated intercept will take the form $\hat{\beta}_0 = \bar{y} = \sum_{i=1}^n y_i/n$.

An Application to the Credit Data

In Figure 6.4, the ridge regression coefficient estimates for the `Credit` data set are displayed. In the left-hand panel, each curve corresponds to the ridge regression coefficient estimate for one of the ten variables, plotted as a function of λ . For example, the black solid line represents the ridge regression estimate for the `income` coefficient, as λ is varied. At the extreme left-hand side of the plot, λ is essentially zero, and so the corresponding ridge coefficient estimates are the same as the usual least squares estimates. But as λ increases, the ridge coefficient estimates shrink towards zero. When λ is extremely large, then all of the ridge coefficient estimates are basically zero; this corresponds to the *null model* that contains no predictors. In this plot, the `income`, `limit`, `rating`, and `student` variables are displayed in distinct colors, since these variables tend to have by far the largest coefficient estimates. While the ridge coefficient estimates tend to decrease in aggregate as λ increases, individual coefficients, such as `rating` and `income`, may occasionally increase as λ increases.

The right-hand panel of Figure 6.4 displays the same ridge coefficient estimates as the left-hand panel, but instead of displaying λ on the x -axis, we now display $\|\hat{\beta}_\lambda^R\|_2/\|\hat{\beta}\|_2$, where $\hat{\beta}$ denotes the vector of least squares coefficient estimates. The notation $\|\beta\|_2$ denotes the ℓ_2 norm (pronounced “ell 2”) of a vector, and is defined as $\|\beta\|_2 = \sqrt{\sum_{j=1}^p \beta_j^2}$. It measures the distance of β from zero. As λ increases, the ℓ_2 norm of $\hat{\beta}_\lambda^R$ will *always* decrease, and so will $\|\hat{\beta}_\lambda^R\|_2/\|\hat{\beta}\|_2$. The latter quantity ranges from 1 (when $\lambda = 0$, in which case the ridge regression coefficient estimate is the same as the least squares estimate, and so their ℓ_2 norms are the same) to 0 (when $\lambda = \infty$, in which case the ridge regression coefficient estimate is a vector of zeros, with ℓ_2 norm equal to zero). Therefore, we can think of the x -axis in the right-hand panel of Figure 6.4 as the amount that the ridge regression coefficient estimates have been shrunk towards zero; a small value indicates that they have been shrunk very close to zero.

The standard least squares coefficient estimates discussed in Chapter 3 are *scale equivariant*: multiplying X_j by a constant c simply leads to a scaling of the least squares coefficient estimates by a factor of $1/c$. In other words, regardless of how the j th predictor is scaled, $X_j \hat{\beta}_j$ will remain the same. In contrast, the ridge regression coefficient estimates can change *substantially* when multiplying a given predictor by a constant. For instance, consider the `income` variable, which is measured in dollars. One could reasonably have measured income in thousands of dollars, which would result in a reduction in the observed values of `income` by a factor of 1,000. Now due to the sum of squared coefficients term in the ridge regression formulation (6.5), such a change in scale will not simply cause the ridge regression coefficient estimate for `income` to change by a factor of 1,000. In other words, $X_j \hat{\beta}_{j,\lambda}^R$ will depend not only on the value of λ , but also on the scaling of the j th predictor. In fact, the value of $X_j \hat{\beta}_{j,\lambda}^R$ may even depend on the scaling of the *other* predictors! Therefore, it is best to apply ridge regression after *standardizing the predictors*, using the formula

$$\tilde{x}_{ij} = \frac{x_{ij}}{\sqrt{\frac{1}{n} \sum_{i=1}^n (x_{ij} - \bar{x}_j)^2}}, \quad (6.6)$$

so that they are all on the same scale. In (6.6), the denominator is the estimated standard deviation of the j th predictor. Consequently, all of the standardized predictors will have a standard deviation of one. As a result the final fit will not depend on the scale on which the predictors are measured. In Figure 6.4, the y -axis displays the standardized ridge regression coefficient estimates—that is, the coefficient estimates that result from performing ridge regression using standardized predictors.

Why Does Ridge Regression Improve Over Least Squares?

Ridge regression’s advantage over least squares is rooted in the *bias-variance trade-off*. As λ increases, the flexibility of the ridge regression fit decreases, leading to decreased variance but increased bias. This is illustrated in the left-hand panel of Figure 6.5, using a simulated data set containing $p = 45$ predictors and $n = 50$ observations. The green curve in the left-hand panel

scale
equivariant

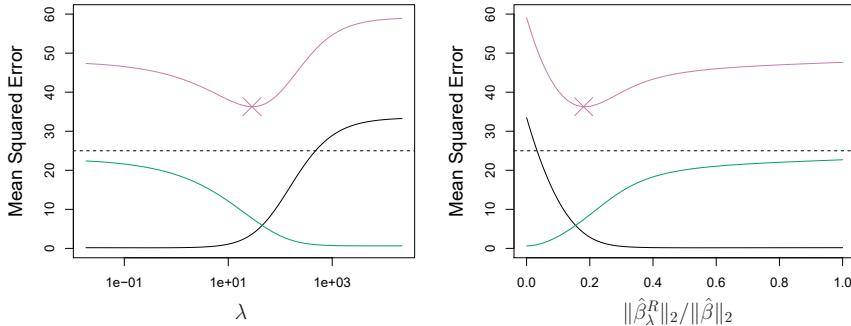


FIGURE 6.5. Squared bias (black), variance (green), and test mean squared error (purple) for the ridge regression predictions on a simulated data set, as a function of λ and $\|\hat{\beta}_\lambda^R\|_2 / \|\hat{\beta}\|_2$. The horizontal dashed lines indicate the minimum possible MSE. The purple crosses indicate the ridge regression models for which the MSE is smallest.

of Figure 6.5 displays the variance of the ridge regression predictions as a function of λ . At the least squares coefficient estimates, which correspond to ridge regression with $\lambda = 0$, the variance is high but there is no bias. But as λ increases, the shrinkage of the ridge coefficient estimates leads to a substantial reduction in the variance of the predictions, at the expense of a slight increase in bias. Recall that the test mean squared error (MSE), plotted in purple, is closely related to the variance plus the squared bias. For values of λ up to about 10, the variance decreases rapidly, with very little increase in bias, plotted in black. Consequently, the MSE drops considerably as λ increases from 0 to 10. Beyond this point, the decrease in variance due to increasing λ slows, and the shrinkage on the coefficients causes them to be significantly underestimated, resulting in a large increase in the bias. The minimum MSE is achieved at approximately $\lambda = 30$. Interestingly, because of its high variance, the MSE associated with the least squares fit, when $\lambda = 0$, is almost as high as that of the null model for which all coefficient estimates are zero, when $\lambda = \infty$. However, for an intermediate value of λ , the MSE is considerably lower.

The right-hand panel of Figure 6.5 displays the same curves as the left-hand panel, this time plotted against the ℓ_2 norm of the ridge regression coefficient estimates divided by the ℓ_2 norm of the least squares estimates. Now as we move from left to right, the fits become more flexible, and so the bias decreases and the variance increases.

In general, in situations where the relationship between the response and the predictors is close to linear, the least squares estimates will have low bias but may have high variance. This means that a small change in the training data can cause a large change in the least squares coefficient estimates. In particular, when the number of variables p is almost as large as the number of observations n , as in the example in Figure 6.5, the least squares estimates will be extremely variable. And if $p > n$, then the least squares estimates do not even have a unique solution, whereas ridge regression can still perform well by trading off a small increase in bias for a

large decrease in variance. Hence, ridge regression works best in situations where the least squares estimates have high variance.

Ridge regression also has substantial computational advantages over best subset selection, which requires searching through 2^p models. As we discussed previously, even for moderate values of p , such a search can be computationally infeasible. In contrast, for any fixed value of λ , ridge regression only fits a single model, and the model-fitting procedure can be performed quite quickly. In fact, one can show that the computations required to solve (6.5), *simultaneously for all values of λ* , are almost identical to those for fitting a model using least squares.

6.2.2 The Lasso

Ridge regression does have one obvious disadvantage. Unlike best subset, forward stepwise, and backward stepwise selection, which will generally select models that involve just a subset of the variables, ridge regression will include all p predictors in the final model. The penalty $\lambda \sum \beta_j^2$ in (6.5) will shrink all of the coefficients towards zero, but it will not set any of them exactly to zero (unless $\lambda = \infty$). This may not be a problem for prediction accuracy, but it can create a challenge in model interpretation in settings in which the number of variables p is quite large. For example, in the `Credit` data set, it appears that the most important variables are `income`, `limit`, `rating`, and `student`. So we might wish to build a model including just these predictors. However, ridge regression will always generate a model involving all ten predictors. Increasing the value of λ will tend to reduce the magnitudes of the coefficients, but will not result in exclusion of any of the variables.

The *lasso* is a relatively recent alternative to ridge regression that overcomes this disadvantage. The lasso coefficients, $\hat{\beta}_\lambda^L$, minimize the quantity

$$\sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^p |\beta_j| = \text{RSS} + \lambda \sum_{j=1}^p |\beta_j|. \quad (6.7)$$

Comparing (6.7) to (6.5), we see that the lasso and ridge regression have similar formulations. The only difference is that the β_j^2 term in the ridge regression penalty (6.5) has been replaced by $|\beta_j|$ in the lasso penalty (6.7). In statistical parlance, the lasso uses an ℓ_1 (pronounced “ell 1”) penalty instead of an ℓ_2 penalty. The ℓ_1 norm of a coefficient vector β is given by $\|\beta\|_1 = \sum |\beta_j|$.

As with ridge regression, the lasso shrinks the coefficient estimates towards zero. However, in the case of the lasso, the ℓ_1 penalty has the effect of forcing some of the coefficient estimates to be exactly equal to zero when the tuning parameter λ is sufficiently large. Hence, much like best subset selection, the lasso performs *variable selection*. As a result, models generated from the lasso are generally much easier to interpret than those produced by ridge regression. We say that the lasso yields *sparse* models—that is, models that involve only a subset of the variables. As in ridge regression, selecting a good value of λ for the lasso is critical; we defer this discussion to Section 6.2.3, where we use cross-validation.

lasso

sparse

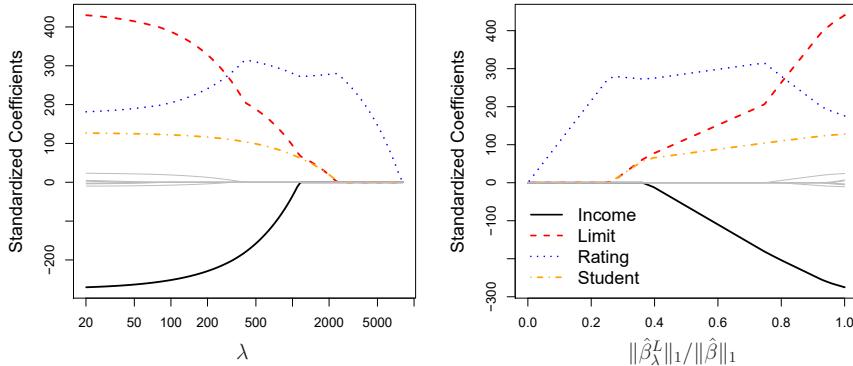


FIGURE 6.6. The standardized lasso coefficients on the `Credit` data set are shown as a function of λ and $\|\hat{\beta}_\lambda^L\|_1 / \|\hat{\beta}\|_1$.

As an example, consider the coefficient plots in Figure 6.6, which are generated from applying the lasso to the `Credit` data set. When $\lambda = 0$, then the lasso simply gives the least squares fit, and when λ becomes sufficiently large, the lasso gives the null model in which all coefficient estimates equal zero. However, in between these two extremes, the ridge regression and lasso models are quite different from each other. Moving from left to right in the right-hand panel of Figure 6.6, we observe that at first the lasso results in a model that contains only the `rating` predictor. Then `student` and `limit` enter the model almost simultaneously, shortly followed by `income`. Eventually, the remaining variables enter the model. Hence, depending on the value of λ , the lasso can produce a model involving any number of variables. In contrast, ridge regression will always include all of the variables in the model, although the magnitude of the coefficient estimates will depend on λ .

Another Formulation for Ridge Regression and the Lasso

One can show that the lasso and ridge regression coefficient estimates solve the problems

$$\underset{\beta}{\text{minimize}} \left\{ \sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2 \right\} \quad \text{subject to} \quad \sum_{j=1}^p |\beta_j| \leq s \quad (6.8)$$

and

$$\underset{\beta}{\text{minimize}} \left\{ \sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2 \right\} \quad \text{subject to} \quad \sum_{j=1}^p \beta_j^2 \leq s, \quad (6.9)$$

respectively. In other words, for every value of λ , there is some s such that the Equations (6.7) and (6.8) will give the same lasso coefficient estimates. Similarly, for every value of λ there is a corresponding s such that Equations (6.5) and (6.9) will give the same ridge regression coefficient estimates.

When $p = 2$, then (6.8) indicates that the lasso coefficient estimates have the smallest RSS out of all points that lie within the diamond defined by $|\beta_1| + |\beta_2| \leq s$. Similarly, the ridge regression estimates have the smallest RSS out of all points that lie within the circle defined by $\beta_1^2 + \beta_2^2 \leq s$.

We can think of (6.8) as follows. When we perform the lasso we are trying to find the set of coefficient estimates that lead to the smallest RSS, subject to the constraint that there is a *budget* s for how large $\sum_{j=1}^p |\beta_j|$ can be. When s is extremely large, then this budget is not very restrictive, and so the coefficient estimates can be large. In fact, if s is large enough that the least squares solution falls within the budget, then (6.8) will simply yield the least squares solution. In contrast, if s is small, then $\sum_{j=1}^p |\beta_j|$ must be small in order to avoid violating the budget. Similarly, (6.9) indicates that when we perform ridge regression, we seek a set of coefficient estimates such that the RSS is as small as possible, subject to the requirement that $\sum_{j=1}^p \beta_j^2$ not exceed the budget s .

The formulations (6.8) and (6.9) reveal a close connection between the lasso, ridge regression, and best subset selection. Consider the problem

$$\underset{\beta}{\text{minimize}} \left\{ \sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2 \right\} \quad \text{subject to} \quad \sum_{j=1}^p I(\beta_j \neq 0) \leq s. \quad (6.10)$$

Here $I(\beta_j \neq 0)$ is an indicator variable: it takes on a value of 1 if $\beta_j \neq 0$, and equals zero otherwise. Then (6.10) amounts to finding a set of coefficient estimates such that RSS is as small as possible, subject to the constraint that no more than s coefficients can be nonzero. The problem (6.10) is equivalent to best subset selection. Unfortunately, solving (6.10) is computationally infeasible when p is large, since it requires considering all $\binom{p}{s}$ models containing s predictors. Therefore, we can interpret ridge regression and the lasso as computationally feasible alternatives to best subset selection that replace the intractable form of the budget in (6.10) with forms that are much easier to solve. Of course, the lasso is much more closely related to best subset selection, since the lasso performs feature selection for s sufficiently small in (6.8), while ridge regression does not.

The Variable Selection Property of the Lasso

Why is it that the lasso, unlike ridge regression, results in coefficient estimates that are exactly equal to zero? The formulations (6.8) and (6.9) can be used to shed light on the issue. Figure 6.7 illustrates the situation. The least squares solution is marked as $\hat{\beta}$, while the blue diamond and circle represent the lasso and ridge regression constraints in (6.8) and (6.9), respectively. If s is sufficiently large, then the constraint regions will contain $\hat{\beta}$, and so the ridge regression and lasso estimates will be the same as the least squares estimates. (Such a large value of s corresponds to $\lambda = 0$ in (6.5) and (6.7).) However, in Figure 6.7 the least squares estimates lie outside of the diamond and the circle, and so the least squares estimates are not the same as the lasso and ridge regression estimates.

Each of the ellipses centered around $\hat{\beta}$ represents a *contour*: this means that all of the points on a particular ellipse have the same RSS value. As

contour

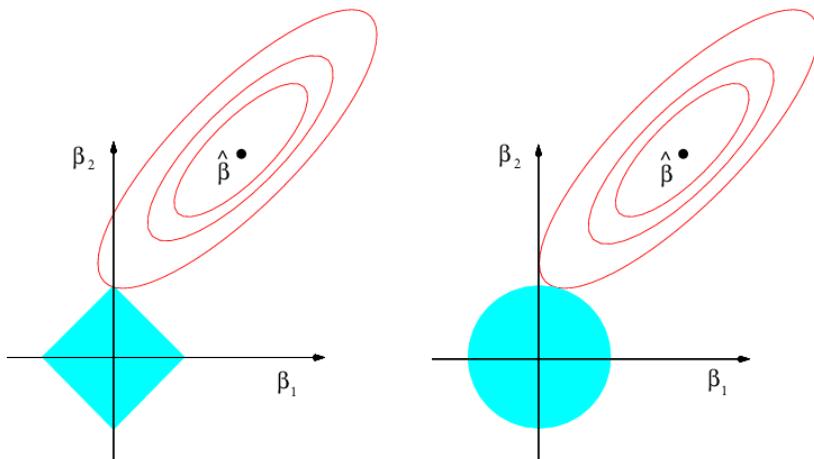


FIGURE 6.7. Contours of the error and constraint functions for the lasso (left) and ridge regression (right). The solid blue areas are the constraint regions, $|\beta_1| + |\beta_2| \leq s$ and $\beta_1^2 + \beta_2^2 \leq s$, while the red ellipses are the contours of the RSS.

the ellipses expand away from the least squares coefficient estimates, the RSS increases. Equations (6.8) and (6.9) indicate that the lasso and ridge regression coefficient estimates are given by the first point at which an ellipse contacts the constraint region. Since ridge regression has a circular constraint with no sharp points, this intersection will not generally occur on an axis, and so the ridge regression coefficient estimates will be exclusively non-zero. However, the lasso constraint has *corners* at each of the axes, and so the ellipse will often intersect the constraint region at an axis. When this occurs, one of the coefficients will equal zero. In higher dimensions, many of the coefficient estimates may equal zero simultaneously. In Figure 6.7, the intersection occurs at $\beta_1 = 0$, and so the resulting model will only include β_2 .

In Figure 6.7, we considered the simple case of $p = 2$. When $p = 3$, then the constraint region for ridge regression becomes a sphere, and the constraint region for the lasso becomes a polyhedron. When $p > 3$, the constraint for ridge regression becomes a hypersphere, and the constraint for the lasso becomes a polytope. However, the key ideas depicted in Figure 6.7 still hold. In particular, the lasso leads to feature selection when $p > 2$ due to the sharp corners of the polyhedron or polytope.

Comparing the Lasso and Ridge Regression

It is clear that the lasso has a major advantage over ridge regression, in that it produces simpler and more interpretable models that involve only a subset of the predictors. However, which method leads to better prediction accuracy? Figure 6.8 displays the variance, squared bias, and test MSE of the lasso applied to the same simulated data as in Figure 6.5. Clearly the lasso leads to qualitatively similar behavior to ridge regression, in that as λ increases, the variance decreases and the bias increases. In the right-hand

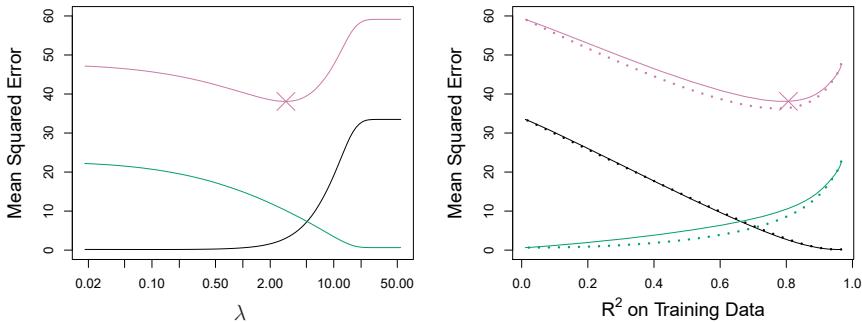


FIGURE 6.8. Left: Plots of squared bias (black), variance (green), and test MSE (purple) for the lasso on a simulated data set. Right: Comparison of squared bias, variance, and test MSE between lasso (solid) and ridge (dotted). Both are plotted against their R^2 on the training data, as a common form of indexing. The crosses in both plots indicate the lasso model for which the MSE is smallest.

panel of Figure 6.8, the dotted lines represent the ridge regression fits. Here we plot both against their R^2 on the training data. This is another useful way to index models, and can be used to compare models with different types of regularization, as is the case here. In this example, the lasso and ridge regression result in almost identical biases. However, the variance of ridge regression is slightly lower than the variance of the lasso. Consequently, the minimum MSE of ridge regression is slightly smaller than that of the lasso.

However, the data in Figure 6.8 were generated in such a way that all 45 predictors were related to the response—that is, none of the true coefficients $\beta_1, \dots, \beta_{45}$ equaled zero. The lasso implicitly assumes that a number of the coefficients truly equal zero. Consequently, it is not surprising that ridge regression outperforms the lasso in terms of prediction error in this setting. Figure 6.9 illustrates a similar situation, except that now the response is a function of only 2 out of 45 predictors. Now the lasso tends to outperform ridge regression in terms of bias, variance, and MSE.

These two examples illustrate that neither ridge regression nor the lasso will universally dominate the other. In general, one might expect the lasso to perform better in a setting where a relatively small number of predictors have substantial coefficients, and the remaining predictors have coefficients that are very small or that equal zero. Ridge regression will perform better when the response is a function of many predictors, all with coefficients of roughly equal size. However, the number of predictors that is related to the response is never known *a priori* for real data sets. A technique such as cross-validation can be used in order to determine which approach is better on a particular data set.

As with ridge regression, when the least squares estimates have excessively high variance, the lasso solution can yield a reduction in variance at the expense of a small increase in bias, and consequently can generate more accurate predictions. Unlike ridge regression, the lasso performs variable selection, and hence results in models that are easier to interpret.

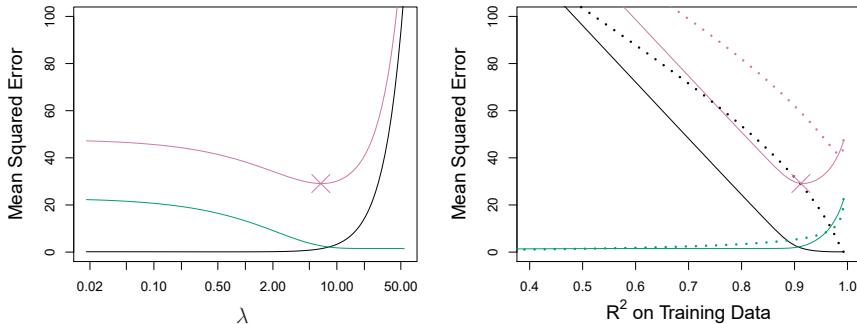


FIGURE 6.9. Left: Plots of squared bias (black), variance (green), and test MSE (purple) for the lasso. The simulated data is similar to that in Figure 6.8, except that now only two predictors are related to the response. Right: Comparison of squared bias, variance, and test MSE between lasso (solid) and ridge (dotted). Both are plotted against their R^2 on the training data, as a common form of indexing. The crosses in both plots indicate the lasso model for which the MSE is smallest.

There are very efficient algorithms for fitting both ridge and lasso models; in both cases the entire coefficient paths can be computed with about the same amount of work as a single least squares fit. We will explore this further in the lab at the end of this chapter.

A Simple Special Case for Ridge Regression and the Lasso

In order to obtain a better intuition about the behavior of ridge regression and the lasso, consider a simple special case with $n = p$, and \mathbf{X} a diagonal matrix with 1's on the diagonal and 0's in all off-diagonal elements. To simplify the problem further, assume also that we are performing regression without an intercept. With these assumptions, the usual least squares problem simplifies to finding β_1, \dots, β_p that minimize

$$\sum_{j=1}^p (y_j - \beta_j)^2. \quad (6.11)$$

In this case, the least squares solution is given by

$$\hat{\beta}_j = y_j.$$

And in this setting, ridge regression amounts to finding β_1, \dots, β_p such that

$$\sum_{j=1}^p (y_j - \beta_j)^2 + \lambda \sum_{j=1}^p \beta_j^2 \quad (6.12)$$

is minimized, and the lasso amounts to finding the coefficients such that

$$\sum_{j=1}^p (y_j - \beta_j)^2 + \lambda \sum_{j=1}^p |\beta_j| \quad (6.13)$$

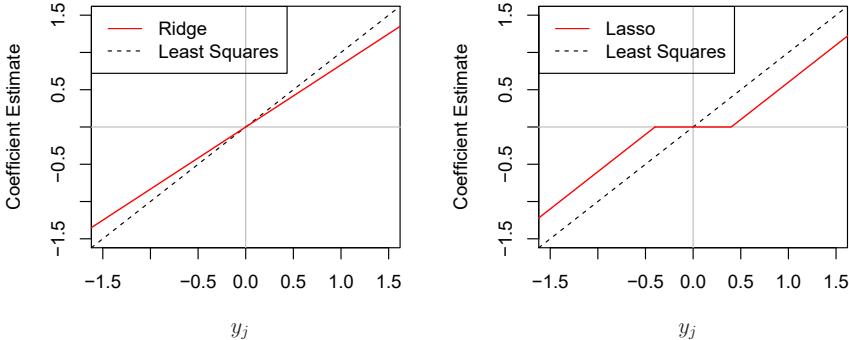


FIGURE 6.10. The ridge regression and lasso coefficient estimates for a simple setting with $n = p$ and \mathbf{X} a diagonal matrix with 1's on the diagonal. Left: The ridge regression coefficient estimates are shrunken proportionally towards zero, relative to the least squares estimates. Right: The lasso coefficient estimates are soft-thresholded towards zero.

is minimized. One can show that in this setting, the ridge regression estimates take the form

$$\hat{\beta}_j^R = y_j / (1 + \lambda), \quad (6.14)$$

and the lasso estimates take the form

$$\hat{\beta}_j^L = \begin{cases} y_j - \lambda/2 & \text{if } y_j > \lambda/2; \\ y_j + \lambda/2 & \text{if } y_j < -\lambda/2; \\ 0 & \text{if } |y_j| \leq \lambda/2. \end{cases} \quad (6.15)$$

Figure 6.10 displays the situation. We can see that ridge regression and the lasso perform two very different types of shrinkage. In ridge regression, each least squares coefficient estimate is shrunken by the same proportion. In contrast, the lasso shrinks each least squares coefficient towards zero by a constant amount, $\lambda/2$; the least squares coefficients that are less than $\lambda/2$ in absolute value are shrunk entirely to zero. The type of shrinkage performed by the lasso in this simple setting (6.15) is known as *soft-thresholding*. The fact that some lasso coefficients are shrunk entirely to zero explains why the lasso performs feature selection.

soft-thresholding

In the case of a more general data matrix \mathbf{X} , the story is a little more complicated than what is depicted in Figure 6.10, but the main ideas still hold approximately: ridge regression more or less shrinks every dimension of the data by the same proportion, whereas the lasso more or less shrinks all coefficients toward zero by a similar amount, and sufficiently small coefficients are shrunk all the way to zero.

Bayesian Interpretation of Ridge Regression and the Lasso



We now show that one can view ridge regression and the lasso through a Bayesian lens. A Bayesian viewpoint for regression assumes that the coefficient vector β has some *prior* distribution, say $p(\beta)$, where $\beta = (\beta_0, \beta_1, \dots, \beta_p)^T$. The likelihood of the data can be written as $f(Y|X, \beta)$,

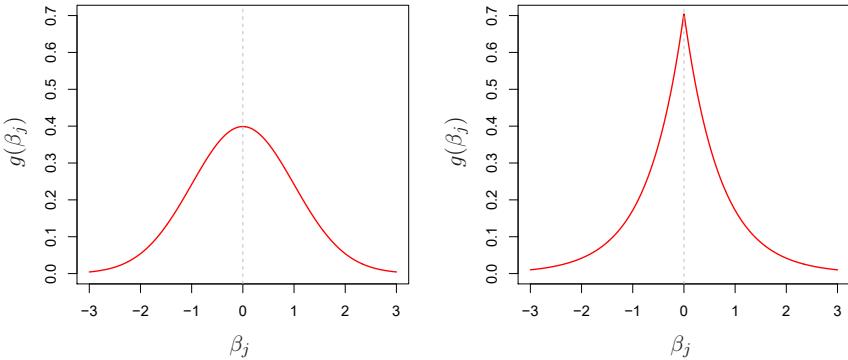


FIGURE 6.11. Left: Ridge regression is the posterior mode for β under a Gaussian prior. Right: The lasso is the posterior mode for β under a double-exponential prior.

where $X = (X_1, \dots, X_p)$. Multiplying the prior distribution by the likelihood gives us (up to a proportionality constant) the *posterior distribution*, which takes the form

$$p(\beta|X, Y) \propto f(Y|X, \beta)p(\beta|X) = f(Y|X, \beta)p(\beta),$$

where the proportionality above follows from Bayes' theorem, and the equality above follows from the assumption that X is fixed.

We assume the usual linear model,

$$Y = \beta_0 + X_1\beta_1 + \dots + X_p\beta_p + \epsilon,$$

and suppose that the errors are independent and drawn from a normal distribution. Furthermore, assume that $p(\beta) = \prod_{j=1}^p g(\beta_j)$, for some density function g . It turns out that ridge regression and the lasso follow naturally from two special cases of g :

- If g is a Gaussian distribution with mean zero and standard deviation a function of λ , then it follows that the *posterior mode* for β —that is, the most likely value for β , given the data—is given by the ridge regression solution. (In fact, the ridge regression solution is also the posterior mean.)
- If g is a double-exponential (Laplace) distribution with mean zero and scale parameter a function of λ , then it follows that the posterior mode for β is the lasso solution. (However, the lasso solution is *not* the posterior mean, and in fact, the posterior mean does not yield a sparse coefficient vector.)

posterior distribution

posterior mode

The Gaussian and double-exponential priors are displayed in Figure 6.11. Therefore, from a Bayesian viewpoint, ridge regression and the lasso follow directly from assuming the usual linear model with normal errors, together with a simple prior distribution for β . Notice that the lasso prior is steeply peaked at zero, while the Gaussian is flatter and fatter at zero. Hence, the lasso expects a priori that many of the coefficients are (exactly) zero, while ridge assumes the coefficients are randomly distributed about zero.

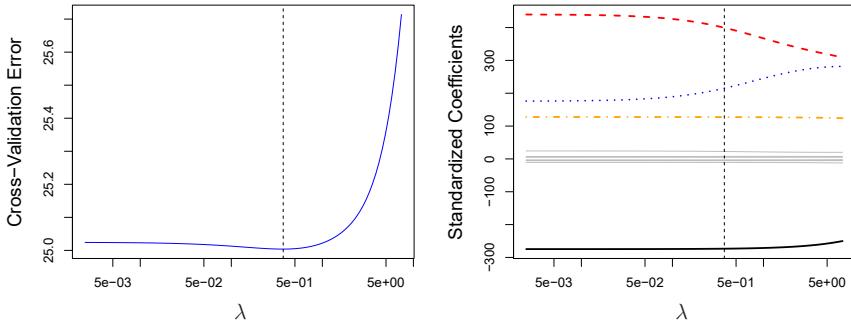


FIGURE 6.12. Left: Cross-validation errors that result from applying ridge regression to the `Credit` data set with various values of λ . Right: The coefficient estimates as a function of λ . The vertical dashed lines indicate the value of λ selected by cross-validation.

6.2.3 Selecting the Tuning Parameter

Just as the subset selection approaches considered in Section 6.1 require a method to determine which of the models under consideration is best, implementing ridge regression and the lasso requires a method for selecting a value for the tuning parameter λ in (6.5) and (6.7), or equivalently, the value of the constraint s in (6.9) and (6.8). Cross-validation provides a simple way to tackle this problem. We choose a grid of λ values, and compute the cross-validation error for each value of λ , as described in Chapter 5. We then select the tuning parameter value for which the cross-validation error is smallest. Finally, the model is re-fit using all of the available observations and the selected value of the tuning parameter.

Figure 6.12 displays the choice of λ that results from performing leave-one-out cross-validation on the ridge regression fits from the `Credit` data set. The dashed vertical lines indicate the selected value of λ . In this case the value is relatively small, indicating that the optimal fit only involves a small amount of shrinkage relative to the least squares solution. In addition, the dip is not very pronounced, so there is rather a wide range of values that would give a very similar error. In a case like this we might simply use the least squares solution.

Figure 6.13 provides an illustration of ten-fold cross-validation applied to the lasso fits on the sparse simulated data from Figure 6.9. The left-hand panel of Figure 6.13 displays the cross-validation error, while the right-hand panel displays the coefficient estimates. The vertical dashed lines indicate the point at which the cross-validation error is smallest. The two colored lines in the right-hand panel of Figure 6.13 represent the two predictors that are related to the response, while the grey lines represent the unrelated predictors; these are often referred to as *signal* and *noise* variables, respectively. Not only has the lasso correctly given much larger coefficient estimates to the two signal predictors, but also the minimum cross-validation error corresponds to a set of coefficient estimates for which only the signal variables are non-zero. Hence cross-validation together with the lasso has correctly identified the two signal variables in the model, even though this is a challenging setting, with $p = 45$ variables and only $n = 50$

signal

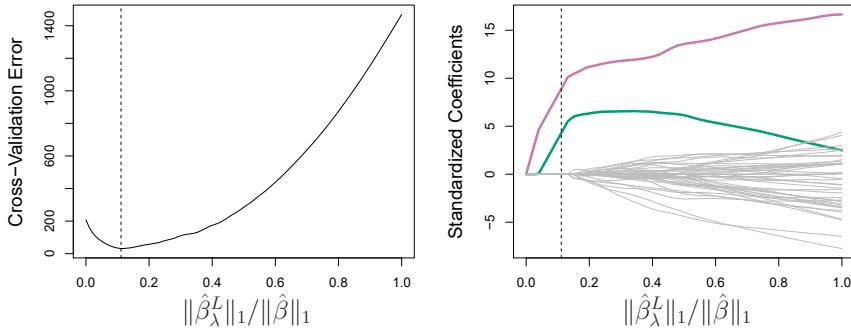


FIGURE 6.13. Left: Ten-fold cross-validation MSE for the lasso, applied to the sparse simulated data set from Figure 6.9. Right: The corresponding lasso coefficient estimates are displayed. The two signal variables are shown in color, and the noise variables are in gray. The vertical dashed lines indicate the lasso fit for which the cross-validation error is smallest.

observations. In contrast, the least squares solution—displayed on the far right of the right-hand panel of Figure 6.13—assigns a large coefficient estimate to only one of the two signal variables.

6.3 Dimension Reduction Methods

The methods that we have discussed so far in this chapter have controlled variance in two different ways, either by using a subset of the original variables, or by shrinking their coefficients toward zero. All of these methods are defined using the original predictors, X_1, X_2, \dots, X_p . We now explore a class of approaches that *transform* the predictors and then fit a least squares model using the transformed variables. We will refer to these techniques as *dimension reduction* methods.

Let Z_1, Z_2, \dots, Z_M represent $M < p$ linear combinations of our original p predictors. That is,

$$Z_m = \sum_{j=1}^p \phi_{jm} X_j \quad (6.16)$$

dimension
reduction
linear
combination

for some constants $\phi_{1m}, \phi_{2m}, \dots, \phi_{pm}$, $m = 1, \dots, M$. We can then fit the linear regression model

$$y_i = \theta_0 + \sum_{m=1}^M \theta_m z_{im} + \epsilon_i, \quad i = 1, \dots, n, \quad (6.17)$$

using least squares. Note that in (6.17), the regression coefficients are given by $\theta_0, \theta_1, \dots, \theta_M$. If the constants $\phi_{1m}, \phi_{2m}, \dots, \phi_{pm}$ are chosen wisely, then such dimension reduction approaches can often outperform least squares regression. In other words, fitting (6.17) using least squares can lead to better results than fitting (6.1) using least squares.

The term *dimension reduction* comes from the fact that this approach reduces the problem of estimating the $p+1$ coefficients $\beta_0, \beta_1, \dots, \beta_p$ to the

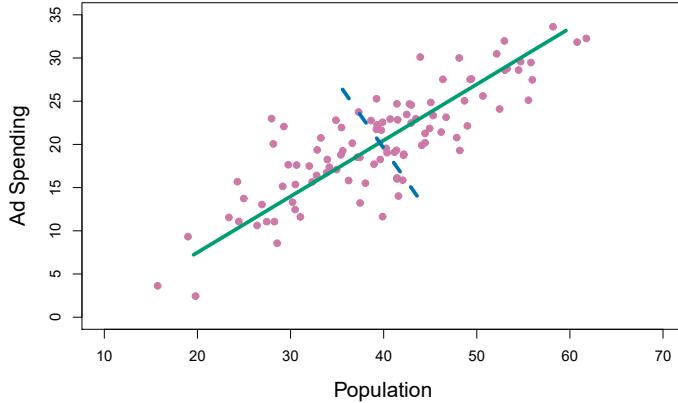


FIGURE 6.14. The population size (pop) and ad spending (ad) for 100 different cities are shown as purple circles. The green solid line indicates the first principal component, and the blue dashed line indicates the second principal component.

simpler problem of estimating the $M + 1$ coefficients $\theta_0, \theta_1, \dots, \theta_M$, where $M < p$. In other words, the dimension of the problem has been reduced from $p + 1$ to $M + 1$.

Notice that from (6.16),

$$\sum_{m=1}^M \theta_m z_{im} = \sum_{m=1}^M \theta_m \sum_{j=1}^p \phi_{jm} x_{ij} = \sum_{j=1}^p \sum_{m=1}^M \theta_m \phi_{jm} x_{ij} = \sum_{j=1}^p \beta_j x_{ij},$$

where

$$\beta_j = \sum_{m=1}^M \theta_m \phi_{jm}. \quad (6.18)$$

Hence (6.17) can be thought of as a special case of the original linear regression model given by (6.1). Dimension reduction serves to constrain the estimated β_j coefficients, since now they must take the form (6.18). This constraint on the form of the coefficients has the potential to bias the coefficient estimates. However, in situations where p is large relative to n , selecting a value of $M \ll p$ can significantly reduce the variance of the fitted coefficients. If $M = p$, and all the Z_m are linearly independent, then (6.18) poses no constraints. In this case, no dimension reduction occurs, and so fitting (6.17) is equivalent to performing least squares on the original p predictors.

All dimension reduction methods work in two steps. First, the transformed predictors Z_1, Z_2, \dots, Z_M are obtained. Second, the model is fit using these M predictors. However, the choice of Z_1, Z_2, \dots, Z_M , or equivalently, the selection of the ϕ_{jm} 's, can be achieved in different ways. In this chapter, we will consider two approaches for this task: *principal components* and *partial least squares*.

6.3.1 Principal Components Regression

Principal components analysis (PCA) is a popular approach for deriving

principal
components
analysis

a low-dimensional set of features from a large set of variables. PCA is discussed in greater detail as a tool for *unsupervised learning* in Chapter 12. Here we describe its use as a dimension reduction technique for regression.

An Overview of Principal Components Analysis

PCA is a technique for reducing the dimension of an $n \times p$ data matrix \mathbf{X} . The *first principal component* direction of the data is that along which the observations *vary the most*. For instance, consider Figure 6.14, which shows population size (`pop`) in tens of thousands of people, and ad spending for a particular company (`ad`) in thousands of dollars, for 100 cities.⁶ The green solid line represents the first principal component direction of the data. We can see by eye that this is the direction along which there is the greatest variability in the data. That is, if we *projected* the 100 observations onto this line (as shown in the left-hand panel of Figure 6.15), then the resulting projected observations would have the largest possible variance; projecting the observations onto any other line would yield projected observations with lower variance. Projecting a point onto a line simply involves finding the location on the line which is closest to the point.

The first principal component is displayed graphically in Figure 6.14, but how can it be summarized mathematically? It is given by the formula

$$Z_1 = 0.839 \times (\text{pop} - \overline{\text{pop}}) + 0.544 \times (\text{ad} - \overline{\text{ad}}). \quad (6.19)$$

Here $\phi_{11} = 0.839$ and $\phi_{21} = 0.544$ are the principal component loadings, which define the direction referred to above. In (6.19), $\overline{\text{pop}}$ indicates the mean of all `pop` values in this data set, and $\overline{\text{ad}}$ indicates the mean of all advertising spending. The idea is that out of every possible *linear combination* of `pop` and `ad` such that $\phi_{11}^2 + \phi_{21}^2 = 1$, this particular linear combination yields the highest variance: i.e. this is the linear combination for which $\text{Var}(\phi_{11} \times (\text{pop} - \overline{\text{pop}}) + \phi_{21} \times (\text{ad} - \overline{\text{ad}}))$ is maximized. It is necessary to consider only linear combinations of the form $\phi_{11}^2 + \phi_{21}^2 = 1$, since otherwise we could increase ϕ_{11} and ϕ_{21} arbitrarily in order to blow up the variance. In (6.19), the two loadings are both positive and have similar size, and so Z_1 is almost an *average* of the two variables.

Since $n = 100$, `pop` and `ad` are vectors of length 100, and so is Z_1 in (6.19). For instance,

$$z_{i1} = 0.839 \times (\text{pop}_i - \overline{\text{pop}}) + 0.544 \times (\text{ad}_i - \overline{\text{ad}}). \quad (6.20)$$

The values of z_{11}, \dots, z_{n1} are known as the *principal component scores*, and can be seen in the right-hand panel of Figure 6.15.

There is also another interpretation of PCA: the first principal component vector defines the line that is *as close as possible* to the data. For instance, in Figure 6.14, the first principal component line minimizes the sum of the squared perpendicular distances between each point and the line. These distances are plotted as dashed line segments in the left-hand

⁶This dataset is distinct from the `Advertising` data discussed in Chapter 3.

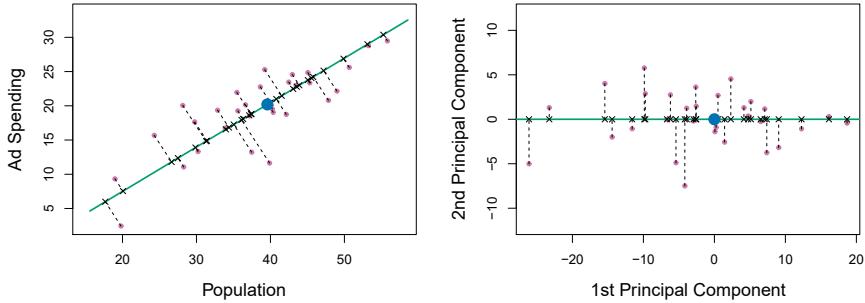


FIGURE 6.15. A subset of the advertising data. The mean `pop` and `ad` budgets are indicated with a blue circle. Left: The first principal component direction is shown in green. It is the dimension along which the data vary the most, and it also defines the line that is closest to all n of the observations. The distances from each observation to the principal component are represented using the black dashed line segments. The blue dot represents $(\bar{\text{pop}}, \bar{\text{ad}})$. Right: The left-hand panel has been rotated so that the first principal component direction coincides with the x -axis.

panel of Figure 6.15, in which the crosses represent the *projection* of each point onto the first principal component line. The first principal component has been chosen so that the projected observations are *as close as possible* to the original observations.

In the right-hand panel of Figure 6.15, the left-hand panel has been rotated so that the first principal component direction coincides with the x -axis. It is possible to show that the *first principal component score* for the i th observation, given in (6.20), is the distance in the x -direction of the i th cross from zero. So for example, the point in the bottom-left corner of the left-hand panel of Figure 6.15 has a large negative principal component score, $z_{i1} = -26.1$, while the point in the top-right corner has a large positive score, $z_{i1} = 18.7$. These scores can be computed directly using (6.20).

We can think of the values of the principal component Z_1 as single-number summaries of the joint `pop` and `ad` budgets for each location. In this example, if $z_{i1} = 0.839 \times (\text{pop}_i - \bar{\text{pop}}) + 0.544 \times (\text{ad}_i - \bar{\text{ad}}) < 0$, then this indicates a city with below-average population size and below-average ad spending. A positive score suggests the opposite. How well can a single number represent both `pop` and `ad`? In this case, Figure 6.14 indicates that `pop` and `ad` have approximately a linear relationship, and so we might expect that a single-number summary will work well. Figure 6.16 displays z_{i1} versus both `pop` and `ad`.⁷ The plots show a strong relationship between the first principal component and the two features. In other words, the first principal component appears to capture most of the information contained in the `pop` and `ad` predictors.

So far we have concentrated on the first principal component. In general, one can construct up to p distinct principal components. The second

⁷The principal components were calculated after first standardizing both `pop` and `ad`, a common approach. Hence, the x -axes on Figures 6.15 and 6.16 are not on the same scale.

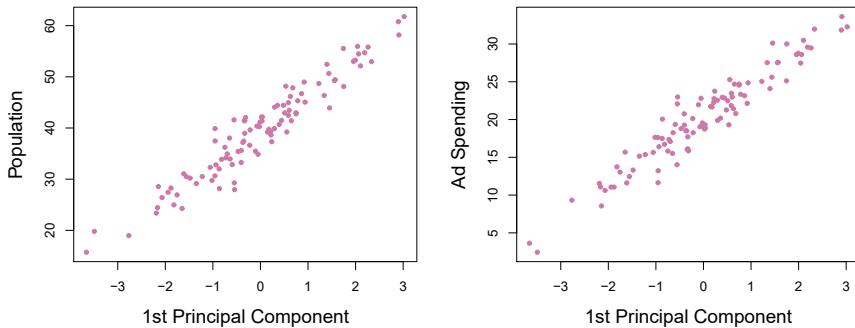


FIGURE 6.16. Plots of the first principal component scores z_{i1} versus pop and ad . The relationships are strong.

principal component Z_2 is a linear combination of the variables that is uncorrelated with Z_1 , and has largest variance subject to this constraint. The second principal component direction is illustrated as a dashed blue line in Figure 6.14. It turns out that the zero correlation condition of Z_1 with Z_2 is equivalent to the condition that the direction must be *perpendicular*, or *orthogonal*, to the first principal component direction. The second principal component is given by the formula

$$Z_2 = 0.544 \times (\text{pop} - \overline{\text{pop}}) - 0.839 \times (\text{ad} - \overline{\text{ad}}).$$

perpendicular
orthogonal

Since the advertising data has two predictors, the first two principal components contain all of the information that is in pop and ad . However, by construction, the first component will contain the most information. Consider, for example, the much larger variability of z_{i1} (the x -axis) versus z_{i2} (the y -axis) in the right-hand panel of Figure 6.15. The fact that the second principal component scores are much closer to zero indicates that this component captures far less information. As another illustration, Figure 6.17 displays z_{i2} versus pop and ad . There is little relationship between the second principal component and these two predictors, again suggesting that in this case, one only needs the first principal component in order to accurately represent the pop and ad budgets.

With two-dimensional data, such as in our advertising example, we can construct at most two principal components. However, if we had other predictors, such as population age, income level, education, and so forth, then additional components could be constructed. They would successively maximize variance, subject to the constraint of being uncorrelated with the preceding components.

The Principal Components Regression Approach

The *principal components regression* (PCR) approach involves constructing the first M principal components, Z_1, \dots, Z_M , and then using these components as the predictors in a linear regression model that is fit using least squares. The key idea is that often a small number of principal components suffice to explain most of the variability in the data, as well as the relationship with the response. In other words, we assume that *the*

principal
components
regression

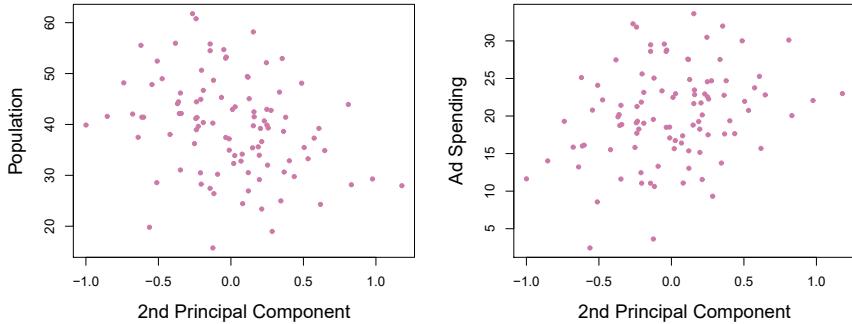


FIGURE 6.17. Plots of the second principal component scores z_{i2} versus `pop` and `ad`. The relationships are weak.

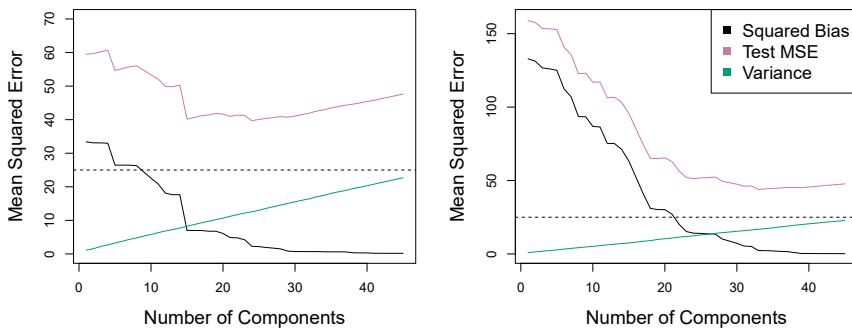


FIGURE 6.18. PCR was applied to two simulated data sets. In each panel, the horizontal dashed line represents the irreducible error. Left: Simulated data from Figure 6.8. Right: Simulated data from Figure 6.9.

directions in which X_1, \dots, X_p show the most variation are the directions that are associated with Y . While this assumption is not guaranteed to be true, it often turns out to be a reasonable enough approximation to give good results.

If the assumption underlying PCR holds, then fitting a least squares model to Z_1, \dots, Z_M will lead to better results than fitting a least squares model to X_1, \dots, X_p , since most or all of the information in the data that relates to the response is contained in Z_1, \dots, Z_M , and by estimating only $M \ll p$ coefficients we can mitigate overfitting. In the advertising data, the first principal component explains most of the variance in both `pop` and `ad`, so a principal component regression that uses this single variable to predict some response of interest, such as `sales`, will likely perform quite well.

Figure 6.18 displays the PCR fits on the simulated data sets from Figures 6.8 and 6.9. Recall that both data sets were generated using $n = 50$ observations and $p = 45$ predictors. However, while the response in the first data set was a function of all the predictors, the response in the second data set was generated using only two of the predictors. The curves are plotted as a function of M , the number of principal components used as predictors in the regression model. As more principal components are used

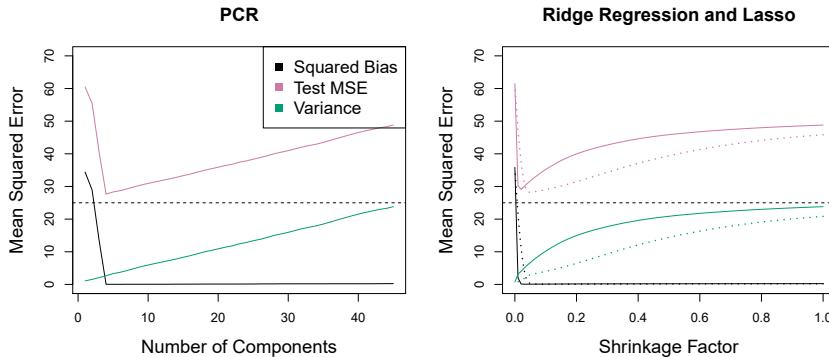


FIGURE 6.19. PCR, ridge regression, and the lasso were applied to a simulated data set in which the first five principal components of X contain all the information about the response Y . In each panel, the irreducible error $\text{Var}(\epsilon)$ is shown as a horizontal dashed line. Left: Results for PCR. Right: Results for lasso (solid) and ridge regression (dotted). The x -axis displays the shrinkage factor of the coefficient estimates, defined as the ℓ_2 norm of the shrunken coefficient estimates divided by the ℓ_2 norm of the least squares estimate.

in the regression model, the bias decreases, but the variance increases. This results in a typical U-shape for the mean squared error. When $M = p = 45$, then PCR amounts simply to a least squares fit using all of the original predictors. The figure indicates that performing PCR with an appropriate choice of M can result in a substantial improvement over least squares, especially in the left-hand panel. However, by examining the ridge regression and lasso results in Figures 6.5, 6.8, and 6.9, we see that PCR does not perform as well as the two shrinkage methods in this example.

The relatively worse performance of PCR in Figure 6.18 is a consequence of the fact that the data were generated in such a way that many principal components are required in order to adequately model the response. In contrast, PCR will tend to do well in cases when the first few principal components are sufficient to capture most of the variation in the predictors as well as the relationship with the response. The left-hand panel of Figure 6.19 illustrates the results from another simulated data set designed to be more favorable to PCR. Here the response was generated in such a way that it depends exclusively on the first five principal components. Now the bias drops to zero rapidly as M , the number of principal components used in PCR, increases. The mean squared error displays a clear minimum at $M = 5$. The right-hand panel of Figure 6.19 displays the results on these data using ridge regression and the lasso. All three methods offer a significant improvement over least squares. However, PCR and ridge regression slightly outperform the lasso.

We note that even though PCR provides a simple way to perform regression using $M < p$ predictors, it is *not* a feature selection method. This is because each of the M principal components used in the regression is a linear combination of all p of the *original* features. For instance, in (6.19), Z_1 was a linear combination of both **pop** and **ad**. Therefore, while PCR often performs quite well in many practical settings, it does not result in the

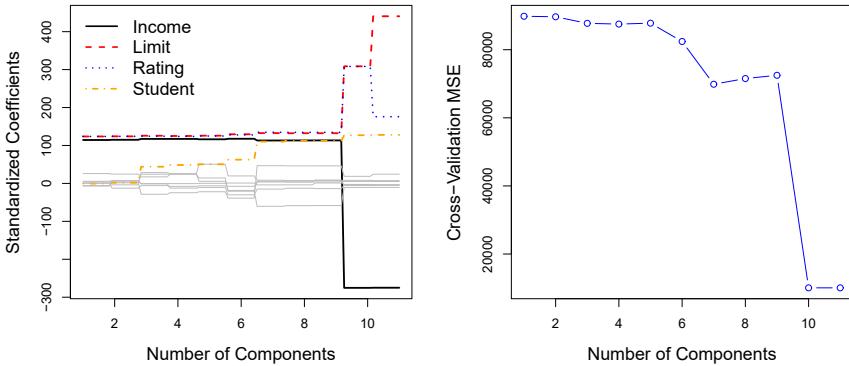


FIGURE 6.20. Left: PCR standardized coefficient estimates on the `Credit` data set for different values of M . Right: The ten-fold cross-validation MSE obtained using PCR, as a function of M .

development of a model that relies upon a small set of the original features. In this sense, PCR is more closely related to ridge regression than to the lasso. In fact, one can show that PCR and ridge regression are very closely related. One can even think of ridge regression as a continuous version of PCR!⁸

In PCR, the number of principal components, M , is typically chosen by cross-validation. The results of applying PCR to the `Credit` data set are shown in Figure 6.20; the right-hand panel displays the cross-validation errors obtained, as a function of M . On these data, the lowest cross-validation error occurs when there are $M = 10$ components; this corresponds to almost no dimension reduction at all, since PCR with $M = 11$ is equivalent to simply performing least squares.

When performing PCR, we generally recommend *standardizing* each predictor, using (6.6), prior to generating the principal components. This standardization ensures that all variables are on the same scale. In the absence of standardization, the high-variance variables will tend to play a larger role in the principal components obtained, and the scale on which the variables are measured will ultimately have an effect on the final PCR model. However, if the variables are all measured in the same units (say, kilograms, or inches), then one might choose not to standardize them.

6.3.2 Partial Least Squares

The PCR approach that we just described involves identifying linear combinations, or *directions*, that best represent the predictors X_1, \dots, X_p . These directions are identified in an *unsupervised* way, since the response Y is not used to help determine the principal component directions. That is, the response does not *supervise* the identification of the principal components. Consequently, PCR suffers from a drawback: there is no guarantee

⁸More details can be found in Section 3.5 of *The Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman.

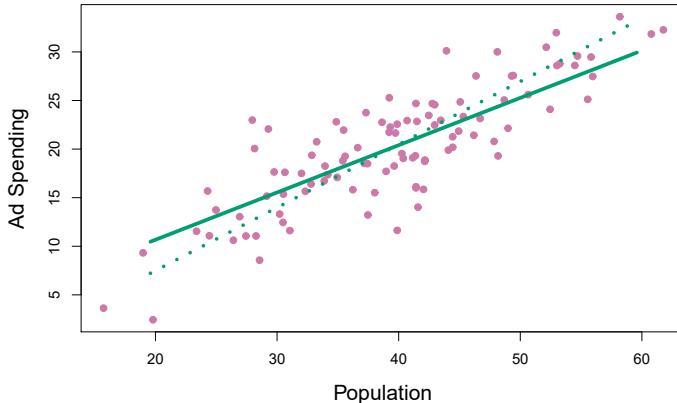


FIGURE 6.21. For the advertising data, the first PLS direction (solid line) and first PCR direction (dotted line) are shown.

that the directions that best explain the predictors will also be the best directions to use for predicting the response. Unsupervised methods are discussed further in Chapter 12.

We now present *partial least squares* (PLS), a *supervised* alternative to PCR. Like PCR, PLS is a dimension reduction method, which first identifies a new set of features Z_1, \dots, Z_M that are linear combinations of the original features, and then fits a linear model via least squares using these M new features. But unlike PCR, PLS identifies these new features in a supervised way—that is, it makes use of the response Y in order to identify new features that not only approximate the old features well, but also that are related to the response. Roughly speaking, the PLS approach attempts to find directions that help explain both the response and the predictors.

partial least squares

We now describe how the first PLS direction is computed. After standardizing the p predictors, PLS computes the first direction Z_1 by setting each ϕ_{j1} in (6.16) equal to the coefficient from the simple linear regression of Y onto X_j . One can show that this coefficient is proportional to the correlation between Y and X_j . Hence, in computing $Z_1 = \sum_{j=1}^p \phi_{j1} X_j$, PLS places the highest weight on the variables that are most strongly related to the response.

Figure 6.21 displays an example of PLS on a synthetic dataset with Sales in each of 100 regions as the response, and two predictors; Population Size and Advertising Spending. The solid green line indicates the first PLS direction, while the dotted line shows the first principal component direction. PLS has chosen a direction that has less change in the `ad` dimension per unit change in the `pop` dimension, relative to PCA. This suggests that `pop` is more highly correlated with the response than is `ad`. The PLS direction does not fit the predictors as closely as does PCA, but it does a better job explaining the response.

To identify the second PLS direction we first *adjust* each of the variables for Z_1 , by regressing each variable on Z_1 and taking *residuals*. These residuals can be interpreted as the remaining information that has not been explained by the first PLS direction. We then compute Z_2 using this *or-*

thogonalized data in exactly the same fashion as Z_1 was computed based on the original data. This iterative approach can be repeated M times to identify multiple PLS components Z_1, \dots, Z_M . Finally, at the end of this procedure, we use least squares to fit a linear model to predict Y using Z_1, \dots, Z_M in exactly the same fashion as for PCR.

As with PCR, the number M of partial least squares directions used in PLS is a tuning parameter that is typically chosen by cross-validation. We generally standardize the predictors and response before performing PLS.

PLS is popular in the field of chemometrics, where many variables arise from digitized spectrometry signals. In practice it often performs no better than ridge regression or PCR. While the supervised dimension reduction of PLS can reduce bias, it also has the potential to increase variance, so that the overall benefit of PLS relative to PCR is a wash.

6.4 Considerations in High Dimensions

6.4.1 High-Dimensional Data

Most traditional statistical techniques for regression and classification are intended for the *low-dimensional* setting in which n , the number of observations, is much greater than p , the number of features. This is due in part to the fact that throughout most of the field's history, the bulk of scientific problems requiring the use of statistics have been low-dimensional. For instance, consider the task of developing a model to predict a patient's blood pressure on the basis of his or her age, sex, and body mass index (BMI). There are three predictors, or four if an intercept is included in the model, and perhaps several thousand patients for whom blood pressure and age, sex, and BMI are available. Hence $n \gg p$, and so the problem is low-dimensional. (By dimension here we are referring to the size of p .)

low-dimensional

In the past 20 years, new technologies have changed the way that data are collected in fields as diverse as finance, marketing, and medicine. It is now commonplace to collect an almost unlimited number of feature measurements (p very large). While p can be extremely large, the number of observations n is often limited due to cost, sample availability, or other considerations. Two examples are as follows:

1. Rather than predicting blood pressure on the basis of just age, sex, and BMI, one might also collect measurements for half a million *single nucleotide polymorphisms* (SNPs; these are individual DNA mutations that are relatively common in the population) for inclusion in the predictive model. Then $n \approx 200$ and $p \approx 500,000$.
2. A marketing analyst interested in understanding people's online shopping patterns could treat as features all of the search terms entered by users of a search engine. This is sometimes known as the "bag-of-words" model. The same researcher might have access to the search histories of only a few hundred or a few thousand search engine users who have consented to share their information with the researcher. For a given user, each of the p search terms is scored present (0) or

absent (1), creating a large binary feature vector. Then $n \approx 1,000$ and p is much larger.

Data sets containing more features than observations are often referred to as *high-dimensional*. Classical approaches such as least squares linear regression are not appropriate in this setting. Many of the issues that arise in the analysis of high-dimensional data were discussed earlier in this book, since they apply also when $n > p$: these include the role of the bias-variance trade-off and the danger of overfitting. Though these issues are always relevant, they can become particularly important when the number of features is very large relative to the number of observations.

high-dimensional

We have defined the *high-dimensional setting* as the case where the number of features p is larger than the number of observations n . But the considerations that we will now discuss certainly also apply if p is slightly smaller than n , and are best always kept in mind when performing supervised learning.

6.4.2 What Goes Wrong in High Dimensions?

In order to illustrate the need for extra care and specialized techniques for regression and classification when $p > n$, we begin by examining what can go wrong if we apply a statistical technique not intended for the high-dimensional setting. For this purpose, we examine least squares regression. But the same concepts apply to logistic regression, linear discriminant analysis, and other classical statistical approaches.

When the number of features p is as large as, or larger than, the number of observations n , least squares as described in Chapter 3 cannot (or rather, *should not*) be performed. The reason is simple: regardless of whether or not there truly is a relationship between the features and the response, least squares will yield a set of coefficient estimates that result in a perfect fit to the data, such that the residuals are zero.

An example is shown in Figure 6.22 with $p = 1$ feature (plus an intercept) in two cases: when there are 20 observations, and when there are only two observations. When there are 20 observations, $n > p$ and the least squares regression line does not perfectly fit the data; instead, the regression line seeks to approximate the 20 observations as well as possible. On the other hand, when there are only two observations, then regardless of the values of those observations, the regression line will fit the data exactly. This is problematic because this perfect fit will almost certainly lead to overfitting of the data. In other words, though it is possible to perfectly fit the training data in the high-dimensional setting, the resulting linear model will perform extremely poorly on an independent test set, and therefore does not constitute a useful model. In fact, we can see that this happened in Figure 6.22: the least squares line obtained in the right-hand panel will perform very poorly on a test set comprised of the observations in the left-hand panel. The problem is simple: when $p > n$ or $p \approx n$, a simple least squares regression line is too *flexible* and hence overfits the data.

Figure 6.23 further illustrates the risk of carelessly applying least squares when the number of features p is large. Data were simulated with $n = 20$ observations, and regression was performed with between 1 and 20 features,

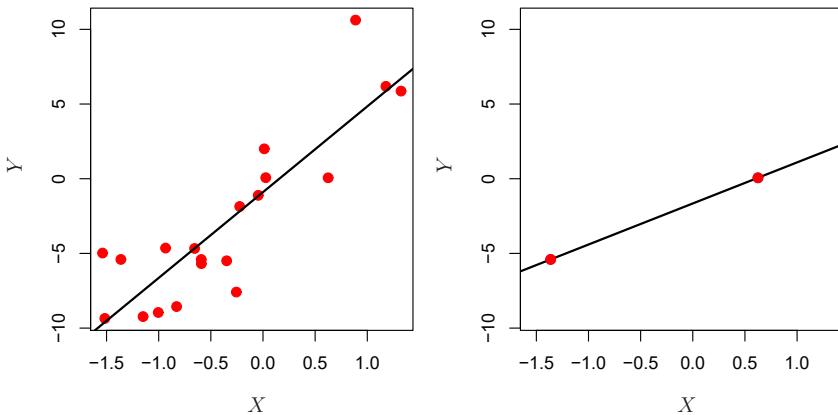


FIGURE 6.22. Left: Least squares regression in the low-dimensional setting. Right: Least squares regression with $n = 2$ observations and two parameters to be estimated (an intercept and a coefficient).

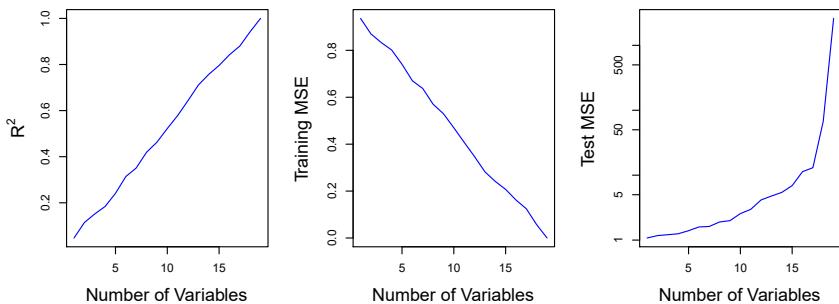


FIGURE 6.23. On a simulated example with $n = 20$ training observations, features that are completely unrelated to the outcome are added to the model. Left: The R^2 increases to 1 as more features are included. Center: The training set MSE decreases to 0 as more features are included. Right: The test set MSE increases as more features are included.

each of which was completely unrelated to the response. As shown in the figure, the model R^2 increases to 1 as the number of features included in the model increases, and correspondingly the training set MSE decreases to 0 as the number of features increases, *even though the features are completely unrelated to the response*. On the other hand, the MSE on an *independent test set* becomes extremely large as the number of features included in the model increases, because including the additional predictors leads to a vast increase in the variance of the coefficient estimates. Looking at the test set MSE, it is clear that the best model contains at most a few variables. However, someone who carelessly examines only the R^2 or the training set MSE might erroneously conclude that the model with the greatest number of variables is best. This indicates the importance of applying extra care when analyzing data sets with a large number of variables, and of always evaluating model performance on an independent test set.

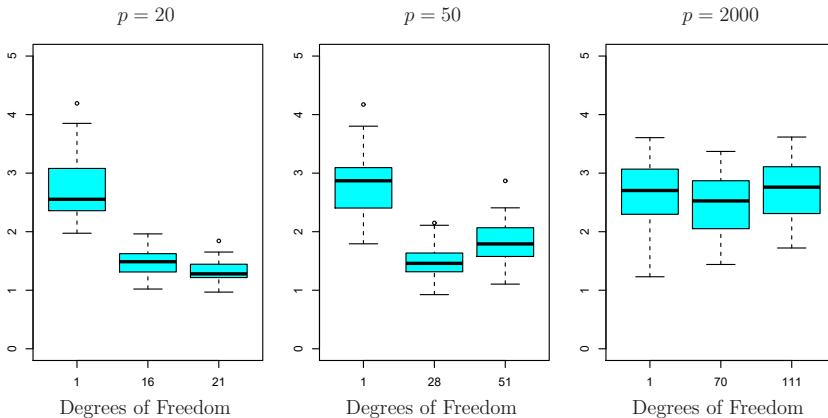


FIGURE 6.24. The lasso was performed with $n = 100$ observations and three values of p , the number of features. Of the p features, 20 were associated with the response. The boxplots show the test MSEs that result using three different values of the tuning parameter λ in (6.7). For ease of interpretation, rather than reporting λ , the degrees of freedom are reported; for the lasso this turns out to be simply the number of estimated non-zero coefficients. When $p = 20$, the lowest test MSE was obtained with the smallest amount of regularization. When $p = 50$, the lowest test MSE was achieved when there is a substantial amount of regularization. When $p = 2,000$ the lasso performed poorly regardless of the amount of regularization, due to the fact that only 20 of the 2,000 features truly are associated with the outcome.

In Section 6.1.3, we saw a number of approaches for adjusting the training set RSS or R^2 in order to account for the number of variables used to fit a least squares model. Unfortunately, the C_p , AIC, and BIC approaches are not appropriate in the high-dimensional setting, because estimating $\hat{\sigma}^2$ is problematic. (For instance, the formula for $\hat{\sigma}^2$ from Chapter 3 yields an estimate $\hat{\sigma}^2 = 0$ in this setting.) Similarly, problems arise in the application of adjusted R^2 in the high-dimensional setting, since one can easily obtain a model with an adjusted R^2 value of 1. Clearly, alternative approaches that are better-suited to the high-dimensional setting are required.

6.4.3 Regression in High Dimensions

It turns out that many of the methods seen in this chapter for fitting less flexible least squares models, such as forward stepwise selection, ridge regression, the lasso, and principal components regression, are particularly useful for performing regression in the high-dimensional setting. Essentially, these approaches avoid overfitting by using a less flexible fitting approach than least squares.

Figure 6.24 illustrates the performance of the lasso in a simple simulated example. There are $p = 20, 50$, or 2,000 features, of which 20 are truly associated with the outcome. The lasso was performed on $n = 100$ training observations, and the mean squared error was evaluated on an independent test set. As the number of features increases, the test set error increases. When $p = 20$, the lowest validation set error was achieved when λ in

(6.7) was small; however, when p was larger then the lowest validation set error was achieved using a larger value of λ . In each boxplot, rather than reporting the values of λ used, the *degrees of freedom* of the resulting lasso solution is displayed; this is simply the number of non-zero coefficient estimates in the lasso solution, and is a measure of the flexibility of the lasso fit. Figure 6.24 highlights three important points: (1) regularization or shrinkage plays a key role in high-dimensional problems, (2) appropriate tuning parameter selection is crucial for good predictive performance, and (3) the test error tends to increase as the dimensionality of the problem (i.e. the number of features or predictors) increases, unless the additional features are truly associated with the response.

The third point above is in fact a key principle in the analysis of high-dimensional data, which is known as the *curse of dimensionality*. One might think that as the number of features used to fit a model increases, the quality of the fitted model will increase as well. However, comparing the left-hand and right-hand panels in Figure 6.24, we see that this is not necessarily the case: in this example, the test set MSE almost doubles as p increases from 20 to 2,000. In general, *adding additional signal features that are truly associated with the response will improve the fitted model*, in the sense of leading to a reduction in test set error. However, adding noise features that are not truly associated with the response will lead to a deterioration in the fitted model, and consequently an increased test set error. This is because noise features increase the dimensionality of the problem, exacerbating the risk of overfitting (since noise features may be assigned nonzero coefficients due to chance associations with the response on the training set) without any potential upside in terms of improved test set error. Thus, we see that new technologies that allow for the collection of measurements for thousands or millions of features are a double-edged sword: they can lead to improved predictive models if these features are in fact relevant to the problem at hand, but will lead to worse results if the features are not relevant. Even if they are relevant, the variance incurred in fitting their coefficients may outweigh the reduction in bias that they bring.

curse of dimensionality

6.4.4 Interpreting Results in High Dimensions

When we perform the lasso, ridge regression, or other regression procedures in the high-dimensional setting, we must be quite cautious in the way that we report the results obtained. In Chapter 3, we learned about *multicollinearity*, the concept that the variables in a regression might be correlated with each other. In the high-dimensional setting, the multicollinearity problem is extreme: any variable in the model can be written as a linear combination of all of the other variables in the model. Essentially, this means that we can never know exactly which variables (if any) truly are predictive of the outcome, and we can never identify the *best* coefficients for use in the regression. At most, we can hope to assign large regression coefficients to variables that are correlated with the variables that truly are predictive of the outcome.

For instance, suppose that we are trying to predict blood pressure on the basis of half a million SNPs, and that forward stepwise selection indicates that 17 of those SNPs lead to a good predictive model on the training data. It would be incorrect to conclude that these 17 SNPs predict blood pressure more effectively than the other SNPs not included in the model. There are likely to be many sets of 17 SNPs that would predict blood pressure just as well as the selected model. If we were to obtain an independent data set and perform forward stepwise selection on that data set, we would likely obtain a model containing a different, and perhaps even non-overlapping, set of SNPs. This does not detract from the value of the model obtained—for instance, the model might turn out to be very effective in predicting blood pressure on an independent set of patients, and might be clinically useful for physicians. But we must be careful not to overstate the results obtained, and to make it clear that what we have identified is simply *one of many possible models* for predicting blood pressure, and that it must be further validated on independent data sets.

It is also important to be particularly careful in reporting errors and measures of model fit in the high-dimensional setting. We have seen that when $p > n$, it is easy to obtain a useless model that has zero residuals. Therefore, one should *never* use sum of squared errors, p-values, R^2 statistics, or other traditional measures of model fit on the training data as evidence of a good model fit in the high-dimensional setting. For instance, as we saw in Figure 6.23, one can easily obtain a model with $R^2 = 1$ when $p > n$. Reporting this fact might mislead others into thinking that a statistically valid and useful model has been obtained, whereas in fact this provides absolutely no evidence of a compelling model. It is important to instead report results on an independent test set, or cross-validation errors. For instance, the MSE or R^2 on an independent test set is a valid measure of model fit, but the MSE on the training set certainly is not.

6.5 Lab: Linear Models and Regularization Methods

In this lab we implement many of the techniques discussed in this chapter. We import some of our libraries at this top level.

```
In [1]: import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
from statsmodels.api import OLS
import sklearn.model_selection as skm
import sklearn.linear_model as skl
from sklearn.preprocessing import StandardScaler
from ISLP import load_data
from ISLP.models import ModelSpec as MS
from functools import partial
```

We again collect the new imports needed for this lab.

```
In [2]: from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
```

10

Deep Learning



This chapter covers the important topic of *deep learning*. At the time of writing (2020), deep learning is a very active area of research in the machine learning and artificial intelligence communities. The cornerstone of deep learning is the *neural network*.

deep
learning

neural
network

Neural networks rose to fame in the late 1980s. There was a lot of excitement and a certain amount of hype associated with this approach, and they were the impetus for the popular *Neural Information Processing Systems* meetings (NeurIPS, formerly NIPS) held every year, typically in exotic places like ski resorts. This was followed by a synthesis stage, where the properties of neural networks were analyzed by machine learners, mathematicians and statisticians; algorithms were improved, and the methodology stabilized. Then along came SVMs, boosting, and random forests, and neural networks fell somewhat from favor. Part of the reason was that neural networks required a lot of tinkering, while the new methods were more automatic. Also, on many problems the new methods outperformed poorly-trained neural networks. This was the *status quo* for the first decade in the new millennium.

All the while, though, a core group of neural-network enthusiasts were pushing their technology harder on ever-larger computing architectures and data sets. Neural networks resurfaced after 2010 with the new name *deep learning*, with new architectures, additional bells and whistles, and a string of success stories on some niche problems such as image and video classification, speech and text modeling. Many in the field believe that the major reason for these successes is the availability of ever-larger training datasets, made possible by the wide-scale use of digitization in science and industry.

In this chapter we discuss the basics of neural networks and deep learning, and then go into some of the specializations for specific problems, such as convolutional neural networks (CNNs) for image classification, and recurrent neural networks (RNNs) for time series and other sequences. We

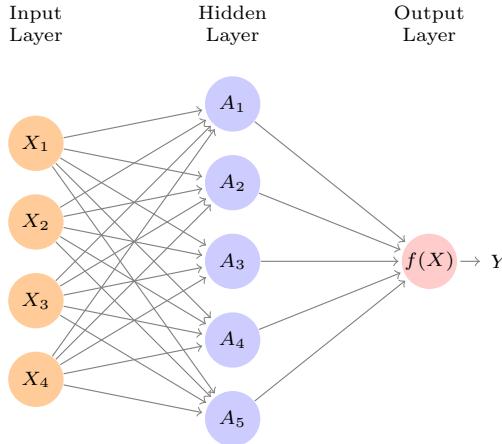


FIGURE 10.1. Neural network with a single hidden layer. The hidden layer computes activations $A_k = h_k(X)$ that are nonlinear transformations of linear combinations of the inputs X_1, X_2, \dots, X_p . Hence these A_k are not directly observed. The functions $h_k(\cdot)$ are not fixed in advance, but are learned during the training of the network. The output layer is a linear model that uses these activations A_k as inputs, resulting in a function $f(X)$.

will also demonstrate these models using the **Python torch** package, along with a number of helper packages.

The material in this chapter is slightly more challenging than elsewhere in this book.

10.1 Single Layer Neural Networks

A neural network takes an input vector of p variables $X = (X_1, X_2, \dots, X_p)$ and builds a nonlinear function $f(X)$ to predict the response Y . We have built nonlinear prediction models in earlier chapters, using trees, boosting and generalized additive models. What distinguishes neural networks from these methods is the particular *structure* of the model. Figure 10.1 shows a simple *feed-forward neural network* for modeling a quantitative response using $p = 4$ predictors. In the terminology of neural networks, the four features X_1, \dots, X_4 make up the units in the *input layer*. The arrows indicate that each of the inputs from the input layer feeds into each of the K *hidden units* (we get to pick K ; here we chose 5). The neural network model has the form

$$\begin{aligned} f(X) &= \beta_0 + \sum_{k=1}^K \beta_k h_k(X) \\ &= \beta_0 + \sum_{k=1}^K \beta_k g(w_{k0} + \sum_{j=1}^p w_{kj} X_j). \end{aligned} \tag{10.1}$$

It is built up here in two steps. First the K *activations* A_k , $k = 1, \dots, K$, in the hidden layer are computed as functions of the input features X_1, \dots, X_p ,

$$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^p w_{kj} X_j), \tag{10.2}$$

feed-forward
neural
network
input layer
hidden units

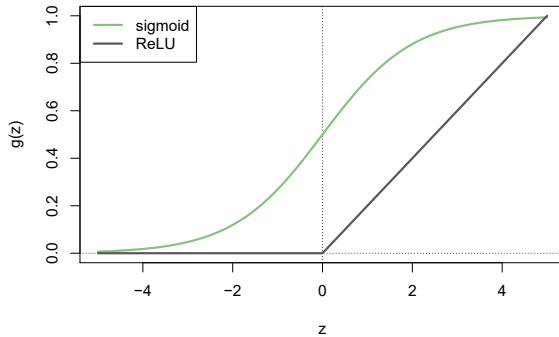


FIGURE 10.2. Activation functions. The piecewise-linear ReLU function is popular for its efficiency and computability. We have scaled it down by a factor of five for ease of comparison.

where $g(z)$ is a nonlinear *activation function* that is specified in advance. We can think of each A_k as a different transformation $h_k(X)$ of the original features, much like the basis functions of Chapter 7. These K activations from the hidden layer then feed into the output layer, resulting in

$$f(X) = \beta_0 + \sum_{k=1}^K \beta_k A_k, \quad (10.3)$$

a linear regression model in the $K = 5$ activations. All the parameters β_0, \dots, β_K and w_{10}, \dots, w_{Kp} need to be estimated from data. In the early instances of neural networks, the *sigmoid* activation function was favored,

$$g(z) = \frac{e^z}{1 + e^z} = \frac{1}{1 + e^{-z}}, \quad (10.4)$$

which is the same function used in logistic regression to convert a linear function into probabilities between zero and one (see Figure 10.2). The preferred choice in modern neural networks is the *ReLU* (*rectified linear unit*) activation function, which takes the form

$$g(z) = (z)_+ = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise.} \end{cases} \quad (10.5)$$

A ReLU activation can be computed and stored more efficiently than a sigmoid activation. Although it thresholds at zero, because we apply it to a linear function (10.2) the constant term w_{k0} will shift this inflection point.

So in words, the model depicted in Figure 10.1 derives five new features by computing five different linear combinations of X , and then squashes each through an activation function $g(\cdot)$ to transform it. The final model is linear in these derived variables.

The name *neural network* originally derived from thinking of these hidden units as analogous to neurons in the brain — values of the activations $A_k = h_k(X)$ close to one are *firing*, while those close to zero are *silent* (using the sigmoid activation function).

The nonlinearity in the activation function $g(\cdot)$ is essential, since without it the model $f(X)$ in (10.1) would collapse into a simple linear model in

activation
function

sigmoid

ReLU
rectified
linear unit

X_1, \dots, X_p . Moreover, having a nonlinear activation function allows the model to capture complex nonlinearities and interaction effects. Consider a very simple example with $p = 2$ input variables $X = (X_1, X_2)$, and $K = 2$ hidden units $h_1(X)$ and $h_2(X)$ with $g(z) = z^2$. We specify the other parameters as

$$\begin{aligned}\beta_0 &= 0, & \beta_1 &= \frac{1}{4}, & \beta_2 &= -\frac{1}{4}, \\ w_{10} &= 0, & w_{11} &= 1, & w_{12} &= 1, \\ w_{20} &= 0, & w_{21} &= 1, & w_{22} &= -1.\end{aligned}\tag{10.6}$$

From (10.2), this means that

$$\begin{aligned}h_1(X) &= (0 + X_1 + X_2)^2, \\ h_2(X) &= (0 + X_1 - X_2)^2.\end{aligned}\tag{10.7}$$

Then plugging (10.7) into (10.1), we get

$$\begin{aligned}f(X) &= 0 + \frac{1}{4} \cdot (0 + X_1 + X_2)^2 - \frac{1}{4} \cdot (0 + X_1 - X_2)^2 \\ &= \frac{1}{4} [(X_1 + X_2)^2 - (X_1 - X_2)^2] \\ &= X_1 X_2.\end{aligned}\tag{10.8}$$

So the sum of two nonlinear transformations of linear functions can give us an interaction! In practice we would not use a quadratic function for $g(z)$, since we would always get a second-degree polynomial in the original coordinates X_1, \dots, X_p . The sigmoid or ReLU activations do not have such a limitation.

Fitting a neural network requires estimating the unknown parameters in (10.1). For a quantitative response, typically squared-error loss is used, so that the parameters are chosen to minimize

$$\sum_{i=1}^n (y_i - f(x_i))^2.\tag{10.9}$$

Details about how to perform this minimization are provided in Section 10.7.

10.2 Multilayer Neural Networks

Modern neural networks typically have more than one hidden layer, and often many units per layer. In theory a single hidden layer with a large number of units has the ability to approximate most functions. However, the learning task of discovering a good solution is made much easier with multiple layers each of modest size.

We will illustrate a large dense network on the famous and publicly available **MNIST** handwritten digit dataset.¹ Figure 10.3 shows examples of these digits. The idea is to build a model to classify the images into their correct digit class 0–9. Every image has $p = 28 \times 28 = 784$ pixels, each of which is an eight-bit grayscale value between 0 and 255 representing

¹See LeCun, Cortes, and Burges (2010) “The MNIST database of handwritten digits”, available at <http://yann.lecun.com/exdb/mnist>.

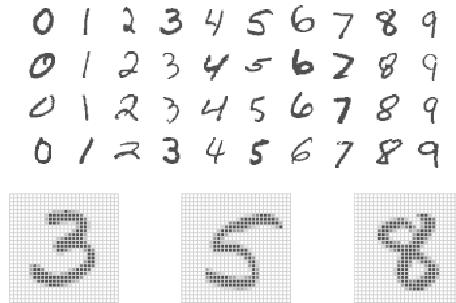


FIGURE 10.3. Examples of handwritten digits from the **MNIST** corpus. Each grayscale image has 28×28 pixels, each of which is an eight-bit number (0–255) which represents how dark that pixel is. The first 3, 5, and 8 are enlarged to show their 784 individual pixel values.

the relative amount of the written digit in that tiny square.² These pixels are stored in the input vector X (in, say, column order). The output is the class label, represented by a vector $Y = (Y_0, Y_1, \dots, Y_9)$ of 10 dummy variables, with a one in the position corresponding to the label, and zeros elsewhere. In the machine learning community, this is known as *one-hot encoding*. There are 60,000 training images, and 10,000 test images.

On a historical note, digit recognition problems were the catalyst that accelerated the development of neural network technology in the late 1980s at AT&T Bell Laboratories and elsewhere. Pattern recognition tasks of this kind are relatively simple for humans. Our visual system occupies a large fraction of our brains, and good recognition is an evolutionary force for survival. These tasks are not so simple for machines, and it has taken more than 30 years to refine the neural-network architectures to match human performance.

Figure 10.4 shows a multilayer network architecture that works well for solving the digit-classification task. It differs from Figure 10.1 in several ways:

- It has two hidden layers L_1 (256 units) and L_2 (128 units) rather than one. Later we will see a network with seven hidden layers.
- It has ten output variables, rather than one. In this case the ten variables really represent a single qualitative variable and so are quite dependent. (We have indexed them by the digit class 0–9 rather than 1–10, for clarity.) More generally, in *multi-task learning* one can predict different responses simultaneously with a single network; they all have a say in the formation of the hidden layers.
- The loss function used for training the network is tailored for the multiclass classification task.

one-hot
encoding

multi-task
learning

²In the analog-to-digital conversion process, only part of the written numeral may fall in the square representing a particular pixel.

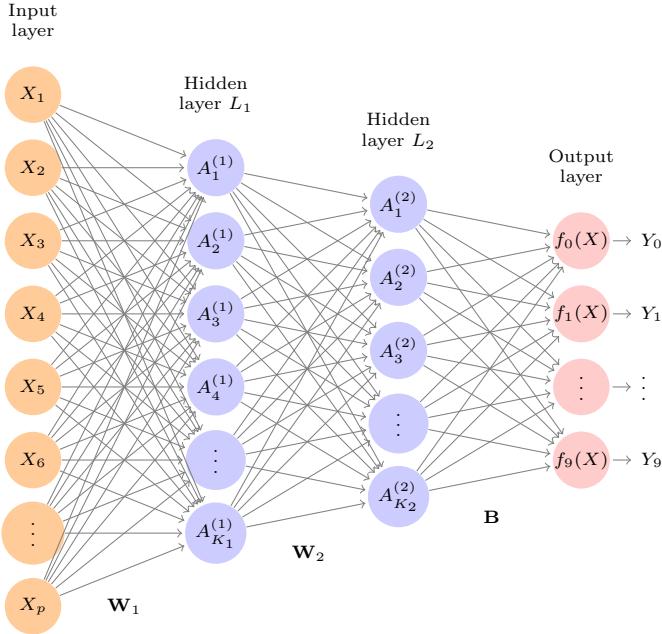


FIGURE 10.4. Neural network diagram with two hidden layers and multiple outputs, suitable for the MNIST handwritten-digit problem. The input layer has $p = 784$ units, the two hidden layers $K_1 = 256$ and $K_2 = 128$ units respectively, and the output layer 10 units. Along with intercepts (referred to as biases in the deep-learning community) this network has 235,146 parameters (referred to as weights).

The first hidden layer is as in (10.2), with

$$\begin{aligned} A_k^{(1)} &= h_k^{(1)}(X) \\ &= g(w_{k0}^{(1)} + \sum_{j=1}^p w_{kj}^{(1)} X_j) \end{aligned} \quad (10.10)$$

for $k = 1, \dots, K_1$. The second hidden layer treats the activations $A_k^{(1)}$ of the first hidden layer as inputs and computes new activations

$$\begin{aligned} A_\ell^{(2)} &= h_\ell^{(2)}(X) \\ &= g(w_{\ell 0}^{(2)} + \sum_{k=1}^{K_1} w_{\ell k}^{(2)} A_k^{(1)}) \end{aligned} \quad (10.11)$$

for $\ell = 1, \dots, K_2$. Notice that each of the activations in the second layer $A_\ell^{(2)} = h_\ell^{(2)}(X)$ is a function of the input vector X . This is the case because while they are explicitly a function of the activations $A_k^{(1)}$ from layer L_1 , these in turn are functions of X . This would also be the case with more hidden layers. Thus, through a chain of transformations, the network is able to build up fairly complex transformations of X that ultimately feed into the output layer as features.

We have introduced additional superscript notation such as $h_\ell^{(2)}(X)$ and $w_{\ell j}^{(2)}$ in (10.10) and (10.11) to indicate to which layer the activations and weights (coefficients) belong, in this case layer 2. The notation \mathbf{W}_1 in Fig-

weights

ure 10.4 represents the entire matrix of weights that feed from the input layer to the first hidden layer L_1 . This matrix will have $785 \times 256 = 200,960$ elements; there are 785 rather than 784 because we must account for the intercept or *bias* term.³

Each element $A_k^{(1)}$ feeds to the second hidden layer L_2 via the matrix of weights \mathbf{W}_2 of dimension $257 \times 128 = 32,896$.

We now get to the output layer, where we now have ten responses rather than one. The first step is to compute ten different linear models similar to our single model (10.1),

$$\begin{aligned} Z_m &= \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} h_{\ell}^{(2)}(X) \\ &= \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} A_{\ell}^{(2)}, \end{aligned} \tag{10.12}$$

for $m = 0, 1, \dots, 9$. The matrix \mathbf{B} stores all $129 \times 10 = 1,290$ of these weights.

If these were all separate quantitative responses, we would simply set each $f_m(X) = Z_m$ and be done. However, we would like our estimates to represent class probabilities $f_m(X) = \Pr(Y = m|X)$, just like in multinomial logistic regression in Section 4.3.5. So we use the special *softmax* activation function (see (4.13) on page 145),

$$f_m(X) = \Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{\ell=0}^9 e^{Z_{\ell}}}, \tag{10.13}$$

for $m = 0, 1, \dots, 9$. This ensures that the 10 numbers behave like probabilities (non-negative and sum to one). Even though the goal is to build a classifier, our model actually estimates a probability for each of the 10 classes. The classifier then assigns the image to the class with the highest probability.

To train this network, since the response is qualitative, we look for coefficient estimates that minimize the negative multinomial log-likelihood

$$-\sum_{i=1}^n \sum_{m=0}^9 y_{im} \log(f_m(x_i)), \tag{10.14}$$

also known as the *cross-entropy*. This is a generalization of the criterion (4.5) for two-class logistic regression. Details on how to minimize this objective are given in Section 10.7. If the response were quantitative, we would instead minimize squared-error loss as in (10.9).

cross-
entropy

Table 10.1 compares the test performance of the neural network with two simple models presented in Chapter 4 that make use of linear decision boundaries: multinomial logistic regression and linear discriminant analysis. The improvement of neural networks over both of these linear methods is dramatic: the network with dropout regularization achieves a test error rate below 2% on the 10,000 test images. (We describe dropout regularization in Section 10.7.3.) In Section 10.9.2 of the lab, we present the code for fitting this model, which runs in just over two minutes on a laptop computer.

³The use of “weights” for coefficients and “bias” for the intercepts w_{k0} in (10.2) is popular in the machine learning community; this use of bias is not to be confused with the “bias-variance” usage elsewhere in this book.

Method	Test Error
Neural Network + Ridge Regularization	2.3%
Neural Network + Dropout Regularization	1.8%
Multinomial Logistic Regression	7.2%
Linear Discriminant Analysis	12.7%

TABLE 10.1. Test error rate on the MNIST data, for neural networks with two forms of regularization, as well as multinomial logistic regression and linear discriminant analysis. In this example, the extra complexity of the neural network leads to a marked improvement in test error.

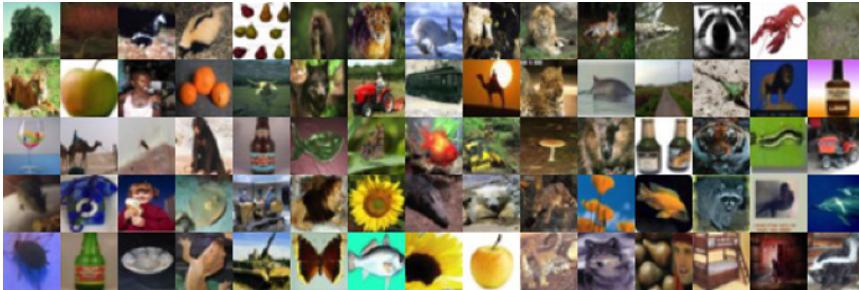


FIGURE 10.5. A sample of images from the CIFAR100 database: a collection of natural images from everyday life, with 100 different classes represented.

Adding the number of coefficients in \mathbf{W}_1 , \mathbf{W}_2 and \mathbf{B} , we get 235,146 in all, more than 33 times the number $785 \times 9 = 7,065$ needed for multinomial logistic regression. Recall that there are 60,000 images in the training set. While this might seem like a large training set, there are almost four times as many coefficients in the neural network model as there are observations in the training set! To avoid overfitting, some regularization is needed. In this example, we used two forms of regularization: ridge regularization, which is similar to ridge regression from Chapter 6, and dropout regularization. We discuss both forms of regularization in Section 10.7.

dropout

10.3 Convolutional Neural Networks

Neural networks rebounded around 2010 with big successes in image classification. Around that time, massive databases of labeled images were being accumulated, with ever-increasing numbers of classes. Figure 10.5 shows 75 images drawn from the CIFAR100 database.⁴ This database consists of 60,000 images labeled according to 20 superclasses (e.g. aquatic mammals), with five classes per superclass (beaver, dolphin, otter, seal, whale). Each image has a resolution of 32×32 pixels, with three eight-bit numbers per pixel representing red, green and blue. The numbers for each image are organized in a three-dimensional array called a *feature map*. The first two

feature map

⁴See Chapter 3 of Krizhevsky (2009) “Learning multiple layers of features from tiny images”, available at <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>.

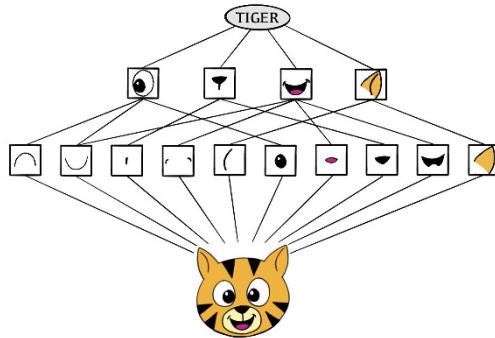


FIGURE 10.6. Schematic showing how a convolutional neural network classifies an image of a tiger. The network takes in the image and identifies local features. It then combines the local features in order to create compound features, which in this example include eyes and ears. These compound features are used to output the label “tiger”.

axes are spatial (both are 32-dimensional), and the third is the *channel* axis,⁵ representing the three colors. There is a designated training set of 50,000 images, and a test set of 10,000.

A special family of *convolutional neural networks* (CNNs) has evolved for classifying images such as these, and has shown spectacular success on a wide range of problems. CNNs mimic to some degree how humans classify images, by recognizing specific features or patterns anywhere in the image that distinguish each particular object class. In this section we give a brief overview of how they work.

Figure 10.6 illustrates the idea behind a convolutional neural network on a cartoon image of a tiger.⁶

The network first identifies low-level features in the input image, such as small edges, patches of color, and the like. These low-level features are then combined to form higher-level features, such as parts of ears, eyes, and so on. Eventually, the presence or absence of these higher-level features contributes to the probability of any given output class.

How does a convolutional neural network build up this hierarchy? It combines two specialized types of hidden layers, called *convolution* layers and *pooling* layers. Convolution layers search for instances of small patterns in the image, whereas pooling layers downsample these to select a prominent subset. In order to achieve state-of-the-art results, contemporary neural-network architectures make use of many convolution and pooling layers. We describe convolution and pooling layers next.

10.3.1 Convolution Layers

A *convolution layer* is made up of a large number of *convolution filters*, each

channel
convolutional
neural
networks

convolution
layer
convolution
filter

⁵The term *channel* is taken from the signal-processing literature. Each channel is a distinct source of information.

⁶Thanks to Elena Tuzhilina for producing the diagram and <https://www.cartooning4kids.com/> for permission to use the cartoon tiger.

of which is a template that determines whether a particular local feature is present in an image. A convolution filter relies on a very simple operation, called a *convolution*, which basically amounts to repeatedly multiplying matrix elements and then adding the results.

To understand how a convolution filter works, consider a very simple example of a 4×3 image:

$$\text{Original Image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}.$$

Now consider a 2×2 filter of the form

$$\text{Convolution Filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}.$$

When we *convolve* the image with the filter, we get the result⁷

$$\text{Convolved Image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}.$$

For instance, the top-left element comes from multiplying each element in the 2×2 filter by the corresponding element in the top left 2×2 portion of the image, and adding the results. The other elements are obtained in a similar way: the convolution filter is applied to every 2×2 submatrix of the original image in order to obtain the convolved image. If a 2×2 submatrix of the original image resembles the convolution filter, then it will have a *large* value in the convolved image; otherwise, it will have a *small* value. Thus, *the convolved image highlights regions of the original image that resemble the convolution filter*. We have used 2×2 as an example; in general convolution filters are small $\ell_1 \times \ell_2$ arrays, with ℓ_1 and ℓ_2 small positive integers that are not necessarily equal.

Figure 10.7 illustrates the application of two convolution filters to a 192×179 image of a tiger, shown on the left-hand side.⁸ Each convolution filter is a 15×15 image containing mostly zeros (black), with a narrow strip of ones (white) oriented either vertically or horizontally within the image. When each filter is convolved with the image of the tiger, areas of the tiger that resemble the filter (i.e. that have either horizontal or vertical stripes or edges) are given large values, and areas of the tiger that do not resemble the feature are given small values. The convolved images are displayed on the right-hand side. We see that the horizontal stripe filter picks out horizontal stripes and edges in the original image, whereas the vertical stripe filter picks out vertical stripes and edges in the original image.

⁷The convolved image is smaller than the original image because its dimension is given by the number of 2×2 submatrices in the original image. Note that 2×2 is the dimension of the convolution filter. If we want the convolved image to have the same dimension as the original image, then padding can be applied.

⁸The tiger image used in Figures 10.7–10.9 was obtained from the public domain image resource <https://www.needpix.com/>.

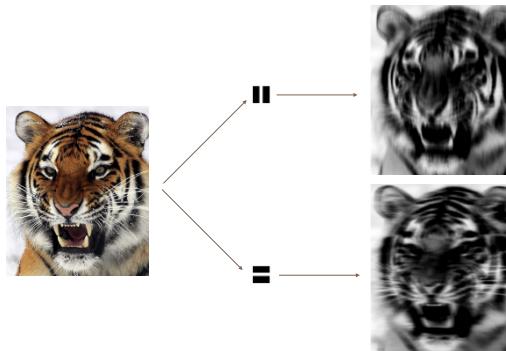


FIGURE 10.7. Convolution filters find local features in an image, such as edges and small shapes. We begin with the image of the tiger shown on the left, and apply the two small convolution filters in the middle. The convolved images highlight areas in the original image where details similar to the filters are found. Specifically, the top convolved image highlights the tiger’s vertical stripes, whereas the bottom convolved image highlights the tiger’s horizontal stripes. We can think of the original image as the input layer in a convolutional neural network, and the convolved images as the units in the first hidden layer.

We have used a large image and two large filters in Figure 10.7 for illustration. For the **CIFAR100** database there are 32×32 color pixels per image, and we use 3×3 convolution filters.

In a convolution layer, we use a whole bank of filters to pick out a variety of differently-oriented edges and shapes in the image. Using predefined filters in this way is standard practice in image processing. By contrast, with CNNs the filters are *learned* for the specific classification task. We can think of the filter weights as the parameters going from an input layer to a hidden layer, with one hidden unit for each pixel in the convolved image. This is in fact the case, though the parameters are highly structured and constrained (see Exercise 4 for more details). They operate on localized patches in the input image (so there are many structural zeros), and the same weights in a given filter are reused for all possible patches in the image (so the weights are constrained).⁹

We now give some additional details.

- Since the input image is in color, it has three channels represented by a three-dimensional feature map (array). Each channel is a two-dimensional (32×32) feature map — one for red, one for green, and one for blue. A single convolution filter will also have three channels, one per color, each of dimension 3×3 , with potentially different filter weights. The results of the three convolutions are summed to form a two-dimensional output feature map. Note that at this point the color information has been used, and is not passed on to subsequent layers except through its role in the convolution.

⁹This used to be called *weight sharing* in the early years of neural networks.

- If we use K different convolution filters at this first hidden layer, we get K two-dimensional output feature maps, which together are treated as a single three-dimensional feature map. We view each of the K output feature maps as a separate channel of information, so now we have K channels in contrast to the three color channels of the original input feature map. The three-dimensional feature map is just like the activations in a hidden layer of a simple neural network, except organized and produced in a spatially structured way.
- We typically apply the ReLU activation function (10.5) to the convolved image. This step is sometimes viewed as a separate layer in the convolutional neural network, in which case it is referred to as a *detector layer*.

detector
layer

10.3.2 Pooling Layers

A *pooling* layer provides a way to condense a large image into a smaller summary image. While there are a number of possible ways to perform pooling, the *max pooling* operation summarizes each non-overlapping 2×2 block of pixels in an image using the maximum value in the block. This reduces the size of the image by a factor of two in each direction, and it also provides some *location invariance*: i.e. as long as there is a large value in one of the four pixels in the block, the whole block registers as a large value in the reduced image.

pooling

Here is a simple example of max pooling:

$$\text{Max pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}.$$

10.3.3 Architecture of a Convolutional Neural Network

So far we have defined a single convolution layer — each filter produces a new two-dimensional feature map. The number of convolution filters in a convolution layer is akin to the number of units at a particular hidden layer in a fully-connected neural network of the type we saw in Section 10.2. This number also defines the number of channels in the resulting three-dimensional feature map. We have also described a pooling layer, which reduces the first two dimensions of each three-dimensional feature map. Deep CNNs have many such layers. Figure 10.8 shows a typical architecture for a CNN for the **CIFAR100** image classification task.

At the input layer, we see the three-dimensional feature map of a color image, where the channel axis represents each color by a 32×32 two-dimensional feature map of pixels. Each convolution filter produces a new channel at the first hidden layer, each of which is a 32×32 feature map (after some padding at the edges). After this first round of convolutions, we now have a new “image”; a feature map with considerably more channels than the three color input channels (six in the figure, since we used six convolution filters).

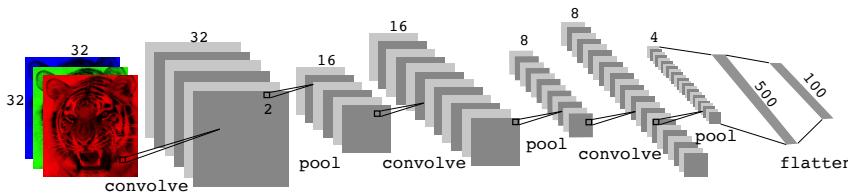


FIGURE 10.8. Architecture of a deep CNN for the **CIFAR100** classification task. Convolution layers are interspersed with 2×2 max-pool layers, which reduce the size by a factor of 2 in both dimensions.

This is followed by a max-pool layer, which reduces the size of the feature map in each channel by a factor of four: two in each dimension.

This convolve-then-pool sequence is now repeated for the next two layers. Some details are as follows:

- Each subsequent convolve layer is similar to the first. It takes as input the three-dimensional feature map from the previous layer and treats it like a single multi-channel image. Each convolution filter learned has as many channels as this feature map.
- Since the channel feature maps are reduced in size after each pool layer, we usually increase the number of filters in the next convolve layer to compensate.
- Sometimes we repeat several convolve layers before a pool layer. This effectively increases the dimension of the filter.

These operations are repeated until the pooling has reduced each channel feature map down to just a few pixels in each dimension. At this point the three-dimensional feature maps are *flattened* — the pixels are treated as separate units — and fed into one or more fully-connected layers before reaching the output layer, which is a *softmax activation* for the 100 classes (as in (10.13)).

There are many tuning parameters to be selected in constructing such a network, apart from the number, nature, and sizes of each layer. Dropout learning can be used at each layer, as well as lasso or ridge regularization (see Section 10.7). The details of constructing a convolutional neural network can seem daunting. Fortunately, terrific software is available, with extensive examples and vignettes that provide guidance on sensible choices for the parameters. For the **CIFAR100** official test set, the best accuracy as of this writing is just above 75%, but undoubtedly this performance will continue to improve.

10.3.4 Data Augmentation

An additional important trick used with image modeling is *data augmentation*. Essentially, each training image is replicated many times, with each replicate randomly distorted in a natural way such that human recognition is unaffected. Figure 10.9 shows some examples. Typical distortions are

data augmentation



FIGURE 10.9. *Data augmentation. The original image (leftmost) is distorted in natural ways to produce different images with the same class label. These distortions do not fool humans, and act as a form of regularization when fitting the CNN.*

zoom, horizontal and vertical shift, shear, small rotations, and in this case horizontal flips. At face value this is a way of increasing the training set considerably with somewhat different examples, and thus protects against overfitting. In fact we can see this as a form of regularization: we build a cloud of images around each original image, all with the same label. This kind of fattening of the data is similar in spirit to ridge regularization.

We will see in Section 10.7.2 that the stochastic gradient descent algorithms for fitting deep learning models repeatedly process randomly-selected batches of, say, 128 training images at a time. This works hand-in-glove with augmentation, because we can distort each image in the batch on the fly, and hence do not have to store all the new images.

10.3.5 Results Using a Pretrained Classifier

Here we use an industry-level pretrained classifier to predict the class of some new images. The `resnet50` classifier is a convolutional neural network that was trained using the `imagenet` data set, which consists of millions of images that belong to an ever-growing number of categories.¹⁰ Figure 10.10 demonstrates the performance of `resnet50` on six photographs (private collection of one of the authors).¹¹ The CNN does a reasonable job classifying the hawk in the second image. If we zoom out as in the third image, it gets confused and chooses the fountain rather than the hawk. In the final image a “jacamar” is a tropical bird from South and Central America with similar coloring to the South African Cape Weaver. We give more details on this example in Section 10.9.4.

Much of the work in fitting a CNN is in learning the convolution filters at the hidden layers; these are the coefficients of a CNN. For models fit to massive corpora such as `imagenet` with many classes, the output of these filters can serve as features for general natural-image classification problems. One can use these pretrained hidden layers for new problems with much smaller training sets (a process referred to as *weight freezing*), and just train the last few layers of the network, which requires much less data.

weight
freezing

¹⁰For more information about `resnet50`, see He, Zhang, Ren, and Sun (2015) “Deep residual learning for image recognition”, <https://arxiv.org/abs/1512.03385>. For details about `imagenet`, see Russakovsky, Deng, et al. (2015) “ImageNet Large Scale Visual Recognition Challenge”, in *International Journal of Computer Vision*.

¹¹These `resnet` results can change with time, since the publicly-trained model gets updated periodically.

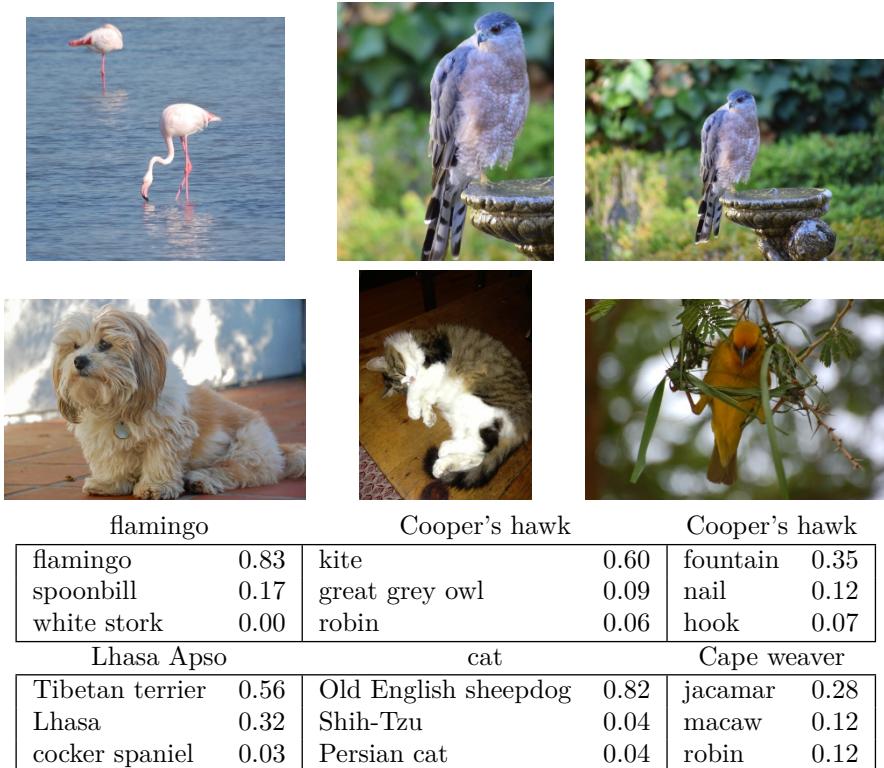


FIGURE 10.10. Classification of six photographs using the `resnet50` CNN trained on the `imagenet` corpus. The table below the images displays the true (intended) label at the top of each panel, and the top three choices of the classifier (out of 100). The numbers are the estimated probabilities for each choice. (A kite is a raptor, but not a hawk.)

The vignettes and book¹² that accompany the `keras` package give more details on such applications.

10.4 Document Classification

In this section we introduce a new type of example that has important applications in industry and science: predicting attributes of documents. Examples of documents include articles in medical journals, Reuters news feeds, emails, tweets, and so on. Our example will be `IMDb` (Internet Movie Database) ratings — short documents where viewers have written critiques of movies.¹³ The response in this case is the `sentiment` of the review, which will be *positive* or *negative*.

¹²Deep Learning with R by F. Chollet and J.J. Allaire, 2018, Manning Publications.

¹³For details, see Maas et al. (2011) “Learning word vectors for sentiment analysis”, in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150.

Here is the beginning of a rather amusing negative review:

This has to be one of the worst films of the 1990s. When my friends & I were watching this film (being the target audience it was aimed at) we just sat & watched the first half an hour with our jaws touching the floor at how bad it really was. The rest of the time, everyone else in the theater just started talking to each other, leaving or generally crying into their popcorn . . .

Each review can be a different length, include slang or non-words, have spelling errors, etc. We need to find a way to *featurize* such a document. This is modern parlance for defining a set of predictors.

featurize

The simplest and most common featurization is the *bag-of-words* model. We score each document for the presence or absence of each of the words in a language dictionary — in this case an English dictionary. If the dictionary contains M words, that means for each document we create a binary feature vector of length M , and score a 1 for every word present, and 0 otherwise. That can be a very wide feature vector, so we limit the dictionary — in this case to the 10,000 most frequently occurring words in the training corpus of 25,000 reviews. Fortunately there are nice tools for doing this automatically. Here is the beginning of a positive review that has been redacted in this way:

bag-of-words

<START> this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could just imagine being there robert <UNK> is an amazing actor and now the same being director <UNK> father came from the same scottish island as myself so i loved . . .

Here we can see many words have been omitted, and some unknown words (UNK) have been marked as such. With this reduction the binary feature vector has length 10,000, and consists mostly of 0's and a smattering of 1's in the positions corresponding to words that are present in the document. We have a training set and test set, each with 25,000 examples, and each balanced with regard to *sentiment*. The resulting training feature matrix \mathbf{X} has dimension $25,000 \times 10,000$, but only 1.3% of the binary entries are non-zero. We call such a matrix sparse, because most of the values are the same (zero in this case); it can be stored efficiently in *sparse matrix format*.¹⁴ There are a variety of ways to account for the document length; here we only score a word as in or out of the document, but for example one could instead record the relative frequency of words. We split off a validation set of size 2,000 from the 25,000 training observations (for model tuning), and fit two model sequences:

sparse matrix format

- A lasso logistic regression using the `glmnet` package;
- A two-class neural network with two hidden layers, each with 16 ReLU units.

¹⁴Rather than store the whole matrix, we can store instead the location and values for the nonzero entries. In this case, since the nonzero entries are all 1, just the locations are stored.

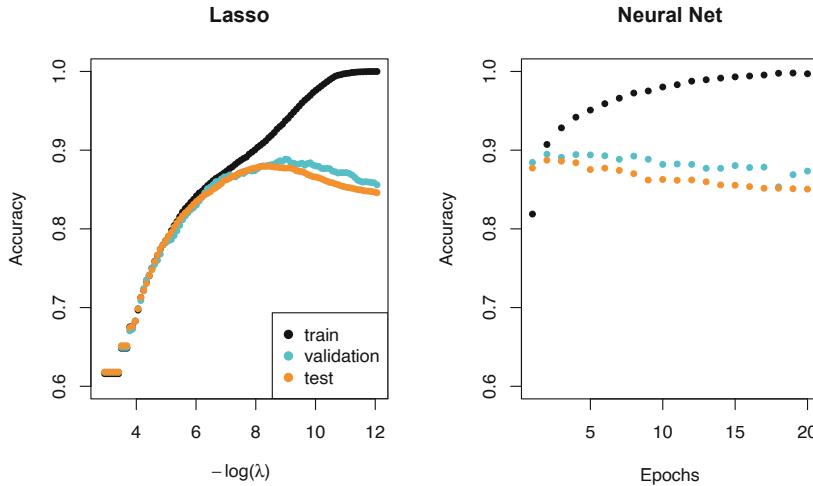


FIGURE 10.11. Accuracy of the lasso and a two-hidden-layer neural network on the **IMDb** data. For the lasso, the x -axis displays $-\log(\lambda)$, while for the neural network it displays epochs (number of times the fitting algorithm passes through the training set). Both show a tendency to overfit, and achieve approximately the same test accuracy.

Both methods produce a sequence of solutions. The lasso sequence is indexed by the regularization parameter λ . The neural-net sequence is indexed by the number of gradient-descent iterations used in the fitting, as measured by training epochs or passes through the training set (Section 10.7). Notice that the training accuracy in Figure 10.11 (black points) increases monotonically in both cases. We can use the validation error to pick a good solution from each sequence (blue points in the plots), which would then be used to make predictions on the test data set.

Note that a two-class neural network amounts to a nonlinear logistic regression model. From (10.12) and (10.13) we can see that

$$\begin{aligned} \log \left(\frac{\Pr(Y=1|X)}{\Pr(Y=0|X)} \right) &= Z_1 - Z_0 \\ &= (\beta_{10} - \beta_{00}) + \sum_{\ell=1}^{K_2} (\beta_{1\ell} - \beta_{0\ell}) A_\ell^{(2)}. \end{aligned} \quad (10.15)$$

(This shows the redundancy in the softmax function; for K classes we really only need to estimate $K-1$ sets of coefficients. See Section 4.3.5.) In Figure 10.11 we show *accuracy* (fraction correct) rather than classification error (fraction incorrect), the former being more popular in the machine learning community. Both models achieve a test-set accuracy of about 88%.

accuracy

The bag-of-words model summarizes a document by the words present, and ignores their context. There are at least two popular ways to take the context into account:

- The *bag-of-n-grams* model. For example, a bag of 2-grams records

bag-of-n-
grams

the consecutive co-occurrence of every distinct pair of words. “Blissfully long” can be seen as a positive phrase in a movie review, while “blissfully short” a negative.

- Treat the document as a sequence, taking account of all the words in the context of those that preceded and those that follow.

In the next section we explore models for sequences of data, which have applications in weather forecasting, speech recognition, language translation, and time-series prediction, to name a few. We continue with this [IMDb](#) example there.

10.5 Recurrent Neural Networks

Many data sources are sequential in nature, and call for special treatment when building predictive models. Examples include:

- Documents such as book and movie reviews, newspaper articles, and tweets. The sequence and relative positions of words in a document capture the narrative, theme and tone, and can be exploited in tasks such as topic classification, sentiment analysis, and language translation.
- Time series of temperature, rainfall, wind speed, air quality, and so on. We may want to forecast the weather several days ahead, or climate several decades ahead.
- Financial time series, where we track market indices, trading volumes, stock and bond prices, and exchange rates. Here prediction is often difficult, but as we will see, certain indices can be predicted with reasonable accuracy.
- Recorded speech, musical recordings, and other sound recordings. We may want to give a text transcription of a speech, or perhaps a language translation. We may want to assess the quality of a piece of music, or assign certain attributes.
- Handwriting, such as doctor’s notes, and handwritten digits such as zip codes. Here we want to turn the handwriting into digital text, or read the digits (optical character recognition).

In a *recurrent neural network* (RNN), the input object X is a *sequence*. Consider a corpus of documents, such as the collection of [IMDb](#) movie reviews. Each document can be represented as a sequence of L words, so $X = \{X_1, X_2, \dots, X_L\}$, where each X_ℓ represents a word. The order of the words, and closeness of certain words in a sentence, convey semantic meaning. RNNs are designed to accommodate and take advantage of the sequential nature of such input objects, much like convolutional neural networks accommodate the spatial structure of image inputs. The output Y can also be a sequence (such as in language translation), but often is a scalar, like the binary sentiment label of a movie review document.

recurrent
neural
network

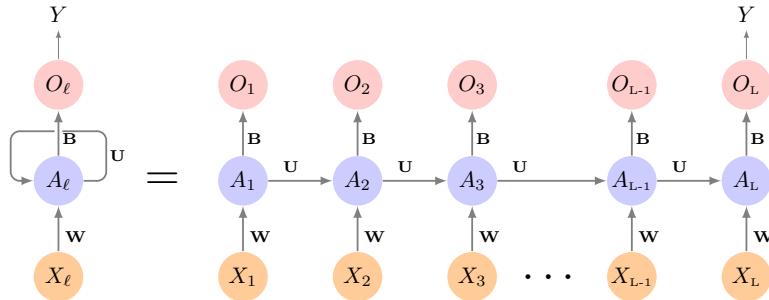


FIGURE 10.12. Schematic of a simple recurrent neural network. The input is a sequence of vectors $\{X_\ell\}_1^L$, and here the target is a single response. The network processes the input sequence X sequentially; each X_ℓ feeds into the hidden layer, which also has as input the activation vector $A_{\ell-1}$ from the previous element in the sequence, and produces the current activation vector A_ℓ . The same collections of weights \mathbf{W} , \mathbf{U} and \mathbf{B} are used as each element of the sequence is processed. The output layer produces a sequence of predictions O_ℓ from the current activation A_ℓ , but typically only the last of these, O_L , is of relevance. To the left of the equal sign is a concise representation of the network, which is unrolled into a more explicit version on the right.

Figure 10.12 illustrates the structure of a very basic RNN with a sequence $X = \{X_1, X_2, \dots, X_L\}$ as input, a simple output Y , and a hidden-layer sequence $\{A_\ell\}_1^L = \{A_1, A_2, \dots, A_L\}$. Each X_ℓ is a vector; in the document example X_ℓ could represent a one-hot encoding for the ℓ th word based on the language dictionary for the corpus (see the top panel in Figure 10.13 for a simple example). As the sequence is processed one vector X_ℓ at a time, the network updates the activations A_ℓ in the hidden layer, taking as input the vector X_ℓ and the activation vector $A_{\ell-1}$ from the previous step in the sequence. Each A_ℓ feeds into the output layer and produces a prediction O_ℓ for Y . O_L , the last of these, is the most relevant.

In detail, suppose each vector X_ℓ of the input sequence has p components $X_\ell^T = (X_{\ell 1}, X_{\ell 2}, \dots, X_{\ell p})$, and the hidden layer consists of K units $A_\ell^T = (A_{\ell 1}, A_{\ell 2}, \dots, A_{\ell K})$. As in Figure 10.4, we represent the collection of $K \times (p+1)$ shared weights w_{kj} for the input layer by a matrix \mathbf{W} , and similarly \mathbf{U} is a $K \times K$ matrix of the weights u_{ks} for the hidden-to-hidden layers, and \mathbf{B} is a $K + 1$ vector of weights β_k for the output layer. Then

$$A_{\ell k} = g\left(w_{k0} + \sum_{j=1}^p w_{kj} X_{\ell j} + \sum_{s=1}^K u_{ks} A_{\ell-1,s}\right), \quad (10.16)$$

and the output O_ℓ is computed as

$$O_\ell = \beta_0 + \sum_{k=1}^K \beta_k A_{\ell k} \quad (10.17)$$

for a quantitative response, or with an additional sigmoid activation function for a binary response, for example. Here $g(\cdot)$ is an activation function such as ReLU. Notice that the same weights \mathbf{W} , \mathbf{U} and \mathbf{B} are used as we

process each element in the sequence, i.e. they are not functions of ℓ . This is a form of *weight sharing* used by RNNs, and similar to the use of filters in convolutional neural networks (Section 10.3.1.) As we proceed from beginning to end, the activations A_ℓ accumulate a history of what has been seen before, so that the learned context can be used for prediction.

weight sharing

For regression problems the loss function for an observation (X, Y) is

$$(Y - O_L)^2, \quad (10.18)$$

which only references the final output $O_L = \beta_0 + \sum_{k=1}^K \beta_k A_{Lk}$. Thus O_1, O_2, \dots, O_{L-1} are not used. When we fit the model, each element X_ℓ of the input sequence X contributes to O_L via the chain (10.16), and hence contributes indirectly to learning the shared parameters \mathbf{W} , \mathbf{U} and \mathbf{B} via the loss (10.18). With n input sequence/response pairs (x_i, y_i) , the parameters are found by minimizing the sum of squares

$$\sum_{i=1}^n (y_i - o_{iL})^2 = \sum_{i=1}^n \left(y_i - \left(\beta_0 + \sum_{k=1}^K \beta_k g\left(w_{k0} + \sum_{j=1}^p w_{kj} x_{iLj} + \sum_{s=1}^K u_{ks} a_{i,L-1,s} \right) \right) \right)^2. \quad (10.19)$$

Here we use lowercase letters for the observed y_i and vector sequences $x_i = \{x_{i1}, x_{i2}, \dots, x_{iL}\}$,¹⁵ as well as the derived activations.

Since the intermediate outputs O_ℓ are not used, one may well ask why they are there at all. First of all, they come for free, since they use the same output weights \mathbf{B} needed to produce O_L , and provide an evolving prediction for the output. Furthermore, for some learning tasks the response is also a sequence, and so the output sequence $\{O_1, O_2, \dots, O_L\}$ is explicitly needed.

When used at full strength, recurrent neural networks can be quite complex. We illustrate their use in two simple applications. In the first, we continue with the [IMDb](#) sentiment analysis of the previous section, where we process the words in the reviews sequentially. In the second application, we illustrate their use in a financial time series forecasting problem.

10.5.1 Sequential Models for Document Classification

Here we return to our classification task with the [IMDb](#) reviews. Our approach in Section 10.4 was to use the bag-of-words model. Here the plan is to use instead the sequence of words occurring in a document to make predictions about the label for the entire document.

We have, however, a dimensionality problem: each word in our document is represented by a one-hot-encoded vector (dummy variable) with 10,000 elements (one per word in the dictionary)! An approach that has become popular is to represent each word in a much lower-dimensional *embedding* space. This means that rather than representing each word by a binary vector with 9,999 zeros and a single one in some position, we will represent it instead by a set of m real numbers, none of which are typically zero. Here m is the embedding dimension, and can be in the low 100s, or even less. This means (in our case) that we need a matrix \mathbf{E} of dimension $m \times 10,000$,

embedding

¹⁵This is a sequence of vectors; each element $x_{i\ell}$ is a p -vector.

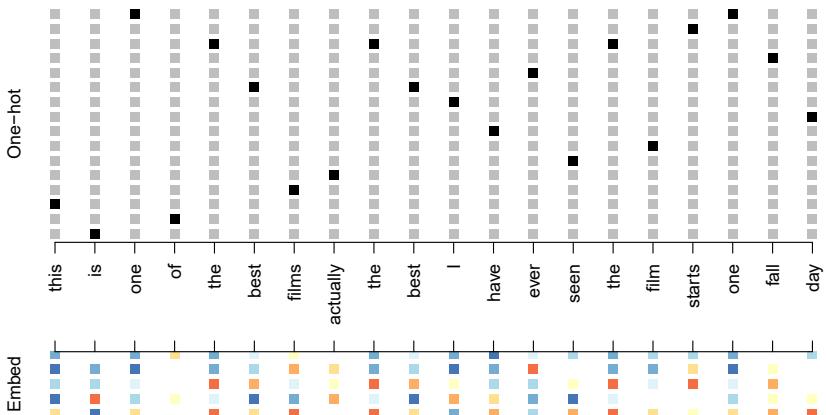


FIGURE 10.13. Depiction of a sequence of 20 words representing a single document: one-hot encoded using a dictionary of 16 words (top panel) and embedded in an m -dimensional space with $m = 5$ (bottom panel).

where each column is indexed by one of the 10,000 words in our dictionary, and the values in that column give the m coordinates for that word in the embedding space.

Figure 10.13 illustrates the idea (with a dictionary of 16 rather than 10,000, and $m = 5$). Where does \mathbf{E} come from? If we have a large corpus of labeled documents, we can have the neural network learn \mathbf{E} as part of the optimization. In this case \mathbf{E} is referred to as an *embedding layer*, and a specialized \mathbf{E} is learned for the task at hand. Otherwise we can insert a precomputed matrix \mathbf{E} in the embedding layer, a process known as *weight freezing*. Two pretrained embeddings, `word2vec` and `GloVe`, are widely used.¹⁶ These are built from a very large corpus of documents by a variant of principal components analysis (Section 12.2). The idea is that the positions of words in the embedding space preserve semantic meaning; e.g. synonyms should appear near each other.

So far, so good. Each document is now represented as a sequence of m -vectors that represents the sequence of words. The next step is to limit each document to the last L words. Documents that are shorter than L get padded with zeros upfront. So now each document is represented by a series consisting of L vectors $X = \{X_1, X_2, \dots, X_L\}$, and each X_ℓ in the sequence has m components.

We now use the RNN structure in Figure 10.12. The training corpus consists of n separate series (documents) of length L , each of which gets processed sequentially from left to right. In the process, a parallel series of hidden activation vectors A_ℓ , $\ell = 1, \dots, L$ is created as in (10.16) for each document. A_ℓ feeds into the output layer to produce the evolving prediction O_ℓ . We use the final value O_L to predict the response: the sentiment of the review.

embedding layer
weight freezing
`word2vec`
`GloVe`

¹⁶ `word2vec` is described in Mikolov, Chen, Corrado, and Dean (2013), available at <https://code.google.com/archive/p/word2vec>. `GloVe` is described in Pennington, Socher, and Manning (2014), available at <https://nlp.stanford.edu/projects/glove>.

This is a simple RNN, and has relatively few parameters. If there are K hidden units, the common weight matrix \mathbf{W} has $K \times (m + 1)$ parameters, the matrix \mathbf{U} has $K \times K$ parameters, and \mathbf{B} has $2(K + 1)$ for the two-class logistic regression as in (10.15). These are used repeatedly as we process the sequence $X = \{X_\ell\}_1^L$ from left to right, much like we use a single convolution filter to process each patch in an image (Section 10.3.1). If the embedding layer \mathbf{E} is learned, that adds an additional $m \times D$ parameters ($D = 10,000$ here), and is by far the biggest cost.

We fit the RNN as described in Figure 10.12 and the accompanying text to the **IMDb** data. The model had an embedding matrix \mathbf{E} with $m = 32$ (which was learned in training as opposed to precomputed), followed by a single recurrent layer with $K = 32$ hidden units. The model was trained with dropout regularization on the 25,000 reviews in the designated training set, and achieved a disappointing 76% accuracy on the **IMDb** test data. A network using the **GloVe** pretrained embedding matrix \mathbf{E} performed slightly worse.

For ease of exposition we have presented a very simple RNN. More elaborate versions use *long term* and *short term* memory (LSTM). Two tracks of hidden-layer activations are maintained, so that when the activation A_ℓ is computed, it gets input from hidden units both further back in time, and closer in time — a so-called *LSTM RNN*. With long sequences, this overcomes the problem of early signals being washed out by the time they get propagated through the chain to the final activation vector A_L .

LSTM RNN

When we refit our model using the LSTM architecture for the hidden layer, the performance improved to 87% on the **IMDb** test data. This is comparable with the 88% achieved by the bag-of-words model in Section 10.4. We give details on fitting these models in Section 10.9.6.

Despite this added LSTM complexity, our RNN is still somewhat “entry level”. We could probably achieve slightly better results by changing the size of the model, changing the regularization, and including additional hidden layers. However, LSTM models take a long time to train, which makes exploring many architectures and parameter optimization tedious.

RNNs provide a rich framework for modeling data sequences, and they continue to evolve. There have been many advances in the development of RNNs — in architecture, data augmentation, and in the learning algorithms. At the time of this writing (early 2020) the leading RNN configurations report accuracy above 95% on the **IMDb** data. The details are beyond the scope of this book.¹⁷

10.5.2 Time Series Forecasting

Figure 10.14 shows historical trading statistics from the New York Stock Exchange. Shown are three daily time series covering the period December 3, 1962 to December 31, 1986.¹⁸

¹⁷An **IMDb** leaderboard can be found at <https://paperswithcode.com/sota/sentiment-analysis-on-imdb>.

¹⁸These data were assembled by LeBaron and Weigend (1998) *IEEE Transactions on Neural Networks*, 9(1): 213–220.

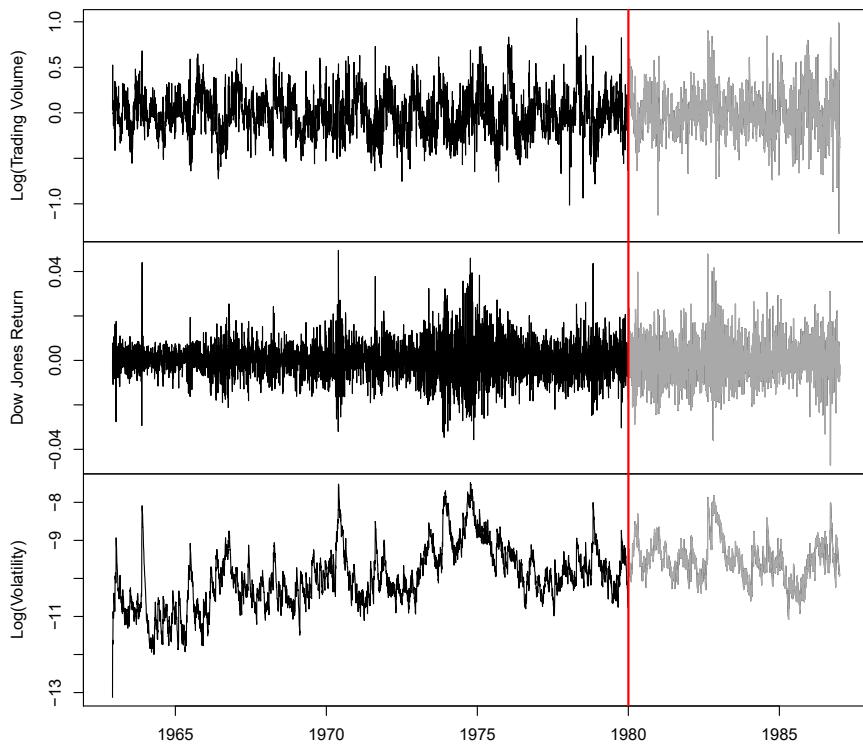


FIGURE 10.14. Historical trading statistics from the New York Stock Exchange. Daily values of the normalized log trading volume, DJIA return, and log volatility are shown for a 24-year period from 1962–1986. We wish to predict trading volume on any day, given the history on all earlier days. To the left of the red bar (January 2, 1980) is training data, and to the right test data.

- **Log trading volume.** This is the fraction of all outstanding shares that are traded on that day, relative to a 100-day moving average of past turnover, on the log scale.
- **Dow Jones return.** This is the difference between the log of the Dow Jones Industrial Index on consecutive trading days.
- **Log volatility.** This is based on the absolute values of daily price movements.

Predicting stock prices is a notoriously hard problem, but it turns out that predicting trading volume based on recent past history is more manageable (and is useful for planning trading strategies).

An observation here consists of the measurements (v_t, r_t, z_t) on day t , in this case the values for `log_volume`, `DJ_return` and `log_volatility`. There are a total of $T = 6,051$ such triples, each of which is plotted as a time series in Figure 10.14. One feature that strikes us immediately is that the day-to-day observations are not independent of each other. The series exhibit *auto-correlation* — in this case values nearby in time tend to be similar to each other. This distinguishes time series from other data sets we have encountered, in which observations can be assumed to be independent of

auto-correlation

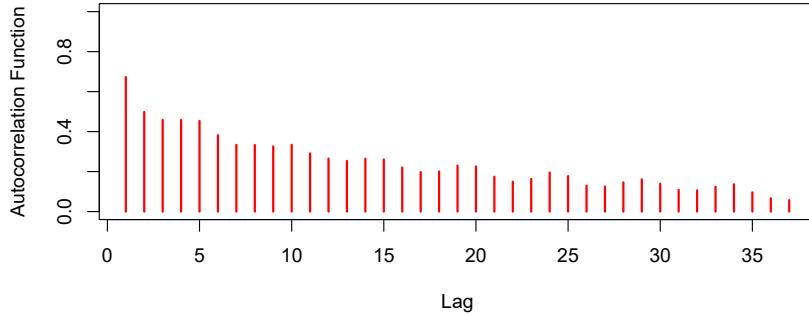


FIGURE 10.15. The autocorrelation function for `log_volume`. We see that nearby values are fairly strongly correlated, with correlations above 0.2 as far as 20 days apart.

each other. To be clear, consider pairs of observations $(v_t, v_{t-\ell})$, a *lag* of ℓ days apart. If we take all such pairs in the v_t series and compute their correlation coefficient, this gives the autocorrelation at lag ℓ . Figure 10.15 shows the autocorrelation function for all lags up to 37, and we see considerable correlation.

Another interesting characteristic of this forecasting problem is that the response variable v_t — `log_volume` — is also a predictor! In particular, we will use the past values of `log_volume` to predict values in the future.

RNN forecaster

We wish to predict a value v_t from past values v_{t-1}, v_{t-2}, \dots , and also to make use of past values of the other series r_{t-1}, r_{t-2}, \dots and z_{t-1}, z_{t-2}, \dots . Although our combined data is quite a long series with 6,051 trading days, the structure of the problem is different from the previous document-classification example.

- We only have one series of data, not 25,000.
- We have an entire *series* of targets v_t , and the inputs include past values of this series.

How do we represent this problem in terms of the structure displayed in Figure 10.12? The idea is to extract many short mini-series of input sequences $X = \{X_1, X_2, \dots, X_L\}$ with a predefined length L (called the *lag* in this context), and a corresponding target Y . They have the form

$$X_1 = \begin{pmatrix} v_{t-L} \\ r_{t-L} \\ z_{t-L} \end{pmatrix}, \quad X_2 = \begin{pmatrix} v_{t-L+1} \\ r_{t-L+1} \\ z_{t-L+1} \end{pmatrix}, \dots, \quad X_L = \begin{pmatrix} v_{t-1} \\ r_{t-1} \\ z_{t-1} \end{pmatrix}, \quad \text{and } Y = v_t. \quad (10.20)$$

So here the target Y is the value of `log_volume` v_t at a single timepoint t , and the input sequence X is the series of 3-vectors $\{X_\ell\}_1^L$ each consisting of the three measurements `log_volume`, `DJ_return` and `log_volatility` from day $t - L$, $t - L + 1$, up to $t - 1$. Each value of t makes a separate (X, Y) pair, for t running from $L + 1$ to T . For the `NYSE` data we will use the past

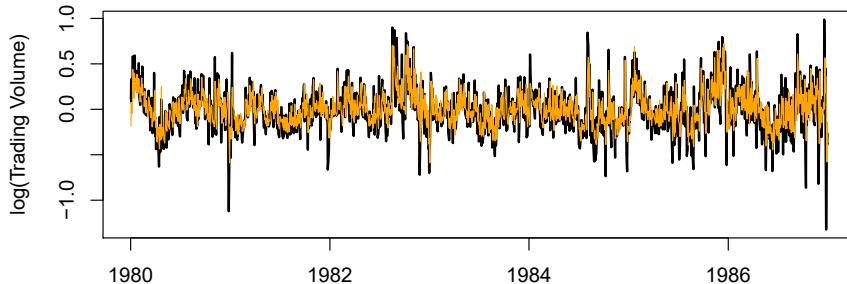


FIGURE 10.16. RNN forecast of `log_volume` on the NYSE test data. The black lines are the true volumes, and the superimposed orange the forecasts. The forecasted series accounts for 42% of the variance of `log_volume`.

five trading days to predict the next day's trading volume. Hence, we use $L = 5$. Since $T = 6,051$, we can create 6,046 such (X, Y) pairs. Clearly L is a parameter that should be chosen with care, perhaps using validation data.

We fit this model with $K = 12$ hidden units using the 4,281 training sequences derived from the data before January 2, 1980 (see Figure 10.14), and then used it to forecast the 1,770 values of `log_volume` after this date. We achieve an $R^2 = 0.42$ on the test data. Details are given in Section 10.9.6. As a *straw man*,¹⁹ using yesterday's value for `log_volume` as the prediction for today has $R^2 = 0.18$. Figure 10.16 shows the forecast results. We have plotted the observed values of the daily `log_volume` for the test period 1980–1986 in black, and superimposed the predicted series in orange. The correspondence seems rather good.

In forecasting the value of `log_volume` in the test period, we have to use the test data itself in forming the input sequences X . This may feel like cheating, but in fact it is not; we are always using past data to predict the future.

Autoregression

The RNN we just fit has much in common with a traditional *autoregression* (AR) linear model, which we present now for comparison. We first consider the response sequence v_t alone, and construct a response vector \mathbf{y} and a matrix \mathbf{M} of predictors for least squares regression as follows:

$$\mathbf{y} = \begin{bmatrix} v_{L+1} \\ v_{L+2} \\ v_{L+3} \\ \vdots \\ v_T \end{bmatrix} \quad \mathbf{M} = \begin{bmatrix} 1 & v_L & v_{L-1} & \cdots & v_1 \\ 1 & v_{L+1} & v_L & \cdots & v_2 \\ 1 & v_{L+2} & v_{L+1} & \cdots & v_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & v_{T-1} & v_{T-2} & \cdots & v_{T-L} \end{bmatrix}. \quad (10.21)$$

\mathbf{M} and \mathbf{y} each have $T - L$ rows, one per observation. We see that the predictors for any given response v_t on day t are the previous L values

auto-regression

¹⁹A straw man here refers to a simple and sensible prediction that can be used as a baseline for comparison.

of the same series. Fitting a regression of \mathbf{y} on \mathbf{M} amounts to fitting the model

$$\hat{v}_t = \hat{\beta}_0 + \hat{\beta}_1 v_{t-1} + \hat{\beta}_2 v_{t-2} + \cdots + \hat{\beta}_L v_{t-L}, \quad (10.22)$$

and is called an order- L autoregressive model, or simply AR(L). For the **NYSE** data we can include lagged versions of **DJ_return** and **log_volatility**, r_t and z_t , in the predictor matrix \mathbf{M} , resulting in $3L + 1$ columns. An AR model with $L = 5$ achieves a test R^2 of 0.41, slightly inferior to the 0.42 achieved by the RNN.

Of course the RNN and AR models are very similar. They both use the same response Y and input sequences X of length $L = 5$ and dimension $p = 3$ in this case. The RNN processes this sequence from left to right with the same weights \mathbf{W} (for the input layer), while the AR model simply treats all L elements of the sequence equally as a vector of $L \times p$ predictors — a process called *flattening* in the neural network literature. Of course the RNN also includes the hidden layer activations A_ℓ which transfer information along the sequence, and introduces additional nonlinearity. From (10.19) with $K = 12$ hidden units, we see that the RNN has $13 + 12 \times (1 + 3 + 12) = 205$ parameters, compared to the 16 for the AR(5) model.

flattening

An obvious extension of the AR model is to use the set of lagged predictors as the input vector to an ordinary feedforward neural network (10.1), and hence add more flexibility. This achieved a test $R^2 = 0.42$, slightly better than the linear AR, and the same as the RNN.

All the models can be improved by including the variable **day_of_week** corresponding to the day t of the target v_t (which can be learned from the calendar dates supplied with the data); trading volume is often higher on Mondays and Fridays. Since there are five trading days, this one-hot encodes to five binary variables. The performance of the AR model improved to $R^2 = 0.46$ as did the RNN, and the nonlinear AR model improved to $R^2 = 0.47$.

We used the most simple version of the RNN in our examples here. Additional experiments with the LSTM extension of the RNN yielded small improvements, typically of up to 1% in R^2 in these examples.

We give details of how we fit all three models in Section 10.9.6.

10.5.3 Summary of RNNs

We have illustrated RNNs through two simple use cases, and have only scratched the surface.

There are many variations and enhancements of the simple RNN we used for sequence modeling. One approach we did not discuss uses a one-dimensional convolutional neural network, treating the sequence of vectors (say words, as represented in the embedding space) as an image. The convolution filter slides along the sequence in a one-dimensional fashion, with the potential to learn particular phrases or short subsequences relevant to the learning task.

One can also have additional hidden layers in an RNN. For example, with two hidden layers, the sequence A_ℓ is treated as an input sequence to the next hidden layer in an obvious fashion.

The RNN we used scanned the document from beginning to end; alternative *bidirectional* RNNs scan the sequences in both directions.

In language translation the target is also a sequence of words, in a language different from that of the input sequence. Both the input sequence and the target sequence are represented by a structure similar to Figure 10.12, and they share the hidden units. In this so-called *Seq2Seq* learning, the hidden units are thought to capture the semantic meaning of the sentences. Some of the big breakthroughs in language modeling and translation resulted from the relatively recent improvements in such RNNs.

Algorithms used to fit RNNs can be complex and computationally costly. Fortunately, good software protects users somewhat from these complexities, and makes specifying and fitting these models relatively painless. Many of the models that we enjoy in daily life (like *Google Translate*) use state-of-the-art architectures developed by teams of highly skilled engineers, and have been trained using massive computational and data resources.

bidirectional
Seq2Seq

10.6 When to Use Deep Learning

The performance of deep learning in this chapter has been rather impressive. It nailed the digit classification problem, and deep CNNs have really revolutionized image classification. We see daily reports of new success stories for deep learning. Many of these are related to image classification tasks, such as machine diagnosis of mammograms or digital X-ray images, ophthalmology eye scans, annotations of MRI scans, and so on. Likewise there are numerous successes of RNNs in speech and language translation, forecasting, and document modeling. The question that then begs an answer is: *should we discard all our older tools, and use deep learning on every problem with data?* To address this question, we revisit our **Hitters** dataset from Chapter 6.

This is a regression problem, where the goal is to predict the **Salary** of a baseball player in 1987 using his performance statistics from 1986. After removing players with missing responses, we are left with 263 players and 19 variables. We randomly split the data into a training set of 176 players (two thirds), and a test set of 87 players (one third). We used three methods for fitting a regression model to these data.

- A linear model was used to fit the training data, and make predictions on the test data. The model has 20 parameters.
- The same linear model was fit with lasso regularization. The tuning parameter was selected by 10-fold cross-validation on the training data. It selected a model with 12 variables having nonzero coefficients.
- A neural network with one hidden layer consisting of 64 **ReLU** units was fit to the data. This model has 1,345 parameters.²⁰

²⁰The model was fit by stochastic gradient descent with a batch size of 32 for 1,000 epochs, and 10% dropout regularization. The test error performance flattened out and started to slowly increase after 1,000 epochs. These fitting details are discussed in Section 10.7.

Model	# Parameters	Mean Abs. Error	Test Set R^2
Linear Regression	20	254.7	0.56
Lasso	12	252.3	0.51
Neural Network	1345	257.4	0.54

TABLE 10.2. Prediction results on the `Hitters` test data for linear models fit by ordinary least squares and lasso, compared to a neural network fit by stochastic gradient descent with dropout regularization.

	Coefficient	Std. error	t-statistic	p-value
<code>Intercept</code>	-226.67	86.26	-2.63	0.0103
<code>Hits</code>	3.06	1.02	3.00	0.0036
<code>Walks</code>	0.181	2.04	0.09	0.9294
<code>CRuns</code>	0.859	0.12	7.09	< 0.0001
<code>PutOuts</code>	0.465	0.13	3.60	0.0005

TABLE 10.3. Least squares coefficient estimates associated with the regression of `Salary` on four variables chosen by lasso on the `Hitters` data set. This model achieved the best performance on the test data, with a mean absolute error of 224.8. The results reported here were obtained from a regression on the test data, which was not used in fitting the lasso model.

Table 10.2 compares the results. We see similar performance for all three models. We report the mean absolute error on the test data, as well as the test R^2 for each method, which are all respectable (see Exercise 5). We spent a fair bit of time fiddling with the configuration parameters of the neural network to achieve these results. It is possible that if we were to spend more time, and got the form and amount of regularization just right, that we might be able to match or even outperform linear regression and the lasso. But with great ease we obtained linear models that work well. Linear models are much easier to present and understand than the neural network, which is essentially a black box. The lasso selected 12 of the 19 variables in making its prediction. So in cases like this we are much better off following the *Occam's razor* principle: when faced with several methods that give roughly equivalent performance, pick the simplest.

After a bit more exploration with the lasso model, we identified an even simpler model with four variables. We then refit the linear model with these four variables to the training data (the so-called *relaxed lasso*), and achieved a test mean absolute error of 224.8, the overall winner! It is tempting to present the summary table from this fit, so we can see coefficients and p-values; however, since the model was selected on the training data, there would be *selection bias*. Instead, we refit the model on the test data, which was not used in the selection. Table 10.3 shows the results.

We have a number of very powerful tools at our disposal, including neural networks, random forests and boosting, support vector machines and generalized additive models, to name a few. And then we have linear models, and simple variants of these. When faced with new data modeling and prediction problems, it's tempting to always go for the trendy new methods. Often they give extremely impressive results, especially when the datasets are very large and can support the fitting of high-dimensional nonlinear models. However, if we can produce models with the simpler tools that

Occam's
razor

perform as well, they are likely to be easier to fit and understand, and potentially less fragile than the more complex approaches. Wherever possible, it makes sense to try the simpler models as well, and then make a choice based on the performance/complexity tradeoff.

Typically we expect deep learning to be an attractive choice when the sample size of the training set is extremely large, and when interpretability of the model is not a high priority.

10.7 Fitting a Neural Network



Fitting neural networks is somewhat complex, and we give a brief overview here. The ideas generalize to much more complex networks. Readers who find this material challenging can safely skip it. Fortunately, as we see in the lab at the end of this chapter, good software is available to fit neural network models in a relatively automated way, without worrying about the technical details of the model-fitting procedure.

We start with the simple network depicted in Figure 10.1 in Section 10.1. In model (10.1) the parameters are $\beta = (\beta_0, \beta_1, \dots, \beta_K)$, as well as each of the $w_k = (w_{k0}, w_{k1}, \dots, w_{kp})$, $k = 1, \dots, K$. Given observations (x_i, y_i) , $i = 1, \dots, n$, we could fit the model by solving a nonlinear least squares problem

$$\underset{\{w_k\}_1^K, \beta}{\text{minimize}} \frac{1}{2} \sum_{i=1}^n (y_i - f(x_i))^2, \quad (10.23)$$

where

$$f(x_i) = \beta_0 + \sum_{k=1}^K \beta_k g\left(w_{k0} + \sum_{j=1}^p w_{kj} x_{ij}\right). \quad (10.24)$$

The objective in (10.23) looks simple enough, but because of the nested arrangement of the parameters and the symmetry of the hidden units, it is not straightforward to minimize. The problem is nonconvex in the parameters, and hence there are multiple solutions. As an example, Figure 10.17 shows a simple nonconvex function of a single variable θ ; there are two solutions: one is a *local minimum* and the other is a *global minimum*. Furthermore, (10.1) is the very simplest of neural networks; in this chapter we have presented much more complex ones where these problems are compounded. To overcome some of these issues and to protect from overfitting, two general strategies are employed when fitting neural networks.

local
minimum
global
minimum

- *Slow Learning:* the model is fit in a somewhat slow iterative fashion, using *gradient descent*. The fitting process is then stopped when overfitting is detected.
- *Regularization:* penalties are imposed on the parameters, usually lasso or ridge as discussed in Section 6.2.

gradient
descent

Suppose we represent all the parameters in one long vector θ . Then we can rewrite the objective in (10.23) as

$$R(\theta) = \frac{1}{2} \sum_{i=1}^n (y_i - f_\theta(x_i))^2, \quad (10.25)$$

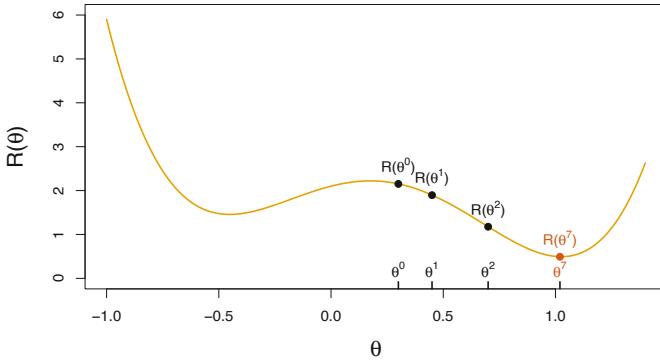


FIGURE 10.17. Illustration of gradient descent for one-dimensional θ . The objective function $R(\theta)$ is not convex, and has two minima, one at $\theta = -0.46$ (local), the other at $\theta = 1.02$ (global). Starting at some value θ^0 (typically randomly chosen), each step in θ moves downhill — against the gradient — until it cannot go down any further. Here gradient descent reached the global minimum in 7 steps.

where we make explicit the dependence of f on the parameters. The idea of gradient descent is very simple.

1. Start with a guess θ^0 for all the parameters in θ , and set $t = 0$.
2. Iterate until the objective (10.25) fails to decrease:
 - (a) Find a vector δ that reflects a small change in θ , such that $\theta^{t+1} = \theta^t + \delta$ reduces the objective; i.e. such that $R(\theta^{t+1}) < R(\theta^t)$.
 - (b) Set $t \leftarrow t + 1$.

One can visualize (Figure 10.17) standing in a mountainous terrain, and the goal is to get to the bottom through a series of steps. As long as each step goes downhill, we must eventually get to the bottom. In this case we were lucky, because with our starting guess θ^0 we end up at the global minimum. In general we can hope to end up at a (good) local minimum.

10.7.1 Backpropagation

How do we find the directions to move θ so as to decrease the objective $R(\theta)$ in (10.25)? The *gradient* of $R(\theta)$, evaluated at some current value $\theta = \theta^m$, is the vector of partial derivatives at that point:

$$\nabla R(\theta^m) = \frac{\partial R(\theta)}{\partial \theta} \Big|_{\theta=\theta^m}. \quad (10.26)$$

The subscript $\theta = \theta^m$ means that after computing the vector of derivatives, we evaluate it at the current guess, θ^m . This gives the direction in θ -space in which $R(\theta)$ increases most rapidly. The idea of gradient descent is to move θ a little in the *opposite* direction (since we wish to go downhill):

$$\theta^{m+1} \leftarrow \theta^m - \rho \nabla R(\theta^m). \quad (10.27)$$

For a small enough value of the *learning rate* ρ , this step will decrease the objective $R(\theta)$; i.e. $R(\theta^{m+1}) \leq R(\theta^m)$. If the gradient vector is zero, then we may have arrived at a minimum of the objective.

How complicated is the calculation (10.26)? It turns out that it is quite simple here, and remains simple even for much more complex networks, because of the *chain rule* of differentiation.

Since $R(\theta) = \sum_{i=1}^n R_i(\theta) = \frac{1}{2} \sum_{i=1}^n (y_i - f_\theta(x_i))^2$ is a sum, its gradient is also a sum over the n observations, so we will just examine one of these terms,

$$R_i(\theta) = \frac{1}{2} \left(y_i - \beta_0 - \sum_{k=1}^K \beta_k g\left(w_{k0} + \sum_{j=1}^p w_{kj} x_{ij}\right) \right)^2. \quad (10.28)$$

To simplify the expressions to follow, we write $z_{ik} = w_{k0} + \sum_{j=1}^p w_{kj} x_{ij}$. First we take the derivative with respect to β_k :

$$\begin{aligned} \frac{\partial R_i(\theta)}{\partial \beta_k} &= \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial \beta_k} \\ &= -(y_i - f_\theta(x_i)) \cdot g(z_{ik}). \end{aligned} \quad (10.29)$$

And now we take the derivative with respect to w_{kj} :

$$\begin{aligned} \frac{\partial R_i(\theta)}{\partial w_{kj}} &= \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial g(z_{ik})} \cdot \frac{\partial g(z_{ik})}{\partial z_{ik}} \cdot \frac{\partial z_{ik}}{\partial w_{kj}} \\ &= -(y_i - f_\theta(x_i)) \cdot \beta_k \cdot g'(z_{ik}) \cdot x_{ij}. \end{aligned} \quad (10.30)$$

Notice that both these expressions contain the residual $y_i - f_\theta(x_i)$. In (10.29) we see that a fraction of that residual gets attributed to each of the hidden units according to the value of $g(z_{ik})$. Then in (10.30) we see a similar attribution to input j via hidden unit k . So the act of differentiation assigns a fraction of the residual to each of the parameters via the chain rule — a process known as *backpropagation* in the neural network literature. Although these calculations are straightforward, it takes careful bookkeeping to keep track of all the pieces.

learning rate

chain rule

backpropagation

minibatch

stochastic gradient descent

10.7.2 Regularization and Stochastic Gradient Descent

Gradient descent usually takes many steps to reach a local minimum. In practice, there are a number of approaches for accelerating the process. Also, when n is large, instead of summing (10.29)–(10.30) over all n observations, we can sample a small fraction or *minibatch* of them each time we compute a gradient step. This process is known as *stochastic gradient descent* (SGD) and is the state of the art for learning deep neural networks. Fortunately, there is very good software for setting up deep learning models, and for fitting them to data, so most of the technicalities are hidden from the user.

We now turn to the multilayer network (Figure 10.4) used in the digit recognition problem. The network has over 235,000 weights, which is around four times the number of training examples. Regularization is essential here

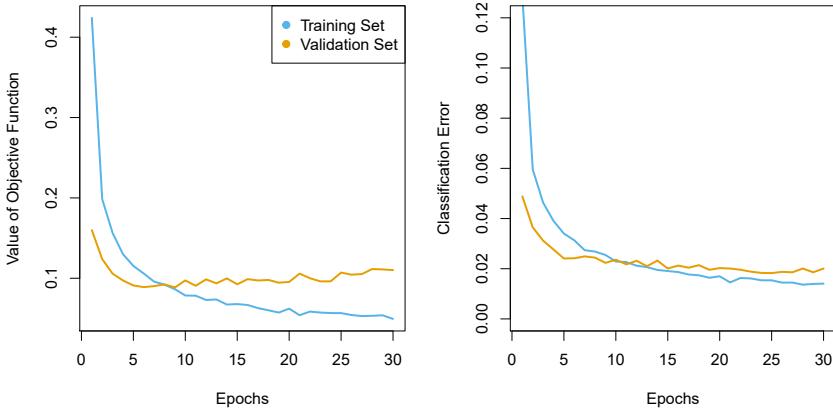


FIGURE 10.18. Evolution of training and validation errors for the **MNIST** neural network depicted in Figure 10.4, as a function of training epochs. The objective refers to the log-likelihood (10.14).

to avoid overfitting. The first row in Table 10.1 uses ridge regularization on the weights. This is achieved by augmenting the objective function (10.14) with a penalty term:

$$R(\theta; \lambda) = -\sum_{i=1}^n \sum_{m=0}^9 y_{im} \log(f_m(x_i)) + \lambda \sum_j \theta_j^2. \quad (10.31)$$

The parameter λ is often preset at a small value, or else it is found using the validation-set approach of Section 5.3.1. We can also use different values of λ for the groups of weights from different layers; in this case \mathbf{W}_1 and \mathbf{W}_2 were penalized, while the relatively few weights \mathbf{B} of the output layer were not penalized at all. Lasso regularization is also popular as an additional form of regularization, or as an alternative to ridge.

Figure 10.18 shows some metrics that evolve during the training of the network on the **MNIST** data. It turns out that SGD naturally enforces its own form of approximately quadratic regularization.²¹ Here the minibatch size was 128 observations per gradient update. The term *epochs* labeling the horizontal axis in Figure 10.18 counts the number of times an equivalent of the full training set has been processed. For this network, 20% of the 60,000 training observations were used as a validation set in order to determine when training should stop. So in fact 48,000 observations were used for training, and hence there are $48,000/128 \approx 375$ minibatch gradient updates per epoch. We see that the value of the validation objective actually starts to increase by 30 epochs, so *early stopping* can also be used as an additional form of regularization.

epochs
early
stopping

²¹This and other properties of SGD for deep learning are the subject of much research in the machine learning literature at the time of writing.

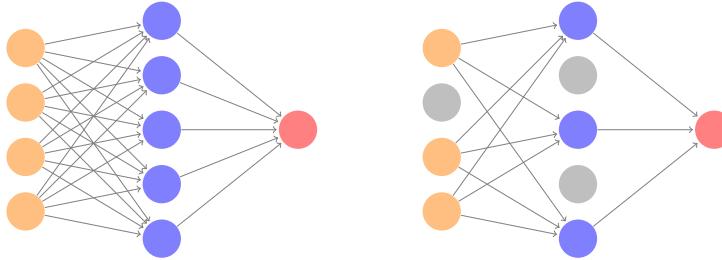


FIGURE 10.19. *Dropout Learning.* Left: a fully connected network. Right: network with dropout in the input and hidden layer. The nodes in grey are selected at random, and ignored in an instance of training.

10.7.3 Dropout Learning

The second row in Table 10.1 is labeled *dropout*. This is a relatively new and efficient form of regularization, similar in some respects to ridge regularization. Inspired by random forests (Section 8.2), the idea is to randomly remove a fraction ϕ of the units in a layer when fitting the model. Figure 10.19 illustrates this. This is done separately each time a training observation is processed. The surviving units stand in for those missing, and their weights are scaled up by a factor of $1/(1 - \phi)$ to compensate. This prevents nodes from becoming over-specialized, and can be seen as a form of regularization. In practice dropout is achieved by randomly setting the activations for the “dropped out” units to zero, while keeping the architecture intact.

dropout

10.7.4 Network Tuning

The network in Figure 10.4 is considered to be relatively straightforward; it nevertheless requires a number of choices that all have an effect on the performance:

- *The number of hidden layers, and the number of units per layer.* Modern thinking is that the number of units per hidden layer can be large, and overfitting can be controlled via the various forms of regularization.
- *Regularization tuning parameters.* These include the dropout rate ϕ and the strength λ of lasso and ridge regularization, and are typically set separately at each layer.
- *Details of stochastic gradient descent.* These include the batch size, the number of epochs, and if used, details of data augmentation (Section 10.3.4.)

Choices such as these can make a difference. In preparing this MNIST example, we achieved a respectable 1.8% misclassification error after some trial and error. Finer tuning and training of a similar network can get under 1% error on these data, but the tinkering process can be tedious, and can result in overfitting if done carelessly.

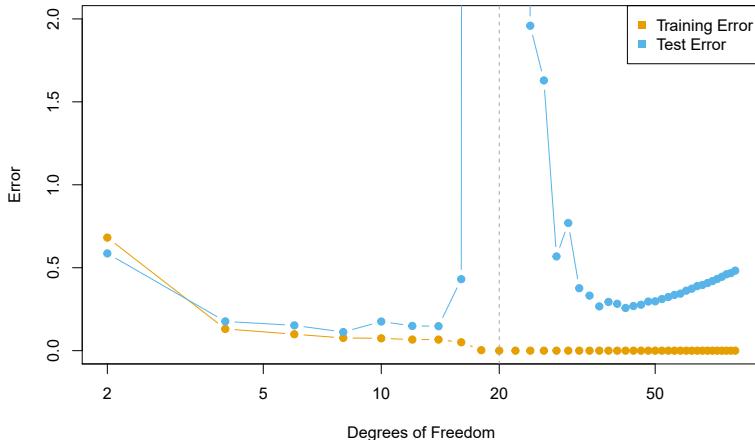


FIGURE 10.20. Double descent phenomenon, illustrated using error plots for a one-dimensional natural spline example. The horizontal axis refers to the number of spline basis functions on the log scale. The training error hits zero when the degrees of freedom coincides with the sample size $n = 20$, the “interpolation threshold”, and remains zero thereafter. The test error increases dramatically at this threshold, but then descends again to a reasonable value before finally increasing again.

10.8 Interpolation and Double Descent

Throughout this book, we have repeatedly discussed the bias-variance trade-off, first presented in Section 2.2.2. This trade-off indicates that statistical learning methods tend to perform the best, in terms of test-set error, for an intermediate level of model complexity. In particular, if we plot “flexibility” on the x -axis and error on the y -axis, then we generally expect to see that test error has a U-shape, whereas training error decreases monotonically. Two “typical” examples of this behavior can be seen in the right-hand panel of Figure 2.9 on page 29, and in Figure 2.17 on page 39. One implication of the bias-variance trade-off is that it is generally not a good idea to *interpolate* the training data — that is, to get zero training error — since that will often result in very high test error.

However, it turns out that in certain specific settings it can be possible for a statistical learning method that interpolates the training data to perform well — or at least, better than a slightly less complex model that does not quite interpolate the data. This phenomenon is known as *double descent*, and is displayed in Figure 10.20. “Double descent” gets its name from the fact that the test error has a U-shape before the interpolation threshold is reached, and then it descends again (for a while, at least) as an increasingly flexible model is fit.

We now describe the set-up that resulted in Figure 10.20. We simulated $n = 20$ observations from the model

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

where $X \sim U[-5, 5]$ (uniform distribution), and $\epsilon \sim N(0, \sigma^2)$ with $\sigma = 0.3$. We then fit a natural spline to the data, as described in Section 7.4, with d

interpolate

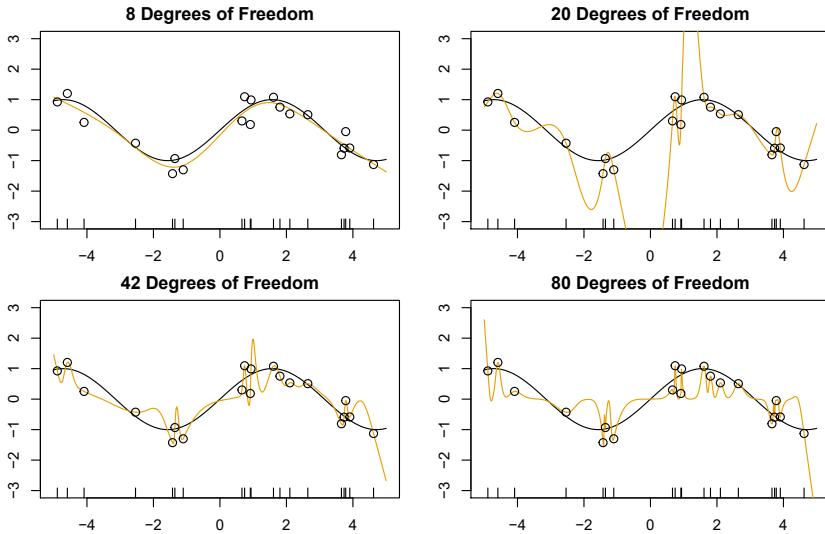


FIGURE 10.21. Fitted functions $\hat{f}_d(X)$ (orange), true function $f(X)$ (black) and the observed 20 training data points. A different value of d (degrees of freedom) is used in each panel. For $d \geq 20$ the orange curves all interpolate the training points, and hence the training error is zero.

degrees of freedom.²² Recall from Section 7.4 that fitting a natural spline with d degrees of freedom amounts to fitting a least-squares regression of the response onto a set of d basis functions. The upper-left panel of Figure 10.21 shows the data, the true function $f(X)$, and $\hat{f}_8(X)$, the fitted natural spline with $d = 8$ degrees of freedom.

Next, we fit a natural spline with $d = 20$ degrees of freedom. Since $n = 20$, this means that $n = d$, and we have zero training error; in other words, we have interpolated the training data! We can see from the top-right panel of Figure 10.21 that $\hat{f}_{20}(X)$ makes wild excursions, and hence the test error will be large.

We now continue to fit natural splines to the data, with increasing values of d . For $d > 20$, the least squares regression of Y onto d basis functions is not unique: there are an infinite number of least squares coefficient estimates that achieve zero error. To select among them, we choose the one with the smallest sum of squared coefficients, $\sum_{j=1}^d \hat{\beta}_j^2$. This is known as the *minimum-norm* solution.

The two lower panels of Figure 10.21 show the minimum-norm natural spline fits with $d = 42$ and $d = 80$ degrees of freedom. Incredibly, $\hat{f}_{42}(X)$ is quite a bit less wild than $\hat{f}_{20}(X)$, even though it makes use of more degrees of freedom. And $\hat{f}_{80}(X)$ is not much different. How can this be? Essentially, $\hat{f}_{20}(X)$ is very wild because there is just a single way to interpolate $n = 20$ observations using $d = 20$ basis functions, and that single way results in a somewhat extreme fitted function. By contrast, there are an

²²This implies the choice of d knots, here chosen at d equi-probability quantiles of the training data. When $d > n$, the quantiles are found by interpolation.

infinite number of ways to interpolate $n = 20$ observations using $d = 42$ or $d = 80$ basis functions, and the smoothest of them — that is, the minimum norm solution — is much less wild than $\hat{f}_{20}(X)$!

In Figure 10.20, we display the training error and test error associated with $\hat{f}_d(X)$, for a range of values of the degrees of freedom d . We see that the training error drops to zero once $d = 20$ and beyond; i.e. once the interpolation threshold is reached. By contrast, the test error shows a U -shape for $d \leq 20$, grows extremely large around $d = 20$, and then shows a second region of descent for $d > 20$. For this example the signal-to-noise ratio — $\text{Var}(f(X))/\sigma^2$ — is 5.9, which is quite high (the data points are close to the true curve). So an estimate that interpolates the data and does not wander too far inbetween the observed data points will likely do well.

In Figures 10.20 and 10.21, we have illustrated the double descent phenomenon in a simple one-dimensional setting using natural splines. However, it turns out that the same phenomenon can arise for deep learning. Basically, when we fit neural networks with a huge number of parameters, we are sometimes able to get good results with zero training error. This is particularly true in problems with high signal-to-noise ratio, such as natural image recognition and language translation, for example. This is because the techniques used to fit neural networks, including stochastic gradient descent, naturally lend themselves to selecting a “smooth” interpolating model that has good test-set performance on these kinds of problems.

Some points are worth emphasizing:

- *The double-descent phenomenon does not contradict the bias-variance trade-off, as presented in Section 2.2.2.* Rather, the double-descent curve seen in the right-hand side of Figure 10.20 is a consequence of the fact that the x -axis displays the number of spline basis functions used, which does not properly capture the true “flexibility” of models that interpolate the training data. Stated another way, in this example, the minimum-norm natural spline with $d = 42$ has lower variance than the natural spline with $d = 20$.
- *Most of the statistical learning methods seen in this book do not exhibit double descent.* For instance, regularization approaches typically do not interpolate the training data, and thus double descent does not occur. This is not a drawback of regularized methods: they can give great results *without interpolating the data!*

In particular, in the examples here, if we had fit the natural splines using ridge regression with an appropriately-chosen penalty rather than least squares, then we would not have seen double descent, and in fact would have obtained better test error results.

- *In Chapter 9, we saw that maximal margin classifiers and SVMs that have zero training error nonetheless often achieve very good test error.* This is in part because those methods seek smooth minimum norm solutions. This is similar to the fact that the minimum-norm natural spline can give good results with zero training error.
- *The double-descent phenomenon has been used by the machine learning community to explain the successful practice of using an over-*

parametrized neural network (many layers, and many hidden units), and then fitting all the way to zero training error. However, fitting to zero error is not always optimal, and whether it is advisable depends on the signal-to-noise ratio. For instance, we may use ridge regularization to avoid overfitting a neural network, as in (10.31). In this case, provided that we use an appropriate choice for the tuning parameter λ , we will never interpolate the training data, and thus will not see the double descent phenomenon. Nonetheless we can get very good test-set performance, likely much better than we would have achieved had we interpolated the training data. Early stopping during stochastic gradient descent can also serve as a form of regularization that prevents us from interpolating the training data, while still getting very good results on test data.

To summarize: though double descent can sometimes occur in neural networks, we typically do not want to rely on this behavior. Moreover, it is important to remember that the bias-variance trade-off always holds (though it is possible that test error as a function of flexibility may not exhibit a U-shape, depending on how we have parametrized the notion of “flexibility” on the x -axis).

10.9 Lab: Deep Learning

In this section we demonstrate how to fit the examples discussed in the text. We use the `Python torch` package, along with the `pytorch_lightning` package which provides utilities to simplify fitting and evaluating models. This code can be impressively fast with certain special processors, such as Apple’s new M1 chip. The package is well-structured, flexible, and will feel comfortable to `Python` users. A good companion is the site pytorch.org/tutorials. Much of our code is adapted from there, as well as the `pytorch_lightning` documentation.²³

`torch`
`pytorch_`
`lightning`

We start with several standard imports that we have seen before.

```
In [1]: import numpy as np, pandas as pd
from matplotlib.pyplot import subplots
from sklearn.linear_model import \
    (LinearRegression,
     LogisticRegression,
     Lasso)
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.pipeline import Pipeline
from ISLP import load_data
from ISLP.models import ModelSpec as MS
from sklearn.model_selection import \
    (train_test_split,
     GridSearchCV)
```

²³The precise URLs at the time of writing are <https://pytorch.org/tutorials/beginner/basics/intro.html> and <https://pytorch-lightning.readthedocs.io/en/latest/>.