THE ART OF MACHINE LEARNING

Algorithms+Data+R

by Norm Matloff

no starch press

San Francisco

For information on book distributors or translations, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
555 De Haro Street, Suite 250, San Francisco, CA 94107
phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

AUTHOR'S BIOGRAPHICAL SKETCH

Dr. Norm Matloff is a professor of computer science at the University of California at Davis, and was formerly a professor of statistics at that university.

Dr. Matloff was born in Los Angeles, and grew up in East Los Angeles and the San Gabriel Valley. He has a PhD in pure mathematics from UCLA, His current research interests are in machine learning, parallel processing, statistical computing, and statistical methodology for handling missing data.

Prof. Matloff is a former appointed member of IFIP Working Group 11.3, an international committee concerned with database software security, established under the United Nations. He was a founding member of the UC Davis Department of Statistics, and participated in the formation of the UCD Computer Science Department as well. He is a recipient of the campuswide Distinguished Teaching Award and Distinguished Public Service Award at UC Davis.

Dr. Matloff has served as Editor-in-Chief of the *R Journal*, and served on the editorial board of the *Journal of Statistical Software*.

Dr. Matloff is the author of several published textbooks. His book *Statistical Regression and Classification: from Linear Models to Machine Learning*, won the Ziegel Award in 2017.

A NOTE ON SOFTWARE

This book uses software in the R programming language, in contrast to many based on Python. There has been much pioneering work in machine learning done in both languages — neural networks in the Python case, and random forests in the R case, for instance — but that is merely personal preference, not inherent features of the languages, and thus not a reason for choosing one language over the other for a book like this. I do use and teach Python, and find it to be very elegant and expressive, but from my point of view, R is the clear choice between the two, for several reasons:

- R libraries are much easier to install. Python's often require computer systems expertise beyond what typical readers of this book have.
- R is "statistically correct," written *by* statisticians *for* statisticians.

- R has outstanding graphics capabilities.

There is relatively little R coding itself in the book, just support code to prepare data for analysis. Instead, the book makes extensive use of R packages from the Comprehensive R Archive Network (CRAN), such as the **partykit** package for decision trees and random forests. I am the author of one of the CRAN packages used, **regtools**, and prefer it over others for specific operations, such as hyperparameter grid search.

However, as noted in the Preface, the prime goal of this book is to empower readers with good machine learning skills and insight, rather than inordinate focus on the syntax and semantics of certain packages. To be sure, this is absolutely a data-oriented book, with computer analysis of real datasets on virtually every page, and the details of the software used *are* presented. But my point is that CRAN bestows on R a rich variety of user choices, and readers who prefer packages other than those in the book should still derive that same empowerment.

THANKS

Over my many years in the business, a number of people have been influential in my thinking on machine learning and statistical issues. I'll just mention a few.

I'll first cite the late Chuck Stone. He was my fellow native of Los Angeles, my statistical mentor in grad school, and my close friend over the several decades he lived in the Bay Area. He was one of the four original developers of the CART method, and did pioneering work in the theory of k-Nearest Neighbor methodology. He was quite highly opinionated and blunt — but that is exactly the kind of person who makes one think, and question and defend one's assumptions, whether they be on statistics, climate change or the economy. Of course, I responded in kind, challenging *his* assumptions, and we had numerous lively discussions.

In 2016 I was appointed to the Organizing Committee of the useR! conference, held that year at Stanford University. While having planning meetings for the conference there, I started attending the weekly Statistics Department seminar, which I have done frequently ever since. In addition to the remarkable camaraderie I've enjoyed observing among the faculty there, a number of speakers, both external and internal, have stimulated my thinking on various aspects of the fascinating field of Prediction.

I learn as much from my research students as they learn from me. I'd cite in particular these who've recently done ML work with me: Yu-Shih

Chen, Melissa Goh, Tiffany Jiang, Robert Tucker, and especially Zhiyuan (Daniel) Guo, Robin Yancey and Wenxuan (Allan) Zhao.

I don't know Sir David Cox, and other than his having accepted an early paper of mine for publication, I have never interacted with him. But I have tremendously admired his writings, which I might describe as Theory Meets Common Sense, just enough mathematics to clarify practical problems. For better or worse, my own style, even in the avowedly nonmathematical book here, follows that philosophy. Hopefully I do this at least somewhat as effectively as Sir David.

I owe more concrete thanks to Achim Zeileis, one of the authors of the **partykit** package of R routines that form excellent implementations of data partitioning methods. He has been quite helpful in guiding me through the nuances of the package. Nello Cristianini kindly read an initially flawed draft of the SVM chapter of my earlier book, *Statistical Regression and Classification: from Linear Models to Machine Learning*, thus improving my present chapter on the topic as well.

I am really indebted to Bill Pollock of No Starch Press, publisher of this book and two previous ones, and John Kimmel, editor of my three books at CRC Press, for their longtime encouragement. Thanks to their help, hopefully I've figured out this book-writing business by now.

As with all my books, I must thank my wife Gamis and our daughter Laura for their inspiration and just for being such wonderful, interesting people.

TO THE READER

Why is this book different from all other machine learning (ML) books?

There are many great books out there, of course, but none really *empowers* the reader to use ML effectively in real-world problems. Many are too theoretical, but my main concern is that the applied books tend to be "cookbooks," too "recipe-oriented," treating the subject in a Step 1, Step 2, Step 3 manner. I wrote this book because:

- **ML is not a recipe.** It is not a matter of knowing the syntax and mechanics of various software packages.
- **ML is an art, not a science.** (Hence the title of this book.)
- One does not have to be a math whiz or know advanced math in orer to use ML effectively, but **one does need to understand the concepts well — the Why? and How? of ML methods.**

My goal is then this:

> We will *empower* the reader with a strong *practical*, real-world knowledge of ML methods — their strengths and weaknesses, what makes them work and fail, what to watch out for. We will do so without much formal math, and will definitely take a hands-on approach, using prominent software packages on real datasets. But we will do so in a savvy manner. We will be "informed consumers."

Style details:

- This book is not aimed at mathematicians or computer scientists. **It's "ML for the rest of us,"** accessible to anyone who understands histograms, scatter plots and the slope of a line. Thus, the book is not mathematical, and has rather few formal equations.
(For readers who do wish to engage with the underlying math of ML methods, there is a companion Web book, freely available at *heather.cs.ucdavis.edu/artofml/MathCompanion.pdf*.)
- On the other hand, as noted, **this is not a "cookbook."** *Those dazzling ML successes you've heard about come only after careful, lengthy tuning and thought on the analyst's part, requiring real insight.* This book aims to develop that insight.
- The book pays **special attention to the all-important issue of overfitting.** Recently I found an especially compelling statement in that regard, by Prof. Yaser Abu-Mostafa of Caltech: "It can be said that the ability to avoid overfitting is what separates professionals from amateurs in ML."[1] Other books mention overfitting, of course, but in a vague, unsatisfying manner. Here we explain what bias and variance really mean in any given ML method.
- Readers want to **get started right away**, not have to wade through a couple of "fluff" chapters with titles like "What IS ML?" Not so with this book. Our first example — real data and code to analyze it — begins right there on the second page of Chapter 1. General issues and concepts are brought in, but they are interwoven though the discussions of methodology.
- Though the book is aimed at those with no prior background in statistics or ML, readers who do have some prior exposure will find that they gain new insights here. I believe they will find themselves saying, "Oh, so THAT is what it means!", "Hmm, I never realized you could do that," and so on.

Recurring special sections:

There are special recurring sections and chapters throughout book:

- **Bias vs. Variance** sections explain in concrete terms — no superstition! — how these two central notions play out for each specific ML method.
- **Pitfalls** sections warn the reader of potential problems, and show how to avoid them.
- **Diagnostic** sections show how to find areas for improvement of one's ML model.

1. *https://www.youtube.com/watch?v=EQWr3GGCdzw*

- **Clearing the Confusion** sections dispel various misunderstandings regarding ML that one often sees on the Web, in books and so on.
- **The Verdict** sections summarize the pros and cons of the methods under discussion.

Background needed:

Hopefully the reader will find these features attractive. What kind of background will she need to use the book profitably?

- No prior exposure to ML or statistics is assumed.
- As to math in general, as noted, the book is mostly devoid of formal equations. Again, as long as the reader is comfortable with basic graphs, such as histograms and scatter plots, and simple algebra notions such as the slope of a line, that is quite sufficient.
- The book does assume some prior background in R coding, e.g. familarity with vectors and data frames. There is a review in an appendix, which should also suffice for readers with some coding experience but not in R.

Software used:

Please note, **this is a book on ML, not a book on using R in ML.** True, R plays a major supporting role and we use prominent R packages for ML throughout the book, with code on almost every page. But in order to be able to *use* ML well, the reader should focus on the structure and interpretation of the ML models themselves; R is just a tool toward that end.

Indeed, wrapper packages such as **caret** and **mlr3** internally use many of the same R packages that are used in this book, such as **glmnet** and **ranger**. Readers who use those wrapper packages should find this book just as useful as do readers who use **glmnet** etc. directly. For that matter, one who uses non-R packages, should still find that the book greatly enhances his/her insight into ML, and skill in applying it to real-world problems.

‘

ONE MORE POINT

In reading this book, keep in mind that **the prose is just as important as the code.** Avoid the temptation to focus only on the code and graphs. The verbiage is where the reader should gain the most insight!

So, let’s get started. Happy ML-ing!

BRIEF CONTENTS

CONTENTS IN DETAIL

PART I

PROLOGUE, AND NEIGHBORHOOD-BASED METHODS

1
PROLOGUE: REGRESSION MODELS

Most books of this sort begin with a "fluff" chapter, presenting a broad overview of the topic, defining a few terms and possibly giving the historical background, but with no technical content. Yet I know that you, the reader, want to get started right away!

Accordingly, this and the next two chapters will do both, bringing in specific technical material and introducing general concepts:

- We'll present our first machine learning method, k-Nearest Neighbors (k-NN), applying it to real data.
- We'll weave in general concepts that will recur throughout the book, such as regression functions, dummy variables, overfitting, p-hacking, "dirty" data and so on.[1]

By the way, we will usually abbreviate "machine learning" as just ML.

1. Some readers may have learned about linear regression analysis in a statistics course, but actually the notion of a regression function is much broader than this. The term *regression* is NOT synonymous with *linear regression*. As you will see, all ML methods are regression methods in some form.

So, make sure you have R and the **regtools** package installed on your computer, and let's proceed.[2]

1.1 Reminder to the Reader

Just one more point before getting into the thick of things: Don't just read the code examples — the prose parts of the book are just as important than the code, actually more so.

A quote from our Preface:

> This book is not aimed at mathematicians or computer scientists. **It's "ML for the rest of us,"** accessible to anyone who understands histograms, scatter plots and the slope of a line. Thus, the book is not mathematical, and has rather few formal equations.
> On the other hand, as noted, **this is not a "cookbook."** *Those dazzling ML successes you've heard about come only after careful, lengthy tuning and thought on the analyst's part, requiring real insight.* This book aims to develop that insight.

Keep that last point in mind! Math will give way to prose that describes may key issues. A page that is all prose — no equations, no graphs and no code — may be one of the most important pages in the book, crucial to your goal of being adept at ML.

Protip: READ THE PROSE!

1.2 The Bike Sharing Dataset

This is a famous dataset from the UC Irvine Machine Learning Repository (*https://archive.ics.uci.edu*). It is included as the **day** dataset in **regtools** by permission of the data curator. A detailed description of the data is available on the UCI site, *https://archive.ics.uci.edu/ml/datasets/bike+sharing+dataset*. Note, though, that we will use a slightly modified version **day1** (also included in **regtools**), in which the numeric weather features are given in their original scale, rather than transformed to the interval [0,1].

1.2.1 Let's Take a Look

The data comes in hourly and daily forms, with the latter being the one in the **regtools** package. Load the data:

```
> library(regtools)
> data(day1)
```

2. All code displays in this book assume that the user has already made the call **library(regtools** to load the package.

With any dataset, it's always a good idea to take a look around, using R's **head()** function to view the top of the data:

```
> head(day1)
  instant     dteday season yr mnth holiday
1       1 2011-01-01      1  0    1       0
2       2 2011-01-02      1  0    1       0
3       3 2011-01-03      1  0    1       0
4       4 2011-01-04      1  0    1       0
5       5 2011-01-05      1  0    1       0
6       6 2011-01-06      1  0    1       0
  weekday workingday weathersit     temp
1       6          0          2 8.175849
2       0          0          2 9.083466
3       1          1          1 1.229108
4       2          1          1 1.400000
5       3          1          1 2.666979
6       4          1          1 1.604356
      atemp      hum windspeed casual registered
1  7.999250 0.805833 10.749882    331        654
2  7.346774 0.696087 16.652113    131        670
3 -3.499270 0.437273 16.636703    120       1229
4 -1.999948 0.590435 10.739832    108       1454
5 -0.868180 0.436957 12.522300     82       1518
6 -0.608206 0.518261  6.000868     88       1518
   tot
1  985
2  801
3 1349
4 1562
5 1600
6 1606
```

We see there is data on the date, nature of the date, and weather conditions. The last three columns measure ridership, from casual users, registered users and the total.

1.2.2 Dummy Variables

There are several columns in the data that represent *dummy variables*, which take on values 1 and 0 only. They are sometimes more formally called *indicator variables*. The latter term is quite apt, because an indicator variable *indicates* whether a certain condition holds (code 1) or not (code 0). An alternate term popular in ML circles is *one-hot coding*.

Look at the 'workingday' column, for instance. The documentation defines this to be a day during Monday-Friday that is not a holiday. The row for 2011-01-05 has workingday = 1, meaning, yes, this was a working day.

In the display of the top few lines of our bike sharing data above, we see dummy variables **holiday** and **workingday**.

One very common usage of dummy variables is coding of *categorical data*. In a marketing study, for instance, a factor of interest might be type of region of residence, say Urban, Suburban or Rural. Our original data might code these as 1, 2 or 3. But those are just arbitrary codes, so for example there is no implication that Rural is 3 times as good as Urban, yet ML algorithms may take it that way, which is not what we want.

The solution is to use dummy variables. We could have a dummy variable for Urban (1 yes, 0 no) and one for Suburban. Rural-ness would then be coded by having both Urban and Suburban set to 0, so we don't need a third dummy (and having one might cause technical problems beyond the scope of this book). And of course there is nothing special about using the first two values as dummies; we could have, say one for Urban and one for Rural, without Suburban; the latter would then be indicated by 0 values in Urban and Rural.

In our display above of the top of the bike sharing data, we see categorical variables **mnth** and **weekday**.[3]

1.2.3 Conversions: between R Factors and Dummy Variables, between Data Frames and Matrices

Note for future reference: **Throughout the book, we will have situations in which we will need to convert between R factors and dummy variables, and between data frames and matrices.**

In R, a categorical variable has a formal class, **factor**. It actually is one of the most useful aspects of R, but one must be adept at switching back and forth between factors and the corresponding dummies.

Similarly, though we will mostly work with data frames in this book, there are some algorithms that need matrices, e.g. because they calculate distances between rows and do matrix multiplication and inversion. You do not need to know how, say, what matrix inversion is, but some of the software packages will require you to present only marix inputs, not data frames.

Some highly popular R ML packages automatically generate dummies from factors, but others do not. For example, **glmnet** for LASSO models, requires that categorical features be in the form of dummies, while **ranger**, for random forests, accepts factors.

So it's important to be able to generate dummy variables ourselves. The **regtools** functions **factorToDummies()** and **factorsToDummies()** do this. We will discuss those two functions later in the book. We'll be using dummy variables throughout the book, including in this chapter.

3. There is also a feature **weathersit** (1 = clear, 2 = mist or cloudy, 3 = light snow/light rain, 4 = heavy rain or ice pallets or thunderstorm). That one could be considered categorical as well. Since there is an implied ordering, it is also *ordinal*. Fully exploiting that property is beyond the scope of this book.

We will also use the built-in R function **as.matrix()** to convert from data frames to matrices.

1.3 Machine Learning Is about Prediction

Prediction is hard, especially about the future — famous baseball player and malapropist Yogi Berra

Both statistics and ML do prediction — in the statistics case, since way back in the early 1800s, continuing strong ever since — but ML is virtually 100% about prediction.

Early in the morning, the manager of the bike sharing service might want to predict the total number of riders for the day. Our predictor variables — called *features* in the ML world[4] — might be the various weather variables, the work status of the day (weekday, holiday), and so on. Of course, predictions are not perfect, but if we are in the ballpark of what turns out to be the actual number, this can be quite helpful. For instance, it can help us decide how many bikes to make available, with pumped up tires and so on. (An advanced version would be to predict the demand for bikes at each station, so that bikes could be reallocated accordingly.)

Note that, amusing as the Yogi Berra quote is, he had a point; prediction is not always about the future. A researcher may wish to estimate the mean wages workers made back in the 1700s, say. Or a physician may wish to make a diagnoisis, say as to whether a patient has a particular disease, based on blood tests, symptoms and so on; the trait we are guessing here is in the present, not the future.

1.4 Classification Applications

A very common special case of prediction is *classification*. Here the entity to be predicted is a dummy variable, or even a categorical variable.

For instance, as noted above, a physician may wish to make a diagnosis (probably preliminary), as to whether a patient has cancer, based on an image from a histology slide. The outcome has two classes, malignant or benign, and is thus represented by a yes-no dummy variable.

There is also the multiclass case. For instance, within breast cancer, there are subtypes Basal, Luminal A, Luminal B, and HER2.[5] Here the cancer type is categorical, and we have four classes, which we'd represent by four dummies.[6] So, if a person in a dataset of breast cancer patients, is of

4. The word *predictor* is also used, and occasionally one still hears the old-fashioned term *independent variable*.
5. Singh N., Couture H.D., Marron J.S., Perou C., Niethammer M. (2014) Topological Descriptors of Histology Images. In: Wu G., Zhang D., Zhou L. (eds) Machine Learning in Medical Imaging. MLMI 2014. *Lecture Notes in Computer Science*, vol 8679. Springer.
6. The reader may wonder why we use four dummies, rather than three. For features, we generally have one fewer dummy than the number of categories in that categorical variable, but for

that third type, she would have Basal = 0, Luminal A = 0, Luminal B = 1 and HER2 = 0.

1.5 A Bit of History: Statistics vs. Machine Learning

Starting in the next section, and recurring at various points in the book, I will bring in some statistical issues. But as seen above in a debate over the question, "Who *really* does prediction?", there is sometimes a friendly rivalry between the statistics and ML communities, even down to a separate terminology for each (see list, Appendix B). I'd like to put all this in perspective, especially since many readers of this book may have some background in statistics.

Many of the methods now regarded as ML were originally developed in the statistics community, such as k-Nearest Neighbor (k-NN), decision trees/random forests, logistic regression, the E-M algorithm, clustering and so on. These evolved from the linear models formulated way back in the 19th century, but which statisticians felt were inadequate for some applications. The latter consideration sparked interest in methods that had less-restrictive assumptions, leading to invention first of k-Nearest Neighbor and later other techniques.

On the other hand, two of the most prominent ML methods, support vector machines (SVMs) and neural networks, have been developed almost entirely outside of statistics, notably in university computer science departments. (Another method, *boosting*, has had major contributions from both factions.) Their impetus was not statistical at all. Neural networks, as we often hear in the media, were studied originally as a means to understand the workings of the human brain. SVMs were viewed simply in computer science terms — given a set of data points of two classes, how can we compute the best line or plane separating them?

As a former statistics professor who has spent most of his career in a computer science department, I have foot in both camps. I will present ML methods in computational terms, but with insights provided by statistical principles.

At first glance, ML may seem to consist of a hodgepodge of unrelated methods. But actually, they all deal, directly or indirectly, with estimating — *learning* — something called the *regression function*.[7] We'll lead up to this notion by a series of examples in this section.

1.6 Example: Bike Rental Data

Here we resume the bike sharing example. To make things easier, let's first look at predicting ridership from a single feature, temperature. Say the day's

outcome variables, i.e. predicting class membership, we have as many dummies as classes. The reasons for this distinction will come later in the book.

7. Again, as noted at the outset of this chapter, the term *regression* is much more general than the concept of linear regression that some readers may have learned in a statistics course.

temperature is forecast to be 28 degrees. (That itself is a prediction, but let's assume it's accurate.) How should we predict ridership for the day, using the 28 figure and our historical ridership dataset? A person we randomly ask on the street, without background in ML, might reason this way:

> Well, let's look at all the days in our data, culling out those of temperature closest to 28. We'll find the average of their ridership values, and use that number as our predicted ridership for this day.

Actually, the "person on the street" here would be correct! This in fact is the basis for many common ML methods. In this form, it's called the *k-Nearest Neighbors* (k-NN) method.

We could take, say, the 5 days with temperatures closest to 28:

```
> data(day1)
> tmps <- day1$temp
> dists <- abs(tmps - 28)  # distances of the temps to 28
> do5 <- order(dists)[1:5]  # which are the 5 closest?
> dists[do5]  # and how close are they?
[1] 0.005849 0.033349 0.045000 0.045000 0.084151
```

So, the 5 closest are quite close to 28, much less than 1.0 degree away. What were their ridership values?

```
> day1$tot[do5]
[1] 7175 4780 4326 5687 3974
```

So our predicted value for today would be the average of those numbers, about 5200 riders:

```
> mean(day1$tot[do5])
[1] 5188.4
```

Let *k* denote the number of "neighbors" we use (the method is called k-NN), in this case 5.

1.7 Central ML Issue: the Famous Bias-vs.-Variance Tradeoff

There are issues left hanging, notably:

> Why take the 5 days nearest in temperature to 28? It would seem that 5 is too small a sample. Is that enough? If not, how much is enough?

1.7.1 Overview

This illustrates the famous *Bias-vs.-Variance tradeoff*. The value *k* is called a *tuning parameter* or *hyperparameter*. How should we set it?

- Even those outside the technology world might intuitively feel that 5 is "too small a sample." There is too much variability in average ridership from one set of 5 days to another, even if their temperatures are near 28. This argues for choosing a larger value than 5 for **k**. **This is the *variance* issue.** Choosing too small a value for *k* brings us too much sampling variability.
- On the other hand, if we use, say, the 25 days with temperatures closest to 28, we risk getting some days whose temperatures are rather far from 28. Say for instance the 25th-closest day had a temperature of 35; that's rather hot. People do not want to ride bikes in such hot weather. If we include too many hot days in our prediction for the 28-degree day, we will have a tendency to underpredict the true value. So in such a situation, maybe having $k = 25$ is too large. **That's a *bias* issue.** Choosing too large a value of *k* may bring us into portions of the data that induce a systemic tendency to under- or overpredict.
- Clearly, variance and bias are at odds with each other: Given our dataset, we can reduce one only at the expense of the other. It is indeed a tradeoff, and is one of the most important notions in both ML and statistics. Clearly, it is central to choosing the values of tuning parameters/hyperparameters, and will arise frequently throughout this book.

If we use too small a **k**, i.e. try to reduce bias below what is possible on this data, we say we have *overfit*. Too large a **k**, i.e. an overly conservative one, is termed *underfitting*.

1.7.2 Relation to Overall Dataset Size

But there's more. How large is our overall dataset?

```
> nrow(day1)
[1] 731
```

Only 731 days' worth of data. Is that large enough to make good predictions? Why should that number matter? Actually, it relates directly to the Bias-Variance Tradeoff. Here's why:

In our example above, we worried that with $k = 25$ nearest neighbors, we might have some days among the 25 whose temperatures are rather far from 28. But if we had, say 2,000 days instead 731, the 25th-closest might still be pretty close to 28. In other words:

Let n denote our overall number of data points. The larger n is, the larger we can make k.

As noted, we do want to make k large, to keep variance low, but are worried about bias. The larger n is, the less likely we' ll have such problems.

1.7.3 Well Then, What Is the Best Value of k?

Mind you, this still doesn't tell us how to set a good, "Goldilocks" value of k. Contrary to website rumors, the truth is that there is no surefire way to choose the best **k**. However, various methods are used in practice that generally work pretty well. *This is a general theme in ML methodology* — there are no failsafe ways to choose hyperparameters, but there are ones that generally work well.

A rough rule of thumb, stemming from some mathematical theory,[8] is to follow the limitation

$$k < \sqrt{n} \tag{1.1}$$

i.e. the number of nearest neighbors should be less than the square root of the number of data points.

Even with that limitation, though, there still is the question of how to choose k within that imposed range. We will introduce the our first way of handling this, the "leaving one out" method, later in this chapter.

1.7.4 And What about p, the Number of Predictors?

A dataset is not "large" or "small" on its own. Instead, its size n must be viewed relative to the number of features p. Another way overfitting can arise is by using too many features for a given dataset size.

1.7.4.1 Example: ZIP Code as a Feature

For instance, say one of our features is ZIP code (postal zone in the US). Here is the tradeoff:

- There are more than 42,000 ZIP codes in the US. That would mean 42,000 dummy variables! If we have only, say, 100,000 rows in our data, on average each ZIP code would have only about 2 rows, hardly enough for a good estimate of the effect of that code. Again, the small sample size for a given ZIP code gives us a *variance* problem.
- On the other hand, suppose we have ZIP code but don't use it in our analysis. For instance, say in a marketing study we wish to predict whether a given customer will buy a heavy winter parka, and in addition to ZIP code, we have information on the customer's income, age and gender.. Since ZIP code roughly corresponds to ge-

8. There are a number of reasons why I don't use stronger language here, e.g. "proven by mathematical theory." The derivations are asymptotic, they involve unknown mulitiplicative constants and so on. We thus settle for saying this is a "rule of thumb."

ography in the US — there is a code 10001 in New York City and one 92027 in San Diego — ZIP tells us about the climate the customer lives in, very relevant information for parka purchases! If we were to NOT make use of ZIP code information, our estimates of the probability of a parka purchase would tend to be too high for San Diego customers, and too low for those in New York. In other words, we'd be biased upwards in San Diego and biased downwards in New York.

1.7.4.2 Bias Reflecting Lack of Detail

The above argument shows that the amount of detail in our data, say geography, may really matter. So, **bias can be viewed as a lack of detail.** But may we can find a middle ground in that regard.

In the ZIP code example, we choose, for instance, to use only the first digit of the ZIP code. We would still get fairly good geographic detail for the purpose of predicting parka purchases. That way, we'd have a lot of data points for each (abbreviated) ZIP code, thus addressing the variance issue.

Better yet, we might choose to use an *embedding*. We could fetch the average daily winter temperature per ZIP code from government data sites, and use that temperature instead of ZIP code as our feature.

1.7.4.3 Well Then, What Is the Best Value of p?

Again, the question of how many features is too many has vexed analysts from the time ML began (and earlier, from the start of statistics), but as mentioned, methods for dealing with this exist, and will be presented in this book, starting with Section 3.1.2.

A rough — though in my experience conservative — rule of thumb is to follow the limitation

$$p < \sqrt{n} \tag{1.2}$$

i.e. the number of features should be less than the square root of the number of data points.[9]

If our data frame has, say, 1000 rows, it can support about 30 features. Again, this should not be taken too literally, but it's not a bad rough guide.

However: Though the condition $p < \sqrt{n}$ is not universally held by all analysts, it was always taken for granted that one should have at least $p < n$. In modern statistics and ML, though, it is now common to have — or at least start with — a value of p much larger than n. We will see this with certain methods used later in the book. For now, let's stick with $p < \sqrt{n}$ as a reasonable rule.

9. For those reader with some math background, technically the theory requires that $p/ qrtn$ goes to 0 as n goes to infinity.

1.8 The Regression Function: What Is Being "Learned" in Machine Learning

In our example above, we took as our predicted value for a 28-degree day the mean ridership of all days of that temperature. (We actually took the mean of days *near* that temperature, a little different, but put that aside for the moment.) If we were to predict the ridership on a 15-degree day, we would use the mean ridership of all days of temperature 15, and so on.

Hmm, this sounds like a function! For every input (a temperature), we get an output (an associated mean). Stating this a bit more abstractly:

> We predict the ridership on a day of temperature t to be $r(t)$, the mean ridership of all days of temperature t. This function, $r(t)$, is called the *regression function* of ridership on temperature.

1.8.1 A Bit More on the Function View

The regression function is also termed the *conditional mean*. In predicting ridership from temperature, $r(28)$ is the mean ridership, subject to the *condition* that temperature is 28. That's a subpopulation mean, quite different from the overall population mean ridership.

A regression function has as many arguments as we have features. Let's take humidity as a second feature, for instance. So, to predict ridership for a day with temperature 28 and humidity 0.51, we would use the mean ridership in our dataset, among days with temperature and humidity approximately 28 and 0.51. In regression function notation, that's $r(28, 0.51)$.

Don't get carried away with this math abstraction, in which we are speaking of functions. But understanding ML methods, throughout this book, will of course require knowing *what* machine "learning" is "learning" — which is $r()$. We do this learning based on data for which we know the values of both the features and the outcome variable (e.g. temperature and humidity, and ridership). This data is known as the *training set*, the word "training" again reflecting the idea of "learning" the regression function.

1.8.2 But Wait --- Did You Say Exact Mean Ridership Given Temperature and Humidity?

In the field of statistics, a central issue is the distinction between *samples* and *populations*. For instance, during an election campaign, voter polls will be taken to estimate the popularity of the various candidates. Lists of say, telephone numbers, will be sampled, and opinions solicited from those sampled. We are interested in p, the proportion of the entire population who favor Candidate X. But since we just have a sample from that population, we can only *estimate* the value of p, using the proportion who like X in our *sample*. Accordingly, the poll results are accompanied by a *margin of error*, to recognize that the reported proportion is only an estimate of p.[10]

In the bike sharing example, we treat the data as a sample from the (rather conceptual) population of all days, past, present and future. Using the k-NN method (or any other ML method), we are only obtaining an estimate of the true population regression function $r()$. First, it's an estimate as it is based only on a sample of k data points. Second, we base that estimate by looking at data points that are *near* the given value, e.g. near 28 or near (28,0.51), rather than at that exact value.

This distinction between sample and population is central to statistics. And though this distinction is rarely mentioned in ML discussions, it is implicit, and is crucial there as well. As mentioned, the Bias-Variance Tradeoff is a key issue for statisticians and ML people alike, and it implies that there is some kind of population being sampled from. Even non-statisticians know that it's difficult to make inferences from small samples (small k), after all. Note too that predicting new cases depends on their being "similar" to the training data, i.e. from the same population.

In summary, we wish to learn the population regression function, but since we have only sample data, we must settle for using only estimated regression values. In discussing the various ML methods in this book, the question will boil down to: In any particular application using any particular feature set, which method will give us the most accurate estimates?[11]

By the way, in statistics it is common to use "hat" notation (a caret symbol) as shorthand for "estimate of." So, our estimate of $r(t)$ is denoted $\widehat{r}(t)$.

1.9 Informal Notation: the "X" and the "Y"

We're ready to do some actual analysis, but should mention one more thing concerning some loose terminology:

- Traditionally, one collectively refers to the features as "X", and the outcome to be predicted as "Y."
- Indeed, X typically is a set of columns in a data frame or matrix. If we are predicting ridership from temperature and humidity, X consists of those columns in our data. Y here is the ridership column. In classification applications, Y will typically be a column of 1s and 0s, or an R factor.
- In multiclass classification applications, Y is a set of columns, one for each dummy variable. Or Y could be an R factor.

Again, this is just loose terminology, a convenient shorthand. It's easier, for instance, to say "X" rather than the more cumbersome "our feature set."

One more bit of standard notation:

10. Those who have studied statistics may know that the margin of error is the radius of a 95% confidence interval.

11. There will also be another important factor, computation time. If the best method takes too long to run or uses too much memory, we may end up choosing a different method.

- The number of rows in X, i.e. the number of data points, is typically denoted by n.
- The number of columns in X, i.e. the number of feaures, is typically denoted by p.

1.10 So, Let's Do Some Analysis

1.10.1 Example: Bike Sharing Data

Our featured method in this chapter will be k-NN. It's arguably the oldest ML method, going back to the early 1950s, but is still widely used today (generally if the number of features is small, for reasons that will become clear later). It's also simple to explain, and easy to implement — thus the perfect choice for this introductory chapter.

For the bike sharing example, we'll predict total ridership. Let's start out using as features just the dummy for working day, and the numeric weather variables, columns 8, 10-13 and 16:

```
> day1 <- day1[,c(8,10:13,16)]
> head(day1)
  workingday     temp     atemp      hum windspeed  tot
1          0 8.175849  7.999250 0.805833 10.749882  985
2          0 9.083466  7.346774 0.696087 16.652113  801
3          1 1.229108 -3.499270 0.437273 16.636703 1349
4          1 1.400000 -1.999948 0.590435 10.739832 1562
5          1 2.666979 -0.868180 0.436957 12.522300 1600
6          1 1.604356 -0.608206 0.518261  6.000868 1606
```

1.10.1.1 Prediction Using kNN()

We can use the **kNN()** function in **regtools**.[12] The call form, without options (they'll come in shortly), is

```
kNN(x,y,newx,k)
```

where the arguments are:

- **x:** the "X" matrix for the training set (it cannot be a data frame, as nearest-neighbor distances between rows must be computed)
- **y:** the "Y" vector for the training set, in this case the **tot** column

12. The function is named "basic," because the package includes a more advanced set of k-NN tools, mainly the function **knnest()** and a couple of related routines. These make use of a much faster nearest-neighbor algorithm, and thus are preferable for large datasets. But in the interest of simplicity, we stick to the basic function.

- **newx:** a vector of feature values for a new case to be predicted, or a matrix of such vectors, where we are predicting several new cases
- **k:** the number of nearest neighbors we wish to use

So, say you are the manager this morning, and the day is a working day, with temperature 12.0, atemp 11.8, humidity 23% and wind at 5 miles per hour. What is your prediction for the ridership?

The arguments to **kNN()** will be

- **x: day1[,1:5]**, all the features but not the outcome variable **tot**
- **y: day1[,6]**, the **tot** column
- **newx:** a vector of the feaure values for the new case to be predicted, i.e. work day 1, temperature 12.0 and so on, **c(1,12.0,11.8,0.23,5),5)**
- **k:** the number of nearest neighbors; let's use 5

So, let's call the function:

```
> day1x <- day1[,1:5]
> tot <- day1$tot
> knnout <- kNN(day1x,tot,c(1,12.0,11.8,0.23,5),5)
> knnout
$whichClosest
     [,1] [,2] [,3] [,4] [,5]
[1,]  459  481  469  452   67

$regests
[1] 5320.2
```

The output shows which rows in the training set were closest to the point to be predicted — rows 459, 481 and so on — and the prediction itself. Our prediction would then be about 5320 riders. Let's see how this came about.

First, after the code found the 5 closest rows, it found the ridership values in those rows:

```
> day1[c(459,481,469,452,67),6]
[1] 6772 6196 6398 5102 2133
```

It then averaged the Y values:

```
> mean(day1[c(459,481,469,452,67),6])
[1] 5320.2
```

Note, though, that point 67 had a lower ridership, 2133. Could be an outlier, and it might be better to take the median than the mean:

```
> kNN(day1x,totx,c(1,12.0,11.8,0.23,5),5,smoothingFtn=median)
$whichClosest
         [,1] [,2] [,3] [,4] [,5]
kClosest  459  481  469  452   67

$regests
kClosest
    6196
```

Which prediction is better? We'll soon see ways to answer this question, though really, the reader must keep in mind that at the end of the day, it is up to the analyst.

By the way, **kNN()** allows us to make multiple predictions in one call, using a matrix instead of a vector for the argument **newx**. Each row of the matrix will consist of one point to be predicted.

So in our above example, say we wish to predict both work day and non-work day cases, with the weather conditions being as before:

```
> newx <- rbind(c(1,12.0,11.8,0.23,5),c(0,12.0,11.8,0.23,5))
> kNN(day1x,tot,newx,5)
$whichClosest
     [,1] [,2] [,3] [,4] [,5]
[1,]  459  481  469  452   67
[2,]  484  505  652  464  498

$regests
[1] 5320.2 6444.2
```

Interesting; a day being a work day makes quite a difference, with a predicted value of about 6444 for the non-work day, more than 1000 riders more than the work day case with the same weather conditions.

1.10.2 The mlb Dataset

Let's briefly discuss another dataset, then return to bike sharing. This other data, **mlb**, is also included in **regtools**. This is data on Major League Baseball players (provided courtesy of the UCLA Statistics Department):

```
> data(mlb)
> head(mlb)
             Name Team       Position Height
1   Adam_Donachie  BAL        Catcher     74
2       Paul_Bako  BAL        Catcher     74
3 Ramon_Hernandez  BAL        Catcher     72
```

```
4    Kevin_Millar  BAL  First_Baseman     72
5     Chris_Gomez  BAL  First_Baseman     73
6   Brian_Roberts  BAL Second_Baseman     69
  Weight   Age PosCategory
1    180 22.99     Catcher
2    215 34.69     Catcher
3    210 30.78     Catcher
4    210 35.43   Infielder
5    188 35.71   Infielder
6    176 29.39   Infielder
```

1.10.3 A Point Regarding Distances

The "near" in k-Nearest Neighbor involves distances. Those in turn involve sums of squares. In our ballplayer example, say we wish to predict the weight of player whose height is 74 and age is 28. In our sample data, one of the players is of height 70 and age 26. The distance between them[13] is

$$\sqrt{(70-74)^2 + (26-28)^2} = 4.47$$

The idea is that 78 is 4 away from 74 and 26 is 2 away from 28. We take differences, then square and add. Why square them? If we just added up raw differences, positive and negative values could largely cancel out, giving us a small "distance" between two data points that might not be close at all.

In an application with, say, five features, we would take the square root of the sum of five squared differences, rather than two as in the above example. In this manner, we can find all the distances, and then find the nearest to the point of interest.[14]

1.10.4 Scaling

A problem, though, is that this would place higher priority on height. It's much larger than age; it would thus dominate the distance computation, especially after squaring. We could change height from inches to centimeters, say, and this would work, but it's better to have something that always works.

For this, we divide each predictor/feature by its standard deviation. This gives everything a standard deviation of 1. We also subtract the mean, giving everything mean 0. Now all the features are commensurate. Of course, we must remember to do the same scaling — e.g. dividing by the same standard

13. Actually this is the Pythagorean Theorem from high-school geometry, but we need not go into that.
14. Though this sum-of-squares definition of distance is the time-honored one, some analysts prefer the sum of absolute values rather than squares (without the square root). They believe the traditional method places undue emphasis on large components. Our **preprocessx()** function does give this option.

deviations — in the X values of new cases that we predict, such as new days in our bike sharing example.

The function **kNN()** does scaling by default. We can turn this feature off by setting **scaleX=FALSE** in the call. For instance, we may wish to do this if we use the original version of the bike sharing data, **day**, rather than **day1**; there the weather variables are transformed to [0,1], i.e. a different kind of scaling. If we like that scheme, we might direct **kNN()** to refrain from any further scaling.

In the bike sharing data, **day1** uses unscaled data, for instructional purposes. The "official" version of the data, **day**, does scale. It does so in a different manner, though: They scale such that the values of the variables lie in the interval [0,1]. One way to do that is do transform by

$$x \rightarrow \frac{x - \min(x)}{\max(x) - \min(x)}$$

This has an advantage over **scale()** by producing bounded variables; **scale()** produces variables in the interval $(-\infty, \infty)$.

> Good practice: Keep scaling in mind, as it is used in many ML methods, and may produce better results even if not required by a method.

1.11 Choosing Hyperparameters

One nice thing about k-NN is that there is only one hyperparameter, k. Later in the book you'll find that most ML methods have several hyperparameters, in some cases even a dozen or more. Imagine how hard it is to choose the "right" combination of values for several hyperparameters. Yet even choosing a good value for that one is nontrivial.

As noted in our discussion of the bias-variance tradeoff, we need to find the "Goldilocks" value of k, not too large and not too small. Setting tuning parameters — also known as *hyperparameters* — is one of the most difficult problems for any ML method. But there are a number of ways to tackle the problem.

1.11.1 Predicting Our Original Data

Almost all methods for choosing tuning parameters involve simulation of prediction. In the most basic form, we go back and predict our original data. Sounds odd — we know the ridership values in the data, so why predict them? But the idea is that we try various values of k, and see which one predicts the best. That then would be the value of k that we use for predicting new X data in the future.

1.11.1.1 A First Try

So here we go! First, just as an illustration, let's predict

```
> knnout <- kNN(day1x,tot,day1x,8,allK=TRUE)
> str(knnout)
List of 2
 $ whichClosest: int [1:731, 1:8] 1 2 3 4 5 6 7 8 9 10 ...
 $ regests     : num [1:8, 1:731] 985 2817 2696 2746 2399 ...
```

Note that we set the **newx** argument to **day1x**, reflecting our plan to go back and predict our known data. But note that new option, **allK = TRUE**. What is this doing?

In choosing a good value of *k*, we'd like to try several of them at once. Setting **allK = TRUE** stipulates that we wish to try all the values 1,2,3,...*k*. The argument *k* is then specifying the largest value of *k* that we wish to try.

In the above call, we had *k = 8*, so we are going to get 8 sets of 731 predictions (since the Y data has 731 values to be predicted). Thus, **knnout$regests** here is an 8 × 731 R matrix. Let's look at a bit of it:

```
> knnout$regests[1:3,1:8]
         [,1]     [,2]   [,3]   [,4]     [,5] [,6]     [,7]     [,8]
[1,]  985.000  801.000 1349.0 1562.0 1600.000 1606 1510.000  959.000
[2,] 2817.000 2735.000 1449.5 2145.5 2381.500 1676 3442.500 1628.500
[3,] 2696.333 3901.667 1479.0 2462.0 2728.333 1591 2828.333 1998.667
```

Row 1 has all the predicted values for *k = 1*, row 2 has these for *k = 2* and so on.

Of course, we know the real Y values:

```
> tot[1:8]
[1]  985  801 1349 1562 1600 1606 1510  959
```

Now, wait a minute...these numbers are exactly what we see in row 1 of **knnout$regests**! In other words, *k = 1* seems to give us perfect predictions. That would seem to be quite at odds with the bias/variance tradeoff. Too good to be true? Well, yes...

Look at that first component of **knnout**:

```
 $ whichClosest: int [1:731, 1:8] 1 2 3 4 5 6 7 8 9 10 ...
```

That pattern certainly looks suspicious. Let's look closer:

```
> knnout$whichClosest[1:5,]
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1  702   99  100  725  709  275  716
```

```
[2,]    2  688  456   64  366  275  105  647
[3,]    3   34   41  384   10  385  430   84
[4,]    4  731  728   31   35   25   53   14
[5,]    5  385  409  431  322    7  395  706
```

Recall the **x** and **newx** arguments in **kNN()**. The (i,j) element in the above output tells us which row of **x** is the jth closest to row i of **newx**. Now remember, we had set both **x** and **newx** to **day1x**. So for instance that 2 element in the first column says that the closest row in **day1x** to the second row in **day1x** is...the second row in **day1x**! In other words, the closest data point is itself. So of course we'll (misleadingly) get perfect prediction by using $k = 1$.

1.11.1.2 Resolving the Problem

The **kNN()** function has an optional argument to deal with this. Setting **leave1out = TRUE** means, "In finding the k closest neighbors, skip the first, and take the second through the kth." Let's use that:

```
> knnout <- kNN(day1x,tot,day1x,8,allK=TRUE,leave1out=TRUE)
> str(knnout)
List of 2
 $ whichClosest: int [1:731, 1:7] 702 688 34 731 385 42 706 381 408 41 ...
 $ regests     : num [1:7, 1:731] 4649 3552 3333 2753 2848 ...
```

(The reason for the argument name "leave1out" will become clearer in future chapters.)

1.11.2 Evaluating Prediction Accuracy

Well, then, how well did we predict? For each value in **tot**, we take the absolute difference between the actual value and predicted values, then average those absolute differences to get the Mean Absolute Prediction Error (MAPE). We do this for each value of k, using another **regtools** function, **findOverallLoss()**:

```
> findOverallLoss(knnout$regests,tot)
[1] 1362.696 1181.111 1121.280 1071.653 1085.303
[6] 1069.830 1060.946
```

These are the MAPE values for $k = 1,2,...,7$, again not counting the data point to be predicted. The best value of k appears to be larger than 7, as MAPE was still decreasing at $k = 6$.

Disclaimer: There are numerous examples in this book, with one or more computer runs for each. There is no claim that any of these analyses is optimal. On the contrary, the reader is encouraged to find better solutions.

1.11.3 Loss Functions

A couple of further points, which will come up again at various points in this book.

Consider the prediction error for the i^{th} data point,

$$\text{predicted value}_i - \text{actual value}_i \tag{1.3}$$

This is called the *residual* for data point i.

What is “losss” in the function name **findOverallLoss()**? In computing MAPE, our loss for each data point was defined to be the absolute value of the residual for that point. In classification applications the loss may be defined in terms of perfect prediction.

For instance, in Section 2.2, the goal is to predict whether a customer of a phone service decides to leave to another provider, say coded 1 for yes, 0 for no. Then the predicted and actual values come from the set {0,1}. Here, the absolute residual works out to 0 or 1 as well, with 0 loss if we predicted correctly and 1 loss if not.

And look what happens to MAPE in this setting. It’s the average of those 0 and 1 loss values. Averaging a set of 0s and 1s just gives us the proportion of 1s. (Try it yourself, say with the numbers 0,1,1,0,1.) In other words, here MAPE boils down to our misclassification rate.

In classification applications with more than two classes, our loss again is defined in terms of whether predicted value$_i$ = actual value$_i$, though the misclassification rate can no longer be described in terms of MAPE.

A very common loss function is MSPE, Mean Squared Prediction Error, the average squared residual.

1.11.4 Cross-Validation

We saw above that, in predicting our original dataset in order to find a good value of k, we needed to “leave one out”: For each row i in our dataset, we predict that row via a nearest-neighbor analsis on *the rest* of our data. This is a special case of *cross-validation*, in which we predict one subset of rows from the rest. In the above presentation, our our subset size was 1, but in general it could be larger, in ways to be discussed in Chapter 3.

1.12 Can We Improve on This?

The average overall ridership in our sample is about 4500 (just call **mean(tot)**), so a MAPE value of about 1000 is substantial. But recall, we are not using all of our data. Look again at page 5. There were several variables involving timing of the observation: date, season, year and month.

To investigate the possible usefulness of the timing data, let’s graph ridership against row number (since the data is in chronological order):

```
> plot(1:731,tot,type='l')
```

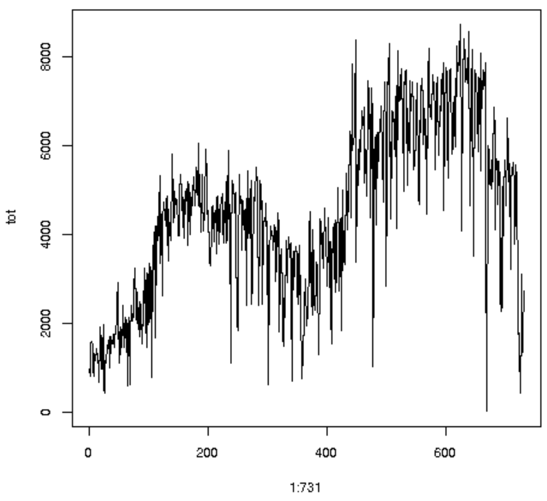

Figure 1-1: Time trends, bike data

The result is in Figure 1-1. Clearly, there is not only a seasonal trend, but also an overall upward trend. The bike sharing service seems to have gotten much more popular over time.

So let's add the day number, 1,2,...,731 as another feature, and try again:

```
> day1 <- data.frame(day1,dayNum=1:731)
> day1x <- day1[,c(1:5,7)]
> knnout <- kNN(day1x,tot,day1x,25,allK=TRUE,leave1out=TRUE)
> findOverallLoss(knnout$regests,tot)
 [1] 712.7346 630.1799 614.2239 594.9231 574.0337
 [6] 566.1569 569.5325 571.1919 569.5338 568.0204
[11] 569.3462 570.7351 573.9481 574.6321 579.2849
[16] 585.4124 587.3991 588.5394 588.1793 592.0328
[21] 593.0713 594.0240 597.2949 598.3143 600.7795
```

Ah, much better! The best *k* in this data is 6, but there is sampling variation and other issues at play here, but at least now we have a ballpark figure.

But we can probably improve accuracy even more. This is *time series* data. In such settings, one typically predicts the current value from the past several values of the same variable. In other words, we might predict today's value of **tot** from not just today's weather but also the value of **tot** in the last three days. We'll return to this topic in Chapter 14.

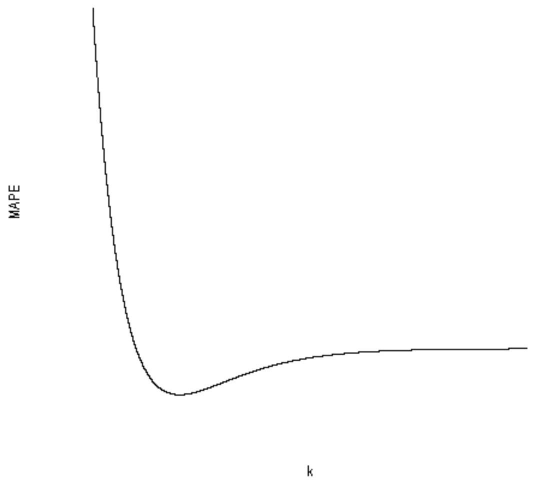

Figure 1-2: Typical mean loss vs. k

1.13 Going Further with This Data

We attained a huge improvement by adding the day number, 1 through 731, as a feature. But we may be able to make further improvements.

For instance, it would seem plausible that the summer vacation season would be an especially good indicator of higher ridership. Though that is already taken into account in the day number, it may be that giving it special emphasis would be helpful. Thus we may wish to try setting a new feature, a dummy variable for summer:

```
day1$summer <- (day1$season == 3)
```

The reader is invited to tackle this problem further.

1.14 General Behavior of MAPE/Other Loss As a Function of k

A typical shape of MAPE vs. *k* might look like that shown in Figure 1-2. As *k* moves away from 0, the reduction in variance dominates the small increase in bias. But after hitting a minimum, the bias becomes the dominant component. And for very large *k* the neighbors of a given point expand to eventually include the entire dataset. In that situation, our estimate $\widehat{r}(t)$ becomes the overall mean value of our outcome variable Y— so we make the same prediction no matter what value of X we are predicting from.

1.15 Pitfall: Beware of "p-Hacking"!

In recent years there has been much concern over something that has acquired the name *p-hacking*. Though such issues had always been known and discussed, it really came to a head with the publication of John Ioannidis' highly provocatively titled paper, "Why Most Published Research Findings Are False."[15] One aspect of this controversy can be desribed as follows.

Say we have 250 coins, and we suspect that some are unbalanced. (Any coin is unbalanced to at least some degree, but let's put that aside.) We toss each coin 100 times, and if a coin yields fewer than 40 or more than 60 heads, we will decide that it's unbalanced. For those who know some statistics, this range was chosen so that a balanced coin would have only a 5% chance of straying more than 10 heads away from 50 out of 100. So, while this chance is only 5% for each particular coin, with 250 coins, the chances are high that at least one of them falls outside that [40,60] range, *even if none of the coins is unbalanced*. We will falsely declare some coins unbalanced. In reality, it was just a random accident that those coins look unbalanced.

Or, to give a somewhat frivolous example that still will make the point, say we are investigating whether there is any genetic component to sense of humor. Is there a Humor gene? There are many, many genes to consider, many more than 250 actually. Testing each one for relation to sense of humor is like checking each coin for being unbalanced: Even if there is no Humor gene, eventually just by accident, we'll stumble upon one that seems to be related to humor, even if there actually is no such gene.

In a complex scientific study, the analyst is testing many genes, or many risk factors for cancer, or many exoplanets for possibility of life, or many economic inflation factors, etc. The term *p-hacking* means that the analyst looks at so many different factors that one is likely to emerge as "statistically significant" even if no factor has any true impact. A common joke is that the analyst "beats the data until they confess," alluding to a researcher testing so many factors that one finally comes out "significant."

Cassie Kozyrkov, Head of Decision Intelligence, Google, said it quite well:

> What the mind does with inkblots it also does with data. Complex datasets practically beg you to find false meaning in them.

This has major implications for ML analysis. For instance, a popular thing in the ML community is to have competitions, in which many analysts try their own tweaks on ML methods to outdo each other on a certain dataset. Typically these are classification problems, and "winning" means getting the lowest rate of misclassification.

The trouble is, having, say, 250 ML analysts attacking the same dataset is like having 250 coins in our example above. Even if the 250 methods they try are all equally effective, one of them will emerge by accident as the victor, and her method will be annointed as a "technological advance."

15. *PLOS Medicine*, August 30, 2005.

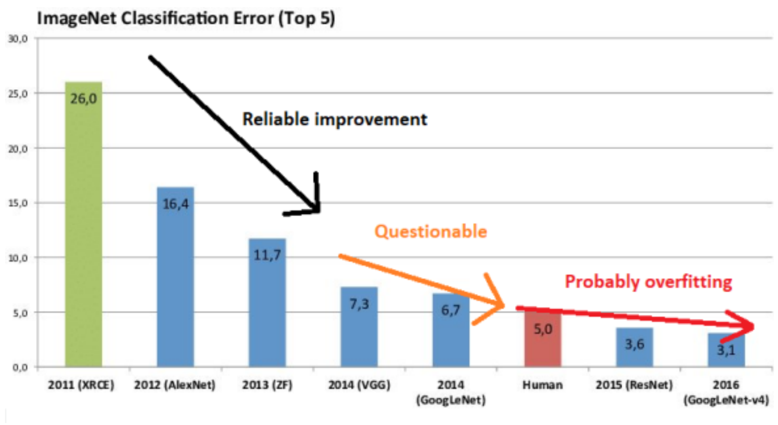

Figure 1-3: AI p-hacking

Of course, it may well be that one of the 250 methods really is superior. But without careful statistical analysis of the 250 data points, it is not clear what's real and what's just accident. Note too that even if one of the 250 methods is in fact superior, there is a high probability that it won't be the winner in the competition, again due to random variation.

As mentioned, this concept is second nature to statisticians, but it is seldom mentioned in ML circles. An exception is Luke Oakden-Rayner's blog post, "AI Competitions Don't Produce Useful Models," whose excellent graphic is reproduced in Figure 1-3 with Dr. Rayner's permission.[16]

Rayner uses a simple statistical power analysis to analyze ImageNet, a contest in ML image classification. He reckons that at least those "new records" starting in 2014 are overfitting, just noise. With more sophisticated statistical tools, a more refined analysis could be done, but the principle is clear.

This also has a big implication for the setting of tuning parameters. Let's say we have four tuning parameters in an ML method, and we try 10 values of each. That $10^4 = 10000$ possible combinations, a lot more than 250! So again, what seems to be the "best" setting for the tuning parameters may be illusory.

> **Good practice:** Any context in which some "best" entity is being calculated is at risk for the p-hacking phenomenon. The "best" may not be as good as it looks, and may even hide a better entity.

1.16 Pitfall: Dirty Data

Look at the entry in the bike sharing data for 2011-01-01.

```
> head(day1)
```

16. https://lukeoakdenrayner.wordpress.com/2019/09/19/ai-competitions-dont-produce-useful-models/

```
  instant     dteday season yr mnth holiday
1       1 2011-01-01      1  0    1       0
```

It has holiday = 0, meaning, no, this is not a holiday. But of course January 1 is a federal holiday in the US.

Also, although the documentation for the dataset states there are 4 values for the categorical variable **weathersit**, there actually are just values 1, 2 and 3:

```
> table(day1$weathersit)

  1   2   3
463 247  21
```

Errors in data are quite common, and of course an obstacle to good analysis. For instance, consider the famous Pima diabetes dataset. One of the features is diastolic blood pressure, a histogram of which is in Figure 1-4. There are a number of 0 values, medically impossible. The same is true for glucose and so on. Clearly, those who compiled the dataset simply used 0s for missing values. If the analyst is unware of this, his/her analysis will be compromised.

Another example is the New York City Taxi Data.[17] It contains pickup and dropoff locations, trip times and so on. One of the dropoff locations, if one believes the numbers, is in Antartica!

Whenever working with a new dataset, the analysis should do quite a bit of exploring, e.g. with **hist()** and **table()** as we've seen here.

There is also the possibility of multivariate outliers, meaning a data point that is not extreme in any one of its components, but viewed collectively in unusual. For instance, suppose a person is recorded as having height 74 inches (29.1 cm) and age 6. Neither that height nor that age would be cause for concern individually (assume we have people of all ages in our data), but in combination it seems quite suspicious. This is too specialized a topic for this book, but the interested reader would do well to start with the CRAN Task View on Anomaly Detection.

> **Good practice:** Assume any dataset is "dirty." Perform careful screening before starting analysis. This will also help you acquire a feel for the data.

1.17 Pitfall: Missing Data

In R, the value NA means the data is not available, i.e. missing. It is common that a dataset will have NAs, maybe many. How should we deal with this?

17. See e.g. *https://data.cityofnewyork.us/Transportation/2018-Yellow-Taxi-Trip-Data/t29m-gskq*.

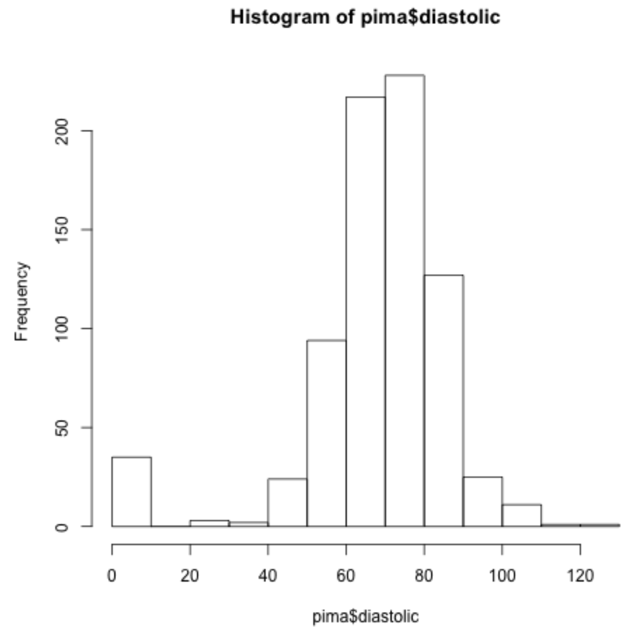

Figure 1-4: Diastolic blood pressure

One common method is *listwise deletion*. Say our data consists of people, and we have variables Age, Gender, Years of Education and so on. If for a particular person Age is missing but the other variables are intact, this method would simply skip over that case. But there are problems with this:

- If we have a large number of features, odds are that many cases will have at least one NA. This would mean throwing out a lot of precious data.
- Skipping over cases with NAs may induce a bias. In survey data, for instance, the people who decline to respond to a certain question may be different from those who do respond to it, and this may affect the accuracy of our predictions.

Missing value analysis has been the subject of much research, and we will return to this issue later in the book. But for now, note that when we find that missing values are coded numerically rather than as NAs, we should change such values to NAs. In the Pima data, for instance, blood pressure values of 0 should be changed to NA. Code for this would look something like this little example:

```
> w
[1] 102 140   0 129   0
> w[w == 0] <- NA
> w
[1] 102 140  NA 129  NA
```

1.18 Tweaking k-NN

The k-NN method is quite useful, but may be improved upon.

1.18.1 Weighted Distance Measures

In Section 1.10.4, we stressed the importance of scaling the data, so that all the features played an equal role in the distance computation. But we might consider deliberately weighting some features more than others.

In the bike sharing data, for instance, maybe humidity should be weighted more, or maybe wind speed should be weighted less. The **kNN()** function has an option for this, via the arguments **expandVars** and **expandVals**. The former indicates which features to reweight, and the latter gives the reweighting values.

1.18.2 Example: Programmer and Engineer Wages

Here we tried the method on the **prgeng** data from our **regtools** package. This is US Census data on programmers and engineers from the year 2000. We predicted wage income from age, various dummy education and occupation variables, and so on. We used $k = 25$ nearest-neighbors.

Figure 1-5 shows the Mean Absolute Prediction Error (MAPE) versus the expansion factor w; $w = 1.0$ means ordinary weighting, while $w - 0.0$ and $w = \infty$ corresponding to discarding the age variable or using it as the only predictor. The method was applied to a training set, then assessed on a holdout set of 1000 data points.

Interestingly, we see that *less* weight should be placed on the age variable.

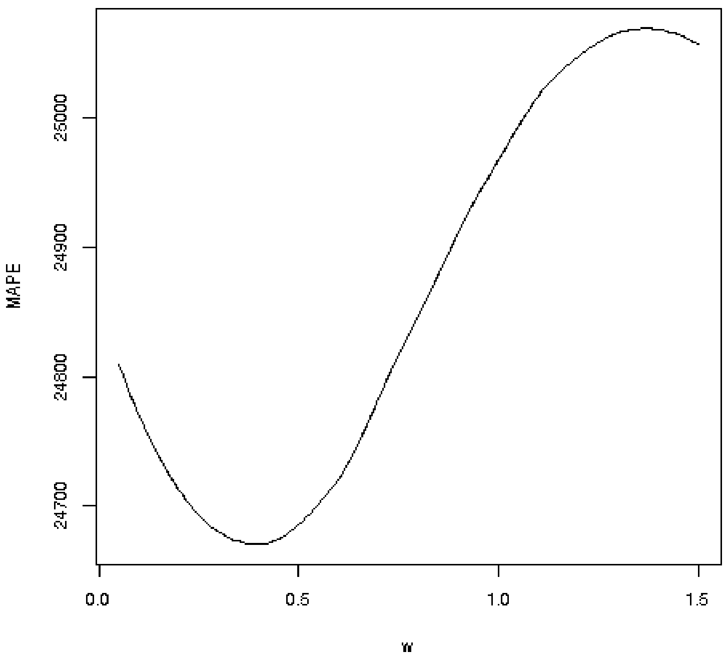

Figure 1-5: Effects of reweighting age

2

PROLOGUE: CLASSIFICATION MODELS

2.1 Classification Is a Special Case of Regression

Classification applications are quite common in ML. In fact, they probably form the majority of ML applications. How does the regression function $r(t)$ — recall, the mean Y for the subpopulation corresponding to $X = t$, i.e. a conditional mean for the condition $X = t$ — work then?

2.1.1 What Happens When the Response Is a Dummy Variable

Recall that in classification applications, the outcome is represented by dummy variables, which are coded 1 or 0. In a marketing study, for instance, the outcome might be represented by 1 — yes, the customer buys the product, or 0, no the customer declines to purchase.

So, after collecting our k nearest neighbors, we will be averaging a bunch of 1s and 0s, representing yes's and no's. Say for example k is 8, and the outcomes for those 8 neighbors are 0,1,1,0,0,0,1,0. The average is then

$$(0 + 1 + 1 + 0 + 0 + 0 + 1 + 0)/8 = 3/8 = 0.375$$

But it's also true that 3/8 of the outcomes were 1s. So you can see that:

> The average of 0-1 outcomes is the proportion of 1s. And this can be thought of as the probability of a 1.

So the regression function, a conditional mean, becomes a conditional probability in classification settings. In the marketing example above, it is the probability that a customer will buy a certain product, conditioned on his/her feature values, such as age, gender, income and so on. If the estimated probability is larger than 0.5, we would predict Buy, otherwise Not Buy.

The same is true in multiclass settings. Consider our cancer example earlier, with 4 types, thus 4 dummy variables. For a new patient, ML would give the physician 4 probabilities, one for each type, and the preliminary diagnosis would be the type with the highest estimated probability.

The reader may wonder why we use 4 dummy variables here. As noted in Section 1.2.2, just 3 should suffice. The same would be true here, but not for some other methods covered later on. For consistency, we'll always take this approach, with as many dummies as classes.

So, the function *r*() defined for use in predicting numeric quantities applies to classification settings as well; in those settings, means reduce to probabilities, and we use those to predict class. This is nice as a unifying concept. The algorithm used to *estimate* the *r*() function may differ in the two cases, as we'll see in the coming chapters, but the entity to be estimated is the same.

Note that it is customary in the ML world to speak of *regression applications* vs. *classification applications*, as we do in this book. Thus the word *regression* is used in two different senses — regression *applications* and the regression *function*. Use of the same word in two different contexts is common in computer and statistics, but one can easily avoid confusion here by keeping in mind these two separate terms. At any rate, the above discussion can be summarized by saying the regression *function* is central to both types of *applications*.

2.1.2 We May Not Need a Formal Class Prediction

Note that instead of making formal prediction of class, we may wish to simply report estimates of the class probabilities. Consider again the breast cancer example. Say two of the four types are essentially tied for having the highest probability. We may wish to report this rather than simply predict the one with the slightly higher probability.

Or, consider the example of the next section, in which we are predicting whether a phone service customer will switch service providers. Even if the estimated probability is less than 0.5, it may be a signal that further consideration is warranted, indeed mandated. If for instance the probability is, say, 0.22, it still may be worthwhile for the customer's current provider to contact her, giving her special perks, and so on, in order to try to retain her as a customer.

We will return to this issue in Section 2.6. But now, on to our first example.

2.1.3 An Important Note of Terminology

So, classification problems are really special cases of regression. This point will come up repeatedly in this book. However, it is also common to use the term "regression problem" to refer to cases in the *Y* variable is non-categorical, such as predicting bike ridership in our first example in the book.

2.2 Example: Telco Churn Data

In marketing circles, the term *churn* refers to customers moving often from one purveyor of a service to another. A service will then hope to identify customers who are likely "flight risks," i.e. have a substantial probability of leaving.

This example involves a customer retention program, in the Telco Customer Churn dataset.[1] Let's load it and take a look:

```
> telco <- read.csv('WA_Fn-UseC_-Telco-Customer-Churn.csv',header=T)
> head(telco)
  customerID gender SeniorCitizen Partner Dependents tenure
1 7590-VHVEG Female             0     Yes         No      1
2 5575-GNVDE   Male             0      No         No     34
3 3668-QPYBK   Male             0      No         No      2
4 7795-CFOCW   Male             0      No         No     45
5 9237-HQITU Female             0      No         No      2
6 9305-CDSKC Female             0      No         No      8
  PhoneService    MultipleLines InternetService
1           No No phone service             DSL
2          Yes               No             DSL
3          Yes               No             DSL
4           No No phone service             DSL
5          Yes               No     Fiber optic
6          Yes              Yes     Fiber optic
...
> names(telco)
 [1] "customerID"       "gender"
 [3] "SeniorCitizen"    "Partner"
 [5] "Dependents"       "tenure"
 [7] "PhoneService"     "MultipleLines"
 [9] "InternetService"  "OnlineSecurity"
[11] "OnlineBackup"     "DeviceProtection"
[13] "TechSupport"      "StreamingTV"
[15] "StreamingMovies"  "Contract"
[17] "PaperlessBilling" "PaymentMethod"
[19] "MonthlyCharges"   "TotalCharges"
[21] "Churn"
```

1. *https://www.kaggle.com/blastchar/telco-customer-churn*

That last column is the response; 'Yes' means the customer bolted.

2.2.1 Data Preparation

Many of these columns appear to be R factors. Let's check:

```
> for(i in 1:21) print(class(telco[,i]))
[1] "factor"
[1] "factor"
[1] "integer"
[1] "factor"
[1] "factor"
[1] "integer"
[1] "factor"
[1] "factor"
[1] "factor"
[1] "factor"
[1] "factor"
[1] "factor"
[1] "factor"
[1] "factor"
[1] "factor"
[1] "factor"
[1] "factor"
[1] "factor"
[1] "numeric"
[1] "numeric"
[1] "factor"
```

As noted in Section 1.2.3, many R ML packages accept factor variables, but some don't. Our **kNN()** function requires dummies, so we'll apply the **factorsToDummies()** utility from **regtools**. While we are at it, we'll remove the Customer ID column, which is useless to us:

```
> telco$customerID <- NULL
> tc <- factorsToDummies(telco,omitLast=TRUE)
> head(tc)
  gender.Female SeniorCitizen Partner.No Dependents.No
1             1             0          0             1
2             0             0          1             1
3             0             0          1             1
4             0             0          1             1
5             1             0          1             1
6             1             0          1             1
  tenure PhoneService.No MultipleLines.No
1      1               1                0
```

```
2     34              0                 1
3      2              0                 1
4     45              1                 0
5      2              0                 1
6      8              0                 0
...
```

Our data frame now has more columns than before:

```
> names(tc)
 [1] "gender.Female"
 [2] "SeniorCitizen"
 [3] "Partner.No"
 [4] "Dependents.No"
 [5] "tenure"
 [6] "PhoneService.No"
 [7] "MultipleLines.No"
 [8] "MultipleLines.No phone service"
 [9] "InternetService.DSL"
[10] "InternetService.Fiber optic"
[11] "OnlineSecurity.No"
[12] "OnlineSecurity.No internet service"
[13] "OnlineBackup.No"
[14] "OnlineBackup.No internet service"
[15] "DeviceProtection.No"
[16] "DeviceProtection.No internet service"
[17] "TechSupport.No"
[18] "TechSupport.No internet service"
[19] "StreamingTV.No"
[20] "StreamingTV.No internet service"
[21] "StreamingMovies.No"
[22] "StreamingMovies.No internet service"
[23] "Contract.Month-to-month"
[24] "Contract.One year"
[25] "PaperlessBilling.No"
[26] "PaymentMethod.Bank transfer (automatic)"
[27] "PaymentMethod.Credit card (automatic)"
[28] "PaymentMethod.Electronic check"
[29] "MonthlyCharges"
[30] "TotalCharges"
[31] "Churn.No"j
```

As noted before, if we convert an R factor with m levels to dummies, we create m-1 dummies, the -1 being indicated in our **factorsToDummies()** argument **omitLast=TRUE**. An exception is for categorical Y, where we will prefer to set **omitLast=FALSE**.

The original **gender** column is now a dummy variable, 1 for Female, 0 otherwise. Similar changes have been made for the other factors that had had two levels. On the other hand, look at the original **Contract** variable:

```
> table(telco$Contract)
Month-to-month       One year       Two year
          3875           1473           1695
```

It had 3 levels. As noted, for use as a feature (as opposed to an outcome variable), this means we need 3-1 = 2 dummies. In **tc** these two dummies are **Contract.Month-to-month** and **Contract.One year**. Numeric variables, e.g. **MonthlyCharges**, are unchanged.

But a warning — **factorsToDummies()** has reversed the roles of yes and no here. In the original data, Yes for the **Churn** variable meant yes, there was churn, i.e. the customer left for another provider. But **factorsToDummies()** created just one dummy, and since "No" comes alphabetically before "Yes," it created a **Churn.No** variable rather than a **Churn.Yes**. A value of 1 for that variable means there was no churn, i.e. happily our customer stayed put.

To avoid confusion, let's change the name of the 'Churn.No' column to 'Stay':

```
> dim(tc)
[1] 7032   31
> colnames(tc)[31] <- 'Stay'
> colnames(tc)
 [1] "gender.Female"
 [2] "SeniorCitizen"
 [3] "Partner.No"
...
[30] "TotalCharges"
[31] "Stay"
```

Another warning: As noted in Section 1.17, many datasets contain NA values. Let see if this is the case here:

```
> sum(is.na(tc))
[1] 11
```

Why, yes! We'll use listwise deletion here as a first-level analysis, but if we were to pursue the matter further, we may look more closely at the NA pattern. R actually has a **complete.cases()** function, returning TRUE for the rows that are intact:

```
> ccIdxs <- which(complete.cases(tc))
> tc <- tc[ccIdxs,]
```

If we don't need to know which particular cases are excluded, we could simply run

```
tc <- na.exclude(tc)
```

2.2.2 Fitting the Model

Again, a major advantage of k-NN is that we only have one hyperparameter, *k*. So, what value of *k* should we use for this data? As will be seen, not an easy question, but for now let's just see how to call the function. Say we set *k* to 75:

```
knnout <- kNN(tc[,-31],tc[,31],tc[,-31],75,allK=FALSE,leave1out=TRUE,
             classif=TRUE)
```

We have a new argument here, **classif=TRUE**, notifying **kNN()** that we are in a classification setting, rather than regression. Accordingly, we have an extra output, **ypreds** ("Y predictions"):[2]

```
> str(knnout)
List of 3
 $ whichClosest: int [1:7032, 1:75] 186 2254 1257 3032 4855 5866 1215
3896 6598 1880 ...
 $ regests     : num [1:7032] 0.507 0.92 0.653 0.947 0.347 ...
 $ ypreds      : num [1:7032] 1 1 1 1 0 0 1 1 0 1 ...
```

The **ypreds** component are the predicted values for Y, 1 (yes, the customer will stay) or 0 (no, the customer will leave), which are obtained by comparing the **regests** value to 0.5.

2.2.3 Pitfall: Failure to Do a ``Sanity Check"

The first time we use a package in a new context, it's wise to do a quick "sanity check," to guard against major errors in the way we called the function.

For instance, it can be shown mathematically that the mean of a conditional mean equals the unconditional mean. In our context here, that means the average of the **regests** should be about the same as the overall proportion of data points having Stay = 1. Let's see:

```
> mean(knnout$regests)
[1] 0.7073929
> table(telco$Churn) / sum(table(telco$Churn))
       No       Yes
0.7346301 0.2653699
```

2. This argument may be set only if **allK** is FALSE.

In that second call we ran **table()**, and indeed about 73% of the customers stayed. (Remember, this is all approximate, due to sampling variation, model issues etc.)

So, good, no obvious major errors in our call of **kNN()**. Again, this is just a check for big errors in our call. It does NOT mean that almost all our predictions were correct.

By the way, note the call to **sum()** after running **table()**. The latter is a vector, so this works out. Or, we could do

```
> mean(tc[,31])
[1] 0.734215
```

but if we had more than two clases, the table approach would be better, showing all the class proportions at a glance.

2.2.4 Fitting the Model (cont'd.)

Recall what **kNN()** returns to us — the row numbers of the nearest neighbors and the estimated regression function values. Since we are trying *k* up through 75, then for each of the 7043 cases in our data, we will have 75 nearest neighbors and 75 regression estimates:

For instance, consider row 143 in our data. If we were to use $k = 75$, what would be our estimate of $r(t)$ at that data point, and what would we predict for Y?

```
> knnout$regests[143]
[1] 0.8
> knnout$ypreds[143]
[1] 1
```

In other words, we estimate that for customers with characteristics like those of this customer, the probability of staying would be about 80%. Since that is larger than 0.5, we'd guess Y = 1, staying put. (The inclusion of the **ypreds** component is there just as a convenience.)

Well, then, how well did we predict overall?

```
> mean(knnout$ypreds == tc[,31])
[1] 0.7916667
```

To review the computational issue here: Recall that the expression

```
knnout$ypreds == tc[,31]
```

gives us a vector of TRUEs and FALSEs, which in the **mean()** context are treated as 1s and 0s. The mean of a bunch of 1s and 0s is the proportion of 1s, thus the proportion of correction predictions here.

There is a convenience function for this too:

```
> findOverallLoss(knnout$regests,tc[,31],probIncorrectClass)
[1] 0.2083333
```

We used this for MAPE loss before; **probIncorrectClass** loss does what the name implies, compute the overall probability of incorrect prediction.

But that was for 75 nearest neighbors. What about other values of k?

```
> knnout <- kNN(tc[,-31],tc[,31],tc[,-31],75,allK=TRUE,leave1out=TRUE)
> str(knnout)
List of 2
$ whichClosest: int [1:7032, 1:75] 186 2254 1257 3032 4855 5866 1215 3896 6598 1880 ...
$ regests: num [1:75, 1:7032] 0 0.5 0.333 0.5 0.4 ...
> findOverallLoss(knnout$regests,tc[,31],probIncorrectClass)
 [1] 0.2842719 0.3230944 0.2542662 0.2691980 0.2401877 0.2500000 0.2342150
 [8] 0.2400455 0.2290956 0.2335040 0.2219852 0.2261092 0.2221274 0.2245449
[15] 0.2162969 0.2199943 0.2168658 0.2184300 0.2160125 0.2155859 0.2127418
[22] 0.2143060 0.2107509 0.2154437 0.2131684 0.2128840 0.2104664 0.2107509
[29] 0.2098976 0.2108931 0.2107509 0.2107509 0.2081911 0.2098976 0.2086177
[36] 0.2094710 0.2083333 0.2100398 0.2070535 0.2079067 0.2069113 0.2101820
[43] 0.2070535 0.2086177 0.2067691 0.2097554 0.2071957 0.2084755 0.2081911
[50] 0.2106086 0.2089022 0.2090444 0.2089022 0.2096132 0.2083333 0.2098976
[57] 0.2089022 0.2103242 0.2093288 0.2106086 0.2097554 0.2104664 0.2083333
[64] 0.2100398 0.2081911 0.2093288 0.2089022 0.2084755 0.2074801 0.2080489
[71] 0.2084755 0.2090444 0.2073379 0.2081911 0.2083333
```

The minimum value occurs for $k = 45$. However, as noted earlier, these numbers are subject to sampling variation, so we should not take this result too literally. But it's clear that performance tapers off a bit after the 40s, and a reasonable strategy for us to choose for k would be somewhere around that value.

2.2.5 Pitfall: Factors with Too Many Levels

Suppose we had not removed the **customerID** column in our original data, **telco**. How many distinct IDs are there?

```
> length(levels(telco$customerID))
[1] 7043
```

Actually, that also is the number of rows; there is one record per customer. Thus customer ID is not helpful information.

If we had not removed this column, there would have been 7042 columns in **tc** just stemming from this ID column! Not only would the result be unwieldy, but also the presence of all these meaningless columns would dilute the power of k-NN.

So, even with ML packages that directly accept factor data, one must keep an eye on what the package is doing — and what we are feeding into it.

> **Good practice:** Watch out for R factors with a large number of levels. They may appear useful, and may in fact be so. But they can also lead to overfitting and computational/memory problems.

2.3 Example: Vertebrae Data

Consider another UCI dataset, Vertebral Column Data.[3], described by the curator as a "Data set containing values for six biomechanical features used to classify orthopaedic patients into 3 classes (normal, disk hernia or spondilolysthesis)." They abbreviate the three classes by NO, DH and SP.

Our "Y" data here will then consist of three columns of dummy variables, one for each class. Say the fifth patient in the data is of class DH, for example. Then row 5 in Y will consist of (0,1,0) — no, the patient is not NO; yes, the patient is DH; and no, the patient is not SP.

2.3.1 Data Prep

```
> vert <- read.table('column_3C.dat',header=FALSE)
> head(vert)
     V1    V2    V3    V4     V5    V6 V7
1 63.03 22.55 39.61 40.48  98.67 -0.25 DH
2 39.06 10.06 25.02 29.00 114.41  4.56 DH
3 68.83 22.22 50.09 46.61 105.99 -3.53 DH
4 69.30 24.65 44.31 44.64 101.87 11.21 DH
5 49.71  9.65 28.32 40.06 108.17  7.92 DH
6 40.25 13.92 25.12 26.33 130.33  2.23 DH
```

The patient status is V7. (We see, by the way, that the curator of the dataset decided to group the rows by patient class.)

We also see that V7 is in R factor form. We'll need to convert it to dummies for **kNN()**.

```
> vert1 <- factorsToDummies(vert,omitLast=FALSE)
> head(vert1)
     V1    V2    V3    V4     V5    V6 V7.DH V7.NO V7.SL
1 63.03 22.55 39.61 40.48  98.67 -0.25     1     0     0
2 39.06 10.06 25.02 29.00 114.41  4.56     1     0     0
3 68.83 22.22 50.09 46.61 105.99 -3.53     1     0     0
4 69.30 24.65 44.31 44.64 101.87 11.21     1     0     0
5 49.71  9.65 28.32 40.06 108.17  7.92     1     0     0
6 40.25 13.92 25.12 26.33 130.33  2.23     1     0     0
```

3. http://archive.ics.uci.edu/ml/datasets/vertebral+column

Note that classes DH, NO and SL have been given numeric IDs 0, 1 and 2, respectively. This will be used in the **ypreds** component below.

As an example, consider a patient similar to the first one in our data, but with V2 being 18 rather than 22.55. What would be our predicted class, say using 10 nearest neighbors?

```
> x <- vert1[,1:6]
> y <- vert1[,7:9]
> newx1 <- x[1,]
> newx1[2] <- 18.00
> kNN(x,y,newx1,10,allK=FALSE,classif=TRUE)
$whichClosest
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    4  173  205    3  175   56  236  220   269

$regests
     V7.DH V7.NO V7.SL
[1,]   0.4   0.3   0.3

$ypreds
[1] 0
```

We'd estimate that class DH (class code 0, as noted earlier) would be the most likely, with estimated probability 0.4.

2.3.2 Choosing Hyperparameters

Let's explore using various values of k. The software doesn't accommodate **allK = TRUE** in the multiclass case, so let's try them one-by-one.

We only have 310 cases here, in contrast to the $n = 7032$ we had in the customer churn example. Recall the comment in Chapter 1:

> The larger our number of data points n is, the larger we can set the number of nearest neighbors k.

for this smaller dataset, we should try smaller values of k. The output of the code

```
for(i in 2:20) {
   knnout <- kNN(x,y,x,i,leave1out = TRUE)
   oL <- findOverallLoss(knnout$regests,y,probIncorrectClass)
   cat(i,': ',oL,'\n',sep='')
}
```

is

```
2: 0.216129
```

```
3: 0.2225806
4: 0.2193548
5: 0.2064516
6: 0.2096774
7: 0.2129032
8: 0.216129
9: 0.2129032
10: 0.2064516
11: 0.1935484
12: 0.1967742
13: 0.2064516
14: 0.216129
15: 0.2193548
16: 0.2193548
17: 0.216129
18: 0.2096774
19: 0.2129032
20: 0.216129
```

So, it looks like $k = 11$ or 12 would be a good choice. But again, these accuracy levels are subject to sampling error, so we cannot take that 11 or 12 value literally.

2.3.3 Typical Graph Shape

The situation described in Section 1.14 and in Figure 1-2 holds here in the classification case as well. Our measure of loss now is classification error rate, but the same analysis as to graph shape holds.

2.4 Pitfall: Are We Doing Better Than Random Chance?

In Section 2.2, we found that we could predict customer loyality about 79% of the time. Is 79% considered a "good" level of accuracy? This of course depends on what our goals are, but consider this:

```
> mean(tc[,31])
[1] 0.7346301
```

About 73% of the customers are loyal, and so if we just predict everyone to stay put, our accuracy will be 73%. Using the customer information to predict, we can do somewhat better, and that 5 or 6% improvement is important, but it is not a dramatic gain.

So, keep in mind:

> **Pitfall:** A seemingly "good" error rate may be little or no better than random chance. Check the unconditional class probabilities.

Let's do this analysis for the example in Section 2.3, where we achieved an error rate of about 19%. If we were to not use the features, how would we do?

To that end, let's see what proportion each class has, ignoring our six features. Recalling that the average of a bunch of 1s and 0s is the proportion of 1s, we can answer this question easily.

```
> colMeans(y)
    V7.DH     V7.NO     V7.SL
0.1935484 0.3225806 0.4838710
```

So, if we did not use the features, we'd always guess the SL class, as it is the most common. We would then be wrong a proportion of 1 - 0.4838710 = 0.516129 of the time, much worse than the 19% error rate we attained using the features. So, yes indeed, the features do greatly enhance our predictive ability in this case.

2.5 Diagnostic: the Confusion Matrix

In multiclass problems, the overall error rate is only the start of the story. The next step is to calculate the (unfortunately-named) *confusion matrix.* To see what that is, let's get right into it:

```
> confusion(vert$V7,preds$ypreds)
      pred
actual   0   1   2
     0  43  16   1
     1  19  71  10
     2   4  14 132
```

Of the 43+16+1 = 60 datapoints with actual class 0 (DH), 43 were correctly classified as class 0, but 16 were misclassified as class 1, and 1 was wrongly predicted at class 2.

This type of analysis enables a more finely-detailed assessment of our predictive power.

2.6 Clearing the Confusion: Unbalanced Data

Recall that in our customer churn example earlier in this chapter, about 73% of the customers were "loyal," while 27% moved to another telco. With the 7042 cases in our data, those figures translate to 5141 loyal cases and 1901 cases of churn. Such a situation is termed *unbalanced*.

Many books, Web tutorials etc. on machine learning (ML) classification problems recommend that if one's dataset has unbalanced class sizes, one should modify the data to have equal class counts. Yet this is both unnecessary and harmful.

Illustrations of the (perceived) problem and offered remedies appear in numerous parts of the ML literature, ranging from Web tutorials[4] to major CRAN packages, such as **caret** and **mlr3**. In spite of warnings by statisticians,[5] all of these sources recommend that you artificially equalize the class counts in your data, via various resampling methods. This is generally inadvisable, indeed harmful, for several reasons:

- Downsampling is clearly problematic: Why throw away data? Discarding data weakens our ability to predict new cases.
- The data may be unbalanced for a reason. Thus the imbalance itself is useful information, again resulting in reduced predictive power if it is ignored.
- There is a simple, practical alternative to resampling.

So, there is not a strong case for artificially balancing the data.[6]

2.6.1 Example: Missed-Appointments Dataset

This is a dataset from Kaggle,[7] a firm with the curious business model of operating data science competitions. The goal here is to predict whether a patient will fail to keep a doctor's appointment; if one could flag the patients at risk of not showing up, extra efforts could be made to avoid the economic losses resulting from no-shows.

About 20% of the cases are no-shows.

```
> ma <- read.csv('KaggleV2-May-2016.csv',header=T)
> table(ma$No.show)
   No   Yes
88208 22319
```

Thus, in line with our discussion in Section 2.4, absent feature data, our best strategy would be to guess that everyone *will* show up. Let's see if use of feature data will change that.

```
> idxs <- sample(1:nrow(ma),10000)
> ma1 <- ma[,c(3,6,7:14)]
> ma2 <- factorsToDummies(ma1,omitLast=TRUE)
```

4. *https://www.datacamp.com/community/tutorials/diving-deep-imbalanced-data*
5. *https://www.fharrell.com/post/classification/*
6. Advocates of rebalancing sometimes imbalance can cause a parametric model, say logistic regression (Chapter 7) to "fit the dominant class." Actually, due to a concept known as *leverage*, the opposite is more likely; the cases in the smaller class will drag the fit towards themselves. In any case, there is no inherent reason to believe that rebalancing would ameliorate this problem.
7. *https://www.kaggle.com/joniarroba/noshowappointments*

To save computation time, we are predicting only a random sample of 10000 of the data points. We've also used only some of the features. (One additional possibly helpful feature would be day of the week, since many no-shows might occur on, say, Mondays. This could be derived from the **ScheduledDay**) column.)

Keep in mind the nature of the dummies here:

```
> colnames(ma2)
 [1] "Gender.F"
 [2] "Age"
 [3] "Neighbourhood.AEROPORTO"
 [4] "Neighbourhood.ANDORINHAS"
...
82] "Neighbourhood.UNIVERSITÁRIO"
[83] "Scholarship"
[84] "Hipertension"
[85] "Diabetes"
[86] "Alcoholism"
[87] "Handcap"
[88] "SMS_received"
[89] "No.show.No
```

So, class 1 means the patient *did* keep the appointment.

So, how well can we predict with these features?

```
> preds <- kNN(ma2[,-89],ma2[,89],ma2[idxs,-89],50)
> table(preds$ypreds)
  0    1
 53 9947
```

Indeed, even with the available feature data, we still are almost always predicting that people will show up (class 1)! This is exactly the problem cited by the above sources who recommend resampling the data. But let's look carefully at their remedy.

They recommend that all classes should be represented equally in the data, which of course is certainly not the case here, with 88,208 in class 0 and 22,319 in class 1. They recommend one of the following remedies:

- Downsample: Replace the class 1 data by 22,319 randomly chosen elements from the original 88,208.
- Upsample: Replace the class 0 data by 88,208 randomly selected elements from the original 22,319 (with replacement).
- Resample: Form an entirely new dataset of 88,208 points, by randomly sampling (with replacement) from the original data, but with weights so that there will be 44,104 points from each class.

One would apply one's ML method to the modified data, then predict new cases to be no-shows according to whether the estimated conditional probability is larger than 0.5 or not.

Clearly downsampling is undesirable; data is precious, and shouldn't be discarded. But none of the approaches makes sense. Among other things, they assume equal severity of losses from false negatives and false positives, which is unlikely here.

One could set up formal utility values here for the relative costs of false negatives and false positives. But the easier solution by far is to simply flag the cases in which there is a substantial probability of a no-show. Consider this:

```
> table(preds$regests)
0.34  0.4 0.42 0.44 0.46 0.48  0.5 0.52 0.54 0.56
   5    6    2    7   10    9   14   26   39   47
0.58  0.6 0.62 0.64 0.66 0.68  0.7 0.72 0.74 0.76
  89  143  156  205  273  343  340  480  585  631
0.78  0.8 0.82 0.84 0.86 0.88  0.9 0.92 0.94 0.96
 757  840  861  901  847  778  621  437  285  153
0.98    1
  62   48
```

So there are quite a few patients who, though having a probability > 0.5 of keeping their appointments, still have a substantial risk of no-show. For instance, there are 2731 patients who have at least a 25% of not showing up:

```
> sum(preds$regests < 0.75)
[1] 2779
```

So the reasonable approach here would be to decide on a threshold for no-show probability, then determine which patients fail to meet that threshold. We would then make extra phone calls, explain penalties for missed appointments and so on, to this group of patients,

In other words,

> **Good practice:** The practical solution to the unbalanced-data "problem" is not to artificially resample the data but instead to **identify individual cases of interest**.

3
PROLOGUE: DEALING WITH LARGER OR MORE COMPLEX DATA

In many applications these days, we have "Big Data." What does that mean?" There is no universal definition, so let's just discuss "larger data." Roughly we might say it occurs if one or more of the following conditions hold:

- Large *n*: Our data frame has a large number of rows, say in the hundreds of thousands or even hundreds of millions.
- Large *p*: Our data frame has a large number of columns, say hundreds or more.
- Complex algorithm: Many ML algorithms need much more computation than their classical statistical counterparts. As one writeup of AlexNet, a particular neural network for image classifica-

tion reported,[1] "The network takes...five or six days to train on two [extremely powerful computers]." This is an extreme case, of course, but it certainly illustrates that speed and memory size are issues.

What challenges does Big Data bring? Clearly, there is a potential computation time problem. Actually, computation can be a problem even on "large-ish" datasets, such as some considered in this chapter. And we'll see later in this chapter that it may become especially problematic when we try to use the "leaving one out" method discussed in earlier chapters.

Similarly, memory requirements may also be an issue. A dataset with a million rows and 1000 columns has a billion elements. At 8 bytes each, that's 8 gigabytes of RAM!

But there are also other issues that are equally important, actually more so. Overfitting is a prominent example, particularly in "short and squat" data frames in which p is a substantial fraction of n, maybe even a lot larger than n.

We'll look at these aspects:

- Dimension reduction: How can we reduce the number of columns in our data frame?
- Overfitting: With many columns (even after dimension reduction), the potential for overfitting becomes more acute. How can we deal with this?
- The p-hacking problem (Section 1.15): With many columns and/or many hyperparameters, the potential for finding some set of features and/or

1. https://medium.com/@smallfishbigsea/a-walk-through-of-alexnet-6cbd137a5637

some combination of hyperparameter values that looks really good but actually is a random, meaningless artifact is very high. How do we guard against that?

3.1 Dimension Reduction

Many of the tools we'll use here will be aimed at reducing p, i.e. *dimension reduction*. This has two goals, both equally important:

- Avoid overfitting. If we don't have $p < \sqrt{n}$ (Section 1.7.4.3), we should consider reducing p.
- Reduce computation. With larger data sets, k-NN and most of the methods in this book will have challenging computational requirements.

3.1.1 Example: Million Song Dataset

Say in the course of some investigation, you've run into a recording of an old song, with no identity, and you wish to identify it. Knowing the year of release might be a clue, and the Million Song Dataset may help us in this regard.

3.1.1.1 Data

You may download the data from the UCI site[2] (actually only about 0.5 million songs in this version). Due to the size, I chose to read the data using the **fread()** function from the **data.table** package:

```
> ms <- fread('YearPredictionMSD.txt',header=FALSE)
> dim(ms)
[1] 515345     91
> ms[1,]
     V1       V2       V3      V4      V5        V6
1: 2001 49.94357 21.47114 73.0775 8.74861 -17.40628
          V7        V8        V9     V10      V11     V12
1: -13.09905 -25.01202 -12.23257 7.83089 -2.46783 3.32136
...
```

That first column is the year of release, followed by 90 columns of arcane audio measurements. So column 1 is our outcome variable, and the remaining 90 columns are (potential) features.

2. https://archive.ics.uci.edu/ml/datasets/YearPredictionMSD

3.1.2 Why Reduce the Dimension?

We have 90 features here. With over 500,000 data points, the rough rule of thumb $p < \sqrt{n}$ from Section 1.7.4 says we probably could use all 90 features without overfitting. But k-NN requires a lot of computation, so dimension is still an important issue.

As an example of the computational burden, let's see how long it takes to predict one data point, say the first row of our data:

```
> system.time(knnout <- kNN(ms[,-1],ms[[1]],ms[1,-1],25))
   user  system elapsed
  3.088   0.992   4.044
```

You might wonder about that second argument, *ms[[1]]*, representing the first column, release year of a song. It's needed because **fread()** creates a data table, not a data frame.

Anyway, that's 4 seconds just for predicting one data point. If we predict the entire original dataset, over 500,000 data points (as in Section 1.11.1), the time needed would be prohibitive.

Thus we may want to cut down the size of our feature set, whether out of computational concerns as was the case here, or because of a need to avoid overfitting.

Feature selection is yet another aspect of ML that has no perfect solution. To see how difficult the problem is, consider the following possible approach, based on predicting our original data, as in Section 1.11.1. There we tried to find the best *k* by seeing how well each *k* value fared in predicting total ridership in our original dataset.

Here we could do the same: For each subset of our 90 columns, we could predict our original dataset, then use the set of columns with the best prediction record. There are two big problems here:

- We'd need tons of computation time. The 2-feature sets alone would number over 4,000. The number of feature sets of all sizes is 2^{90}, one of those absurd figures like "number of atoms in the universe" that one often sees in the press.
- We'd risk serious p-hacking issues. The chance of some pair of columns accidentally looking very accurate in predicting release year is probably very high.

But a number of approaches have been developed, one of the more common of which we will take up now.

3.1.3 A Common Approach: PCA

Principal components analysis (PCA) will replace our 90 features by 90 new ones. Sounds like no gain, right? We're hoping to *reduce* the number of features.

But these new features, known as *principal components* (PCs), are different. It will be a lot easier to choose subsets among these new features than among the original 90. They have two special properties:

- The PCs are uncorrelated. One might think of them as not duplicating each other. Why is that important? If, after reducing our number of predictors, some of them partially duplicated each other, it would seem that we should reduce even further. Thus having uncorrelated features means unduplicated features, and we feel like we've achieved a minimal set.
- The PCs are arranged in order of decreasing variance. Since a feature with small variance is essentially constant, it would seem unlikely to be a useful feature. So, we might retain just the PCs that are of substantial size, say m of them, and since the sizes are ordered, that means retaining the first s PCs, say. Note that m then becomes another hyperparameter, in addition to k.

Each PC is a *linear combination* of our original features, i.e. sums of constants times the features. If for instance the latter were Height, Weight and Age, a PC might be, say, 0.12 Height + 0.59 Weight - 0.02 Age. For the purposes of this book, where our focus is Prediction rather than Description, those numbers themselves are not very relevant. Instead the point is that we are creating new features as combinations of the original ones.

With these new features, we have a lot fewer candidate sets to choose from:

- The first PC.
- The first two PCs.
- The first three PCs.
- Etc.

So, we are choosing from just 90 candidate feature sets, rather than the unimaginable 2^{90}. Remember, smaller sets are better. It saves computation time (for k-NN now, and other methods later) and helps avoid overfitting and p-hacking.

Base-R includes several functions to do PCA, including the one we'll use here, **prcomp()**. It's used within the code of *kNN()* when the user requests PCA. CRAN also has packages that implement especially fast PCA algorithms, very useful for large datasets, such as **RSpectra**.

3.1.3.1 PCA in the Song Data

As an artificial but quick and convenient example of k-NN prediction using PCA, say we have a new case whose 90 audio measures are the same as the first song in our dataset, except that there is a value 32.6 in the first column. Again, just as an example, say we've decided to use the first 20 PCs, and 50 for our k value. Here is the code:

```
> newx <- ms[1,-91]
> newx[1] <- 32.6
> kNN(ms[,-1],ms[[1]],newx,50,PCAcomps=20)
$whichClosest
       [,1]   [,2]   [,3]   [,4]   [,5]   [,6]  [,7]   [,8]
[1,] 349626 456006 510959 259656 169737 351353 61427 181601
...

$regests
[1] 1997.9
```

Ah, we'd guess the year 1998 as release date.

Be sure to note what transpired here: **kNN()** took the original **ms** data (X portion), and applied PCA to it, extracting the first 20 PCs. It then transformed **newx** to PCs, using the coefficients found for **ms**. Then, and only then, did it actually apply the k-NN method.

3.1.3.2 Choosing k and the Number of PCs

One issue to deal with is that now we have two hyperparameters, our old one k and now the number of PCs, which we'll denote here by m. At the end of Section 1.12, we tried a range of values for k, but now we'll have to add in a range of values for m.

Say we try a range of 10 values for each, i.e. each of 10 k values paired with each of 10 m values. That's $10 \times 10 = 100$ pairs to try! Recall, there are two major problems with this:

- Each pair will take substantial computation time, and 100 may thus be more time than we wish to endure.
- Again, we must worry about p-hacking. One of those 100 pairs may seem to perform especially well, i.e. predict song release year especially accurately, simply by coincidence.

So, just as we reduced the number of features, we should consider reducing the number of (k,m) pairs that we examine. To that end, let's look at the variances of the PCs (squares of standard deviations):

```
> pcout <- prcomp(ms[,-91])
> pcout$sdev^2
 [1] 4.471212e+06 1.376757e+06 8.730969e+05 4.709903e+05
 [5] 2.951400e+05 2.163123e+05 1.701326e+05 1.553153e+05
 [9] 1.477271e+05 1.206304e+05 1.099533e+05 9.319449e+04
...
```

That "e+" notation means powers of 10. The first one, for instance, means 4.471212×10^6.

You can see the variances are rapidly decreasing. The 12th one, for instance, is about 90,000, which is small relative to the first, over 4 million. So we might decide to take $m = 12$. Once we've done that, we are back to the case of choosing just one hyperparameter, k. Another approach will be illustrated later in this chapter.

3.1.3.3 Should We Scale the Data?

One issue regarding PCA is whether to scale our data before applying PCA. The documentation for **prcomp** says this is ”advisable.” This is a common recommendation in the ML world, and in fact some authors consider it mandatory. However, there are better approaches, as follows.

First, keep in mind:

> The goal of PCA is to derive new features from our original ones, retaining the new high-variance features, while discarding the new low-variance onces.

What then is the problem people are concerned about? Consider two versions of the same dataset, one with numbers in the metric system, and the other in the British system. Say one of our features is Travel Speed, per hour. Since one mile is about 1.6 kilometers, the variance of Travel Speed in the metric dataset will be 1.6^2 times that in the British-system set. As a result, PCA will be more likely to pick up Travel Speed in the top PCs in the metric set, whereas intuitively the scale shouldm't matter.

Scaling, i.e. dividing each feature by its standard deviation (typically after subtracting its mean) does solve this problem, but its propriety is questionable, as follows.

Consider a setting with two features, A and B, with variances 500 and 2, respectively. Let A' and B' denote these features after centering and scaling.

As noted, PCA is all about removing features with small variance, as they are essentially constant. If we work with A and B, we would of course use only A. But if we work with A' and B', we would use both of them, as they both have variance 1.0.

So, dealing with the disparate-variance problem (e.g. miles vs. kilometers) shouldn't generally be solved by ordinary scaling, i.e. by dividing by the standard deviation. What can be done instead?

- Do nothing. In many data sets, the features of interest are already commensurate. Consider survey data, say, with each survey question asking for a response on a scale of 1 to 5. No need to transform the data here, and worse, standardizing would have the distoritionary effect of exaggerating rare values in items with small variance.
- Map each feature to the interval [0,1], i.e. t -> (t-m)/(M-m), where m and M are the minimum and maximum values of the given feature. We discussed this earlier, in Section 1.10.4. This is typically better than standardizing, but it does have some problems. First, it is sensitive to outliers. This might be ameliorated with a modified form of the transformation, but a second problem is that new data – new data in prediction applications, say – may stray from this [0,1] world.
- A variance is only large in relation to the mean; this is the reason for the classical measure *coefficient of variation*, the ratio of the standard deviation to the mean. So, instead of changing the *standard deviation* of a feature to 1.0, change its *mean* to 1.0, i.e. divide each feature by its mean.

This addresses the miles-vs.-kilometers concern more directly, without inducing the distortions I described above. And if one is worried about outliers, then divide the feature by the median.

3.1.3.4 Other Issues with PCA

PCA often does the job well, and will be an important part of your ML toolkit.

But to be sure, like any ML method, this approach is not perfect. It is conceivable that one of the later PCs, a nearly-constant variable, somehow has a strong relation with Y.[3]

And, possibly the relation of our Y to some PC is nonmonotonic, i.e. not always increasing or always decreasing. For instance, mean income as a function of age tends to increase through, say, age 40, then decline somewhat. To deal properly with this, one might add an age^2 column to the data before applying PCA.

3.2 Cross-Validation in Larger Datasets

Recall the "leaving one out" method from the last chapter, which we said is a special case of *cross-validation*. Actually, the latter is often called *K-fold cross-validation*. (Keep in mind, K here is not k, the number of nearest neighbors.) Leaving-one-out is in that context called n-fold cross validation, for reasons we'll see shortly.

3.2.1 Problems with Leave-One-Out

In this chapter on larger datasets, n-fold cross-validation presents a problem. We would have to loop around n times, one for each subset of size 1, and each such time would have us processing about n data items. That's at least n^2 amount of work (actually much more). If say n is a million, then n^2 is a trillion, i.e. 1000 billion. Modern computers are fast, and a clock rate in the gigaherz range means billions of (small) operations per second, but you can see that some ML computations can indeed take long periods of time to complete.

In other words, the "leaving one out" method is impractical for larger data. A remedy, though, would be to "leave one out" only $q < n$ times: We would choose q of our n data points at random, and do leave-one-out with them, one at a time. If one of those points were, say, X_{22}, we would predict Y_{22} from the other $n - 1$ X-Y data points, and compare the predicted value with the real one. We would choose q large enough to get a good sample. Note that we may need to employ this strategy in general K-fold cross-validation too, as will be seen later.

Another issue to deal with is that there is mathematical theory that suggests that with certain kinds of ML methods, leave-one-out may not work

3. See Nina Zumel's blog post, *http://www.win-vector.com/blog/2016/05/pcr_part1_xonly/*, which in turn cites an old paper by Joliffe, *https://pdfs.semanticscholar.org/f219/2a76327e28de77b8d27db3432b6a20e7eb*
.

well even if we have time to do the full computation with $q = n$. This is not a problem with k-NN,[4] but there are issues in this regard with linear models,[5] which we will cover in Chapter 7.

However, if we keep the number of features p down to a reasonable value, say following the rule of thumb $p < \sqrt{n}$ as discussed in Section 1.7.4, this is not a problem.

At any rate, in the following sections, we'll explore common alternatives.

3.2.2 Holdout Sets

A simple idea is to set aside part of our data as a *holdout set*. We treat the remaining part of our data as our training set. For each of our candidate models, e.g. each candidate value of k in k-NN, we fit the model to the training data, then use the result to predict the Y values in the holdout set.

The key point is that *the holdout portion is playing the role of "new" data.* If we overfit to the training set, we probably won't do well in predicting the "fresh, new" holdout data.

3.2.2.1 Example: Programmer and Engineer Wages

Recall the dataset in Section 1.18.2, on programmer and engineer salaries in the 2000 US Census. Say we take 1000 for the size of our holdout set, out of 20090 total cases.

```
> data(peDumms)
> pe <- peDumms[,c(1,18:29,32,31)]  # just take a few features for example
 head(pe)
       age educ.12 educ.13 educ.14 educ.15 educ.16 occ.100 occ.101 occ.102
1 50.30082       0       1       0       0       0       0       0       1
2 41.10139       0       0       0       0       0       0       1       0
3 24.67374       0       0       0       0       0       0       0       1
4 50.19951       0       0       0       0       0       1       0       0
5 51.18112       0       0       0       0       0       1       0       0
6 57.70413       0       0       0       0       0       1       0       0
  occ.106 occ.140 occ.141 sex.1 wkswrkd wageinc
1       0       0       0     0      52   75000
2       0       0       0     1      20   12300
3       0       0       0     0      52   15400
4       0       0       0     1      52       0
5       0       0       0     0       1     160
6       0       0       0     1       0       0
```

4. KC Li, *Annals of Statistics, 1984, 12.*
5. J. Shao, JASA, 1993.

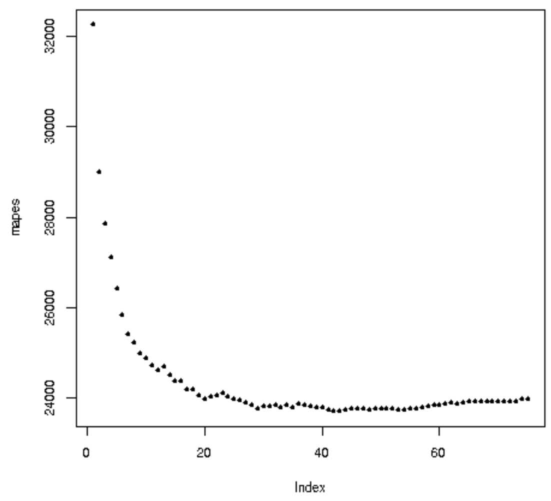

Figure 3-1: Cross-Validation, k = 1,...,75

Now let's randomly choose the training and holdout sets:[6] Then we will apply k-NN to the training set, for all values of *k* through, say, 75. Whichever value gives us the smallest MAPE will be our choice for *k* in making future predictions.

```
holdIdxs <- sample(1:nrow(pe),1000)
petrain <- pe[-holdIdxs,]
pehold <- pe[holdIdxs,]
knnout <- kNN(petrain[,-15],petrain[,15],pehold[,-15],75,allK=TRUE)
MAPEy <- function(x) MAPE(pehold[,15],x)
mapes <- apply(knnout$regests,1,MAPEy)
plot(mapes,,pch=16,cex=0.75)
```

We see the results in Figure 3-1. The optimal *k* for this dataset seems to be around 45.

3.2.2.2 Why Hold Out (Only) 1000?

We have 19090 in the training set and 1000 in the holdout set. Why 1000? Actually, the size of the training and holdout sets is yet another instance of the many tradeoffs that arise in ML.

To see this, note first that after choosing our value of *k* through cross-validation, we will re-fit the k-NN method to our *full*, 20090-cases, dataset (training + holdout), with that value of *k*, and use it to predict all future cases.

6. It is important to do this randomly, as opposed to say choosing the first 1000 rows as hold-out. Many datasets are ordered on some variable, so this would bias our analysis.

(a) What we did above was find the best *k* for the 19090-case data. Hopefully 19090 is close enough to 20090 so that we get a good estimate of MAPE for the full set.
But if we had saved, say, 7500 cases for the holdout, we would then be finding the best *k* for a 12590-case dataset, a poorer approximation to 20090.

(b) On the other hand, there is the concern that the MAPE values we got may have been based on too small a sample, only 1000. In that light, 7500 seems better.

In fact, this is the Bias-vs.-Variance Tradeoff again. If we have a large holdout set, (a) says our MAPE estimates are biased, while (b) says they have a lower variance, and vice versa.

But 1000 *is* enough for low variance. Actually, political pollsters typically use just a little more than that sample size for polls, abut 1200, because it gives them an acceptable 3% margin of error in the (statistically) least favorable case of a 50/50 tie between two candidates. So in this example, using a holdout of 1000 would be fine from both bias and variance points of view. What about in general?

3.2.3 K-Fold Cross-Validation

If we were more concerned about the variance issue in the above example, we could do *many* 19090/1000 splits, thereby basing our MAPE values on a larger sample and thus reducing the variance. If we were to do this for *all* possible such splits, 20 in this case, it would be called *20-fold cross validation.*

You can see now that Leave-One-Out is actually *n*-fold cross-validation. We can have a variance problem there if *n* is small, so again K-fold cross-validation may be a better strategy than Leave-One-Out.

A single holdout split is enough for a cursory analysis, and we will often adopt this approach in this book. But a thorough analysis should use K-fold.

3.3 Pitfall: p-Hacking in Another Form

Take another look at Figure 3-1. Like similar graphs in earlier chapters, this one is "bumpy" This stems from sampling variation, of which we have plenty here: The overall dataset is considered a sample from some population, on top of which we have the randomness arising from the random 19090/1000 split and so on. Different samples would give us somewhat different graphs.

So the bumps and dips are not real, and that means the minimizing value of *k* — the place of the lowest dip — is not fully real either. Though we are cross-validating by only one hyperparameter in this case, *k*, in some other ML applications we have several hyperparameters, even a dozen in some cases.

Consider the song data in Section 3.1.1. If we look at, say, 50 different values of *k* and 60 different values of the number of PCs *m*, that's 3000 combinations to evaluate in our cross validation. Putting aside concerns about computation time for so many combinations, a question arises concerning p-hacking:

Of those 3000 combinations, it's possible that one will be extreme by accident; some (k, m) pair may accidentally have an especially small MAPE value. Though this possibility is tempered by the fact that the various pairs have correlated MAPE values, we may indeed have a p-hacking problem, in which we choose some (k, m) pair that in actuality is not very good.

The **regtools** package has a function **fineTuning()** for doing all this — generating all the parameter combinations, running the model on each one, and most importantly, dealing with the p-hacking problem. We will use this function later.

3.4 Triple Cross Validation

Suppose one splits one's data into training and test sets, and then fits many different combinations of hyperparameters, choosing the combination that does best on the test set. Again we run into the problem of potential p-hacking, as noted above, so that the accuracy rates reported in the test set may be overly optimistic.

One common solution is to partition the data into three subsets rather than two, with the intermediate one being termed the *validation set*. One fits the various combinations of hyperparameters to the training set and evaluates them on the validation set. After choosing the best combination, one then evaluates (only) that combination on the test set, to obtain an accuracy estimate untainted by p-hacking.

3.5 Discussion: the "Curse of Dimensionality"

The Curse of Dimensionality (CoD) says that ML gets harder and harder as the number of features grow. For instance, there is mathematical theory that shows that in high dimensions, every point is approximately the same distance to every other point. The intuition underlying this bizarre situation is discussed briefly level below, but clearly it has implications for k-NN, which relies on distances, and indeed for some other ML methods as well.

Now to get an idea of CoD, consider student data, consisting of grades in mathematics, literature, history, geography and so on. The distance between the data vectors of Students A and B would be the square root of

$$\begin{aligned}(\text{math grade}_A - \text{math grade}_B)^2 &+ \quad (\text{lit grade}_A - \text{lit grade}_B)^2 \; + \\ (\text{history grade}_A - \text{history grade}_B)^2 &+ \quad (\text{geo grade}_A - \text{geo grade}_B)^2 \qquad (3.1)\end{aligned}$$

That expression is a sum, and one can show that sums with a large number of terms (only 4 here, but we could have many more) have small standard deviations relative to their means. Well, a quantity with small standard deviation is nearly constant. So in high dimensions, i.e. in settings with large p, i.e. a large number of features, distances are nearly constant.

For this reason, it is thought that k-NN fares poorly in high dimensions. However, this issue is not limited to k-NN at all.

Much of the ML toolbox has similar problems. In the computation of linear/logistic regression (Chapter 7), there is a sum of p terms, and similar computations arise in neural networks.

In fact, lots of problems arise in high dimensions. Some analyts lump them all together into the CoD. Whatever one counts as CoD, clearly higher dimensions are a challenge.

3.6 Going Further Computationally

For very large datasets, the **data.table** package is highly recommended for large data frame-types of operations. The **bigmemory** package can help with memory limitations, though it is for specialists who understand computers at the OS level. Also, for those who know SQL databases, there are several packages that interface to such data, such as **RSQLite** and **dplyr**.

4

A STEP BEYOND K-NN: DECISION TREES

In k-NN, we look at the neighborhood of the data point to be predicted. Here again we will look at neighborhoods, but in a more sophisticated way. It will be easy to implement and explain, lends itself to nice pictures, and has more available hyperparameters with which to fine-tune it.

We will first introduce decision trees (DTs), which you will see are basically flow charts. We then look at then sets of trees ("forests").

4.1 Basics of Decision Trees

Though some ideas had been proposed earlier, the decision tree (DT) approach became widely used due to the work of statisticians Leo Breiman, Jerry Friedman, Richard Olshen and Chuck Stone, *Classification and Regression Trees* (CART).[1]

A DT method basically sets up the prediction process as a flow chart, thus the name *decision tree*). At the top of the tree, we split the data into two

1. *Classification and Regression Trees*, Wadsworth, 1984.

parts, according to whether some feature is smaller or larger than a given value. Then we split each of *those* parts into two further parts, and so on. Hence an alternative name for the process, *recursive partitioning*. Any given branch of the tree will end at some leaf node. In the end, our predicted *Y* value for a given data point is the average of all the *Y* values in that node.

Various schemes have been devised to decide (a) *whether* to split a node in the tree, and (b) if so, *how* to do the split. More on this shortly.

4.2 The partykit Package

R's CRAN repository has several DT packages, but one I like especially is **partykit**.[2] To illustrate it, let's run an example from the package.

4.2.1 The Data

The dataset here, **airquality**, is built-in to R. Here is what it looks like:

```
> head(airq)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
6    28      NA 14.9   66     5   6
7    23     299  8.6   65     5   7
```

Our goal is to predict ozone levels from the other features.

4.2.2 Fitting the Data

Continuing the package's built-in example,[3]

```
> library(partykit)
> airq <- subset(airquality, !is.na(Ozone))
> airct <- ctree(Ozone ~ ., data = airq,
             control = ctree_control(maxdepth = 3))
> plot(airct)
```

Like many R packages, this one uses R *formulas*, in this case `Ozone ~ .` . The feature on the left side of the tilde symbol **~**, **Ozone**, is to be treated as the response or outcome variable, and the dot means "everything else," i.e. **Solar.R**, **Wind**, **Temp**, **Month** and **Day**. The **control** argument is used to specify hyperparameters, in this case **maxdepth**; we are saying that we want at most 3 levels in the tree.

The plot is shown in Figure 4-1.

2. A pun on the term *recursive partitioning*.
3. Note the removal of NAs. We'll treat this important topic in Chapter 13.

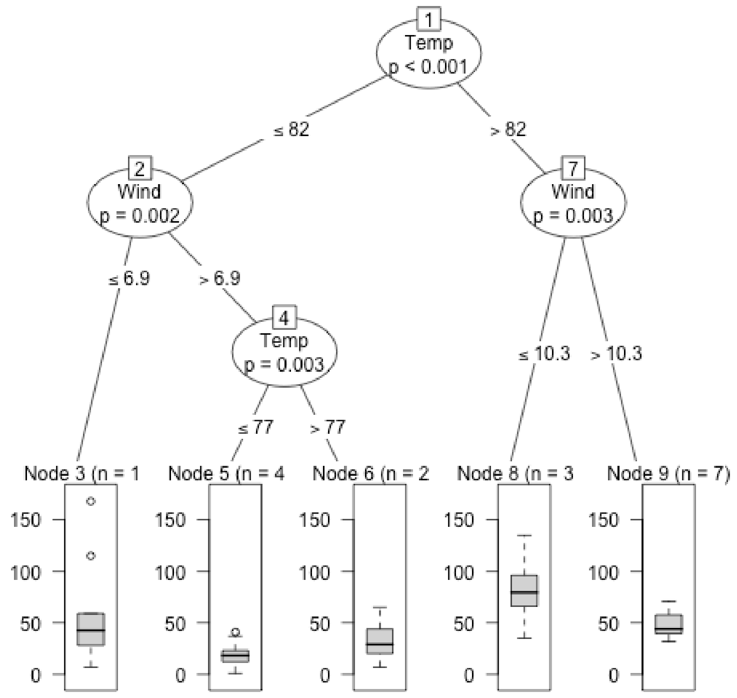

Figure 4-1: Sample plot from ***ctree***

4.2.3 Looking at the Plot

Let's look at the plot, in Figure 4-1. As you can see, a DT indeed takes the form of a flow chart. As we are just getting an overview now, don't try to grasp the entire picture in a single glance.

What the plot is saying is this: For a day with given levels of Solar.R, Wind and so on, what value should be predict for Ozone? The graph shows our prediction procedure:

1. As our first cut in predicting this new day's Ozone, look at Temp. If it is less than or equal to 82 degrees Fahrenheit, go to the left branch of the flow chart; otherwise, go right. So we've now split Node 1 of the tree.
2. If you are on the right branch, now look at Wind. If it is at most 10.3 miles per hour, your Ozone prediction is in Node 8, something like 80 parts per billion (more precise number coming shortly).
3. On the other hand, if you take the left branch from Node 1, there again will be a decision based on Temp, comparing it to 6.9. If the temperature is below 6.9, our prediction is in Node 3. If it is larger than 6.9, compare to Temp again, winding up in either Node 5 or 6.

Note that Nodes 3, 5, 6, 8 and 9 are *terminal* nodes, not split. We can get such information programmatically:

```
> nodeids(airct,terminal=TRUE)
[1] 3 5 6 8 9
```

What exactly is the predicted Ozone value in Node 8? Normally we would let the **predict()** function take care of this for us, but in order to gain insight into the package, note that it is buried rather deeply in the **ctree()** output object **airct**:

```
> nodes(airct,8)[[1]]$prediction
[1] 81.63333
```

This number should be the mean Ozone value in Node 8. Let's check it:

```
> tmp <- airq[airq$Temp > 82,]
> tmp <- tmp[tmp$Wind <= 10.3,]
> mean(tmp$Ozone)
[1] 81.63333
```

Ah, yes.

4.2.4 Printed Version of the Output

We can print the output in text form to our terminal screen:

```
> airct

Model formula:
Ozone ~ Solar.R + Wind + Temp + Month + Day

Fitted party:
[1] root
|   [2] Temp <= 82
|   |   [3] Wind <= 6.9: 55.600 (n = 10, err = 21946.4)
|   |   [4] Wind > 6.9
|   |   |   [5] Temp <= 77: 18.479 (n = 48, err = 3956.0)
|   |   |   [6] Temp > 77: 31.143 (n = 21, err = 4620.6)
|   [7] Temp > 82
|   |   [8] Wind <= 10.3: 81.633 (n = 30, err = 15119.0)
|   |   [9] Wind > 10.3: 48.714 (n = 7, err = 1183.4)

Number of inner nodes:    4
Number of terminal nodes: 5
```

Among other things, the display here shows the number of the original data points ending up in each terminal node and the mean squared prediction error for those points, as well as the mean Y value in each of those nodes. (The mean for Node 8 does jibe with what we found before.)

4.2.5 Generic Functions in R

Before we discuss the plot itself, note that here **plot()** is an example of what are called *generic* functions in R. In your past usage of R, you may have noticed that calls to **plot()** yield different results for different types of objects. For instance, **plot(x)** for a vector **x** will yield a graph of **x** against its indices 1,2,3,... while **plot(x,y)** for vectors **x** and **y** will produce a scatter plot of one vector against the other.

How can **plot()** be so smart? The way R does this is to check the type of object(s) to be plotted, then relay (*dispatch*) the **plot()** call to one specific to the class of the object(s). Here, **ctree()** returns objects of class **'party'**, so the call

```
plot(airct)
```

will be dispatched to a call to another function, **plot.party()**, which then produces the graph.

Similarly, when we typed

```
> airct
```

to get the printed output above, here is what happened: In interactive mode in R (i.e. typing in response to the '.' prompt), any expression that we type will be fed into R's **print()** function. The latter is also generic, so that call will in turn be dispatched to the class-specific version, in this case **print.party()**.

Note that both **plot.party** and **print.party** were written by the authors of the **party** package, not by the authors of R. The **party** authors had to think about how they wanted to plotted and printed descriptions of **airct** to look like, in order to be most useful to the users of the package.

4.2.6 Summary So Far

So, here are DTs in a nutshell:

1. Start with some feature X, which will form Node 1.
2. Decide on some threshold value v for X, according to some criterion.
3. Form the left and right branches emanating from Node 1, according to whether our data point's value is under or over v.
4. Keep growing the tree until some stopping criterion is met.

5. When we later predict a new data point, we use its *X* values to traverse the tree to the proper terminal node. The mean there is our predicted value for the new data point.

We'll discuss possible splitting and stopping criteria shortly.

By the way, if *X* is a categorical variable, the split question in a node involves equality, e.g. "Day of the week = Saturday?"

4.2.7 Other Helper Functions

The **partykit** package has lots of helper functions. Let's look at **predict_party()**. The call form we'll use here is

```
predict_party(obj,id,FUN=your_desired_function)
```

Here **obj** is the return value of **ctree**, **id** is a vector of terminal nodes, and **FUN** is a user-supplied function we'll describe below.. Then **predict_party()** returns an R list, one element for each node specified in **id**.

Specifying **FUN** is a little tricky. Again, we the user write this function. But it will be called by **predict_party()**. And the latter, in calling our function, will expect that our function will have two arguments, the first of which will be the *Y* values in the given node. Remember, we don't know them, but **predict_party()** has them, and will feed them in as the first argument when it calls our **FUN**. The second argument is the weights corresponding to the *Y* values, which does not concern us in our context here.

In fact, we could query **partykit** about all the *Y* values in that node:

```
> f1 <- function(yvals,weights) c(yvals)
> predict_party(airct,id=3,FUN=f1)
$`3`
 [1]   7 115  48  16  39  59 168  28  46  30
```

What happened here? Recall that R's **c** function does concatenation. Here we concatenated all the *Y* values in that node. They turn out to be 7, 115 and so on.

Or, say, we can find the median at Node 3:

```
> mdn <- function(yvals,weights) median(yvals)
> predict_party(airct,id=3,FUN=mdn)
   3
42.5
```

4.2.8 Diagnostics: Effects of Outliers

Suppose we are concerned about outliers. We might try predicting using the median instead of the mean in the terminal nodes.

In this regard it might be useful, if the tree isn't too large, to print out all the terminal nodes' means and medians. A large discrepancy should be investigated for outliers. Let's do that:

```
> nodeIDs <- nodeids(airct,terminal=TRUE)
> predict_party(airct,id=nodeIDs,FUN=mdn)
   3    5    6    8    9
42.5 18.0 29.0 79.5 44.0
> predict_party(airct,id=nodeIDs,FUN=function(yvals,weights) mean(yvals))
       3        5        6        8        9
55.60000 18.47917 31.14286 81.63333 48.71429
```

That's quite a difference in Node 3. As we saw above, there were two values, 115 and 168, that were much higher than the others. They are probably not errors (we'd consult a domain expert if it were crucial), but it does suggest that indeed we should predict using medians rather than means, e.g.:

```
> newx <- data.frame(Solar.R=172,Wind=22.8,Temp=78,Month=10,Day=12)
> predict(airct,newx,FUN=mdn)
 1
29
```

4.3 Example: New York City Taxi Data

Let's try all this on a larger dataset. Fortunately for us data analysts, the New York City Taxi and Limousine Commmission makes available voluminous data on taxi trips in the city. For the example here, I randomly chose 100,000 records from the January 2019 dataset.

It would be nice if taxi operators were to have an app to predict travel time, as many passengers may wish to know. Let's see how feasible that is.

4.3.1 Data Preparation

First, let's take a look around:

```
> load('~/Research/DataSets/TaxiTripData/YellJan19_100K.RData')
> yell <- yellJan19_100k
> names(yell)
 [1] "VendorID"
 [2] "tpep_pickup_datetime"
 [3] "tpep_dropoff_datetime"
 [4] "passenger_count"
 [5] "trip_distance"
 [6] "RatecodeID"
 [7] "store_and_fwd_flag"
 [8] "PULocationID"
 [9] "DOLocationID"
[10] "payment_type"
```

```
[11] "fare_amount"
[12] "extra"
[13] "mta_tax"
[14] "tip_amount"
[15] "tolls_amount"
[16] "improvement_surcharge"
[17] "total_amount"
[18] "congestion_surcharge"
```

That's quite a few variables. Let's try culling out a few. Since types matter so much, let's check the classes of each one.

```
# use pickup and dropoff dates/times and locations
yell <- yell[,c(2,3,4,5,8,9)]
> for(i in 1:ncol(yell)) print(class(yell[,i]))
[1] "factor"
[1] "factor"
[1] "integer"
[1] "numeric"
[1] "integer"
[1] "integer"
```

Wait, there is a problem here. The last two columns, which are location IDs, are listed as R integers; they need to be changed to R factors. On the other hand, the first two columns are dates/times, which we'll need as characters.

```
> yell[,1] <- as.character(yell[,1])
> yell[,2] <- as.character(yell[,2])
> yell[,5] <- as.factor(yell[,5])
> yell[,6] <- as.factor(yell[,6])
```

Traffic depends on day of the week, so let's compute that, and also find the elapsed trip time. There are lots of R functions and packages for dates and times. Here we'll use **data.table** package:

```
> library(data.table)
> PUweekday <- wday(as.IDate(yell[,1]))
> DOweekday <- wday(as.IDate(yell[,2]))
> PUtime <- as.ITime(yell[,1])
> DOtime <- as.ITime(yell[,2])
```

Here **IDate** and **ITime** are standard date and time functions used by **data.table**. The **wday** ("weekday") function returns the value 1 for a weekday and 0 for a weekend. Of course, we could do a more detailed analysis that separate out each of the weekdays. Fridays, for instance, may be different from the other days.

```
> findTripTime <- function(put,dut,pwk,dwk)
+ {
+    pt <- as.ITime(put)  # pickup time
+    dt <- as.ITime(dut)  # dropoff time
+    tripTimeSecs <- as.integer(dt - pt)
+    ifelse(dwk != pwk,tripTimeSecs+24*60*60,tripTimeSecs)
+ }
>
> tripTime <- findTripTime(PUtime,DOtime,PUweekday,DOweekday)
```

Note that in computing elapsed time, we had to account for the fact that a trip may begin on a certain date but end the next day. In such cases, we had to add 24 hours to the dropoff time.

Finally, add the weekday and trip time information to the data frame, and delete the original date/time columns:

```
> yell$PUweekday <- PUweekday
> yell$DOweekday <- DOweekday
> yell$tripTime <- tripTime
> yell[,1:2] <- NULL
```

Now, what about "dirty data"? At a minimum, we should look for outliers in each of the numeric columns of our data frame **yell**. We might call, say, **table()** on the number of passengers and **hist()** on the trip distance. We won't show the results here, but the one for trip time is certainly interesting:

```
> hist(yell$tripTime)
```

The plot is shown in Figure 4-2. This is trip time in seconds, and, for instance, 20,000 seconds is more than 6 hours! It may be an error, but even if real, probably not representative. Let's remove any trip of more than, say, 2 hours:

```
> yell <- yell[yell$tripTime <= 7200,]
```

After calling **hist()** again, Figure 4-3 looks much more reasonable.

We should also check that our winnowing there still left us with a decent number of cases:

```
> dim(yell)
[1] 99723     7
```

Fewer the 300 were deleted.

4.3.2 Tree-Based Analysis

OK, ready to go. Let's set up for cross-validation, say with 5000 in our holdout set.

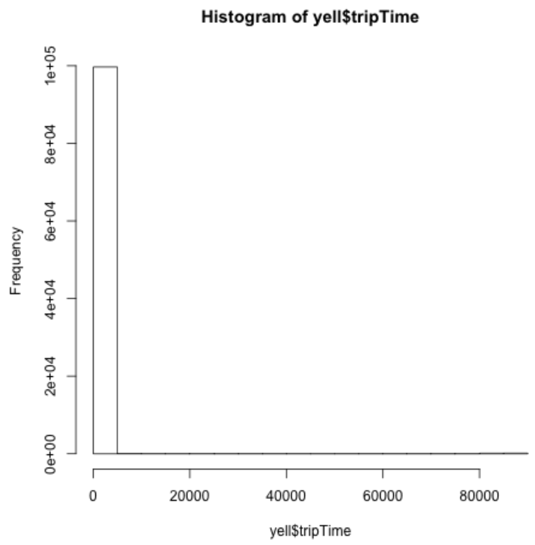

Figure 4-2: Taxi data, trip time

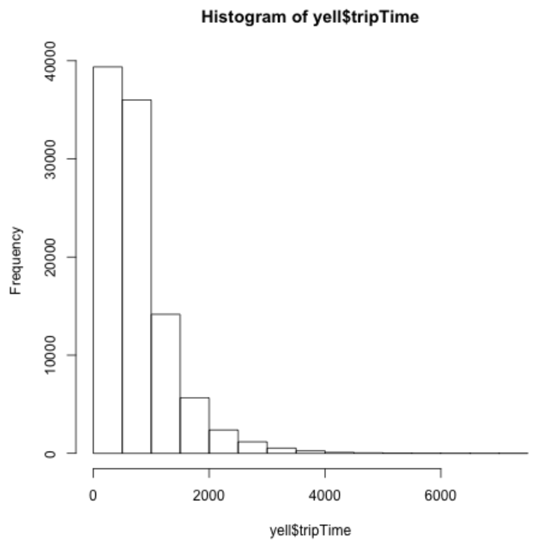

Figure 4-3: Taxi data, modified, trip time

First, though, note that the run shown below was done using the **party** package, a predecessor of **partykit**. (All the other examples in this chapter use **partykit**.) The documentation for the latter recommends using the former if the latter produces a message "Too many levels," which the **partykit** version is not designed to handle. This will often be the case when working with categorical variables with a large number of categories, e.g. here a large number of different pickup and dropoff locations. We'll discuss this further in Section 4.3.3.

```
> tstidxs <- sample(1:nrow(yell),5000)
> yelltst <- yell[tstidxs,]
> yelltrn <- yell[-tstidxs,]
> library(party)
> ctout <- ctree(tripTime ~ .,data=yelltrn)
> preds <- predict(ctout,yelltst)
> mean(abs(preds-yelltst$tripTime))
[1] 202.709
```

So with this analysis — note that we took the default values of all the optional arguments to **ctree()** — taxi drivers would be able to predict trip time for a passenger with an average error of a bit over 3 minutes. There are lots of hyperparameters available, though, so we may be able to do better than this with some tweaking, something to possibly pursue later.

By the way, some readers may be puzzled by the above line

```
> preds <- predict(ctout,yelltst)
```

After all, *yell$tripTime* is in **yelltst**, so aren't we using trip time to predict itself? Actually, no. The **predict()** function here determines from **ctout** what the feature names are, and looks for them in the second argument to **predcit()**, **yelltst**. Any column in the latter not in the former will be ignored.

The resulting tree is too large and complex to call **plot()**, but let's get a sample of it, say node 554:

```
> nodes(ctout,554)
[[1]]
554) trip_distance <= 20.9; criterion = 1, statistic = 231.228
  555) PULocationID == {10, 13, 14, 21, 24, 37, 41, 42, 49, 64, 68,
       76, 86, 87, 88, 107, 116, 136, 142, 144, 147, 151, 158, 161,
       165, 195, 205, 209, 210, 211, 226, 230, 231, 234, 235, 239,
       244, 249, 254, 261}; criterion = 1, statistic = 215.768
    556)*  weights = 260
  555) PULocationID == {22, 26, 28, 33, 38, 40, 43, 45, 48, 50, 51,
       52, 53, 61, 62, 65, 74, 75, 79, 80, 89, 90, 97, 100, 113,
       114, 121, 125, 129, 130, 131, 132, 134, 137, 138, 140, 141,
       143, 145, 146, 148, 152, 162, 163, 164, 166, 167, 170, 182,
       186, 191, 194, 198, 206, 215, 216, 218, 219, 220, 223, 224,
       229, 232, 233, 236, 237, 238, 242, 246, 257, 258, 262, 263, 264}
    557) PUweekday <= 1; criterion = 1, statistic = 150.914
```

```
      558) trip_distance <= 17.21; criterion = 0.999, statistic = 80.654
        559)*  weights = 62
      558) trip_distance > 17.21
        560)*  weights = 131
    557) PUweekday > 1
...
```

So at that particular node, we go left or right, depending on whether the trip distance is below 20.9 or not. If we go left for trip distance, we now go left again if the pickup location has ID 10, 13, 14 and so on. In that case, we reach node 556, which is a terminal node, as indicated by the asterisk.

4.3.3 Use of Embeddings

Recall the notion of an *embedding* mentioned briefly in Section 1.7.4.2. The idea (in simple form) is to replacing a categorical feature with a large number of categories by one or more continuous features that largely summarize the categorical one.

The example given was ZIP code. There are over 40,000 categories (postal localities), thus a risk of overfitting. But one might instead replace ZIP by one or more variables that capture the essence of each locality, e.g. income and age distribution.

With the NYC taxi data above, the pickup and dropoff locations are highly categorical, e.g.

```
> unique(yell$PULocationID)
  [1] 226  50 148 229 262 263  79 132 161 143  90
 [12]  48 163 162 231 186 142 170 249 238  68  75
 [23] 137 138 100 236 237 224 140 166 141 114 244
 [34]  13 164 144 230 239 209 211 152  87 125 234
 [45]  45  12 113 130 107 264  43 246  42   4  41
 [56] 158 151 233   1  74  26  82 261  88 215  10
 [67] 102 247 106  66 179 181 146  24 168 225 197
 [78]   7 112 255  33 145 188  25  18 241  97 193
 [89]  37 194 133 116  52  76  61 232  36 223  65
[100] 256  83 202  40 227  80 265  60 260  49 139
[111]  91  93  85 189  21 235  17  95 119 257  70
[122]  28 121  38  71 207 129 216  62 182   8  89
[133] 219  11 212  72 167 205  55  67 254  35 218
[144] 228 203 149 222 155  53 196 220 127  47  69
[155]  81 123 159 190 169  32 213  86 165  51 134
[166] 174  14 177 258 243  63  16 191   3 250 160
[177] 126 157  56 150 124 122 183 131 210 242 185
[188]  77 180  22 217  59 259  39 135  34 252  20
[199] 198 248 101 206  92 195 136  29  98  15 175
[210]  19   9 173  94 147 153 201  23 111 154  64
[221]  73 208  78 200
```

That's 224 different individual locations, and there are 249 dropoff points. Not as bad as the ZIP code case, but still that's a lot of dummy variables. (Most R ML packages convert factors to dummy variables internally.)

A good embedding here might be the latitude and longitude values of the pickup and dropoff points. At first you might think, "How would that be an improvement? We will still have 224 distinct (latitude,longitude) pairs." But the key difference is that latitude and longitude are *ordinal* variables — it makes sense to ask whether one is less than another — tailor-made for trees. A node might exploit that ordinality, by having the split criterion be, say, pickupLatitude < 41.2, whereas with the categorical form, we can only have a criterion like pickupLocation = 10 or 13 or 28...

It thus may be worth the effort to look up, in some external source, the latitudes and longitudes of those pickup and dropoff locations, and implementating the embedding.

4.4 Example: Forest Cover Data

Another UCI dataset, Covertype, aims to "[predict] forest cover type from cartographic variables only."[4] The idea is that one might determine what kinds of grasses there are in remote, difficult-to-access regions. There are 581012 data points, with 54 features, such as elevation, hill shade at noon and distance to the nearest surface water. There are 7 different cover types, stored in column 55.

This example is useful for a number of reasons. Here we'll see DT in action in a classification problem, with multiple classes, and of a size larger than what we've seen so far. And besides, what could be better in a chapter on trees and random forests than data on forests!

Let's give it a try.

4.4.1 Pitfall: Features Intended as Factors May Not Be Read as Such

When reading data from a file, what is intended as an R factor may be read as numeric or character, depending on what R function you use for the reading. In this case, I used base-R:

```
> cvr <- read.csv('covtype.data',header=FALSE)
```

and found that column 55 came out as **'integer'** class. We need to let **ctree** know that it is categorical. Easily fixed, of course:

```
> cvr$V55 <- as.factor(cvr$V55)
```

4. https://archive.ics.uci.edu/ml/datasets/covertype

4.4.2 Running the Analysis

With the larger sample size n and number of features p here, a really large tree might be generated. This might not only risk overfitting — the larger the tree, the larger the variance even though bias likely goes down — but also there may be some serious computation time problems. So let's make use of one of the hyperparameters available to us, setting the maximum depth of the tree to, say, 6.

Since the number of nodes doubles at each level, that will mean at most $2^6 = 64$ nodes, not large in relation to n at all, nor is $p = 54$ large in that regard either. Thus our accuracy measures on the full dataset will be pretty close to what we obtain by cross-validation To keep things simple, then, let's dispense with cross-validation.

```
> ctout <- ctree(V55 ~ .,data=cvr,control=ctree_control(maxdepth=6))
> preds <- predict(ctout,cvr)
> mean(preds != cvr$V55)
[1] 0.3054515
```

Note that since this is a classification problem, we've taken our accuracy measure to be proportion of wrongly predicted classes.

So, we would guess wrong about 30% of the time. Let's look a bit more closely:

```
> tbl <- table(cvr$V55)
> tbl
     1      2      3      4      5      6      7
211840 283301  35754   2747   9493  17367  20510
> tbl/sum(tbl)
          1           2           3           4
0.364605206 0.487599223 0.061537455 0.004727957
          5           6           7
0.016338733 0.029890949 0.035300476
```

Recalling Section 2.4, we see above that if we simply guess every location to be of cover type 2, we'd be right about 49% of the time. Thus the 70% we got here is promising, and we might do even better with other settings of the hyperparameters.

4.4.3 Diagnostic: the Confusion Matrix

Let's take a closer look:

```
> confusion(cvr$V55,preds)
      pred
actual      1      2      3      4      5      6
     1 150581  57689     73      0     17      0
     2  68834 209528   3073      0    317   1332
     3      0   4397  30124     88      0   1145
     4      0      3   1963    781      0      0
     5      0   8645    319      0    529      0
     6      0   4675  10736     65      0   1891
     7  10290    113      0      0      0      0
      pred
actual      7
     1   3480
     2    217
     3      0
     4      0
     5      0
     6      0
     7  10107
```

Here we've printed out the confusion matrix (Section 2.5). We see that cover types 1 and 2 are often misclassified as each other. Type 6 is often mistaken as class 3. Actually, only 1891 of the 17367 instances of class 6 were correctly identified, and so on.

It would appear, then, that there is room for improvement here. We might be able to achieve this with better values of the hyperparameters, or for that matter, methods called *bagging* and *boosting* that we'll bring in later in this chapter.

4.5 How Is the Splitting Done?

DT packages differ from one another in terms of the details of their node-splitting actions.

The splitting process in **partykit** works by performing nominal significance tests, first to decide whether to split a node and then what split point to use if a split is decided.[5] The *X* feature, if any, most "significantly" related to *Y* for the data within a node will be used for the basis of splitting this node. (The details of the significance tests, here and below, are quite complex, beyond the scope of this book.) If the relation with *Y* is not "significant," we do not split the node. If significant, though, we will split this node, using that particular feature

Once it has been decided to split a node, all possible split points are considered, and the most "significant" one is chosen.

4.6 Hyperparameters and the Bias-Variance Tradeoff

Though one package may differ from another in details, DT software packages generally feature a number of hyperparameters. These control things such as the splitting critera.

5. Many statisticians these days, including me, frown on the use of significance testing. But these are not formal tests, just mechanisms to decide on which feature to split.

4.6.1 Hyperparameters in the partykit Package

The full call form of **ctree** is

```
ctree(formula, data = list(), subset = NULL, weights = NULL,
    control = ctree_control(), xtrafo = ptrafo, ytrafo = ptrafo,
    scores = NULL)
```

We won't go into all the parameters, but the ones of interest here are specified through **control**. That is set in turn by the function

```
ctree_control(teststat = c("quad", "max"), testtype = c("Bonferroni",
    "MonteCarlo", "Univariate", "Teststatistic"), mincriterion = 0.95,
    minsplit = 20, minbucket = 7, stump = FALSE, nresample = 9999,
    maxsurrogate = 0, mtry = 0, savesplitstats = TRUE, maxdepth = 0,
    remove_weights = FALSE)
```

Again, a daunting array of choices, some of which should be considered advanced in nature. Let's look at a few:

- **maxdepth:** The maximum number of levels we are willing to allow our tree to grow to.
- **minsplit:** The minimum number of data points in a node.[6]
- **alpha:** The significance level for the tests. E.g. in deciding whether to split a tree node, a split will be performed if at least one feature has p-value less than **alpha**.
- **testype='Bonferroni':** Again, somewhat advanced, but its goal is to help avoid p-hacking. Roughly, the given **alpha** value is divided by the number of tests performed, thus making the splitting decisions more stringent.

4.6.2 Bias-Variance Tradeoff

Consider each of the above hyperparameters in terms of the bias-variance tradeoff:

- Larger trees allow us to make a more detailed analysis, hence smaller bias.
- Remember, once we have the tree built, our prediction will be the average *Y* value in that node. If the node has very few data points, this average will be based on a small sample, thus high variance. Since larger trees will have smaller numbers of data points per node, we see that larger trees are bad from a variance point of view.

What does this say for the hyperparameters listed above?

6. The package also allows giving different data point different weights, so this hyperparameter actually in the minimum sum of weights, but we will not pursue that here.

- Larger values of **maxdepth** will give us larger trees, thus smaller bias but larger variance.
- Larger values of **minsplit** mean more data points per node, thus larger bias and smaller variance.
- Larger values of **alpha** mean that more tests come out "significant," thus larger trees, hence smaller bias but larger variance.
- Setting **testype='Bonferroni'** has a similar effect to making the actual **alpha** smaller than the value set by the user. This means it's harder to get significance, thus smaller trees, larger bias and smaller variance.

These hyperparameters don't work independently of each other, so setting too many of them probably becomes redundant.

4.6.3 Example: Wisconsin Breast Cancer Data

This dataset, from the CRAN package **TH.data** (originally from the UCI repository), consist of various measures derived from samples of human tissue. Here we will predict the **status** variable, which is 'R' for cases in which the cancer has recurred and 'N' if not.

How big is this dataset?

```
> library(TH.data)
> dim(wpbc)
[1] 198  34
```

So we have $p = 33$ predictors for $n = 198$ cases. We are really at risk for overfitting, and in the DT context that would come from having an overly large tree.

Thus our choice of hyperparameter values may be especially important. We'll first introduce a tool to aid in that choice.

4.7 regtools::fineTuning() — Grid Search Operation, Plus More

Say we decide to use the first three hyperparameters introduced above. We will want to try many different values of those hyperparameters, in combinations. Say, for instance, we wish to try 3 different values of **maxdepth**, 4 for **minsplit** and 3 for **alpha**. That's 36 different combinations. With more hyperparameters, we'd have even more combinations, easily hundreds and maybe far more, necessitating a better way than writing *ad hoc* code for each different dataset. This leads us to our next topic.

It would be nice to have a systematic way to try all those combinations, assess each via cross-validation, and tally the results. This operation is called *grid search*. Many ML packages include a grid search op, but given our earlier discussions of the dangers of p-hacking in grid settings, we want something more – smoothing of the output, seen next.

4.7.1 The Function

The function **fineTuning()** in the **regtools** packages does grid search, *plus more*:[7]

- It will also smooth those results, to help avoid p-hacking, as discussed in Section 3.3.
- It will also display the results graphically, using a method known as *parallel coordinates*.

To see how it works, let's try it on the Wisconsin Breast Cancer Data, using **ctree()**:

```
fineTuning(dataset=wpbc,
   pars=list(minsp=c(5,10,15,20,30),maxd=c(2,3,4,5),
   alpha=c(0.01,0.05,0.10,0.20,0.30)),
   regCall=theCall,nCombs=NULL,nTst=50,nXval=5,k=3)
```

The first few arguments are clear. E.g. "minsp" will denote **minsplit**, and we want the function to try values 5, 10, 20 and 50 for it. Setting **nCombs** to NULL means we want all the combinations.

What about **regCall**? In order to run **fineTuning()**, we need to tell the function how to fit our model on the training set, and how to predict on the test set. We do this by writing a function and providing it to **fineTuning()** thought the argument **regCall**. The **fineTuning()** function will call whatever function we give it, which is for our breast cancer example the following:

```
theCall <- function(dtrn,dtst,cmbi) {
   ctout <- ctree(status ~ .,dtrn,
      control=ctree_control(minsp=cmbi$minsp,
         maxd=cmbi$maxd,alpha=cmbi$alpha))
   preds <- predict(ctout,dtst)
   mean(preds == dtst$status)
}
```

To see why the code is written this way, first note that **fineTuning()** will split our data into training and test sets for us, and pass them as the first two arguments when it calls our function, in this case **theCall()**. We use the first of them in our call to **ctree()**, and the second in our call to **predict()**. But **fineTuning()** must also pass to our function the particular hyperparameter combination it is running right now, which it does through the third argument, **cmbi**. Our code above then follows the pattern seen earlier in this chapter.

The next argument in our call to **fineTuning()**, **nCombs**, specifies the number of hyperparameter combinations we want to try. Here we are asking for all 36, but there could be hundreds or more, and possibly with a long-running **ctree()** call if the data were large. So, we can set **nCombs** to a num-

7. The name is a pun on the fact that hyperparameters are also called *tuning parameters*.

ber less than the total number of possible combinations, and **fineTuning()** will chose **nCombs** of them at random.

The argument **nTst** specifies the size of our test set, while **nXval** says how many folds of a K-fold cross-validation we want. Since we have small *n* here, we might want more than 1. Finally, the value of *k* is the number of k-NN neighbors to use; remember, these are "neighbors" among the various combinations.

4.7.2 Example: Wisconsin Breast Cancer Data

So, here we go:

```
> library(TH.data)
> library(partykit)
> ftout <- fineTuning(dataset=wpbc,
   pars=list(
      minsp=c(5,10,15,20,30),
      maxd=c(2,3,4,5),
      alpha=c(0.01,0.05,0.10,0.20,0.30)),
   regCall=theCall,nCombs=NULL,nTst=50,nXval=5,k=3)
> ftout$outdf
    minsp maxd alpha meanAcc  smoothed
1       5    3  0.20   0.804 0.7306667
2       5    5  0.05   0.756 0.7386667
3       5    2  0.20   0.752 0.7400000
4      20    3  0.10   0.768 0.7413333
5      20    2  0.30   0.768 0.7413333
6      10    2  0.20   0.704 0.7426667
7      30    5  0.01   0.764 0.7426667
8      30    5  0.10   0.764 0.7426667
9      15    4  0.20   0.776 0.7426667
10     30    5  0.05   0.804 0.7426667
...
85     10    2  0.10   0.768 0.7760000
86     30    5  0.30   0.772 0.7760000
87     30    4  0.30   0.784 0.7786667
88     30    5  0.20   0.816 0.7786667
89      5    2  0.30   0.796 0.7786667
90      5    3  0.10   0.800 0.7786667
91     15    5  0.20   0.744 0.7800000
92     15    2  0.05   0.776 0.7800000
93     10    3  0.05   0.808 0.7813333
94      5    3  0.05   0.776 0.7853333
95     20    3  0.20   0.748 0.7866667
96      5    2  0.10   0.764 0.7866667
97      5    3  0.01   0.740 0.7893333
98     10    5  0.05   0.788 0.7906667
99     10    5  0.10   0.764 0.7946667
100     5    2  0.05   0.764 0.8000000
```

These are proportions of correct predictions, so larger is better.

4.7.3 Smoothing, and Mitigating the p-Hacking Problems

Note that the unsmoothed numbers above do seem to be overly extreme, in this case optimistically. The very first combination (5,3,0.20), for instance, claims an 80% accuracy yet the smoothed one is only 73%.

Again, the figures are al subject to sampling variation, but that is exactly the point. The smoothing helps compensate for our small sample size here.

The smoothing also is a direct — though certainly not full — safeguard against p-hacking. When looking at a large number of hyperparameter combinations, the chances are high that at least one or two quite ordinary combinations will appear extraordinary by accident. This was likely the case with the 0.816 value above. And though the discrepancy was not large in this case, with a large number of features and a large number of hyperparameters, the discrepancy can be large.

4.7.4 Plotting fineTuning() Results

There is also an associated generic **plot()** function, using the *parallel coordinates* method for displaying multidimensional data. Let's explain that concept first.

We can display univariate data as a histogram, and show bivariate data as a scatter plot. But graphing data consisting of three or more variables is a challenge. A parallel coordinates plot meets that challenge, for potentially any number of variables, by "connecting the dots," as follows.

4.7.4.1 Example: Baseball Player Data

Let's plot a few players from the **mlb** baseball data:

```
> data(mlb)
> w <- mlb[20:24,c(4,6,5)]
> w
   Height   Age Weight
21     79 25.76    230
22     76 36.33    205
23     74 31.17    230
24     76 32.31    195
25     72 31.03    180
> lattice::parallelplot(w)
> library(MASS)  # included with R
> parcoord(w,col=1:4)
```

The result is shown in Figure 4-4. For instance, player 21 was the tallest, youngest and heaviest. He is represented in the graph by the black line.

4.7.4.2 Parallel Coordinates for fineTuning() Output

The parallel coordinates approach does take a bit of getting used to, but is quite useful for visualizing multidimensional data.[8] In our context here, it enables us to see at a glance the central patterns in the effects of the hyperparameters on our predictive ability.

8. Alfred Inselberg, *Parallel Coordinates: Visual Multidimensional Geometry and Its Applications*, Springer, 2009.

Now let's plot the output of running **fineTuning()** on the breast cancer data above:

```
> library(cdparcoord)
> plot(ftout)
```

The result is depicted in Figure 4-5.

The **cdparcoord** package uses Plotly, which plots on one's Web browser. It's difficult to fit this kind of plot on a printed page and the labels are small, so the reader is encouraged to run the plot herself.

The plot allows us to tell at a glance which grid parameters have which effects. You can see, for example, that for the most part 5 is too large a value for **maxd**; most of the line emanating from the 5 level for that parameter ultimately lead to lower values of the output.

One advantage of using Plotly is that one can use the mouse to permute the order of the columns. For instance, we can click-and-drag the 'minsp' label to the right, to make it the rightmost column, seen in Figure 4-7. This enables to see more clearly the effect of that particular parameter, though we must keep in mind that the effect still depends on the other parameters.

Parallel coordinates plots tend to be rather "full," but **fineTuning()** allows one to "zoom in," say plotting only the lines corresponding to the 10 highest values of **smoothed**, the predictive accuracy:[9]

```
> plot(ftout,disp=10)
```

That result is shown in Figure 4-6.

4.7.4.3 A Note on the Smoothing

Recall the motivation for smoothing the output of a grid search, as described back in Chapter 3:

> Take another look at Figure 3-1. Like similar graphs in earlier chapters, this one is "bumpy" This stems from sampling variation, of which we have plenty here: The overall dataset is considered a sample from some population, on top of which we have the randomness arising from the random 19090/1000 split and so on. Different samples would give us somewhat different graphs. So the bumps and dips are not real, and that means the minimizing value of *k* — the place of the lowest dip — is not fully real either.

As noted, the idea of smoothing is to reduce the variance of the grid values, so those "bumps and dips" are smaller. But we are also inducing a bias, by replacing the value in one row of the output by "nearby" rows. Again, the old bias/variance tradeoff, in a new context.

The implication is that **if we set nXval to a large value, we do not have to smooth**. If we run enough folds of the cross-validation, the variance of the output is small enough that we do not have to resort to smoothing.

9. Use a negative value for **disp** to display the lowest values.

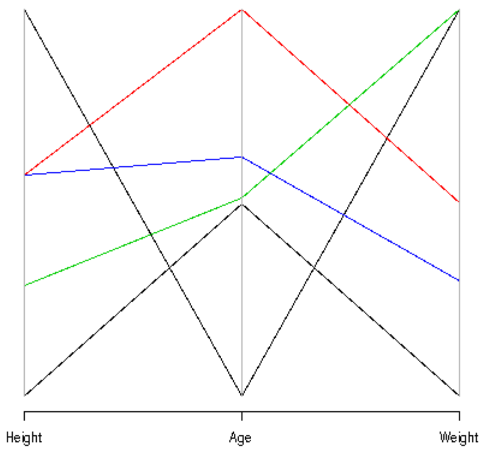

Figure 4-4: Par. coords. plot, mlb data

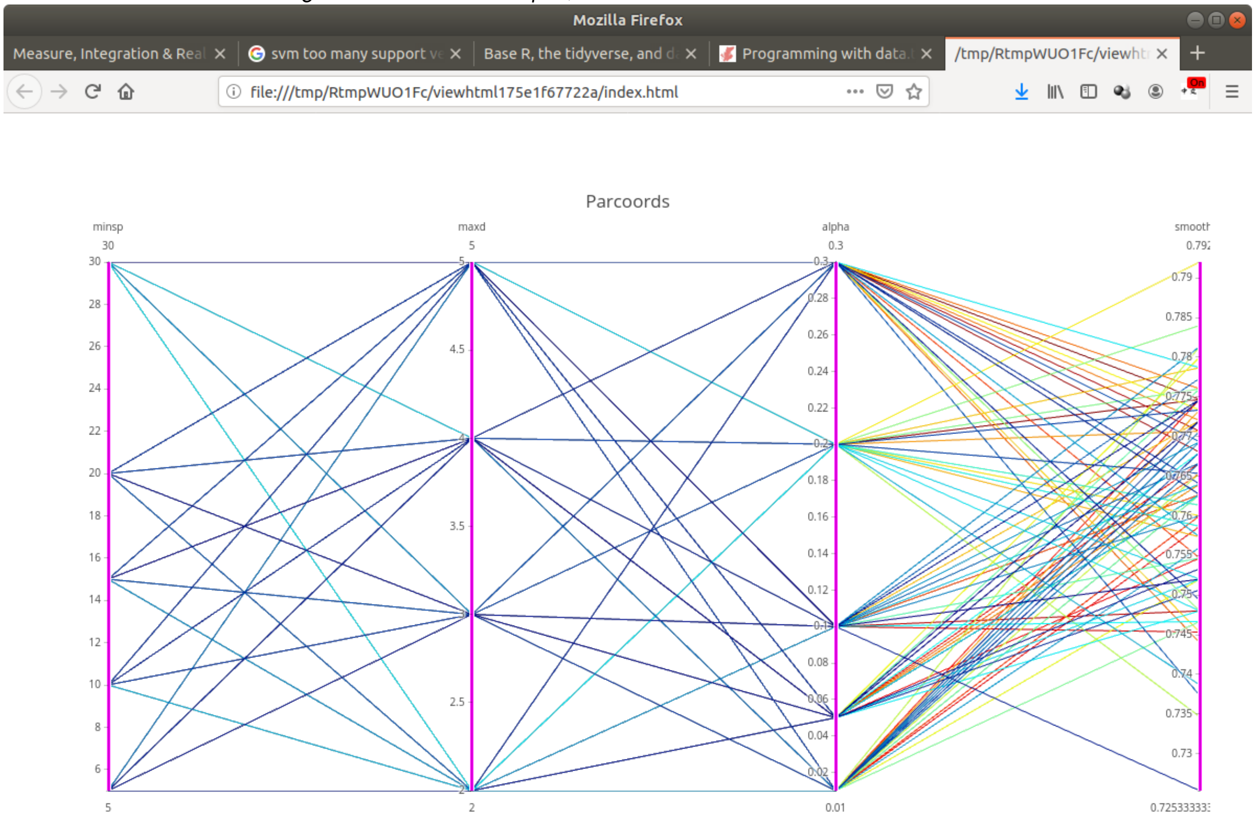

Figure 4-5: Plot from fineTuning()

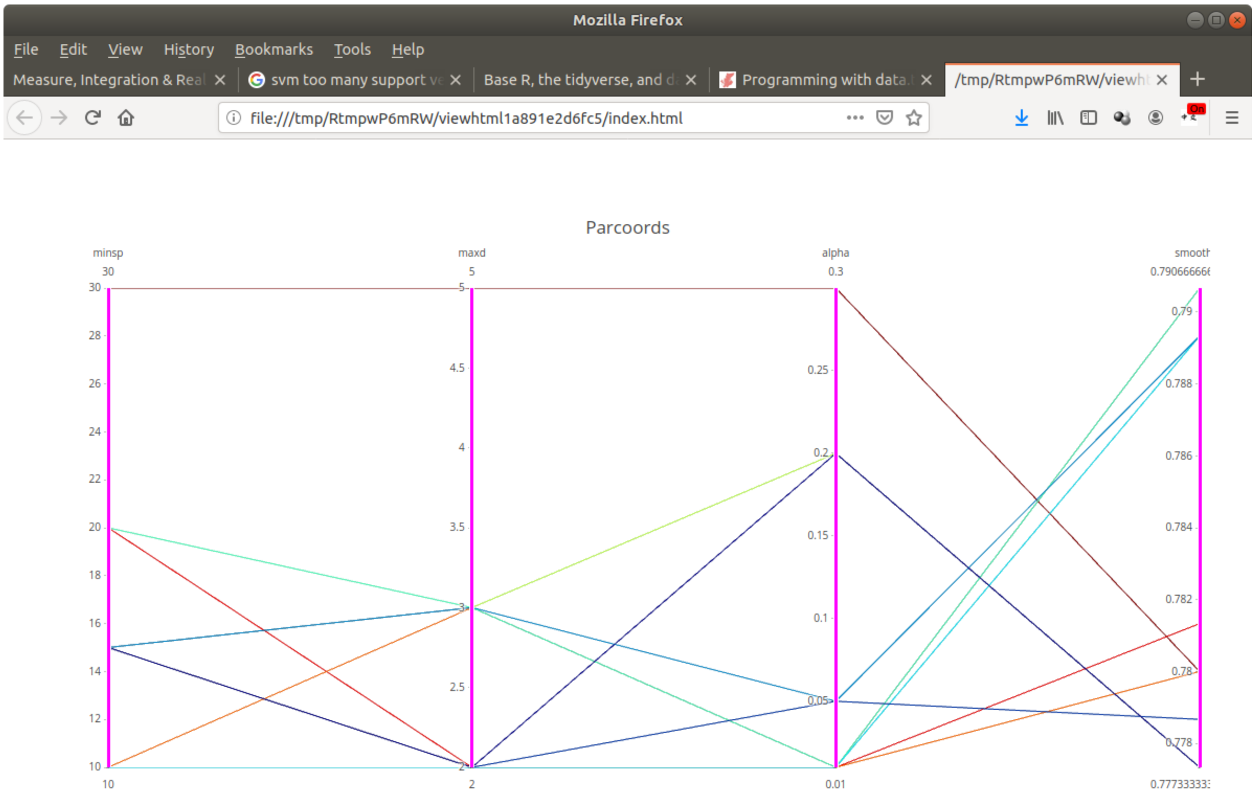
Mozilla Firefox
File Edit View History Bookmarks Tools Help
Measure, Integration & Real
svm too many support ve
Base R, the tidyverse, and d
Programming with data.
/tmp/RtmpwP6mRW/viewht
file:///tmp/RtmpwP6mRW/viewhtml1a891e2d6fc5/index.html
Parcoords
minsp
maxd
alpha
smooth
30
5
0.3
0.79066666
10
2
0.01
0.77733333

Figure 4-6: Plot from fineTuning(), zoomed

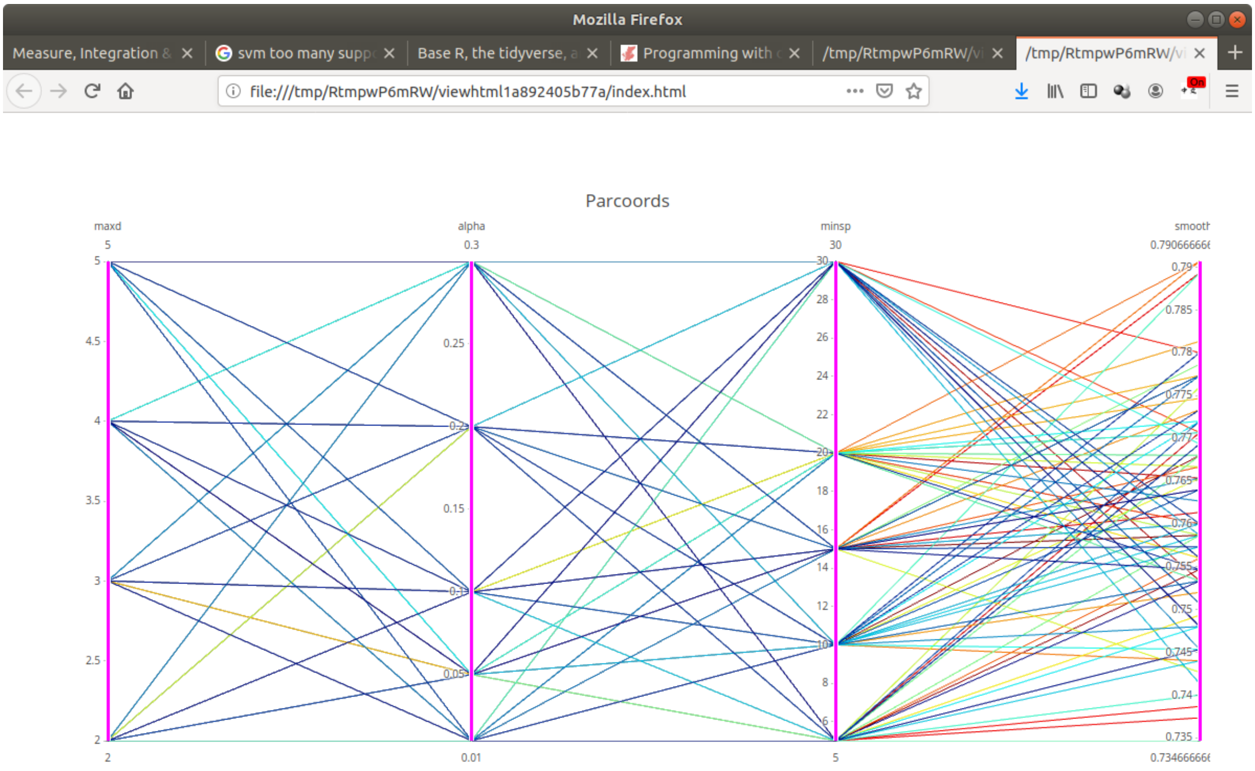

Figure 4-7: Plot from fineTuning(), zoomed and permuted

5

TWEAKING THE TREES: BAGGING, RANDOM FORESTS AND BOOSTING

Here we talk about two extensions of decision tree analysis. They are both widely used, in fact even more than individual tree methods.

5.1 Bias vs. Variance, Bagging and Boosting

For want of a nail, a horse was lost
For want of a horse, the battle was lost
For want of the battle, the war was lost — old proverb

We must always bear in mind that we are dealing with sample data. Sometimes the "population" being sampled is largely conceptual — e.g. in the taxi data, we are considering the data a sample from the ridership in all days, past, present and future — but in any case, there is sampling variation.

In the breast cancer data, say, what if the data collection period had gone one more day? We would have more patients, and even the old patient data would change (e.g. survival time). Even a slight change would affect the exact split at the top of the tree, Node 1. And that effect could then change the splits (or possibly non-splits) at Nodes 2 and 3, and so on, with the those changes cascading down to the lowest levels of the resulting tree. In other words:

> Decision trees can be very sensitive to slight changes in the inputs. That means they are very sensitive to sampling variation, i.e. **decision trees have a high variance.**

In this section, we treat two major methods for handling this problem. Both take the point of view, "Variance too high? Well, that means the sample size is too small, so let's generate more trees!" But how?

5.2 Bagging: Generating New Trees by Resampling

A handy tool from modern statistics is the *bootstrap*. In the *bagging* method, we apply the bootstrap to decision trees.

Starting with our original data, once again considered a sample from some population, we'll generate s new samples from the original one. We generate a new sample by randomly sampling m of our n data points — *with* replacement. Typically we'll get a few duplicates. The quantities s and m here are — you guessed it — hyperparameters.

5.2.1 The Method

Say we have a new case to be predicted. We will then *aggregate* the s trees, by forming a prediction for each tree, then combining all those predicted values to form our final prediction. In a regression setting, say the taxi data, Section 4.3, the combining would take the form of averaging all the predictions.

What about a classification setting, such as the vertebrae example in Section 2.3? Then we could combine by using a *voting* process. For each tree, we would find the predicted class, NO, DH or SP, and then see what class received the most "votes" among the various trees. That would be our predicted class. Or, we could find the estimated class probabilities for this new case, for each tree, then average the probabilities. Our predicted class would be whichever one has the largest average.

So, we do a bootstrap and then aggregation, hence the short name, *bagging*. It is also commonly known as *random forests*, a specific implementation by Breiman.[1] That approach places a limit on the number of features under consideration for splitting at any given node, with a different candidate set at each step.

Why might this strategy work? Ordinary bagging can result in substantially correlated trees, because it tends to choose the same features every time. It can be shown that the average of positively correlated numbers has a higher variance than the average of independent numbers. Thus the approach in which we limit the candidate feature set at each step hopefully reduces variance even more.

1. The earliest proposal along these lines seems to be that of Tin Kam Ho.

5.2.2 Example: Vertebrae Data

The **partykit** package includes an implementation of random forests, **cforest()**. To illustrate it, let's look again at the dataset in Section 2.3, trying both a single tree and bagging. We'll use the default values of the hyperparameters. Since the dataset is small, $n = 100$, we'll try many random partitionings into training and test sets (100 partitionings, not related to the fact that $n = 100$.) Here is the code:

```
nreps <- 100
treeacc <- vector(length=100)
foracc <- vector(length=100)
for (i in 1:nreps) {
   tstidxs <- sample(1:nrow(vert),100)
   trn <- vert[-tstidxs,]
   tst <- vert[tstidxs,]
   ctout <- ctree(V7 ~ .,data=trn)
   predst <- predict(ctout,tst)
   treeacc[i] <- mean(predst == tst$V7)
   cfout <- cforest(V7 ~ .,data=trn)
   predsf <- predict(cfout,tst)
   foracc[i] <- mean(predsf == tst$V7)
}
mean(treeacc)
mean(foracc)
```

Output:

```
> mean(treeacc)
[1] 0.7967
> mean(foracc)
[1] 0.8362
```

So, the use of random forests rather than just single trees did bring some improvement.

5.2.3 The ranger Package

There are a number of R packages for fitting random forests, differing significantly in features. Since fitting these models can have very lengthy run times, the reader may consider **ranger** (as in "forest ranger"), known for its computational speed.[2]

2. Recall that there are two versions of the **partykit** package, each with its own version of **cforest()**. The earlier one is written in C, while the later one is pure R and thus slower.

5.2.3.1 Example: Remote-Sensing Soil Analysis

Here we will analyze the Africa Soil Property dataset on Kaggle.[3] From the data site:

> Advances in rapid, low cost analysis of soil samples using infrared spectroscopy, georeferencing of soil samples, and greater availability of earth remote sensing data provide new opportunities for predicting soil functional properties at unsampled locations...Digital mapping of soil functional properties, especially in data sparse regions such as Africa, is important for planning sustainable agricultural intensification and natural resources management.

One important property of this dataset that we have not encountered before is that is has $p > n$, more columns than rows.

```
> dim(afrsoil)
[1] 1157 3599
```

(The original first column, an ID variable, has been removed.)

Traditionally, the statistics field has been wary of this kind of setting, as linear models (Chapter 7) do not work there. One must first do dimension reduction. Tree-based methods do this as an integral aspect of their operation, so let's give it a try here. The data is large in the "width" sense, and the speed of **ranger** should be useful.

The column names, for the "X" variables, are all code names, while the ones for the "Y" variables are somewhat less cryptic:

```
> names(afrsoil)
...
[3547] "m659.543" "m657.615" "m655.686" "m653.758" "m651.829" "m649.901"
[3553] "m647.972" "m646.044" "m644.115" "m642.187" "m640.258" "m638.33"
[3559] "m636.401" "m634.473" "m632.544" "m630.616" "m628.687" "m626.759"
[3565] "m624.83"  "m622.902" "m620.973" "m619.045" "m617.116" "m615.188"
[3571] "m613.259" "m611.331" "m609.402" "m607.474" "m605.545" "m603.617"
[3577] "m601.688" "m599.76"  "BSAN"     "BSAS"     "BSAV"     "CTI"
[3583] "ELEV"     "EVI"      "LSTD"     "LSTN"     "REF1"     "REF2"
[3589] "REF3"     "REF7"     "RELI"     "TMAP"     "TMFI"     "Depth"
[3595] "Ca"       "P"        "pH"       "SOC"      "Sand"
```

This kind of setting has been considered tough. There is a major potential for overfitting; since with so many columns, one or more of them may accidentally look to be a strong predictor, due to p-hacking. Let's see how well **ranger** does here. There are five Y variables; let's predict pH, the acidity. Of course, since we are worried about overfitting, we'll use cross-validation.

```
> idxs <- sample(1:nrow(afrsoil),250)
> trn <- afrsoil[-idxs,]
```

3. https://www.kaggle.com/c/afsis-soil-properties/data

```
> tst <- afrsoil[idxs,]
> library(ranger)
> rout <- ranger(pH ~ .,data=trn[,c(1:3594,3597)])
> preds <- predict(rout,tst)
> mean(abs(preds$predictions - tst$pH))
[1] 0.3451455
```

Is that MAPE value small or large? Let's get an idea of the scale here:

```
> range(afrsoil$pH)
[1] -1.886946  3.416117
```

In that context, a MAPE of 0.35 seems like it might be good. And here is another way to view it: If we didn't have the features, we would just use the mean of pH as our predictor. The resulting MAPE would be as follows:

```
> mean(abs(mean(tst$pH) - tst$pH))
[1] 0.7307137
```

So, use of the features has cut MAPE in half.

5.2.3.2 A Case of $p > n$

As noted, the above soil analysis example is the first we've had in which $p > n$. Recall from Section 1.7.4.3 that classical statistics refused to consider this case, as the linear model becomes indeterminate in that setting. What about here?

To get an idea, let's do a single-tree run on this data:

```
> ctout <- ctree(pH ~ .,data=afrsoil[,c(1:3578,3597)])
> ctout
...
Fitted party:
[1] root
|   [2] m1359.58 <= 1.40589
|   |   [3] m1679.71 <= 1.4291
|   |   |   [4] m1930.42 <= 0.78926: 0.661 (n = 11, err = 8.8)
|   |   |   [5] m1930.42 > 0.78926: -0.237 (n = 30, err = 3.8)
|   |   [6] m1679.71 > 1.4291
|   |   |   [7] m2850.31 <= 0.83296: 2.228 (n = 18, err = 0.5)
|   |   |   [8] m2850.31 > 0.83296: 1.821 (n = 34, err = 0.7)
|   [9] m1359.58 > 1.40589
|   |   [10] m813.822 <= 1.82726
|   |   |   [11] m1670.07 <= 1.7327
|   |   |   |   [12] m1656.57 <= 1.56458
|   |   |   |   |   [13] m2584.17 <= 0.61801: -0.099 (n = 300, err = 120.5)
|   |   |   |   |   [14] m2584.17 > 0.61801: -0.701 (n = 38, err = 9.6)
|   |   |   |   [15] m1656.57 > 1.56458
|   |   |   |   |   [16] m1556.29 <= 1.46739: 0.891 (n = 66, err = 39.9)
|   |   |   |   |   [17] m1556.29 > 1.46739: -0.205 (n = 124, err = 44.3)
|   |   |   [18] m1670.07 > 1.7327: 1.352 (n = 54, err = 29.0)
|   |   [19] m813.822 > 1.82726
|   |   |   [20] m1641.14 <= 1.92639
|   |   |   |   [21] m2543.68 <= 0.73567
|   |   |   |   |   [22] m3326.64 <= 1.43327: -0.420 (n = 247, err = 98.1)
|   |   |   |   |   [23] m3326.64 > 1.43327: 0.240 (n = 67, err = 33.5)
|   |   |   |   [24] m2543.68 > 0.73567
|   |   |   |   |   [25] m5228.13 <= 0.39942: -0.938 (n = 83, err = 13.4)
|   |   |   |   |   [26] m5228.13 > 0.39942: -0.504 (n = 75, err = 16.3)
|   |   |   [27] m1641.14 > 1.92639: 1.805 (n = 10, err = 2.7)
...
```

So out of the over 3500 features, in the end, only a handful of them were actually used. In other words, $p < n$ after all.

5.3 Boosting: Repeatedly Tweaking a Tree

AdaBoost is the best off-the-shelf classifier in the world — CART co-inventor Leo Breiman, 1996

Imagine a classification problem, just two classes, so $Y = 1$ or 0, and just one feature, X, say age. We fit a tree with just one level. Suppose our rule is to guess $Y = 1$ if $X > 12.5$ and guess $Y = 0$ if $X \leq 12.5$. *Boosting* would involve exploring the effect of small changes to the 12.5 threshold on our overall rate of correct classification.

Consider a data point for which $X = 5.2$. In the original analysis, we'd guess Y to be 0. And — here is the point — if we were to move the threshold to, say, 11.9, we would *still* guess $Y = 0$, but the move may turn some misclassified data points near 12.5 to correctly classified ones. If more formerly misclassified points become correcly classified than vice versa, it's a win.

So the idea of boosting is to tweak the original tree, thus forming a new tree, then in turn tweaking that new tree, forming a second new tree, and so on. After generating s trees — s is a hyperparameter — we predict a new case by plugging it into all those trees and somehow combining the resulting predicted values.

5.3.1 Implementation: AdaBoost

The first proposal made for boosting was *AdaBoost*. The tweaking involves assigning weights to the points in our training set, which change with each tree. Each time we form a new tree, we fit a tree according to the latest set of weights, updating them with each new tree.

In a regression situation, to predict a new case with a certain X value, we plug that value into all the trees, yielding s predicted values. Our final predicted value is a weighted average of the individual predictions. (In a classification setting, we would take a weighted average of the estimated probabilities of $Y = 1$, to get the final probability estimate.)

To make this idea concrete, below is an outline of how the process could be implemented with **ctree()**. It relies on the fact that one of the arguments in **ctree()**, named **weights**, is a vector of nonnegative numbers, one for each data point. Say our response is named **y**, with features *x*. Denote the portion of the data frame **d** for *x* by **dx**.

```
ctboost <- function(d,s) {
   # uniform weights to begin
   wts <- rep(1/n,n)
   trees <- list()
   alpha <- vector(length=s)  # alpha[i] = coefficient for tree i
   for(treeNum in 1:s) {
      trees[[i]] <- ctree(y ~ x,data=d,weights=wts)
      preds <- predict(trees[[i]],dx)
      # update wts, placing larger weight on data points on which
      # we had the largest errors (regression case) or which we
      # misclassified (classification case)
      wts <- (computation not shown)
      # find latest tree weight
      alpha[i] <- (computation not shown)
   }
   l <- list(trees=trees,treeWts=alpha)
   class(l) <- 'ctboost'
   return(l)
}
```

And to predict a new case, **newx**:

```
predict.ctboost <- function(ctbObject,newx)
{
   trees <- ctbObject$trees
   alpha <- ctbObject$alpha
   pred <- 0.0
   for (i in 1:s) {
      pred <-
         pred + alpha[i] * predict(trees[[i]],newx)
   }
   return(pred)
}
```

Since this book is aimed to be nonmathematical, we omit the formulas for **wts** and **alpha**. It should be noted, though, that **alpha** is an increasing sequence, so that when we predict new cases, the later trees play a larger role.

5.3.2 Gradient Boosting

Recall the notion of a *residual*, the difference between a predicted value and actual value. *Gradient boosting* works by fitting trees to residuals. Given our dataset, the process works as follows:

1. Start with an initial tree. Set CurrentTree to it.
2. For each of our data points, calculate the residuals for CurrentTree.
3. Fit a tree *to the residuals*, i.e. take our residuals as the "data" and fit a tree T on it. Set CurrentTree = T.
4. Go to Step 2.

Then to predict a new case, plug it into all the trees. The predicted value is simply the sum of the predicted values from the individual trees.

At any given step, we are saying "OK, we've got a certain predictive ability so far, so let's work on 'what is leftover."' Hence our predicted value for any new case being the sum of what each treee predicts for that case.

The gradient boosting method gets its name from calculus; the gradient of a function is the vector of its partial derivatives. The user sets a loss function (Section 1.11.3), and we are trying to minimize average loss, hence the use of calculus.

In referring to the "residual" above, the technically correct term would be *pseudo-residual*, which is actually the gradient. For linear models (to be covered in Chapter 7), the gradient is the residual, but in general is actually some function of the residual.

5.3.3 The gbm Package

We'll use the **gbm** package for gradient boosting.

5.3.3.1 Example: Call Network Monitoring

Let's first apply it to a dataset titled, Call Test Measurements for Mobile Network Monitoring and Optimization,[4] which rates quality of service on mobile calls. The aim is to predict the quality rating. Here is an introduction to the data:

```
> ds <- read.csv('dataset.csv')
> names(ds)
[1] "Date.Of.Test"             "Signal..dBm."
[3] "Speed..m.s."              "Distance.from.site..m."
[5] "Call.Test.Duration..s."   "Call.Test.Result"
[7] "Call.Test.Technology"     "Call.Test.Setup.Time..s."
[9] "MOS"
> ds <- ds[,-1]
> head(ds)
  Signal..dBm. Speed..m.s. Distance.from.site..m. Call.Test.Duration..s.
1          -61       68.80                1048.60                     90
2          -61       68.77                1855.54                     90
3          -71       69.17                1685.62                     90
4          -65       69.28                1770.92                     90
5         -103        0.82                 256.07                     60
```

4. *https://www.kaggle.com/valeriol93/predict-qoe*

```
6         -61      68.86                 452.50                       90
  Call.Test.Result Call.Test.Technology Call.Test.Setup.Time..s. MOS
1         SUCCESS                 UMTS                     0.56 2.1
2         SUCCESS                 UMTS                     0.45 3.2
3         SUCCESS                 UMTS                     0.51 2.1
4         SUCCESS                 UMTS                     0.00 1.0
5         SUCCESS                 UMTS                     3.35 3.6
6         SUCCESS                 UMTS                     0.00 1.0
...
```

Now, let's fit the model:

```
> gbout <- gbm(MOS ~ .,data=ds,n.trees=500)
Distribution not specified, assuming gaussian ...
```

The comment is a bit misleading; it is simply **gbm()**'s way of saying that it is assuming that this is a regression application, not classification, which is correct.

Let's do a prediction. Say we have a case like **ds[3,]**, but with distance being 1500 and duration 62:

```
> ds3 <- ds[3,]
> ds3[,3] <- 1500
> ds3[,4] <- 62
> predict(gbout,ds3,n.trees=500)
[1] 2.4734
```

Here we told **gbm()** to use all 500 of the trees it had computed and stored in **gbout**.

But should we have used them all? After all, 500 may be overfitting. Maybe the later trees were doing "noise fitting." The package has a couple of ways of addressing that issue.

One approach is to use the auxiliary function **gbm.perf()**, which estimates the optimal number of trees.

```
> gbm.perf(gbout)
```

See the output graph in Figure 5-1.
The dashed vertical line shows the estimated "sweet spot," best number of trees. The curve shows Mean Squared Prediction Error, when the same data is used for estimation and prediction, thus overly optimistic.

The second approach is to direct **gbm()** do cross-validation at each step. Here we specify 90% of the data for the training set:

```
> gbout1 <- gbm(MOS ~ .,data=ds,n.trees=500,train.fraction=0.9)
> gbout1
gbm(formula = MOS ~ ., data = ds, n.trees = 500, train.fraction = 0.9)
A gradient boosted model with gaussian loss function.
500 iterations were performed.
The best test-set iteration was 496.
```

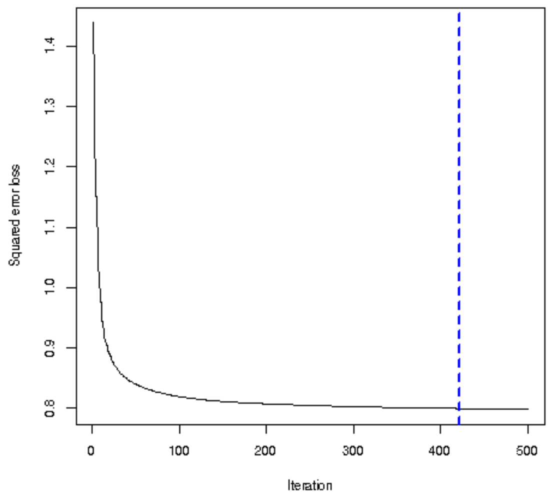

*Figure 5-1: Output from **gbm.perf***

```
There were 7 predictors of which 7 had non-zero influence.
> gbm.perf(gbout1)
[1] 496
```

Here it recommends 496 as the optimal number of trees, which is also seen in the accompanying graph, Figure **??**, considerably more aggressive than the earlier figure of the low 400s. The documentation does say tha **gbm.perf()** gives a conservative number.

The lower curve is in the previous graph, but the upper one is *out-of-band* (i.e. test set) MSPE, again reflecting the fact that accuracy reported from predicting on the estimation dataset is optimistic.

Here is **gbm()** applied to the the vertebrae data (Section 2.3):

```
> vout <- gbm(V7 ~ .,data=vert)
Distribution not specified, assuming multinomial ...
```

The **gbm()** function will infer the type of analysis it needs to do from the nature of the input Y values. If Y just takes the values 0 and 1 (or even is a factor with levels, say, 'Yes' and 'No'), it knows it is fitting conditional probabilities that $Y = 1$. If Y is a factor with multiple levels, **gbm()** knows this to be a multiclass classification problem ("multinomial"), as is the case for the **vert** data here. Otherwise, it is a regression problem, as in our mobile phone example earlier.

By the way, the default value for **n.trees** is 100, so we have that here.

The **vert** dataset has only 310 cases, so there is a real concern regarding overfitting. Let's check:

```
> gbm.perf(vout)
```

```
OOB generally underestimates the optimal number of iterations although
...
[1] 17
...
```

Again, we may wish to request internal cross-validation.

As an example of predicting new cases, let's predict a case similar to **vert[12,]**:

```
> vert[12,]
      V1    V2   V3    V4     V5  V6 V7
12 31.23 17.72 15.5 13.52 120.06 0.5 DH
> predict(vout,vert12,type='response',n.trees=17)
, , 17

           DH        NO         SL
[1,] 0.7173993 0.2472431 0.03535766
```

DH is the mostly likely by far, so we would predict that class.

When *Y* is a dummy variable or a factor, we are modeling a probability. Setting **type='response'** means we want the predictions to take the form of probabilities. (We then rounded to get the 0 or 1 values for predicted *Y*.)

5.3.4 Bias vs. Variance in Boosting

Boosting is "tweaking" a tree, potentially making it more stable, especially since we are averaging many trees, smoothing out "For want of a nail..." problems. So, it may reduce variance. By making small adjustments to a tree, we are potentially developing a more detailed analysis, thus reducing bias.

But all of that is true only "potentially." Though the tweaking process has some theoretical basis, it still can lead us astray, actually *increasing* bias and possibly increasing variance too. If the hyperparameter *s* is set too large, producing too many trees, we may overfit.

5.3.5 Computational Speed

Boosting can take up tons of CPU cycles, so we may need something to speed things up. The **n.cores** argument tries to offload computation to different cores in your machine. If you have a quad core system, you may try setting this argument to 4, or even 8.

5.3.6 Hyperparameters

Boosting algorithms typically have a number of hyperparameters. We have already mentioned **n.trees**, the number of trees to be generated, for instance. As noted above, in terms of the bias-variance tradeoff, increasing the number of trees will reduce bias but increase variance.

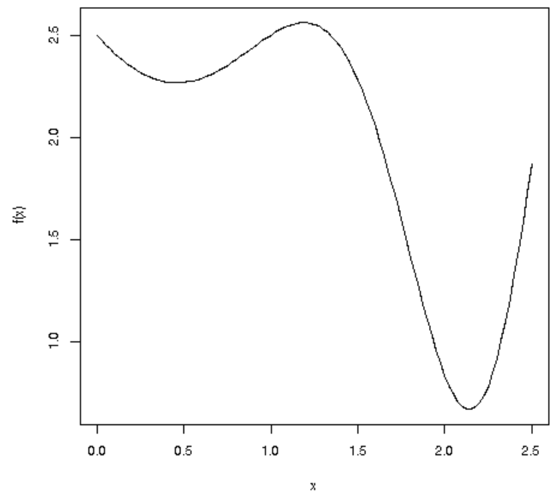

Figure 5-2: A function to be minimized

Another hyperparameters is **n.minobsinnode**, the minimum number of data points we are willing to have in one tree node. As we saw in the last chapter, reducing this value will reduce bias but increase variance.

Another hyperparameter, **shrinkage**, is so important in the general ML context that we'll cover it in a separate subsection, next.

5.3.7 The Learning Rate

The notion of a *learning rate* comes up often in ML. We'll describe it here in general, then explain how it works for gradient boosting, and we'll see it again in our chapters of support vector machines and neural networks.

5.3.7.1 General Concepts

Recall that in ML methods we are usually trying to minimize some loss function, such as MAPE or the overall misclassification error. Computationally, this minimization can be a challenge.

Consider the function graphed in Figure 5-2. It is a function of a one-dimensional variable x, whereas typically our x is high-dimensional, but it will make our point.

There is an overall minimum at approximately $x = 2.2$. This is termed the *global minimum*. But there is also a *local minimum*, at about $x = 0.4$. Let's give the name x_0 to the value of x at the global minimum.

To us humans looking at the graph, it's clear where x_0 is, but we need our software to be able to find it. That may be problematic. Here's why:

Most ML algorithms use an *iterative* approach to finding the desired minimum point x_0. This involves a series of guesses for x_0. The code starts with an initial guess g_0 say randomly chosen, then evaluates $f(g_0)$. Based on

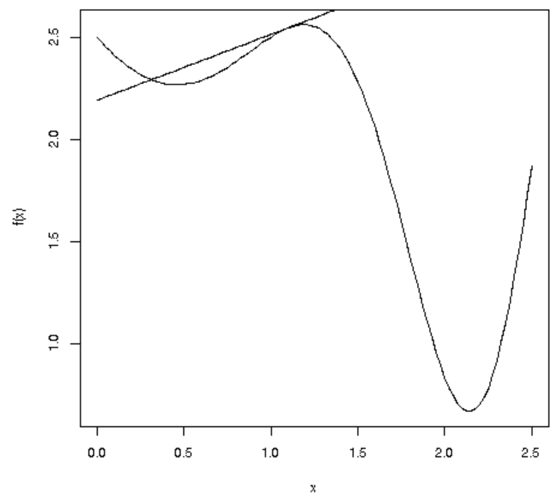

Figure 5-3: function to be minimized, plus tangent

the result, the algorithm then somehow (see below) updates the guess, to g_1. It then evaluates $f(g_1)$, producing the next guess g_2, and so on.

The algorithm keeps generating guesses until they don't change much, say until $|g_{i+1} - g_i| < 0.00000001$ for step i. We say that the algorithm has *converged* to this point. Let's give the name c to that value of i It then reports x_0, the global minimum point, to be the latest guess, g_c.

So, what about that "somehow" alluded to above? How does the algorithm generate the next guess from the present one? The answer lies in the *gradient*, a calculus term for the vector of partial derivatives of a function. In our simple example here with x being one-dimensional, the gradient is simply the slope of the function at the given point, i.e. the slope of the tangent line to the curve.

Say our initial guess $g_0 = 1.1$. The tangent line is shown in Figure 5-3. The line is pointing upward, i.e. has positive slope, so it tells us that by going to the left we will go to smaller values of the function, which is what we want. *The learning rate then tells us how far to move in that direction* in setting our next guess, g_1. This is called the *step size*.

But you can see the problem. We should be moving to the right, not to the left. The function $f(x)$ is fooling the algorithm here. Actually, in this scenario, our algorithm will probably converge to the wrong point.

Worse, in some cases, the algorithm may not converge at all. This is why typical ML packages allow the user to set the learning rate. Small values may be preferable, as large ones may result in our guesses lurching back and forth, always missing the target. On the other hand, if it is too small, we will just inch along, taking a long time to get there. Or worse, we converge to a local minimum.

Once again, we have a hyperparameter for which we need it to be at a "Goldilocks" level, not too large and not too small, and may have to experiment with various values.

5.3.7.2 The Learning Rate in gbm

This is the **shrinkage** argument. Say we set it to 0.2. Recall the pseudocode describing gradient boosting in Section 5.3.2. The revised version is this:

1. Start with an initial tree. Set CurrentTree to it.
2. For each of our data points, calculate the residuals for CurrentTree.
3. Fit a tree *to the residuals*, i.e. take our residuals as the "data" and fit a tree T on it. Set CurrentTree = shrinkage * T.
4. Go to Step 2.

where `shrinkage * T` means multiplying all the values in the terminal nodes of the tree by the factor **shrinkage**. In the end, we still add up all our trees to produce the "supertree" used in prediction of new cases.

Again, a small value of **shrinkage** is more cautious and slower, and it may cause us to need more trees in order to get good predictive power. But it may help prevent overfitting.

5.4 Pitfall: No Free Lunch

There is no such thing as a free lunch — old economics saying

Though Leo Breiman had a point on the considerable value of AdaBoost (especially in saying "off the shelf," meaning usable with just default values of hyperparameters), that old saying about "no free lunch" applies as well. As always, applying cross-validation and so on is indispensable to developing good models.

Similar advice concerns another famous Breiman statement, that it is impossible to overfit using random forests. The reader who has come this far in this book will immediately realize that Breiman did not mean his statement in the way some have interpreted it. Any ML method may overfit. What Breiman meant was that it impossible to set the value of *s*, the number of trees, too high. But the trees themselves still can overfit, e.g. by having too small a minimum value for number of data points in a node.

6

THE VERDICT: NEIGHBORHOOD-BASED MODELS

In finishing Part I, let's review the dual goals of that part of the book:

- Introduce some general concepts that will arise throughout the book, such as cross-validation and the learning rate.
- Introduce the k-NN and decision tree methods, while at the same time using these methods as examples of those general concepts.

In light of that second goal, it's time to take stock of the two methods.

k-NN

Advantages:

- Easy to explain to others.
- Only one hyperparameter to choose.

- No iteration and convergence issues to worry about.
- Long-established record of effectiveness.

Disadvantages:

- Significant computational needs.
- Does not exploit the general monotonicity of relationships (e.g. taller people tend to be heavier).
- Generally considered to have problems in large-*p* settings, i.e. applications with a large number of features.

Decision trees

Advantages:

- Easy to explain.
- Computation for predicting new cases is very fast.
- Exploits monotonicity to some extent.

Disadvantages:

- Computationally intensive at the training set level.

PART II

METHODS BASED ON LINES AND PLANES

7
PARAMETRIC METHODS: LINEAR AND GENERALIZED LINEAR MODELS

Recall the term *regression function*, first introduced in Section 1.8 and denoted by $r(t)$. It's the mean Y in the subpopulation defined by the condition $X = t$. The example we gave then involved the bike ridership data:

> A regression function has as many arguments as we have features. Let's take humidity as a second feature, for instance. So, to predict ridership for a day with temperature 28 and humidity 0.51, we would use the mean ridership in our dataset, among days with temperature and humidity approximately 28 and 0.51. In regression function notation, that's $r(28, 0.51)$.

Basic ML methods all are techniques to estimate the regression function from sample data. With k-NN, we would estimate $r(28, 0.51)$ in the bike ridership example by calculating the mean ridership among the days closest to (28,0.51). With trees, we would plug

(28,0.51) into our tree, follow the proper branches, and then calculate the mean ridership in the resulting terminal node.

The concept of a regression function thus plays a central role in ML. Indeed, this was summarized in the title of Section 1.8, "The Regression Function: What Is Being "Learned" in Machine Learning."

So far, we have not made any assumptions about the shape of the regression function graph. Here we will assume the shape is that of a straight line, or planes and so on in higher dimensions.

7.1 Motivating Example: The Baseball Player Data

Recall the dataset **mlb** in Section 1.10.2 that is included with **regtools**. Let's restrict attention to just heights and weights of the players:

```
> data(mlb)
> hw <- mlb[,c(4,5)]
```

7.1.1 A Graph to Guide Our Intution

Let's consider predicting weight from height. In the $r()$ notation, that means that if we wish to predict the weight of a new player whose height is 71, we need to estimate $r(71)$. This is the mean weight of all players in the subpopulation of players having height 71.

We don't know population values, as we only have a sample from the population. (As noted earlier, we consider our data to be a sample from the population of all players, past, present and future.) How, then, can we estimate $r(71)$? The natural estimate is the analogous sample quantity, the mean weight of all height 71 players in our sample:

```
> mean(mlb$Weight[mlb$Height == 71])
[1] 190.3596
```

Recalling that the the "hat" notation means "estimate of," we have that $\widehat{r}(71)$ = 190.3596. With deft usage of R's **tapply()** function, we get all the estimated **r** values:

```
meanWts <- tapply(hw$Weight,hw$Height,mean)
```

This says, "Group the weight values by height, and find the mean weight in each group."

Let's plot the estimated mean weights against height:

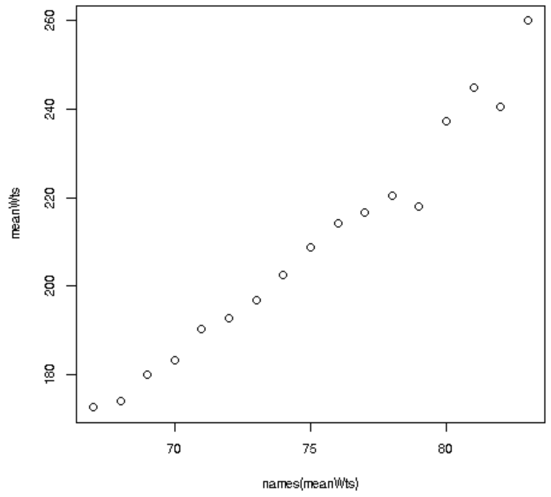

Figure 7-1: Estimated regression function, weight vs. height

```
> plot(names(meanWts),meanWts)
```

with the result seen in Figure 7-1.

Remarkably, the points seem to nearly lie on a straight line. This suggests a model for $r(t)$:

$$r(t) = b + mt \tag{7.1}$$

for some unknown values of the slope m and intercept b. This is the *linear model.*

7.1.2 View as Dimension Reduction

If Equation 7.1 is a valid model (see below) we have greatly simplified our problem, as follows:

Ordinarily, we would need to estimate many different values of $r(t)$,188.46 such as those for t equal to 68, 69, 70, 71, 72, 73 and so on, say 15 or 20 of them. But with the above model, *we need to estimate only two numbers*, m and b. As such, this is a form of dimension reduction.

7.2 The lm() Function

Assuming the linear model (again, we'll address its validity shortly), we can use R's **lm()** function to estimate m and b:

```
> lmout <- lm(Weight ~ .,data=hw)
> lmout
```

```
Call:
lm(formula = Weight ~ ., data = hw)

Coefficients:
(Intercept)      Height
   -151.133       4.783
```

So, $\widehat{m} = 4.783$ and $\widehat{b} = -151.133$.

The form of the call above is: We are requesting that R fit a linear model to our data frame **hw**, predicting weight. The '.' means "all other columns," which in this case is just the height column.

To predict the weight of a new player of height 71, then, we would compute

$$\widehat{r}(71) = \widehat{b} + \widehat{m} \cdot 71 = -151.133 + 4.783 \times 71 = 188.46 \tag{7.2}$$

But hey, we should have the computer do this computation, rather than our doing it by hand:

```
> predict(lmout,data.frame(Height=71))
       1
188.4833
```

(The slight discrepancy is due to roundoff error in the computation by hand, where our data was given only to a few digits.)

7.3 Another Generic Function

In Section 4.2.5 we introduced the notion of of a *generic* function. In the above code, **predict()** is also such a function. That **lmout**, which had been the output of **ctree**, is of class **'lm'**. So the R interpreter dispatched that **predict()** call to **predict.lm()**. Then, we had to place the 71 into a data frame, so that **predict.lm()** knew that 71 corresponded to the Height column in the original data.

7.4 Use of Multiple Features

We can, and typically do, fit the model to more than one feature.

7.4.1 Baseball Player Example, cont'd.

Say we add in Age, so our linear model is

$$\text{mean weight} = b + m_1 \text{ height} + m_2 \text{ age} \tag{7.3}$$

Here is the **lm()** call:

```
> data(mlb)
> lmo <- lm(Weight ~ Height + Age,data=mlb)
> lmo

Call:
lm(formula = Weight ~ Height + Age, data = mlb)

Coefficients:
(Intercept)      Height         Age
  -187.6382      4.9236      0.9115
```

So for instance $\widehat{m}_2 = 0.9115$.

And here is the prediction, say if the player is age 28:

```
> predict(lmo,data.frame(Height=71, Age = 28))
       1
187.4603
```

7.4.2 Impact of Adding More Features

What if the player's age were 31?

```
> predict(lmo,data.frame(Height=71, Age = 31))
       1
190.1949
```

Note that the predicted value is slightly higher than for the 28-year-old. This is a very important point. Consider:

- The fact that $\widehat{m}_2 > 0$ would indicate that ballplayers, in spite of needing to keep fit, do gain weight as they age, in this case an estimated 0.9 pound per year.
- On the other hand, keep in mind the "hat" in $\widehat{m}_2$. We only have sample data, thus only have the estimated age effect. Statistical methods such as confidence intervals could be used here to decide that the aging effect is real, but in any case it is rather small.
- In addition, we must as always worry about overfitting. Here we have $n = 1023$ and $p = 2$, so we need not worry. But if we had many more features (as in the song data), at some point adding one more predictor would move us into overfitting territory.

7.5 How It Works

The quantities $\widehat{m}$ and $\widehat{b}$, etc., are computed using the famous *ordinary least squares* method (OLS), which works as follows:

Imagine that after we compute $\widehat{m}$ and $\widehat{b}$, we go back and "predict" the weight of the first player in our sample data. As implied by the quotation marks, this would be silly; after all, we already know the weight of the first player, 180:

```
> data(mlb)
> mlb[1,]
           Name Team Position Height Weight   Age
1 Adam_Donachie  BAL  Catcher     74    180 22.99
  PosCategory
1     Catcher
```

But think through this exercise anyway; as in Section 1.11.1, it will turn out to be the basis for how things work. Our predicted value would be $\widehat{b} + \widehat{m} \cdot 71$. Thus our prediction error would be

$$180 - (\widehat{b} + \widehat{m} \cdot 71)$$

We'll square that error, as we will be summing errors, and we don't want positive and negative ones to cancel. Now "predict" all the other data points as well, and add up the squared errors:

$$[180 - (\widehat{b} + \widehat{m} \cdot 71)]^2 + [215 - (\widehat{b} + \widehat{m} \cdot 74)]^2 + ... + [195 - (\widehat{b} + \widehat{m} \cdot 73)]^2 \qquad (7.4)$$

Now here is the point: From the beginning, *we choose $\widehat{m}$ and $\widehat{b}$ to minimize the above sum of squared errors.* In other words, we choose the slope and intercept estimates in such a way that we would predict our data best.

By the way, recall the terminology from Section 1.11.3: Those prediction errors — known Y minus predicted Y — are known as *residuals*.

7.6 Diagnostics: Is the Linear Model Valid?

The linearity assumption is pretty strong. Let's take a closer look.

7.6.1 Exactness?

All models are wrong, but some are useful — famous early statistician George Box

The reader may ask, "How can the linear model in Equation (7.1) be valid? Yes, the points in Figure 7-1 look kind of on a straight line, but not exactly so." There are two important answers:

- As the quote from George Box above points out, no model is *exactly* correct. Commonly used physics models ignore things like air resistance and friction, and even models accounting for such things still don't reflect all possible factors. A linear approximation to the regression function $r(t)$ may do a fine job in prediction.
- Even if Equation (7.1) were exactly correct, the points in Figure 7-1 would not lie exactly on the line. Remember, $r(71)$, for instance, is only the *mean* weight of all players of height 71. Individual players of that height are heavier or lighter than that value.

By the way, classical linear model methodology makes some assumptions beyond linearity, such as *Y* having a normal distribution in each subpopulation. But these are not relevant to our prediction context.[1]

7.6.2 Diagnostic Methods

Over the years, analysts have developed a number of methods to check the validity of the linear model. Several are described, for instance, in N. Matloff, *Statistical Regression and Classification: from Linear Models to Machine Learning*, CRC, 2017. One such method, described below, involves k-NN.

The idea is simple: Linear models make an assumption, k-NN models don't. So, plot the predicted values in our data computed by the linear model against the predicted values computed by k-NN. If they tend to be close to each other, the linear model is judged to be valid. (Actually, *approximately* valid, as noted above.)

7.6.3 Example: Song Data

Recall the Million Song dataset from Section 3.1.1. There we predicted the year of release of a song, using 90 features dealing with audio characterisitics. Let's look at the validity of the linear model here:

```
> lmout <- lm(V1 ~ .,data=yr)  # 1st col is song year, to be predicted
# since k-NN's computation is voluminous, just plot at a random subset
> idxs <- sample(1:nrow(yr),1000)
> linPreds <- predict(lmout,yr[idxs,-1])
> knnPreds <- kNN(yr[idxs,-1],yr[idxs,1],yr[idxs,-1],10)$regests
> plot(knnPreds,linPreds)
```

The result is shown in Figure 7-2. The linear predictions largely correlate with those from k-NN, albeit with a few outliers. The linear model thus is usable.

7.7 Handling R Factors

One of the most useful aspects of R is its factor type, representing categorical variables. Among other things, it makes our lives easier in many types of prediction models.

7.7.1 Example: San Francisco Air B&B Data

The short-term housing firm Air B&B makes available voluminous rental data.[2]. Here we look at some data from San Francisco.

1. Actually, even for statistical inference, the normality assumption is not important in large samples.
2. *http://insideairbnb.com/get-the-data.html*

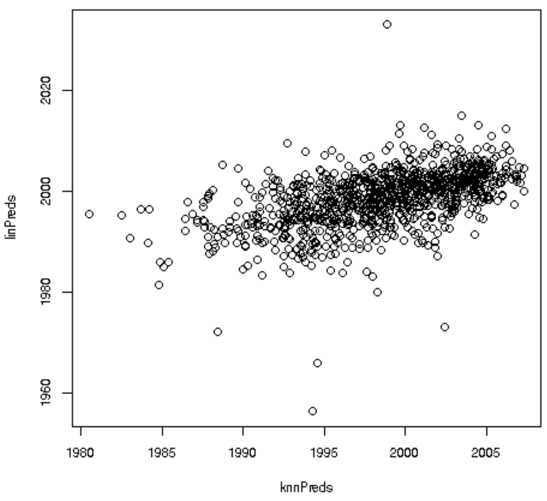

Figure 7-2: Estimated regression function, linear vs. k-NN

Though our primary interest here is to illustrate how R deals with factor in functions such as **lm()**, this dataset will also illustrate a number of other issues.

We will be predicting rental price. Such a prediction model could be useful for a property owner who is pondering how high to set the rent.

7.7.1.1 Data Preparation

There is a **listings.csv** file, one line per real estate property, and a separate customer reviews file. Here we look at the former. Here is a partial list of the column names:

```
> names(abb)
...
11] "notes"                        "transit"
[13] "access"                       "interaction"
[15] "house_rules"                  "thumbnail_url"
[17] "medium_url"                   "picture_url"
[19] "xl_picture_url"               "host_id"
[21] "host_url"                     "host_name"
...
[49] "latitude"                     "longitude"
[51] "is_location_exact"            "property_type"
[53] "room_type"                    "accommodates"
[55] "bathrooms"                    "bedrooms"
[57] "beds"                         "bed_type"
[59] "amenities"                    "square_feet"
[61] "price"                        "weekly_price"
```

```
[63] "monthly_price"                    "security_deposit"
[65] "cleaning_fee"                     "guests_included"
[67] "extra_people"                     "minimum_nights"
...
```

Many of the features are textual, e.g.

```
> abb[1,]$house_rules
[1] "* No Pets - even visiting guests for a short time period. * No Smokers allowed - even
```

We treat the topic of text data later in this book.

Another issue was that prices included dollar signs and commas, e.g.

```
> abb[1,]$monthly_price
[1] "$4,200.00"
```

We wrote this function to convert a column **d** of such numbers to the proper form:

```
convertFromDollars <- function(d) {
   d <- as.character(d)
   # replace dollar sign by ''
   d <- sub('\\$','',d,fixed=F)
   # replace commas by ''
   d <- gsub(',','',d)
   d <- as.numeric(d)
   # some entries were ''
   d[d == ''] <- NA
   d
}
```

And, not suprisingly, this dataset seems to have its share of erroneous entries:

```
> table(abb$square_feet)

   0    1    2   14  120  130  140  150  160  172  175  195  250  280  300  360
   2    3    2    1    1    1    3    2    1    1    1    1    2    2    4    2
 400  450  500  538  550  600  650  700  750  780  800  810  815  840  850  853
   1    2    8    1    1    4    1    3    5    1    4    1    1    1    1    1
 890  900  950 1000 1012 1019 1100 1200 1390 1400 1490 1500 1600 1660 1750 1800
   1    2    3    9    1    1    2    9    1    2    1    7    1    1    1    3
1850 1900 1996 2000 2100 2200 2250 2600 3000
   1    1    1    4    3    2    1    1    4
```

Areas of for instance, 1 and 2 square foot are listed, clearly incorrect. We will not pursue this further here, so as to focus on the R factors issuR e

After selecting a few features, we have a data frame **Abb**:

```
> head(Abb)
```

```
  zipcode bathrooms bedrooms square_feet weekly_price monthly_price
1   94117       1.0        1          NA      1120.00       4200.00
2   94110       1.0        2          NA      1600.00       5500.00
3   94117       4.0        1          NA       485.00       1685.00
4   94117       4.0        1          NA       490.00       1685.00
5   94117       1.5        2          NA         <NA>          <NA>
6   94115       1.0        2          NA         <NA>          <NA>
  security_deposit guests_included minimum_nights maximum_nights
1           100.00               2              1             30
2             <NA>               2             30             60
3           200.00               1             32             60
4           200.00               1             32             90
5             0.00               2              7           1125
6             0.00               1              2            365
  review_scores_rating
1                   97
2                   98
3                   85
4                   93
5                   97
6                   90
```

7.7.1.2 Missing Values

This reveals yet another problem — this is a typical real dataset, loaded with problems — NA values. In fact:

```
> for(i in 1:ncol(Abb)) print(sum(is.na(Abb[,i])))
[1] 193
[1] 19
[1] 1
[1] 6944
[1] 5888
[1] 5887
[1] 1436
[1] 0
[1] 0
[1] 0
[1] 1359
> nrow(Abb)
[1] 7072
```

So for instance, 6944 of the 7072 rows have the square footage missing. We'll delete that feature, but clearly there are serious issues remaining. How does **lm()** handle this?

The answer is that *listwise deletion* is used. As explained in Section 1.17, this means that only fully complete cases are used. That will, alas, greatly reduce the size of our data.

7.7.1.3 The lm() Call

Here is the call, not using the square footage and weekly price columns:

```
> lmout <- lm(monthly_price ~ .,data=Abb[,-c(4,5)])
> lmout

Call:
lm(formula = monthly_price ~ ., data = Abb[, -c(4, 5)])

Coefficients:
         (Intercept)         zipcode94103         zipcode94104
          -4.486e+03           -4.442e+02            6.365e+02
        zipcode94105         zipcode94107         zipcode94108
           1.012e+03           -2.846e+02           -1.650e+03
        zipcode94109         zipcode94110         zipcode94111
          -3.946e+02           -1.113e+03            1.620e+03
        zipcode94112         zipcode94114         zipcode94115
          -2.304e+03           -2.608e+02           -3.881e+02
        zipcode94116         zipcode94117         zipcode94118
          -1.959e+03           -1.543e+02           -1.363e+03
        zipcode94121         zipcode94122         zipcode94123
          -1.315e+03           -1.434e+03            1.640e+03
        zipcode94124         zipcode94127         zipcode94131
          -2.310e+03           -2.128e+03           -1.526e+03
        zipcode94132         zipcode94133         zipcode94134
          -1.676e+03            6.497e+02           -1.370e+03
        zipcode94158            bathrooms             bedrooms
          -2.509e+03            2.025e+02            1.541e+03
    security_deposit      guests_included       minimum_nights
           3.462e-01            3.663e+02           -6.401e-01
      maximum_nights  review_scores_rating
          -2.371e-04            6.613e+01
```

As noted in Section 1.7.4.1, there are more than 40,000 ZIP codes in the US, typically far too many to use directly. However, in San Francisco, the number is manageable.

We could thencall the **factorsToDummies()** function from the **regtools** package to convert the ZIP codes ourselves. But actually, R saves us the trouble. It recognizes that the **zipcode** column is a factor, and automatically converts to dummies internally. The coefficient report above then shows the estimated coefficient for each dummy.

By the way, as real estate agents say, "Location, location, location" — ZIP code matters a lot. A property in district 94105 commands aprice premium of more than $1000 while one in district 94107 will on average be almost $300 cheaper than average for the city, holding all other varibles fixed.

7.7.1.4 More on lm()

We've already seen examples of the notion of a *generic* function in R, such as **print()** and **plot()**. Another common one is **summary()**. A call to that function here will be dispatched to **summary.lm()**, which prints out various measures of interest. Here is part of the output:

```
> summary(lmout)
...
  (6124 observations deleted due to missingness)
Multiple R-squared:  0.5191,    Adjusted R-squared:  0.5028
...
```

The material not shown is mainly of statistical interest, less important in our prediction context. Instead, a useful measure here is R^2.

Recall that the estimated coefficients are caclulated by minimizing the same of squared differences between actual and predicted Y values. (Section 7.5.) R^2 is the squared correlation between actual and predicted Y. It can be shown that this can be interpreted as the proportion of variation in Y due to X. (As always, X refers collectively to all our features.)

As such, we have $0 \leq R^2 \leq 1$, with a value of 1 meaning that X perfectly predicts Y. **HOWEVER**, there is a big problem here, as we are predicting the same data that we used to estimate our prediction machine, the regression coefficients. If we are overfitting, then R^2 will be overly optimistic.

Thus **summary()** also reports a quantity known as *adjusted* R^2, which has been derived to be less susceptible to overfitting. In this case here, there is no indication of overfitting, as the two measures are very close.

7.7.1.5 Pitfall: Overreliance on Adjusted R^2

The adjusted R^2 value is very rough information, and though it can give a quick idea as to whether we are overfitting, in-depth analsys, say for choosing which of several combinations of features to use, really requires cross-validation.

7.8 Polynomial Models

Surprisingly, one can use linear regression methods to model nonlinear effects. We'll see how in this section.

In addition, polynomials will play an important role in our chapter on support vector machines, and even in our treatment of neural networks, where there is a surprising connected to polynomials.

7.8.1 Motivation

We've used the example of programmer and engineer wages earlier in this book. Consider a graph of wage income against age, Figure 7-3. There seems to be a steep rise in a worker's 20s, then a long leveling off, with a hint even of a decline after age 55 or so. This is definitely not a linear relationship.

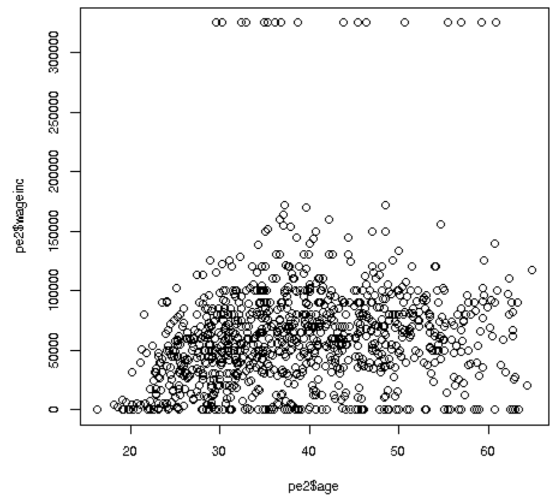

Figure 7-3: Wage income vs. age

Or consider Figure 7-4, for the bike sharing data from Chapter 1, graphing total ridership against temperature. The nonlinear relationship is even clearer here. No surprise, of course — people don't want to go bike riding if the weather is too cold or too hot — but again the point is that a linear model would seem questionable.

Fortunately, these things actually *can* be accommodated with linear models.

7.8.2 Modeling Nonlinearity with a Linear Model

Again, let's look at bike ridership, choosing a few interesting features for illustration.

```
> data(day1)
> head(day1,3)
  instant     dteday season yr mnth holiday
1       1 2011-01-01      1  0    1       0
2       2 2011-01-02      1  0    1       0
3       3 2011-01-03      1  0    1       0
  weekday workingday weathersit     temp
1       6          0          2 8.175849
2       0          0          2 9.083466
3       1          1          1 1.229108
      atemp      hum windspeed casual registered
1  7.999250 0.805833  10.74988    331        654
2  7.346774 0.696087  16.65211    131        670
3 -3.499270 0.437273  16.63670    120       1229
```

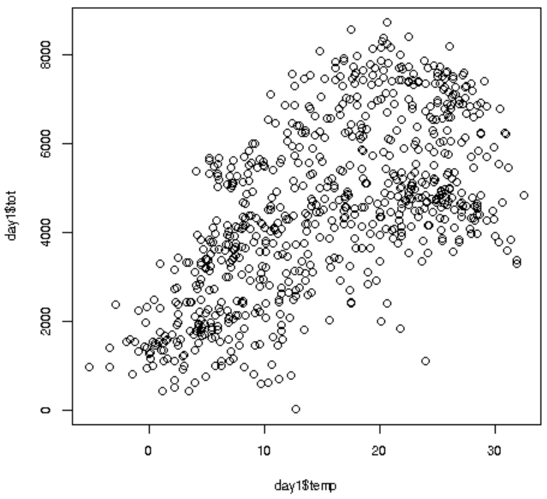

Figure 7-4: Ridershp vs. temperature

```
   tot
1  985
2  801
3 1349
> d1 <- day1[,c(4:8,10,12,13,16)]
> names(d1)
[1] "yr"         "mnth"       "holiday"
[4] "weekday"    "workingday" "temp"
[7] "hum"        "windspeed"  "tot"
```

Let's fit a purely linear model first:

```
> summary(lm(tot ~ .,data=d1))
...
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  3074.246    240.924  12.760  < 2e-16
yr           2006.856     71.639  28.013  < 2e-16
mnth           94.178     10.887   8.651  < 2e-16
holiday      -589.426    220.864  -2.669  0.00778
weekday        53.332     17.850   2.988  0.00291
workingday     84.996     79.055   1.075  0.28267
temp          126.841      4.284  29.608  < 2e-16
hum         -2559.445    264.727  -9.668  < 2e-16
windspeed     -56.512      7.200  -7.848 1.52e-14

(Intercept) ***
yr          ***
```

```
mnth        ***
holiday     **
weekday     **
workingday
temp        ***
hum         ***
windspeed   ***
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘’. 0.1 ‘ ’ 1
...
Multiple R-squared:  0.7571, Adjusted R-squared:  0.7544
```

(We will discuss some other columns of the output in Section 7.9.)

Now let’s try adding a temperature squared term:

```
> d1$temp2 <-  d1$temp^2
> summary(lm(tot ~ .,data=d1))
...
Coefficients:
              Estimate Std. Error t value
(Intercept)  2899.1676   213.1495  13.602
yr           1904.3064    63.6808  29.904
mnth           65.4919     9.8229   6.667
holiday      -501.2602   195.1773  -2.568
weekday        54.2261    15.7661   3.439
workingday     57.4086    69.8525   0.822
temp          337.6918    15.2232  22.183
hum         -3362.2353   240.4677 -13.982
windspeed     -69.9450     6.4288 -10.880
temp2          -6.8940     0.4821 -14.299
            Pr(>|t|)
(Intercept)  < 2e-16 ***
yr           < 2e-16 ***
mnth        5.18e-11 ***
holiday     0.010422 *
weekday     0.000617 ***
workingday  0.411432
temp         < 2e-16 ***
hum          < 2e-16 ***
windspeed    < 2e-16 ***
temp2        < 2e-16 ***
...
Multiple R-squared:  0.8108, Adjusted R-squared:  0.8084
...
```

Whenever we add a new feature to a linear model, the estimated coefficients change. In particular, the **temp** coefficient changed from 126.841

to 337.6918, and we now have a coefficient for the square of the variable, -6.8948. Note that that latter quantity is negative, reflecting the "concave down" relation that we saw in Figure 7-4 and noted that it made intuitive sense.

Now, look at the R^2 and adjusted R^2 values, 0.8108 and 0.8084. That is a substantial increase from the purely linear model, and again there is no indication of overfitting.

7.8.3 Is It Really Still Linear?

In a word, Yes.

Look Equation (7.3). What if we were to multiply each of b, m_1 and m_2 by, say 3.2? Then the entire expression would go up by a factor of 3.2 as well. And *that* is the actual definition of "linear" in the term *linear model*. All of the mathematics, e.g. matrix multiplication, stems from that property.

7.8.4 General Polynomial Models, and the polyreg Package

So, why stop with that **temp** squared term? We can square the other terms too, and for that matter also include their products. And then cubic terms and so on.

Mathematically, one can show that this reduces bias. It does increase variance, so we must guard against overfitting (see next section), but it's worth a try. We can try first-degree models, then second-degree, then third and so on, and choose the best one via cross-validation.

But how do we actually do this? It was easy to add the **temp** squared term "by hand," but a little tricky to write code to do in general. There is one point in particular to worry about: dummy variables.

Consider the Air B&B data, Section 7.7.1. We had a number of dummies there, e.g. **zipcode94132**. By definition, it takes on the values 0 and 1. But $0^2 = 0$ and $1^2 = 1$, so there is no point in add squared terms to our model for any of the ZIP code dummies. In fact, we shouldn't add products of these variables either. E.g.

```
zipcode94132 * zipcode94158
```

will always be 0! Say a house is in ZIP code 94132. Then **zipcode94132** will be 1 but **zipcode94158** will be 0. If a property has ZIP code 94133, both elements of the above product will be 0, so again the product will be 0. So there is no point of adding a term `zipcode94132 * zipcode94158` to our model.

So there are a number of squares and products that we do NOT want to add into our model. Fortunately, the **polyreg** package handles all this for us.

7.8.4.1 Example: Bike Sharing Data

Let's use **polyreg** to fit a degree-2 model.

```
> pfout <- polyFit(d1,2)
```

Note that the first argument is our data, with *Y* assumed to be in the last column.

To illustrate prediction, let's look at a changed version of one of the data points:

```
> d8 <- d1[8,]
> d8
  yr mnth holiday weekday workingday   temp
8  0    1       0       6          0 -0.245
       hum windspeed tot
8 0.535833  17.87587 959
> d8$month <- 4  # what if in April?
> d8$temp <- 10  # what if 10 degrees C?
> predict(pfout,d8)
       1
2693.276  # predict ridership of almost 2700
```

But there is also a warning printed out:

```
Warning message:
In predict.lm(object$fit, plm.newdata) :
  prediction from a rank-deficient fit may be misleading
```

This is linear algebra talk for saying that one or more of the columns in *X* are very nearly a linear function of the others, termed *multicollnearity*. That says in turn that these particular columns are superfluous, so really we are overfitting. We should try deleting some columns.

7.8.5 Polynomials and Overfitting

Let's illustrate the overfitting problem on the dataset **prgeng**, assembled from the 2000 US census, which we first saw in Section 1.18.2. It consists of wage and other information on 20090 programmers and engineers in Silicon Valley. This dataset is included in **tegtools** package. We will also use the **polyreg** package, which fits polynomial models.

As usual, let's take a glance at the data:

```
> head(prgeng)
       age educ occ sex wageinc wkswrkd
1 50.30082   13 102   2   75000      52
2 41.10139    9 101   1   12300      20
3 24.67374    9 102   2   15400      52
4 50.19951   11 100   1       0      52
5 51.18112   11 100   2     160       1
6 57.70413   11 100   1       0       0
```

Note in this version of the dataset, education, occupation and sex are categorical variables, which R will convert to dummies for us. Here is the code:

```
library(regtools)
# polyreg does polynomial regression, forming the powers,
# cross products etc. and then calling lm()
library(polyreg)
data(prgeng)
pe <- prgeng[,c(1:4,6,5)]  # "Y" variable is assumed last column in polyFit()
tstidxs <- sample(1:nrow(prgeng),1000)
petrn <- pe[-tstidxs,]
petst <- pe[tstidxs,]
for (i in 1:4) {
   pfout <- polyFit(petrn,deg=i)
   preds <- predict(pfout,petst)
   print(mean(abs(preds-petst$wageinc)))
   print(length(pfout$fit$coefficients))
}
```

And below are the resulting Mean Absolute Prediction Errors and values of p, the number of features.

degre	MAPE	p
1	24248.43	24
2	23468.56	164
3	24367.16	496
4	818934.9	1020

Note that here p includes both the original features and the newly-created polynomial terms. When we have a degree 2 model, we have all the squared terms (except for the dummies), and the cross-product terms, e.g. interaction between age and gender. So p increases pretty rapidly with degree.

In any event, though, the effects of overfitting are clear. The degree-2 model seems to be an improvement, but after that performance deteriorates rapidly.

7.9 What about Statistical Signficance?

Look at the output pf **summary(lm())** above, reproduced here for convenience:

```
> summary(lm(tot ~ .,data=d1))
...
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  3074.246    240.924  12.760  < 2e-16
yr           2006.856     71.639  28.013  < 2e-16
mnth           94.178     10.887   8.651  < 2e-16
holiday      -589.426    220.864  -2.669  0.00778
```

```
weekday         53.332     17.850   2.988  0.00291
workingday      84.996     79.055   1.075  0.28267
temp           126.841      4.284  29.608  < 2e-16
hum          -2559.445    264.727  -9.668  < 2e-16
windspeed      -56.512      7.200  -7.848 1.52e-14

(Intercept) ***
yr          ***
mnth        ***
holiday     **
weekday     **
workingday
temp        ***
hum         ***
windspeed   ***
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘’. 0.1 ‘ ’ 1
...
Multiple R-squared:  0.7571, Adjusted R-squared:  0.7544
```

What do all those asterisks mean?

7.9.1 Basic Problem

Next to each feature are 0, 1, 2 or 3 asterisks, indicating “statistical significance.” In the past, it was common for statisticians to use this as a basis for dimension reduction: If a feature was “significant,” i.e. had at least one asterisk, it was retained in the model; otherwise, it was discarded.

In the example here, for instance, all features are flagged with an asterisk, except for **workingday**. According to this criterion, one would eliminate that feature.

Use of significance tests is frowned upon by many statisticians (including this author).[3]. The tests are especially unreliable in prediction applications. With large datasets, *every* feature will be declared “very higly significant” (three asterisks) regardless of whether the feature has substantial predictive power. A feature with a very small regression coefficient could be declared “significant,” in spite of being essentially useless as a predictor.

7.9.2 Standard Errors

You can see above that a *standard error* is reported for each estimated coefficient $\widehat{\beta}_i$. It’s the estimated standard deviation of $\widehat{\beta}_i$ over all possible samples for whatever population is being sampled. This gives us an idea as to how

3. https://www.amstat.org/asa/files/pdfs/P-ValueStatement.pdf

accurate $\widehat{\beta}_i$ is. An approximate 95% confidence interval for the true population β_i is

$$\widehat{\beta}_i \pm 1.96 \text{standard error}$$

Note again! If that confidence interval is close to 0 then this feature should probably not be included in our model — **even if the interval excludes 0.**

7.10 Classification Applications: the Logistic Model

The linear model is designed for regression applications. What about classification? A generalization of the linear model, unsurprisingly called the *generalized linear model*, that handles that.

Recall the discussion at the beginning of Chaper 2.1.1, which pointed out that in classification settings, where Y is either 1 or 0, the regression function becomes the probability of $Y = 1$ for the given subpopulation. If we fit a purely linear model with **lm()**, the estimated regression values may be outside the interval [0,1], thus not represent a probability. We could of course truncate any value predicted by **lm()** to [0,1], but the *logistic* model provides a better approach.

We still use a linear form, but run it through the logistic function $l(t) = 1/(1+e^{-t})$ to squeeze it into [0,1]. We still minimize a modified squared-error loss function, similar to that of (7.4), but now there are weights placed in front of each squared term. The computation now becomes iterative, with the weights changing from one iteraton to the next, and the process is called iteratively reweighted least squares (IRLS). Usually this is not a problem (and there is no learning rate etc.

7.10.1 Example: Telco Churn Data

In Section 2.2 we used k-NN to analyze some customer retention data. Let's revisit that data, now using a logistic model:

```
# data prep as before, not shown
> tc <- as.data.frame(tc)  # new
> idxs <- sample(1:nrow(tc),1000)
> trn <- tc[-idxs,]
> tst <- tc[idxs,]
> glout <- glm(Stay ~ .,data=trn,family=binomial)
```

The **glm()** syntax is the same as that of **lm()**, except for the additional argument, **family=binomial**, which specifies the logistic model.

Let's see how well we predict:

```
> preds <- predict(glout,tst,type='response')
Warning message:
In predict.lm(object, newdata, se.fit, scale = 1, type = if (type ==  :
  prediction from a rank-deficient fit may be misleading
> mean(round(preds) == tst$Stay)
```

```
[1] 0.798
```

This corresponds to a misclassification rate of 0.202, slightly better than the 0.210 rate, the best we obtained using k-NN.

By the way, the reader may ask if this is a fair comparison. Here we used cross validation, which we did not do in the k-NN example. So, wasn't the latter result overly optimistic? The answer is No. There we set **leave1out=TRUE**, which is n-fold cross validation.

Finally, what about that warning message? We saw this in Section 7.8.4.1, an indication of overfitting. This is also shown in the list of estimated coefficient:

```
> glout
...
Coefficients:
                          (Intercept)
                            0.5718798
                        gender.Female
                            0.0180493
                        SeniorCitizen
                           -0.2686345
                           Partner.No
                            0.0403715
                        Dependents.No
                           -0.0953884
                               tenure
                            0.0626553
                      PhoneService.No
                            1.1169514
                     MultipleLines.No
                            0.5329380
     `MultipleLines.No phone service`
                                   NA
                                   NA
                  InternetService.DSL
                           -4.1486092
        `InternetService.Fiber optic`
                           -6.4317277
                   OnlineSecurity.No
                           -0.0333799
   `OnlineSecurity.No internet service`
                                   NA
...
```

There just wasn't enough data to estimate some of the coefficients.

7.10.2 Multiclass Case

If there are more than two classes, we have two options. For concreteness, consider the vertebrae data, Section 2.3. There we had three classes, DH, NO and SL.

- **One vs. All (OVA) method:** Here we run **glm()** once for each class, i.e. first with *Y* being DH, then with *Y* being NO, finally with *Y* being SL. That gives use three sets of coefficients, in fact three return objects from **glm()**, say **DHout**, **NOout** and **SLout**. Then when predicting a new case **newx**, we run

  ```
  predict(DHout,newx,type='response')
  predict(NOout,newx,type='response')
  predict(SLout,newx,type='response')
  ```

  and take as our prediction whichever class has the highest probability.
- **All vs. All (AVA) method:** Here we run **glm()** once for each *pair* of classes. First, we restrict the data to just DH and NO, putting SL aside for the moment, and take *Y* as DH. Then we on just DH and SL, taking *Y* to be DH again. Finally, we put DH aside for the moment, running with *Y* to be NO. Again that would give us three objects output from **gm()**.
  Then we would call **predict()** three times on **newx**. Say the outcome with the first object is less than 0.5. That means between DH and NO, we would predict this new case to be NO, i.e. NO "wins." We do this on all three objects, and whichever class wins the most often is our predicted class. (Ties would have to be resolved in some way.)

The **regtools** package has tools to do all this for us. For OVA, the function **ovalogtrn()** and the associated **predict.ovalog()** are the ones to use.

7.10.2.1 Example: Fall Detection Data

This is another dataset from Kaggle.[4]. From the site:

> Four different fall trajectories (forward, backward, left and right), three normal activities (standing, walking and lying down) and near-fall situations are identified and detected.
> Falls are a serious public health problem and possibly life threatening for people in fall risk groups. We develop an automated fall detection system with wearable motion sensor units fitted to the subjects' body at six different positions.

There are six activity types, thus six classes, coded 0 Standing, 1 Walking, 2 Sitting, 3 Falling, 4 Cramps, 5 Running. Let's see how well we can predict:

4. https://www.kaggle.com/pitasr/falldata

```
> fd$ACTIVITY <- as.factor(fd$ACTIVITY)
> idxs <- sample(1:nrow(fd),1000)
> trn <- fd[-idxs,]
> tst <- fd[idxs,]
> ovout <- ovalogtrn(trn,'ACTIVITY')
> predsl <- predict(ovout,predpts=tst[,-7])
> mean(predsl == tst$ACTIVITY)
[1] 0.409
```

As before, we should make sure that this 41% accuracy figure is a genuine improvement over having no features. Proceeding as we did in Section 2.2.3, we have

```
> table(fd$ACTIVITY) / sum(table(fd$ACTIVITY))
         0          1          2          3          4          5
0.28128434 0.03064339 0.15272860 0.21902088 0.21328287 0.10303992
```

So, with no features, we would predict every case to be activity class 0, and we would have 28% accuracy. So, the features did help.

7.11 Bias and Variance in Linear/Generalized Linear Models

As discussed before, the more features we use, the smaller the bias but the larger the variance. With parametric models such as those in this chapter, the larger variance comes in the form of less-stable estimates of the coefficients, which in turn makes later predictions less stable.

7.11.1 Example: Bike Sharing Data

We can make that point about instability of predictions more concrete using the **regtools** function **stdErrPred()**. We'll fit two models, one with a smaller feature set and the other with a somewhat larger one, and then do the same prediction on each; just as an example, we'll predict the third data point. We'll print out the two predictions, and most importantly, print out the estimated standard errors of the two predictions.

```
> e1 <- day1[,c(4,10,12,13,16)]
> e2 <- day1[,c(4,10,12,13,16,6,7)]
> lm1 <- lm(tot ~ .,data=e1)
> lm2 <- lm(tot ~ .,data=e2)
> newx1 <- e1[3,-5]  # exclude tot
> newx2 <- e2[3,-5]  # exclude tot
> predict(lm1,newx1)
       3
1782.503
> predict(lm2,newx2)
       3
1681.662
> stdErrPred(lm1,newx1)
```

```
         [,1]
[1,] 93.45627
> stdErrPred(lm2,newx2)
         [,1]
[1,] 101.4724
```

So the prediction from the larger feature set has a larger standard error. Recall that the standard deviation is the square root of variance. So here we see the bias-variance tradeoff in action. The larger model, though more detailed and thus less biased, does have a larger variance.

Does that mean that we should use the smaller feature set? No. In order to see whether we've hit the switchover point yet, we'd need to use cross-validation. But the reader should keep this concrete illustration of the tradeoff in mind.

8 CUTTING THINGS DOWN TO SIZE: REGULARIZATION

A number of modern statistical methods "shrink" their classical counterparts. This is true for ML methods as well. In particular, the principle may be applied in:

- Boosting (covered already, Section 5.3.7).
- Linear models.
- Support vector machines.
- Neural networks.

In this chapter, we'll see why that may be advantageous, and apply to the linear model case.

8.1 Motivation

Suppose we have sample data on human height, weight and age. Denote the population means of these quantities by μ_{ht}, μ_{wt} and μ_{age}. We estimate them from our sample data as the corresponding sample means, $\overline{X}_{ht}$, $\overline{X}_{wt}$ and $\overline{X}_{age}$.

Just a bit more notation, giving names to vectors:

$$\mu = (\mu_{ht}, \mu_{wt}, \mu_{age}) \tag{8.1}$$

and

$$\overline{X} = (\overline{X}_{ht}, \overline{X}_{wt}, \overline{X}_{age}) \tag{8.2}$$

Amazingly, *James-Stein theory* says the best estimate of μ might NOT be $\overline{X}$. It might be a shrunken-down version of $\overline{X}$, say $0.9\overline{X}$, i.e.

$$(0.9\overline{X}_{ht}, 0.9\overline{X}_{wt}, 0.9\overline{X}_{age}) \tag{8.3}$$

And the higher the dimension (3 here), the more shrinking needs to be done.

The intuition is this: For many samples, there are a few data points that are extreme, on the fringes of the distribution. These points skew our estimators, in the direction of being too large. So, it is optimal to shrink them.

How much shrinking should be done? In practice this is unclear, and typically decided by our usual approach, cross-validation.

Putting aside the mathematical theory — it's quite deep — the implication for us in this book is that, for instance, the least-squares estimator $\widehat{\beta}$ of the population coefficient vector β in the linear model is often too large, and should be shrunken, and that **this is a possible remedy for overfitting**.

8.2 Size of a Vector

Is the vector, say, (15.2,3.0,-6.8) "large"? What do we mean by its size, anyway?

There are two main measures, called ℓ_1 and ℓ_2, and are denoted by the "norm" notation, $|| \; ||$, two pairs of vertical bars. For the example above, the ℓ_1 norm is

$$||(15.2, 3.0, 6.8)||_1 = |15.2| + |3.0| + |-6.8| = 25 \tag{8.4}$$

i.e. the sum of the absolute values of the vector elements. Here is the ℓ_2 case:

$$||(15.2, 3.0, 6.8)||_2 = \sqrt{15.2^2 + 3.0^2 + (-6.8)^2} = 16.9 \tag{8.5}$$

the square root of the sums of squares of the vector elements.

By the way, when we say "shrink" here, it may not take the form of simply multiplying by a constant such as 0.9. We'll use the term broadly, meaning simply that the vector norm is shrunken.

8.3 Ridge Regression and the LASSO

For years, James-Stein theory was mainly a mathematical curiosity, suitable for theoretical research but not affecting mainstream data analysis. There was some usage of *ridge regression*, to be introduced below, but even that was limited. The big change came from the development of the *LASSO* (Least

Absolute Shrinkage and Selection Operator), and its adoption by the ML community.

8.3.1 How They Work

Recall the basics of the least-square method for linear models: We choose $\widehat{\beta} = (\widehat{m}, \widehat{b})$ to minimize the sum of squared prediction errors, as in (7.4). For convenience, here is a copy of that expression:

$$[180 - (\widehat{b} + \widehat{m} \cdot 71)]^2 + [215 - (\widehat{b} + \widehat{m} \cdot 74)]^2 + ... + [195 - (\widehat{b} + \widehat{m} \cdot 73)]^2 \tag{8.6}$$

The idea of ridge regression was to “put a damper” on that, by adding a vector-size-limitation term. Instead of (8.6), we now minimize either

$$[180 - (\widehat{b} + \widehat{m} \cdot 71)]^2 + [215 - (\widehat{b} + \widehat{m} \cdot 74)]^2 + ... + [195 - (\widehat{b} + \widehat{m} \cdot 73)]^2 + \lambda||(\widehat{m}, \widehat{b})||_2^2 \tag{8.7}$$

(ridge case) or

$$[180 - (\widehat{b} + \widehat{m} \cdot 71)]^2 + [215 - (\widehat{b} + \widehat{m} \cdot 74)]^2 + ... + [195 - (\widehat{b} + \widehat{m} \cdot 73)]^2 + \lambda||(\widehat{m}, \widehat{b})||_1 \tag{8.8}$$

(LASSO case), where $\lambda \geq 0$ is a hyperparameter set by the user.

It can be shown that this is equivalent to minimizing (8.6), **subject to the constraint**

$$||(\widehat{m}, \widehat{b})||_2 \leq \eta \tag{8.9}$$

or

$$||(\widehat{m}, \widehat{b})||_1 \leq \eta \tag{8.10}$$

where $\eta > 0$ is a new hyperparameter replacing λ. In this alternate formulation, the goal is now intuitively clear:

> We choose our estimated coefficient vector to minimize the prediction sum of squares, BUT not allowing that vector to be larger than our specification η.

Again, the value η is a hyperparameter, as usual typically chosen by cross-validation.

8.3.2 The Bias/Variance Tradeoff, Avoiding Overfitting

As noted, a major reason that the regularization idea has had such an impact on statistics/ML is that it is a tool to avoid overfitting. Here are the issues:

- On the one hand, we want to make the prediction sum of squares as small as possible, thus reducing bias.

- On the other hand, a small value of that sum of squares may come with a large variance, due to the influence of extreme data points as discussed earlier.

Overfitting occurs when we are on the wrong side of the bias/variance tradeoff. That small sum of squares may be deceptive.

The key point: Shrinkage reduces variance, and if this can be done without increasing bias much, it's a win.

Again, regularization is used not only in the linear model, the case studied in this chapter, but also in SVM, neural nets and so on. It can even be used in PCA.

8.3.3 Relation between η, n and p

Again, the bias/variance tradeoff notion plays a central role here, with implications for dataset size. The larger n is, i.e. the larger the sample size, the smaller the variance in $\widehat{\beta}$ — which means the lesser the need to shrink.

In other words, for large datasets, we probably don't need regularization. We might still use it for dimension reduction, say to reduce computation time in further analyses of this data, though, as we'll see below in Section 8.5.

Note that "large n" here is meant both in absolute terms and relative to p, e.g. by the $\sqrt{n}$ criterion following (1.2) that guarantees variance will go to 0. In any event, the surest way to settle whether shrinkage is going in a particular setting is to try it.

8.3.4 Comparison, Ridge vs. LASSO

The advantage of ridge regression is that its calculation is simple: There is an explicit, closed-form solution, i.e. it is non-iterative; the LASSO requires iterative computation (though it does not have convergence problems).

But the success of the LASSO is due to its providing a *sparse* solution, meaning that usually most of the elements of $\widehat{\beta}$ are 0s. The idea is to then discard the features having $\widehat{\beta}_i = 0$. In other words:

The LASSO is a form of dimension reduction.

8.4 Software

We will use the **glmnet** package here, which allows the user to specify either ridge regression or LASSO, via the argument **alpha**, setting that value to 0 or 1, respectively; the default is LASSO. One can also set **alpha** to an intermediate value, combining the two approaches, termed the *elastic net*.

Input data arguments are **x** and **y** for the X and Y portions of the data. The former must be a matrix, not a data frame.

The **family** argument specifies what model to use, including **'gaussian'** for the linear model, **binomial** for the logistic and **multinomial** for categorical Y. In the latter two cases, **y** may be specified as an R factor.

The package has built-in cross-validation, invoked by calling **cv.glmnet()**. Arguments for **glmnet()**, e.g. **alpha**, may be specified in the **cv.glmnet()** call.

The algorithm will progressively increase λ, thereby dropping one feature at a time. Ordinarily, it will return just the results of the step with the minimum MSPE, but setting **keep = TRUE** instructs the function to retain all steps.

8.5 Example: NYC Taxi Data

Let's return to the New York City taxi data of Section 4.3. As noted, we must convert to dummies.

```
> yelld <- factorsToDummies(yell,TRUE)
> dim(yelld)
[1] 99723   476
```

Note the large number of columns, due mainly to dummy variables corresponding to pickup and dropoff locations.

```
> library(glmnet)
> cv.glout <-
  cv.glmnet(x=yelld[,-476],y=yelld[,476],family='gaussian',alpha=1,keep=TRUE)
> plot(cv.glout)
```

The package includes a generic **plot()** function, the result of which is shown in Figure 8-1. MSPE is on the vertical axis. The length-restraining hyperparameter λ is shown on the bottom horizontal axis (on a $\log_2$ scale). The top border shows the number of nonzero $\widehat{\beta}_i$, i.e. the number of features retained by the algorithm; the larger λ is, the more emphasis on short $\widehat{\beta}$, thus the more 0s.

The striking aspect of the graph is that it is very flat at the beginning. Even for 113 features, MSPE is still near the minimum. Since the full set consists of 476, this dramatic reduction in dimension — possibly enabling major savings in computation time for further analysis of this data — may well be worth the small sacrifice in MSPE (which, after all, is subject to sampling variation anyway[1]).

Well, then, which features make up that set of 113? This takes a bit of work to ferret out:

```
> cv.glout$nzero
 s0  s1  s2  s3  s4  s5  s6  s7  s8  s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19
  0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
s20 s21 s22 s23 s24 s25 s26 s27 s28 s29 s30 s31 s32 s33 s34 s35 s36 s37 s38 s39
  1   1   1   1   1   1   1   1   2   2   2   2   4   4   4   5   6   7   9  12
s40 s41 s42 s43 s44 s45 s46 s47 s48 s49 s50 s51 s52 s53 s54 s55 s56 s57 s58 s59
 19  24  31  36  43  50  55  60  69  85 101 113 133 152 166 188 208 223 238 257
```

1. Standard errors of the MSPE values are also available in the return object from **cvglmnet()**.

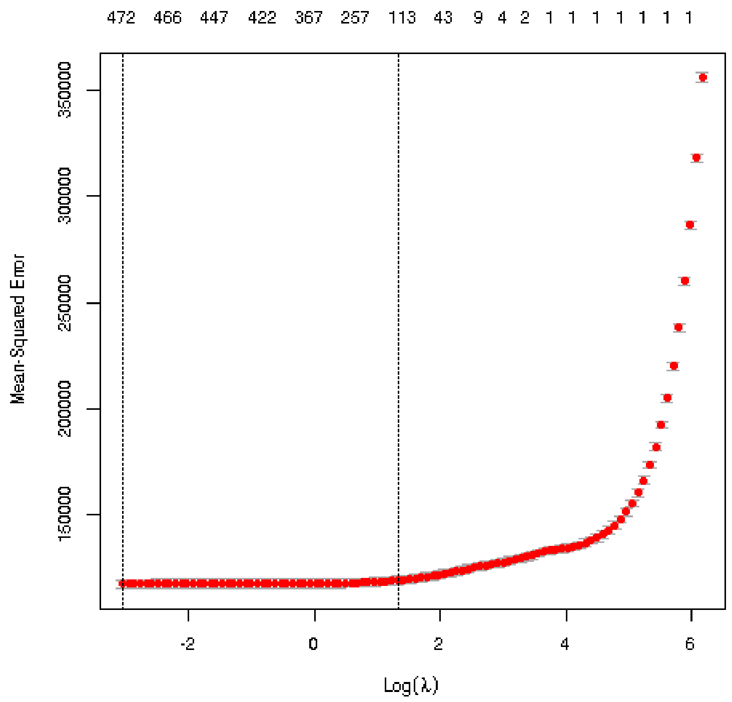

Figure 8-1: MSPE, taxi data

```
s60 s61 s62 s63 s64 s65 s66 s67 s68 s69 s70 s71 s72 s73 s74 s75 s76 s77 s78 s79
273 288 303 316 330 343 354 367 372 384 392 400 406 416 418 422 427 432 439 441
s80 s81 s82 s83 s84 s85 s86 s87 s88 s89 s90 s91 s92 s93 s94 s95 s96 s97 s98 s99
445 447 449 447 454 456 460 459 462 462 464 466 470 469 470 468 471 471 470 472
```

Ah, the step of interest is the one labeled 's51', which is the 52nd one. So we need to obtain $\widehat{\beta}$ for that particular step. As noted, it's rather buried:

```
> glout.m <- as.matrix(cv.glout$glmnet.fit$beta)  # beta-hats for all steps
> bh52 <- glout.m[,52]  # beta-hat for Step 52
> names(bh52[bh52 != 0])
  [1] "trip_distance"    "PULocationID.7"   "PULocationID.16"
  [4] "PULocationID.35"  "PULocationID.43"  "PULocationID.55"
  [7] "PULocationID.60"  "PULocationID.68"  "PULocationID.72"
...
[109] "DOLocationID.259" "DOLocationID.261" "DOLocationID.262"
[112] "DOLocationID.263" "PUweekday"
```

Not surprisingly, the trip distance feature was retained, along with various key locations, and a dummy for whether the trip occurred on a weekday.

The $\widehat{\beta}$ vector is in **bh52**.

8.6 Example: Air B&B Data

Let's revisit the dataset analyzed in Section 7.7.1.

```
> Abb$square_feet <- NULL
> Abb$weekly_price <- NULL
> abbd <- factorsToDummies(Abb,TRUE)
> abbd <- na.exclude(abbd)
> bbout <- cv.glmnet(abbd[,-32],abbd[,32],family='gaussian')
> plot(bbout)
```

The plot here looks much different from the one in the last section. There the LASSO brought us possible saving in computation time, but here we also attain a large reduction in MSPE. (Standard errors are also shown, in the vertical bars extending above and below the curve.)

And of course there is a generic **predict()** function. To try it out, let's take row 18 from our data, and change the security deposit to \$360 and the rating to 92:

```
> x18 <- abbd[18,-32]
> x18[36] <- 92
> x18[32] <- 360
# newx must be a matrix, even if only 1 row
> x18 <- matrix(x18,nrow=1)
> predict(bbout,newx=x18,s='lambda.min')
            1
[1,] 3586.993
```

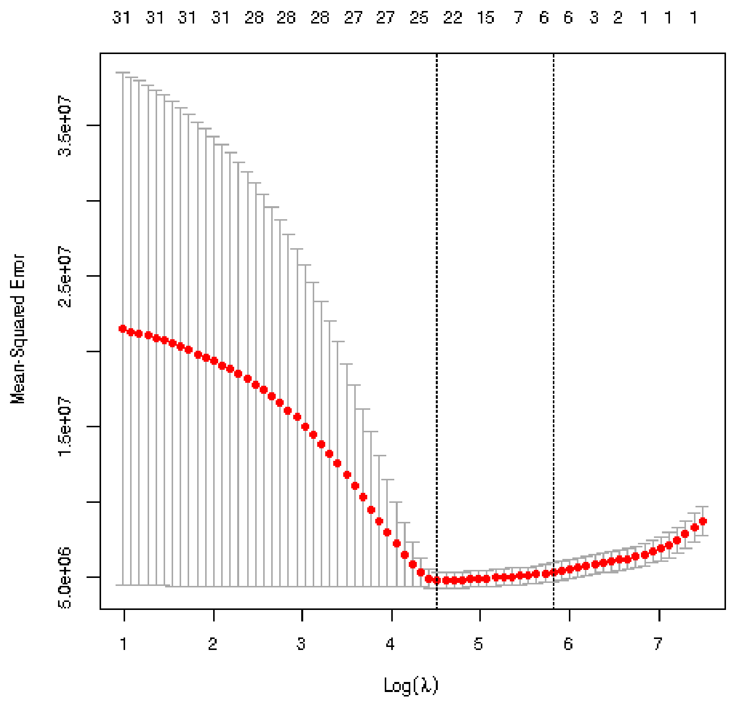

Figure 8-2: MSPE, Air B&B data

8.7 Example: African Soil Data

As noted in Section 5.2.3.1, the importance of this dataset is that it has $p > n$, with almost triple the number of features, considered a very difficult situation, especially for linear models.

Remember, to many analysts, the very essence of LASSO is dimension reduction. So it will be very interesting to see how LASSO does on this data.

8.7.1 Analysis

```
> glout <- cv.glmnet(x=as.matrix(afrsoil[,1:3578]),y=afrsoil[,3597],
   family='gaussian',alpha=1,keep=TRUE)
glout

Call:  cv.glmnet(x = as.matrix(afrsoil[, 1:3578]), y = afrsoil[, 3597],      keep = TRUE, 

Measure: Mean-Squared Error

      Lambda Measure      SE Nonzero
min 0.003426  0.1996 0.01416     156
1se 0.005716  0.2128 0.01385      77
```

So the minimum MSPE came at $\lambda = 0.003426$, which resulted in 156 features retained out of the original 3578. That's quite a dimension reduction, but look at the plot is shown in Figure 8-3 — the curve was still declining as λ decreased. Maybe we actually need to retain more features. We could re-run the analysis with our own custom value of λ to investigate.

8.8 Optional Section: the Famous LASSO Picture

As mentioned, a key property of the LASSO is that it usually provides a *sparse* solution for $\widehat{\beta}$, meaning the many of the $\widehat{\beta}_i$ are 0. In other words, many features are discarded, thus providing a means of dimension reduction. Figure 8-4 shows why. Here is how it works.

The figure is for the case of $p = 2$ predictors, whose coefficients are b_1 and b_2. (For simplicity, we assume there is no constant term β_0.) Let U and V denote the corresponding features. Write $b = (b_1, b_2)$ for the vector of the b_i.

Without shrinkage, we would choose b to minimize the sum of squared errors,

$$SSE = (Y_1 - b_1 U_1 - b_2 V_1)^2 + \ldots + (Y_n - b_1 U_n - b_2 V_n)^2 \qquad (8.11)$$

Recalling that the non-shrunken b is called the Ordinary Least Squares estimator, let's name that b_{OLS}, and name the corresponding SSE value SSE_{OLS}.

The horizontal and vertical axes are for b_1 and b_2, as shown. There is one ellipse for each possible value of SSE. For SSE equal to, say 16.8, the cor-

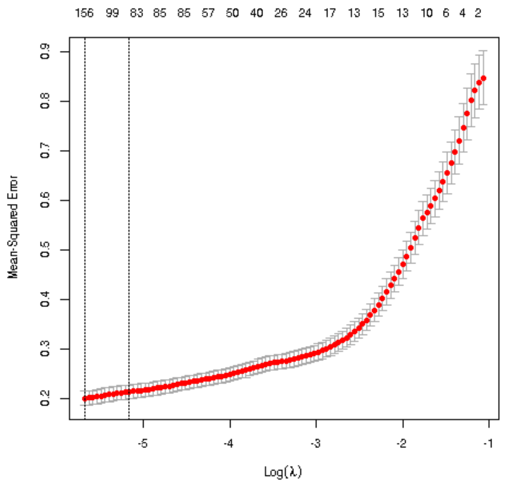

Figure 8-3: African Soil Data

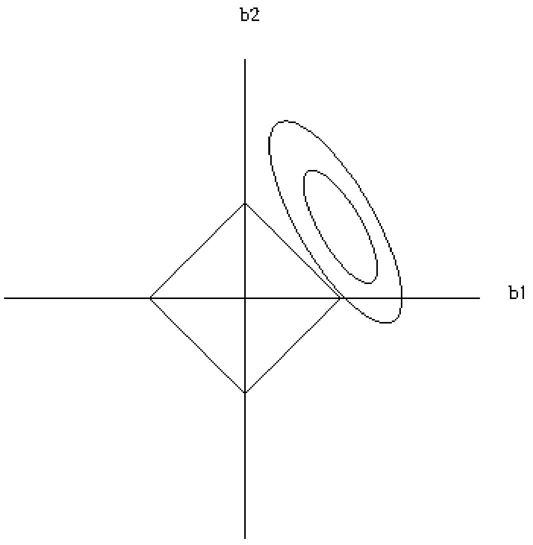

Figure 8-4: Feature subsetting nature of the LASSO

responding ellipse is the set of all points b that have SSE = 16.8. As we vary the SSE value, we get various concentric ellipses, two of which are shown in the picture. The smaller the ellipse, the smaller the value of SSE.

The OLS estimate is unique, so the ellipse for SSE = SSE_{OLS} is degenerate, consisting of the single point b_{OLS}.

The corners of the diamond are at $(\eta, 0)$, $(0, \eta)$ and so on. Due to the constraint (8.10), our LASSO estimator b_{LASSO} must be somewhere within the diamond.

Remember, we want SSE — the sum of square prediction errors — to be as small as possible, but at the same time we must stay within the diamond. So the solution is to choose our ellipse to just barely touch the diamond, as we see with the outer ellipse in the picture. The touchpoint is then our LASSO solution, b_{LASSO}.

But that touchpoint is sparse — $b2 = 0$! And you can see that, for almost any orientation and position of the ellipses, the eventual touchpoint will be one of the corners of the diamond, thus a sparse solution.

Thus the LASSO will usually be sparse, which is the major reason for its popularity. An exception is a situation in which b_{OLS} is itself within the diamond,

And what about ridge regression? In that case, the diamond becomes a circle, so there is no sparseness property.

9

THE VERDICT: PARAMETRIC MODELS

The linear and generalized linear models are in extremely wide usage in statistical contexts, and are definitely considered useful, if not central by ML specialists. Here are the pros and cons:

Pros:

- Thoroughly established, long track record, familiar.
- Makes good use of monotonic relations between *Y* and the features. And in polynomial form, can handle nonmonotonic settings.
- Computationally much faster than other ML methods.
- No local minima for **lm()**, unique solution, no convergence issues.

Cons:

- Polynomial form much less generally understood.
- Polynomial form can lead to multicollinearity (Section 7.8.4.1), and more extreme predictions near the edges of the data.

The reader should note, though, that that last bullet will be seen to also apply to the more central ML methods coming up in the next two chapters.

10

A BOUNDARY APPROACH: SUPPORT VECTOR MACHINES

Support vector machines (SVM), together with *neural networks* (NNs) are arguably the two most "purist" of ML methods, motivated originally by Artificial Intelligence, i.e. nonstatistical concepts. We'll cover SVMs in this chapter, and NNs in the next. SVM is best known for classification applications. It can be used in regression settings as well, but we will focus on classification.

10.1 Motivation

Everything about SVM is centered around boundary lines separating one class from another. To motivate that, we will first do a boundary analysis of the logistic model.

Let's use the Pima diabetes dataset, mentioned briefly in Section 1.16. This kind of data is helpful, not only in indentifying risk factors that contribute to the disease, but more directly related to this prediction-oriented book, predicting whether a patient may develop the disease. One way to access this data is through the **mlbench** package, which has two versions; we'll use the one in which spurious 0s have been replaced by NAs.

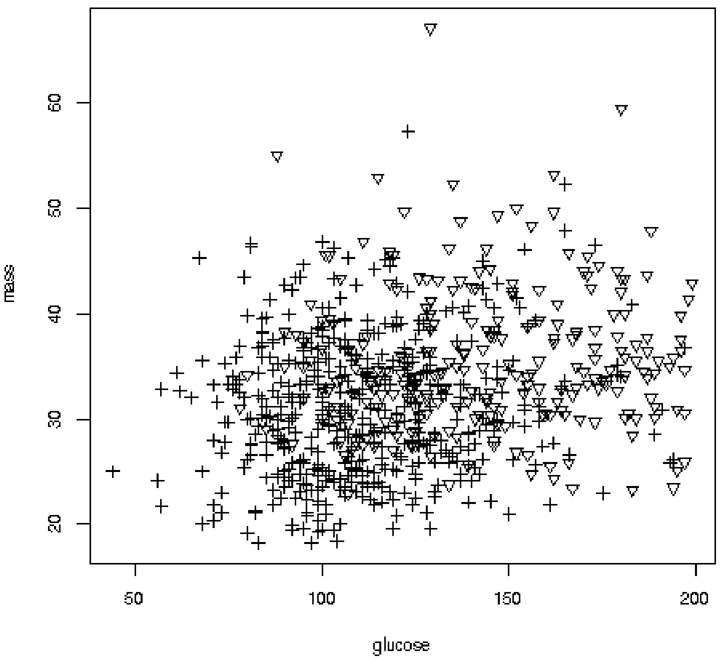

Figure 10-1: Diabetes data, BMI vs. glucose

Ultimately our goal will be a two-dimensional graph, so we will restrict this analysis to just two features, Glucose and Body Mass Index. Let's plot the data, using different symbols for the diabetics (triangles) and nondiabetics (pluses). The code

```
> library(mlbench)
> data(PimaIndiansDiabetes2)
> db <- PimaIndiansDiabetes2
> plot(db[,c(2,6)],pch=3*as.integer(db[,9]))
```

produces Figure 10-1. There no sharp tendency there, but it does look like the diabetics (triangles) have some tendency to be further up and more to the right, whereas the nondiabetics (pluses) are often lower and more to the left.

(The R **pch** argument ("point character") specifies the type of symbol to use for the two classes. I chose the triangle and plus sign, which have codes 0 and 3, hence the expression `3*as.integer(db[,9])`.)

Let's fit a logistic model:

```
> library(mlbench)
> data(PimaIndiansDiabetes2)
> db <- PimaIndiansDiabetes2
> db <- na.exclude(db)  # keep only cases with no NAs
> db1 <- db[,c(2,6,9)]
>
> head(db1)
   glucose mass diabetes
4       89 28.1      neg
5      137 43.1      pos
7       78 31.0      pos
9      197 30.5      pos
14     189 30.1      pos
15     166 25.8      pos
> glout <- glm(diabetes ~ .,data=db1,family=binomial)
> cf <- coef(glout)
> cf
(Intercept)     glucose        mass
-8.19752404  0.03808944  0.08477096
```

(The **coef()** function is yet another *generic* function, like the ones we've seen before such as **print()** and **plot()**. As can be seen, it extracts the estimated regression coefficients.)

Now recall that the logistic model routes the linear model into the logistic function, $\ell(t) = 1/(1+e^{-t})$; the placeholder t is set to the linear form. Denoting glucose and BMI by g and b, the above output says

$$\text{estimated } P(Y = 1 \mid g, b) = \frac{1}{1 + e^{8.19752404 - 0.03808944\, g\, -\, 0.08477096\, b}} \tag{10.1}$$

If our loss criterion is simply our overall misclassification rate, then we guess the person to be diabetic or not, depending on whether the estimated probability (10.1) is greater than 0.5 or not. But saying that quantity is greater than 0.5 is the same as saying that

$$8.19752404 - 0.03808944g - 0.08477096b < 0 \tag{10.2}$$

which in turn is equivalent to

$$b > \frac{8.19752404}{0.08477096} - \frac{0.03808944}{0.08477096} \cdot g \tag{10.3}$$

So the line forming the boundary between predicting $Y = 1$ and $Y = 0$, i.e. the line corresponding to probability 0.5, is

$$b = \frac{8.19752404}{0.08477096} - \frac{0.03808944}{0.08477096} \cdot g \tag{10.4}$$

Now let's superimpose that line onto our plot, using R's **abline()** function, which plays exactly the role implied by the name, i.e. adding a line to an existing plot:

```
> abline(a=-cf[1]/cf[3], b=-cf[2]/cf[3])
```

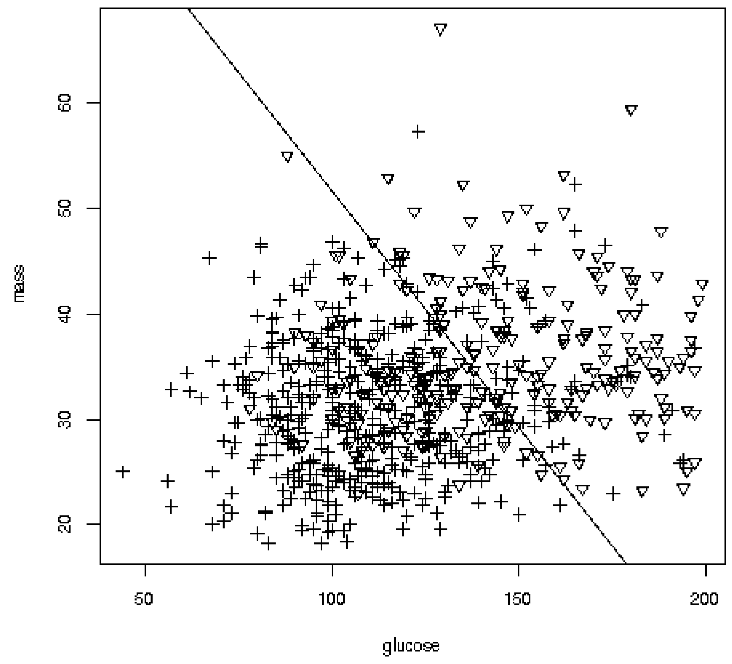

Figure 10-2: Diabetes data, BMI vs. glucose, with Logistic Boundary Line

The result is shown in Figure 10-2. Data points to the right of the line are predicted to be diabetics, with a nondiabetic prediction for those to the left.

Clearly there are quite a few data points that are misclassified (though remember, we are only using two of the features here).

10.1.1 Lines, Planes and Hyperplanes

The derivation above always holds with a logistic model: If we have $p = 2$, i.e. two features such as g and b, the boundary between predicting $Y = 1$ and $Y = 0$ is a straight line, of the form

$$c_1 g + c_2 b = c_3 \tag{10.5}$$

If $p = 3$, say adding age a, the boundary takes the form of a plane, with form

$$c_1 g + c_2 b + c_3 a = c_4 \tag{10.6}$$

Now hard to visualize, though Figure 10-3 may give you some idea.

For $p > 3$ — and typically we do have more than 3 features — we can't visualize the setting at all. But the structures behave mathematically like

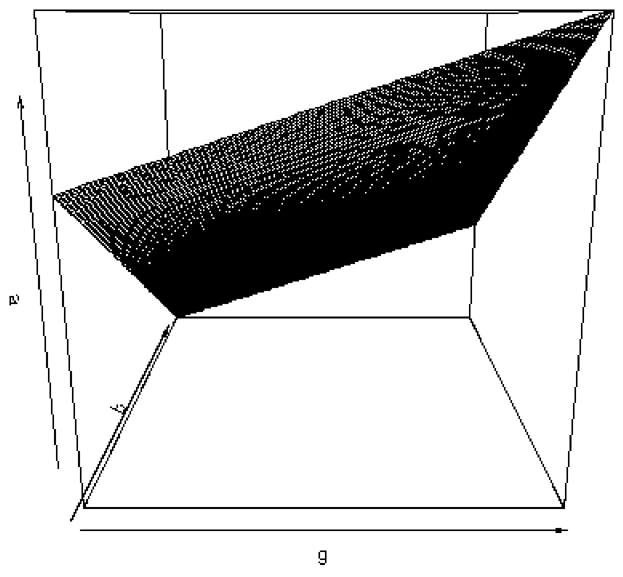

Figure 10-3: 3-D plane, glucose, BMI, age example

planes — they are called *hyperplanes* — and it is helpful to imagine them as planes.

Let's stick to the above $p = 2$ example to continue to motivate our intuition that will lead to SVM. For technical accuracy, we'll use the acronym LPH, Line/Plane/Hyperplane.

10.1.2 Why Not a Boundary Curve Instead of a Line?

The question to ask now is, What if the actual boundary is not a line but a curve? After all, the logistic model is just that, a model, and the boundary separating predicting $Y = 1$ and $Y = 0$ could be modeled to be some kind of curve.

In fact, we can view this in terms of polynomial regression. To see this, first note again that Equation (10.1) is, on the population level,

$$\text{probability(diabetes)} = \frac{1}{1 + e^{\beta_0 \cdot + \beta_1 g + \beta_2 b}} \tag{10.7}$$

Again, the β_i are population values. We have sample estimates, e.g. $\widehat{\beta_0} = 8.19752404$.

To make this a degree-2 polynomial model, we add terms g^2, b^2 and gb:

$$\text{probability(diabetes)} = \frac{1}{1 + e^{\beta_0 + \beta_1 g + \beta_2 m + \beta_3 g^2 + \beta_4 m^2 + \beta_5 gm}} \tag{10.8}$$

Then to fit this model, we could add columns to **db1**, e.g.

```
db1$g2 <- db1$g^2
```

and then call **glm()** as before.

Adding columns to **db1** by hand like this would be a little tedious, even for degree 2; imagine degree 3, which would involve adding a lot more terms. And though we have no dummy variables here, if we did, we would have to avoid forming their powers, as noted earlier in Section 7.8.4. So, it's much easier to use the **polyreg** package.

```
library(polyreg)
pfout <- polyFit(db1,2,use='glm')
```

The argument 2 specifies degree 2. We did not specify the **use** argument back in Section 7.8.4, as the default is 'lm', just what we wanted then, but now we specify 'glm'.

We could later call **predict()** with **pfout**, but instead let's draw the new boundary line — a curve now — updating Figure 10-1. Finding the equation of the curve would be rather involved, and it is easier to use the function **boundaryplot()** from **regtools** to give us a rough idea of the curve:

```
> boundaryplot(db1[,3],db1[,-3],pfout$fit$fitted.values)
```

The resulting plot is in Figure 10-4. The red, filled circles here represent data points for which the estimated regression value is *close to* 0.5. In that

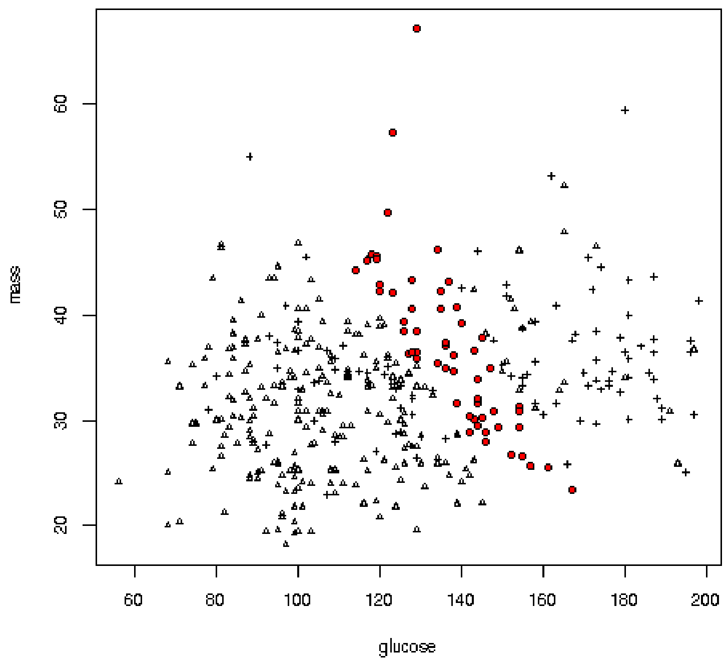

Figure 10-4: Diabetes data, BMI vs. glucose, with quadratic boundary line

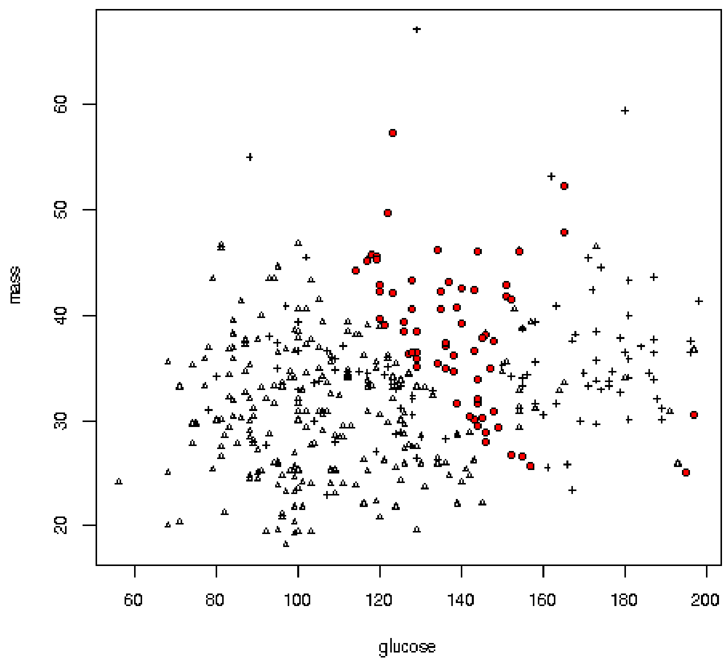

Figure 10-5: Diabetes data, BMI vs. glucose, with Degree 5 Boundary Line

manner, we see an approximation to the curve resulting from setting Equation (10.8) *equal to* 0.5. So we see that the boundary is in fact curved, moving somewhat to the right as it travels upward.

Keep in mind, in our new model, Equation (10.8), the curve represents the equation

$$\beta_0 + \beta_1 g + \beta_2 b + \beta_3 g^2 + \beta_4 b^2 + \beta_5 gb = 0 \qquad (10.9)$$

For various values of the β_i, this curve can be a parabola, an ellipse, a hyperbola — or a straight line. So it includes the straight-line case ($\beta_3 = \beta_4 = \beta_5 = 0$) but allows for many other possibilities. The point is that we now have a much more versatile model than our original one, thus reducing bias.

We might reduce bias even further with degree 3, 4 and so on. But as you may have guessed, this increases variance. In Equation (10.8), for instance, we have 7 parameters to estimate, in contrast to only 2 in Equation (10.7). As we saw in Section 7.11.1, the more parameters, the larger the variance.

One can see this in our present data. Let's skip up to degree 5, shown in Figure 10-5. Those red, filled circles don't seem to hug a curve anymore. We've overfit.

One more point to keep in mind: In (10.8) we basically have $p = 5$ features, with the new ones being g^2, b^2 and gb. As explained in Section 7.8.3, we stlll have a linear model, and the boundary is a 5-dimensional hyperplane. But viewed back in the original setting, where $p = 2$, the boundary is a curve.

10.2 SVM: the Basic Ideas

As mentioned, the central idea of SVM is to estimate the boundary LPH between one class and another. For multiple classes, we will use the OVA or AVA methods from Section 7.10.2, applying the two-class case multiple times. So our discussion here will be mainly for two classes.

10.2.1 Vector Notation

As can be seen from Equations (10.5) and (10.6), an LPH can be represented by its coefficient vector, e.g. (c_1, c_2, c_3, c_4) in (10.6). It's customary in SVM to separate out the constant on the right side, as follows: We rewrite (10.6) as

$$c_1 g + c_2 b + c_3 a - c_4 = 0 \tag{10.10}$$

and set

$$w = (c_1, c_2, c_3,), \quad w_0 = -c_4 \tag{10.11}$$

The vector w and the number w_0 are our description of the LPH.

10.2.2 Optimizing Criterion

Note that the LPH that SVM gives us will be different from than the one the logistic model gives us, because we now will have a different loss function, or more accurately, a different quantity to optimize:

The optimizing criterion used by SVM is quite different. Instead of, say, minimizing a sum of squares, we maximize a distance. That distance, called the *margin*, is the distance from the boundary line to the points in either class that are closest to the line. (Those points are the "support vectors.") SVM then consists of maximizing the margin.

The key idea: A large margin means that the two classes are well separated, which hopefully means that new cases in the future will be correctly classified.

10.2.3 Example: the Famous Iris Dataset

Edgar Anderson's data on Iris flowers, included in R, has been the subject of countless examples in books, Web sites and so. There are three classes, *setosa*, *versicolor* and *virginica*. It turns out that the setosa data points are linearly separable from the other two classes, so this is a good place to start. So, our two classes will be setosa and non-setosa. As before, in order to graph things in two dimensions, let's use only two of the features, say sepal length and petal length:

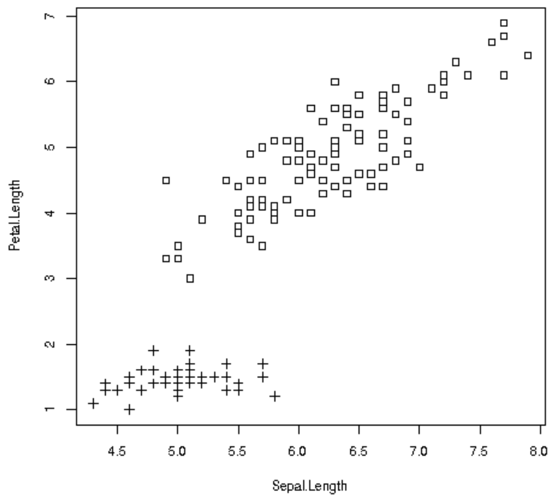

Figure 10-6: Iris data

```
> i2 <- iris[,c(1,3,5)]  # sepal length, petal length, class
> i2[,3] <- as.integer(i2[,3] == 'setosa')  # 1 for setosa, 0 for non-setosa
> head(i2)
  Sepal.Length Petal.Length Species
1          5.1          1.4       1
2          4.9          1.4       1
3          4.7          1.3       1
4          4.6          1.5       1
5          5.0          1.4       1
6          5.4          1.7       1
```

Plot the data:

```
> plot(i2[,1:2],pch=3*i2[,3])
```

The result is shown in Figure 10-6.

Clearly the data are linearly separable — we can draw a straight line completely separating the setosas (pluses) from the non-setosas (squares). In fact, we can draw lots and lots of such lines.

Which line should we use?

10.2.4 A Geometric Look: Convex Hulls

The mathematics underlying SVM is too arcane for an avowedly nonmathematical book like this one. But the process can be explained simply via geometry, using the notion of a *convex hull*. Start with this simple dataset:

```
> set.seed(9999)
> x <- matrix(runif(12),ncol=2)
> x
          [,1]       [,2]
[1,] 0.8608396 0.21975643
[2,] 0.6602416 0.81819992
[3,] 0.8004162 0.99815678
[4,] 0.2076460 0.74069639
[5,] 0.6894843 0.81829430
[6,] 0.8448705 0.02551977
> plot(x)
```

The picture is in Figure 10-7.

The idea of a convex hull is to imagine wrapping a piece string tautly around those points. Some points will end up inside the string, while others will be touching the string (see below).

Convex hulls can be computed in any dimension,[1] an important point, for general number of features *p*. We are in dimension 2 here. One way to find convex hulls is via the **chull()** function in the **mvtnorm** package:

```
> library(mvtnorm)
> chx <- chull(x)
> chx  # indices of rows of x
[1] 1 6 4 3
> lines(x[c(chx,chx[1]),])  # need to add chx[1] for full circuit
> points(x[chx,],pch=21,bg='red')
```

So the hull consists of rows 1, 6, 4 and 3 from our dataset. See Figure 10-8. Four of the six data points are now highlighted in red, filled circles, and are known as the *extreme points* of the hull. The remaining two are in the interior of the hull. The tautly-wrapped string is seen as the lines on the edges of the hull.

Now let's find and draw the convex hulls of the setosa and non-setosa data points in Figure 10-6:

```
> setosIdxs <- which(i2[,3] == 1)
> setos <- i2[setosIdxs,]
> nonsetos <- i2[-setosIdxs,]
> chset <- chull(setos[,1:2])
> points(setos[chset,],cex=1.2,pch=21,bg='red')
> lines(setos[c(chset,chset[1]),])
> chnonset <- chull(nonsetos[,1:2])
> points(nonsetos[chnonset,],cex=1.2,pch=22,bg='blue')
> lines(nonsetos[c(chnonset,chnonset[1]),])
```

See Figure 10-9.

By inspection, the closest pairs of points between setosas and non-setosas are (5.1,1.9) and (5.1,3.0), and (4.8,1.9) and (5.1,3.0).

1. Some graphics packages compute only the two-dimensional case.

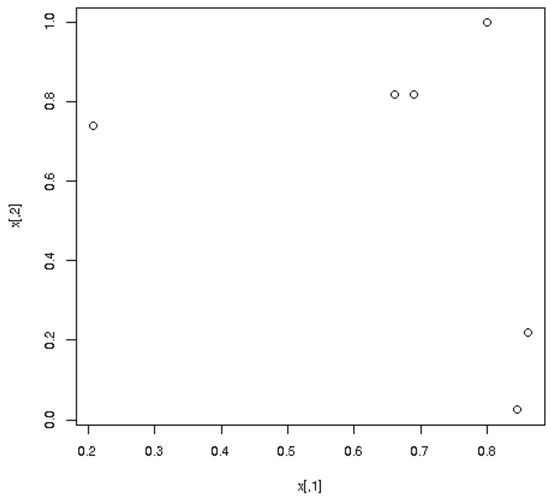

Figure 10-7: Toy example

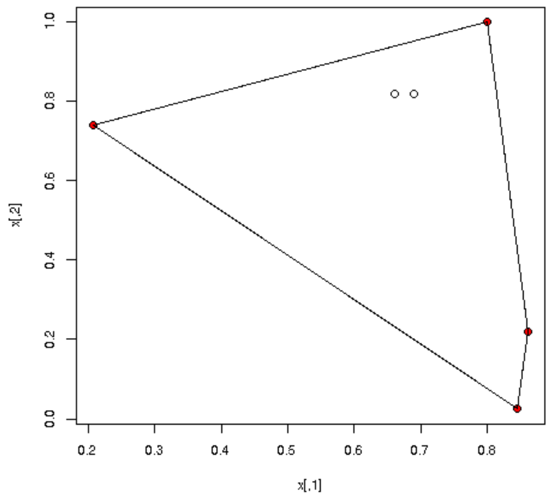

Figure 10-8: Toy example, with convex hull

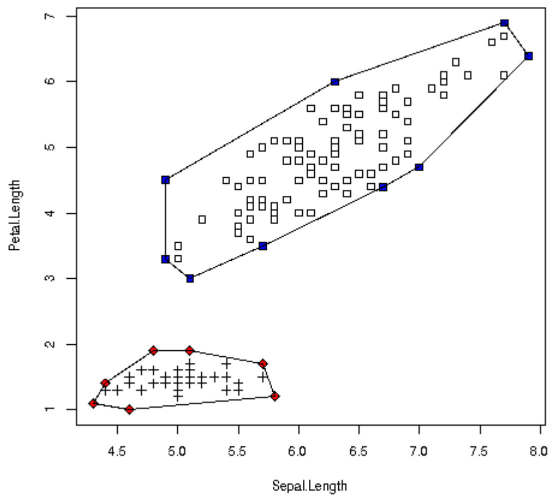

Figure 10-9: Iris data, 2 convex hulls

The SVM boundary itself is then derived as the perpendicular bisector of the shortest line between the two convex sets. That happens here to be horizontal at height 1.9 + 0.55 = 2.45. We add that boundary line (solid) and lines showing the margin (dashed) to the plot; see Figure 10-10.

```
> abline(h=2.45)
> abline(h=3.00,lty='dashed')
> abline(h=1.90,lty='dashed')
```

We omit the details of how to do this in general, especially for $p > 2$. The above derivation is simply meant to provide intuition. The SVM software takes care of it all for us (though with a different but equivalent approach).

So, to predict a new case, we guess setosa if petal length is less than 2.45, non-setosa otherwise.

The three points, (5.1,1.9), (4.8,1.9) and (5.1,3.0), are the *support vectors*. They "support" the fit, in the sense that if any other point is removed, the boundary will not change.

Note again, that:

- It is just a coincidence that the SVM boundary here turned out to be horizontal.
- To enable drawing the convex hulls etc., we needed to stick to $p = 2$, and used just sepal length and petal length as features. In actual analysis, we would use all the features at first, possibly doing some dimension reduction later.

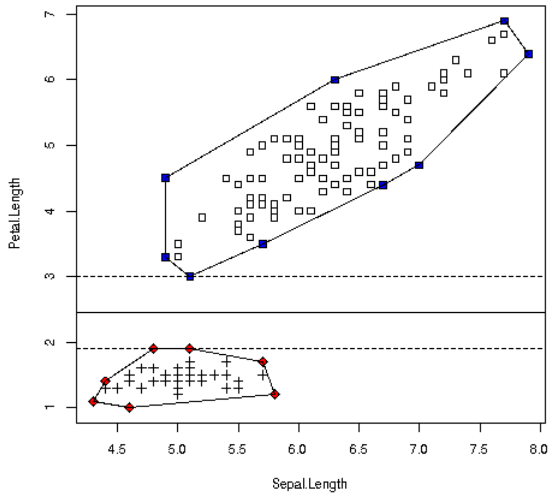

Figure 10-10: Iris, with SVM boundary and margin

10.3 Sample vs. Population

In finding the best LPH, keep in mind that this is a sample quantity, i.e. it is based on our data, which we consider a sample from some population. In the Pima example, we can think of the population of all women of that ethnicity.[2] As always, this consideration is crucial in discussing the bias/-variance tradeoff, as the variance refers to variation from one sample to another.

It will be helpful to think of the population LPH, with coefficient vector ω and constant term ω_0. The coefficient vector for the LPH we find from our data, w and w_0, estimate the population quantities ω and ω_0.

10.4 "Dot Product" Formulation

The underlying theory of SVM makes heavy use of calculus and linear algebra. As noted, such mathematics is beyond the scope of this book. However, it will be useful and easy to use some notation from that subject. Just notation, nothing conceptual.

The *dot product* between two vectors $u = (u_1, ..., u_m)$ and $v = (v_1, ..., v_m)$ is simply

$$u \bullet v = u_1 v_1 + ... + u_m v_m \tag{10.12}$$

Let's go back to our Pima data example. Say we use the features glucose, BMI and age, and that our linear boundary turned to be

2. The Pima are a tribe among the US indigenous peoples.

$$2.3g + 8.9b + 0.12a = 0.24 \qquad (10.13)$$

so that the algorithm says we guess diabetic if $2.3g + 8.9b + 0.12a > 0.24$ and guess non diabetic if $2.3g + 8.9b + 0.12a < 0.24$.

We would write the boundary in dot product form as saying we guess diabetic if

$$(-0.24, 2.3, 8.9, 5.5) \bullet (1, g, b, a) > 0 \qquad (10.14)$$

and nondiabetic if that quantity is less than or greater than 0. Note that the -0.24 is similar to the intercept term in linear models.

What's nice about this formulation (10.14) is that we know which way to guess, diabetic or nondiabetic, by simply noting whether a given quantity is negative or positive, a key point below..

For similar reasons, SVM theorists also like to code the two classes as $Y = +1$ and $Y = -1$, rather than 1 or 0 as is standard in statistics. We'll continue to use the latter in general, but will turn to the former in the next couple of sections.

10.5 Major Problem: Lack of Linear Separability

The margin, actually known as the *hard margin*, is only defined in situations in which some line (or LPH) exists that cleanly separates the data points of the two classes. As can be seen by the above graphs for the Pima data, in almost all practical situations, no such line exists. Even with the Iris data, no line separates the versicolor and virginica:

```
> plot(iris[,c(1,3)],pch=as.numeric(as.factor(iris$Species)))
```

See Figure 10-11.

There are two solutions to that problem:

- **Soft margin:** Here we allow a few — well, maybe quite a few — exceptions to the goal of having a line cleanly separate the data points of the two classes. The optimizing criterion now still consists of maximizing the distance separating two classes, but now:
  - We allow some points to lie within the margin.
  - But we impose a penalty for each exceptional point.

  It's analogous to the LASSO, where we minimize a sum of squares subject to a constraint. (More on this coming below.)
  The reader may already sense from this, correctly, that there will be a bias-variance tradeoff involved.
- **Applying a "kernel":** Here we transform the data, say by applying a polynomial function as we did above with the logistic model, and then finding an LPH separator on the new data.
  Again there will turn out to be a bias-variance tradeoff.

Typically a combination of these two approaches is used. For instance, after doing a transformation, we still may, in fact probably will, find that no

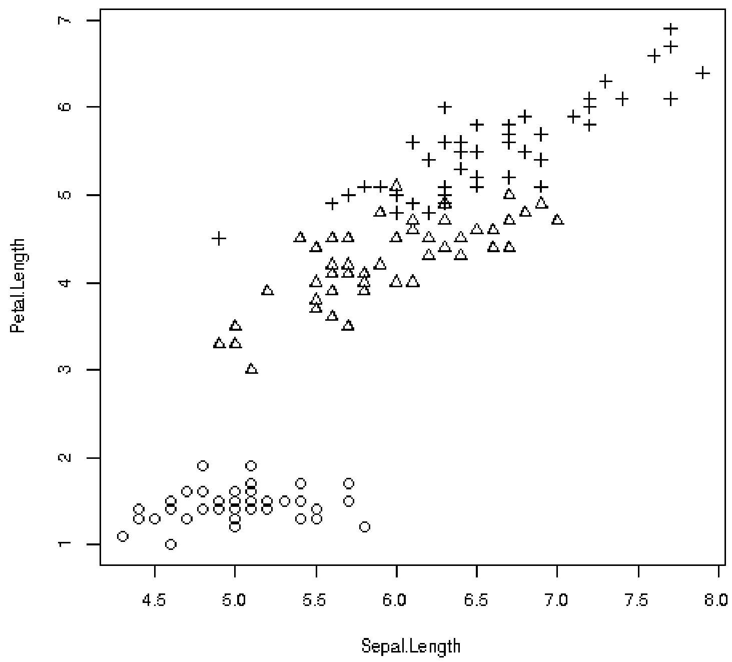

Figure 10-11: Iris data, 3 classes

cleanly-separating LPH exists, and thus will need to resort to allowing some exceptional points to lie within the margin.

10.6 Soft Margin

Here a hyperparameter is defined, termed the *cost* and generally denoted by C. It basically specifies how willing we are to tolerate exceptions to the goal of finding an LPH that fully separates the two classes. If we set a large value of C, we are imposing a high "cost" to exceptions. The SVM algorithm then finds the linear boundary with the maximum margin, subject to the cost being less than C.

10.6.1 Loss Criterion and View as Regularization

Let X_i be our feature vector for the i^{th} data point, $i = 1, ..., n$. This means row i of our X matrix/data.frame. Also let Y_i be the class outcome for data point i, coding it here in the $+1/-1$ form as discussed above.

In the mathematical formulation, the optimal margin is defined as the largest one possible, subject to $|w_0 + w \bullet X_i| \geq 1$ for all data points in the training set. One can show that maximizing the margin is equivalent to *minimizing* $w \bullet w$ subject to that constraint.

Now consider a new data point x to be classified. Then x is within the margin if and only if $|w_0 + w \bullet x| < 1$. In other words:

(a) Classify x as Class +1 if $w_0 + w \bullet x > 0$.

(b) Classify x as Class -1 if $w_0 + w \bullet x < 0$.

(c) If $|w_0 + w \bullet x| < 1$, we are not very confident of our classification, as we are inside the margin, too close to the boundary.

(d) If $|w_0 + w \bullet x| \geq 1$, we are much more confident of our classification, as we are outside the margin.

Of course, we use the word "confident" here loosely, just for the purpose of intuition, not to be quantified.

It turns out that finding the best w is equivalent to minimizing the loss function

$$\max(0, [1 - Y_1(w_0 + w \bullet X_1)]) + ... \quad + \max(0, [1 - Y_n(w_0 + w \bullet X_n)]) + \lambda(w_1^2 + w_2^2 + ... + w_p^2) \tag{10.15}$$

where the minimization is done over all possible values of

$$(w_0, w_1, w_2, ..., w_p) \tag{10.16}$$

The value λ is a reformulated version of C, and thus is a hyperparameter set by the user.

Again, w is rather analogous to the $\widehat{\beta}$ vector in linear models, in that both are estimates of linear forms. In that light, look at the λ term. It is very similar to the "coefficient vector size" penalty in the LASSO. Of course, the loss functions are very different, though.

Now, what is the import of terms like $[1-y(w_0+w \bullet x)]$? This is called *hinge loss*. According to items (a)-(d) above, we reason as follows, in predicting a case x, with unknown class y:

- If $w_0 + w \bullet x$ is positive, then we guess its class to be Class +1.
- If addition $w_0 + w \bullet x$ is greater than 1, we "more confidently" guess Class +1.

Now suppose the true class is $y = +1$. Then in both cases above, we are correct. However, factoring in our "confidence," we set a loss of 0 for the second case but assess a small positive loss in the first. That may seem odd, as our guess is actually correct in that case, but since our goal is to create a wide margin, we impose a penalty in the computation. We want a wide margin, but that will mean more data points are within the margin, so we try not to have too many.

A similar analysis holds for $w_0 + w \bullet x < 0$ and $y = -1$. Note that in both cases, $y(w_0 + w \bullet x)$ will be positive, thus the same effect on the loss term. On the other hand, if the two quantities are of opposite signs, we make an incorrect classification, and incur a loss of at least 1.

Remember, SVM finds the w and w_0 that minimize Equation (10.15). In linearly separable situations, we do not have the λ term, simply minimizing

$$\max(0, [1 - Y_1(w_0 + w \bullet X_1)]) + ... + \max(0, [1 - Y_n(w_0 + w \bullet X_n)]) \qquad (10.17)$$

Adding the λ term imposes a new obstacle in our quest to minimize. As with the LASSO, a larger value of λ means we are forcing w to be smaller, which reduces variance, at the expense of increasing bias.

10.7 Applying a "Kernel"

The other major way to deal with datasets that are not fully linearly separable is transform the data, just as we did with polynomials in the logistic model earlier in this chapter. And indeed, a polynomial transformation is one common way to deal with non-separability in SVM. But there is more to it than that.

10.7.1 Navigating Between the Original and Transformed Spaces

A favorite example in ML presentations is "doughnut shaped" data. Let's generate some:

```
> z <- matrix(rnorm(500),ncol=2)
> plus1 <- z[z[,1]^2 + z[,2]^2 > 4,]  # outer ring
> minus1 <- z[z[,1]^2 + z[,2]^2 < 2,]  # inner disk
> plus1 <- cbind(plus1,+1)
> minus1 <- cbind(minus1,-1)
> head(plus1)
           [,1]       [,2] [,3]
[1,] -2.2440963 -0.6010716    1
[2,] -2.0492483 -0.5398848    1
[3,] -2.1149266 -0.1980786    1
[4,]  1.1959364  1.9502538    1
[5,]  0.4311018  2.1087900    1
[6,] -2.1462675  0.1185750    1
> head(minus1)
           [,1]       [,2] [,3]
[1,]  0.8312536  0.1306220   -1
[2,]  0.6830487  0.2926149   -1
[3,]  0.1952965 -0.7595551   -1
[4,]  0.0120678  0.7575551   -1
[5,] -0.3960729 -0.2216693   -1
[6,]  0.4085649  1.1819958   -1
> pm1 <- rbind(plus1,minus1)
> plot(pm1[,1:2],pch=pm1[,3]+2)  # gives us pluses and circles
```

The data are plotted in Figure 10-12. The two classes, drawn with pluses and circles , are clearly separable — but by a cicle, not by a straight line. But we can fix that by changing variables:

```
> pm2 <- pm1[,1:2]^2
> pm2 <- cbind(pm2,pm1[,3])
> plot(pm2[,1:2],pch=pm2[,3]+2)
```

We took our original dataset and transformed it, replacing each data point by its square. In the new plot, Figure 10-13, the pluses and circles are easily separated by a straight line.

And that is what SVM users typically do — try to find a transformation of the data under which the transformed data will be linearly separable. For computational reasons, this is typically done with a *kernel*.

We'll see how to do this below, but before moving on, let's note that it is important to be able to move back and forth between thinking of our original data and our transformed data, as we did above in visualizing the two different graphs for the logistic analysis.

10.7.2 The Notion of a Kernel

Continuing to use the dot product notation above, a key point is that the mathematical internals of SVM use lots of dot-product expressions $X_i \bullet X_j$. In the kernel approach, we have a function $K()$ — the kernel — that we use in place of that expression. In other words, everywhere in the algorithm where we would use $X_i \bullet X_j$, we would instead use $K(X_i, X_j)$.

The kernel must have certain properties that make it behave mathematically like a dot product. Two of the most popular kernels, offered by almost any SVM software package, are:

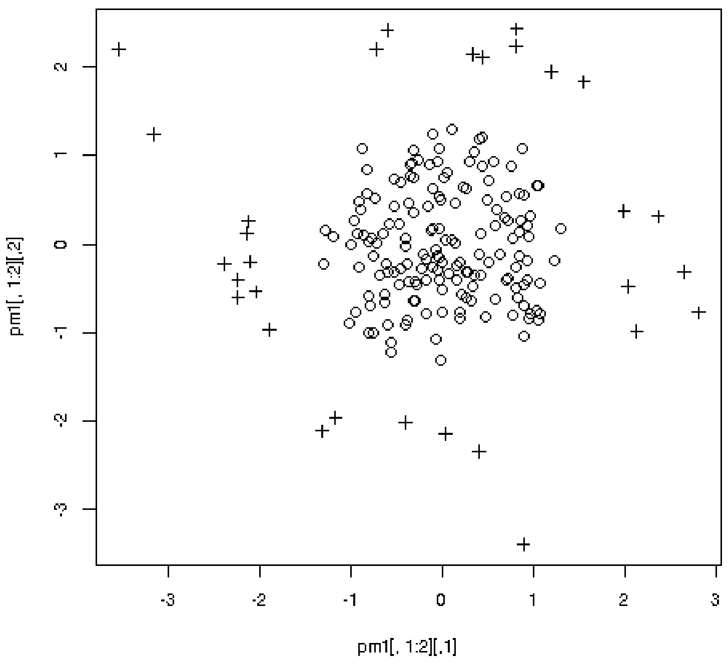

Figure 10-12: "Doughnut" data

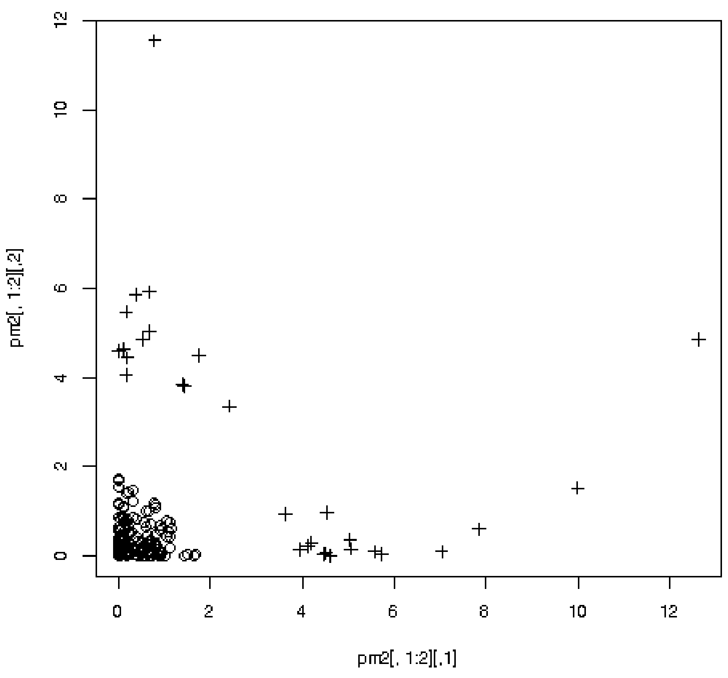

Figure 10-13: "Doughnut" data, transformed

- **polynomial:**

$$K(u,v) = (u \bullet v + 1)^d \tag{10.18}$$

- **radial basis function (RBF:)**

$$K(u,v) = \exp - ||u - v||_2^2/(2\sigma^2) \tag{10.19}$$

As you see above, each of these two kernels has a hyperparameter, the degree d for polynomials and the standard deviation σ for RBF. (RBF is motivated by a Gaussian distribution model, but that will not be relevant.)

The polynomial kernel does what the name implies: It transforms the features by a polynomial of degree d. And in so doing, it reduces computation, by means of a *kernel trick* that we will discuss later.

RBF is probably the more popular of the two, but there really isn't much difference (you'll see why below), and the polynomial kernel is much easier to explain. We'll mostly focus on the latter.

In more sophisticated forms, these kernels have further hyperparameters, but we'll stick to those described above.

10.7.3 Example: Pima Data

We'll use the **kernlab** package here.

```
> kout <- ksvm(diabetes ~ .,data=db1,kernel='vanilladot',C=1)
```

The word "vanilla" in the kernel name is an allusion to the phrase "plain vanilla," meaning here that no transformation is done. In other packages, they may use the term 'linear'. C is the cost, as explained above.

Let's try a prediction:

```
> predict(kout,data.frame(glucose= 73, mass=33))
[1] neg
```

We could try a second-dgree polynomial:

```
> kout2 <- ksvm(diabetes ~ .,data=db1,kernel='polydot',kpar=list(degree=2),C=1)
```

10.8 Choosing the Kernel and Hyperparameters

As usual, our primary tool for choosing the kernel and hyperparameters is cross-validation. We'll use **regtools::fineTuning()** as our exploratory tool.

10.8.1 Example: Pima Data

Let's try the polynomial kernel. We need a **regCall()** function, an argument to **fineTuning()**.

```
pdCall <- function(dtrn,dtst,cmbi) {
   kout <- ksvm(diabetes ~ .,data=dtrn,kernel='polydot',
      kpar=list(d=cmbi$d),C=cmbi$C)
   preds <- predict(kout,dtst)
   mean(preds == dtst$diabetes)
}
```

Note the **kpar** argument, which we didn't have before. The **ksvm()** function has us specify our kernel-related hyperparameters there, in this case the polynomial degree **d**.

So, let's run it:

```
> ft <- fineTuning(db,pars=list(d=c(2:6),C=seq(0.2,2,0.2)),regCall=pdCall,nTst=50,nXval=10
> ft$outdf
   d   C meanAcc       seAcc    bonfAcc
30 6 1.2  0.6838 0.006160775 0.02027220
33 4 1.4  0.6874 0.005795540 0.01907038
14 5 0.6  0.6876 0.005719222 0.01881925
17 3 0.8  0.6878 0.005887120 0.01937173
...
4  5 0.2  0.7072 0.005245643 0.01726093
20 6 0.8  0.7118 0.006513979 0.02143442
31 2 1.4  0.7430 0.005396407 0.01775702
41 2 1.8  0.7450 0.005573204 0.01833878
16 2 0.8  0.7492 0.005758928 0.01894991
46 2 2.0  0.7506 0.005351900 0.01761057
26 2 1.2  0.7516 0.005997845 0.01973607
21 2 1.0  0.7562 0.005161591 0.01698435
6  2 0.4  0.7590 0.005666667 0.01864632
36 2 1.6  0.7612 0.005865978 0.01930216
11 2 0.6  0.7614 0.005615671 0.01847851
1  2 0.2  0.7622 0.005123880 0.01686026
```

Why these choices of hyperparameter ranges? As we've noted, even in the title of this book, ML is an art; there are no magic formulas for these things. A lot comes from experience, including in this case our experience earlier in the chapter, in which the logistic model seemed to do better for this data with a low-degree polynomial. So we tried only a maximum of 6 for **d**.

A general rule of thumb for **C** is to try values up to about 2 or so, starting much less than 1.0. Some analysts favor values in geometric progression, e.g. 0.1, 0.2, 0.4 and so on.

We plot via the call `plot(ft)`; results are plotted in Figure 10-14. We see that most paths from high-degree models go to lower accuracy levels. And Figure 10-15, resulting from the call `plot(ft,disp=10)`, shows the top-10 runs, the trend seems to be that for degree 2, the lower values of **C** lead to higher accuracy.

Let's give RBF a try as well, using

```
rbfCall <- function(dtrn,dtst,cmbi) {
```

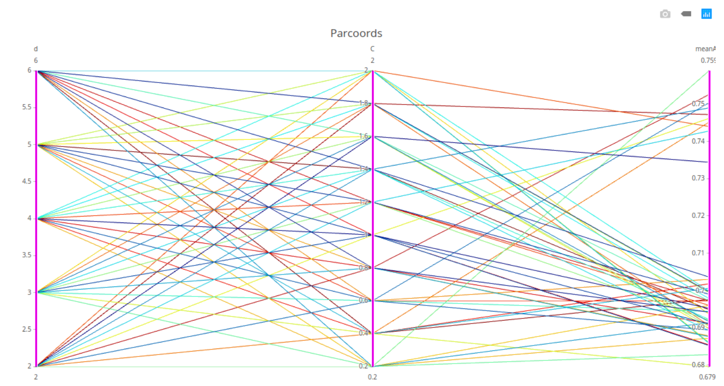

Figure 10-14: Pima, SVM polynomial

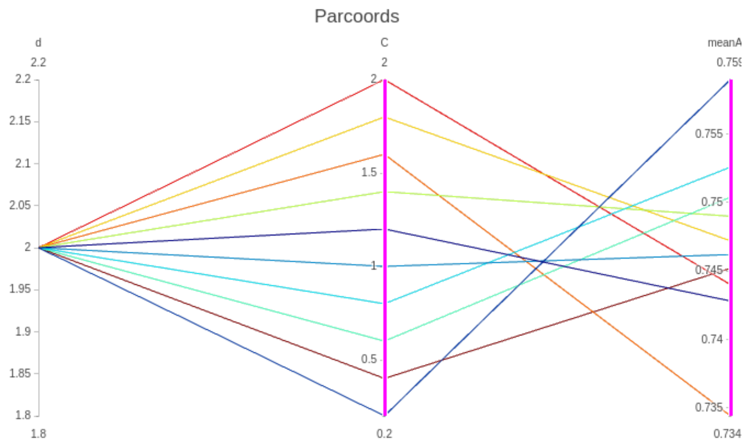

Figure 10-15: Pima, SVM polynomial, highest meanAcc

```
   kout <- ksvm(diabetes ~ .,data=dtrn,kernel='rbfdot',
      kpar=list(sigma=cmbi$sigma),C=cmbi$C)
   preds <- predict(kout,dtst)
   mean(preds == dtst$diabetes)
}
```

```
> ft <- fineTuning(db,pars=list(sigma=c(0.2,0.6,1,1.6),C=seq(0.2,2,0.4)),regCall=rbfCall,n
> ft
$outdf
   sigma   C meanAcc       seAcc    bonfAcc
1    1.0 0.2  0.6516 0.006091425 0.01841646
2    1.6 1.8  0.6708 0.006626668 0.02003468
3    1.6 1.0  0.6724 0.006449681 0.01949959
4    1.6 1.4  0.6724 0.005775462 0.01746119
5    0.6 0.2  0.6744 0.005833238 0.01763587
6    1.6 0.6  0.6746 0.005481687 0.01657301
7    1.0 0.6  0.6758 0.006255793 0.01891340
8    1.6 0.2  0.6770 0.006333333 0.01914783
9    1.0 1.4  0.6836 0.005978953 0.01807642
10   1.0 1.0  0.6844 0.006658055 0.02012957
11   1.0 1.8  0.6880 0.006083593 0.01839278
12   0.6 0.6  0.7084 0.006952538 0.02101989
13   0.6 1.0  0.7294 0.006878190 0.02079512
14   0.6 1.8  0.7350 0.005602128 0.01693715
15   0.2 0.2  0.7398 0.006675721 0.02018298
16   0.6 1.4  0.7430 0.006288518 0.01901234
17   0.2 1.0  0.7460 0.005822284 0.01760275
18   0.2 1.4  0.7484 0.005700824 0.01723554
19   0.2 1.8  0.7514 0.005281988 0.01596925
20   0.2 0.6  0.7642 0.005197474 0.01571374
```

Pretty similar to the polynomial kernel case. We'll explore that in the next section.

10.8.2 Choice of Kernel

Once again, as with so many ML questions, the answer to the question "Which kernel is best?" is "It depends." The type and size of the dataset, the number of features, and so on all make for variation in performance between the two kernels.

On the other hand, it often doesn't really matter. The polynomial and RBF kernels gave us virtually identical accuracy in the Pima example above.

But there's more: RBF is in essence a polynomial method anyway. In calculus, it is shown that functions like e^t can be written in a *Taylor series*:

$$e^t = 1 + t + \frac{t^2}{2!} + \frac{t^3}{3!} + ... \tag{10.20}$$

In other words, e^t acts like an "infinite-degree polynomial, and since the terms in that series eventually get small, the series acts close to an ordinary, finite-degree polynomial. The larger the value of σ, the smaller the degree, though this will also depend on the range of values in our dataset.

10.9 Example: Telco Churn Data

Let's return to the dataset analyzed in Sections 2.2 and 7.10.1. The data preparation is the same is before, except that now we have converted the **Stay** variable to a factor in order to use the R formula specification. One interesting difference of this dataset from the Pima data is that here almost all of the features are dummy variables.

Here is code to explore the data using **fineTuning()**:

```
> polyCall
function(dtrn,dtst,cmbi) {
   kout <- ksvm(Stay ~ .,data=dtrn,kernel='polydot',
      kpar=list(d=cmbi$d),C=cmbi$C)
   preds <- predict(kout,dtst)
   mean(preds == dtst$Stay)
}
> ft <- fineTuning(tcd,pars=list(d=2:4,C=c(0.1,0.5,1)),polyCall,nXval=10)
> ft$outdf
  d   C meanAcc       seAcc    bonfAcc
1 4 1.0  0.6906 0.005978108 0.01657682
2 4 0.1  0.7028 0.006587530 0.01826670
3 4 0.5  0.7052 0.006038396 0.01674400
4 3 0.5  0.7068 0.007555425 0.02095060
5 3 1.0  0.7068 0.007508218 0.02081970
6 3 0.1  0.7362 0.007497852 0.02079095
7 2 0.1  0.7990 0.006227181 0.01726748
8 2 0.5  0.8044 0.005281414 0.01464495
9 2 1.0  0.8046 0.004733333 0.01312516
```

Note first that we did only 10 folds in the cross-validation. We're working with a larger data set than Pima here, both in rows and columns, so the going is slower. It's possible that somehow the large number of dummy features slowed things down as well; SVM can be shown to be minimizing a *convex* function, which has no local minima and thus no convergence problems, but perhaps convergence is slower with the dummy variables.

At any rate, here too we find that degree-2 polynomials are best. The higher-degree ones are overfitting, though remember, that is for our current sample size, about 7000. If we had a larger sample from this population, a higher degree may work well.

Note too that among the degree-2 runs, the value of **C** didn't matter much; indeed, the **bonfAcc** column suggests they are within sampling variation of each other.

10.10 Example: Fall Detection Data

Now let's turn our attention to the fall detection data we first saw in Section 7.10.2.1. Here the *Y* variable is categorical, so let's see how SVM does there. (Recall that the All vs. All method will be used.)

```
> polyCall
function(dtrn,dtst,cmbi) {
   kout <- ksvm(ACTIVITY ~ .,data=dtrn,kernel='polydot',
      kpar=list(d=cmbi$d),C=cmbi$C)
   preds <- predict(kout,dtst)
   mean(preds == dtst$ACTIVITY)
}
> fineTuning(fd,pars=list(d=2:6,C=c(0.1,0.5,1)),polyCall,nXval=10)
$outdf
   d   C meanAcc       seAcc    bonfAcc
1  2 0.1  0.3862 0.008851616 0.02598126
2  2 0.5  0.3958 0.007213876 0.02117416
3  2 1.0  0.4154 0.005190162 0.01523416
4  3 0.1  0.4290 0.007304489 0.02144013
5  3 0.5  0.4534 0.007149514 0.02098525
6  3 1.0  0.4620 0.007757434 0.02276962
7  4 0.1  0.4672 0.009452572 0.02774518
8  4 0.5  0.4942 0.006264184 0.01838663
9  6 1.0  0.4980 0.017025471 0.04997315
10 4 1.0  0.5160 0.005155364 0.01513202
11 5 1.0  0.5168 0.014859939 0.04361688
12 5 0.1  0.5202 0.005822943 0.01709150
13 6 0.1  0.5256 0.011214277 0.03291614
14 5 0.5  0.5400 0.008834277 0.02593036
15 6 0.5  0.5446 0.010464968 0.03071677
```

Comments:

- Keep in mind the variability in the cross-validation process. For instance, an approximate 95% confidence interval for the case d = 6, C = 0.5 is

$$0.5446 \pm 0.03071677 = (0.5139, 0.5753) \qquad (10.21)$$

  So for example we should not necessarily conclude that degree 6 is better than degree 5.

- On the other hand, the above results do suggest that we try even higher-degree polynomials.

10.11 And What about the Kernel Trick?

No presentation of SVM would be complete without discussing the famous *kernel trick*, as it is largely responsible for the success of SVM. As usual, we won't delve into the mathematical details, but the principle itself has major practical implications.

To motivate this, let's get an idea of how large our dataset can expand to when we transform via polynomials, *without* using kernels and SVM. Let's use the forest cover data, Section 4.4, together with the **polyreg** package, the latter being used to count data columns.

Since the latter is our only goal, we can just use a subset of the data, 5000 randomly selected rows. Here is the code:

```
> gpout <- getPoly(cvr5k,2)
> dim(gpout$xdata)
[1] 5000 1314
```

Recall, the original dataset had only 54 features, but in the new form we have over 1300 of them. We would have even more if **getPoly()** did not avoid creating duplicates, e.g. the square of a dummy variable;

With the original dataframe of more than 580,000 rows, the new size would be $581012 \times 1311 = 761706732$ elements. At 8 bytes per element, that would mean 6.1 gigabytes of RAM!

And that was just for degree 2. Imagine degree-3 polynomials and so on! (Degree 3 turns out to generate 16897 columns.) So, some kind of shortcut is badly needed. Kernel trick to the rescue!

The key point is that by using the kernel (10.18), we can avoid having to compute and store all those extra columns. In the forest cover data, we can stick to those original 54 features — the vectors u and v in that expression are each 54 elements long — rather than computing and storing the 1311. We get the same calculations mathematically as if we were to take u and v to be 1311-element vectors. (We omit the derivation here, but it is elementary algebraic manipulation.)

10.12 Further Calculation Issues

To be sure, in some situations the kernel trick by itself may not be enough. It merely prevents large problems from getting larger. As each any nonparametric ML algorithm SVM requires large amounts of computation time for some datasets, more than k-NN though less than neural networks. The running time is roughly a constant times n^3.

10.12.1 Speeding Up the Computation

What can be done to speed things up?

- Dimension reduction, e.g. via PCA.
- Use a fast implementation of SVM, such as the R package **liquidSVM** (next sections).

10.12.1.1 The liquidSVM Package

The best way to speed up R code is to convert some of it to C/C++. Actually, the author of the package started with the latter, and wrote interfaces to several higher-level languages, including R

On the plus side, the code is quite fast. A negative, though, it does not offer the user much choice. The package does its own cross-validation, on a grid chosen by the package. Only RBF is offered.

10.12.1.2 Example: Network Attack

This dataset's goal is to determine whether a section of the Internet is under attack.[3].

The data required some preprocessing (details now shown).

- File was in **.arff** file format, so it was read using **foreign::read.arff()**.
- NAs were removed using **na.exclude()**.
- One column, **Packet Size_Byte**, was constant and thus set to NULL.
- Blanks and '-' signs were replace in the column names by underscores:

```
names(netAtt) <- gsub(' ','_',names(netAtt))
names(netAtt) <- gsub('-','_',names(netAtt))
```

In the end, the processed data frame was named **netAtt**.

```
> head(netAtt,2)
  Node Utilised_Bandwith_Rate Packet_Drop_Rate Full_Bandwidth
1    3               0.822038         0.190381           1000
2    9               0.275513         0.729111            100
...
> dim(netAtt)
[1] 1060   21
```

```
> lq <- svmMulticlass(Class ~ .,trn)
> preds <- predict(lq,tst)
> mean(preds == tst$Class)
[1] 0.99
```

Excellent accuracy.

10.13 SVM: Bias/Variance Tradeoff

To recap: The bias/variance tradeoff is definitely an issue in SVM:

- Cost parameter **C**: The larger **C** is, the smaller w will be, thus reducing variance though increasing bias due to less choice for w.

3. http://archive.ics.uci.edu/ml/datasets/Burst+Header+Packet+(BHP)+flooding+attack+on+Optical+Burst+S

- In the polynomial kernel, larger **d** means more flexibility in the model, hence lower bias. But now we have more parameters to estimate, thus higher variance. RBF is similar.

11

LINEAR MODELS ON STEROIDS: NEURAL NETWORKS

The method of neural networks (NNs) is probably the best known ML technology among the general public. The science fiction-sounding name is catchy — even more so with the advent of the term *deep learning* — and they have become the favorite approach to image classification, on applications that also intrigue the general public such as facial recognition.

Yet NNs are probably the most challenging ML technology to use well, with problems such as:

- "Black Box" operation, not clear what's going on inside.
- Numerous hyperparameters to tune.
- Tendency toward overfitting.
- Possibly lengthy computation time, in some large-data cases running to hours or even days. In addition, large amounts of RAM may be needed.
- Convergence problems,

The term *neural network* (NN) alludes to a machine learning method that is inspired by the biology of human thought. In a two-class classification problem, for instance, the predictor variables serve as inputs to a "neuron," with output 1 or 0, with 1 meaning that the neuron "fires" and we decide Class 1. NNs of several *hidden layers*, in which the outputs of one layer of neurons are fed into the next layer and so on, until the process reaches the final output layer, were also given biological interpretation.

The method was later generalized, using *activation functions* with outputs more general than just 1 and 0, and allowing backward feedback from later layers to earlier ones. This led development of the field somewhat away from the biological motivation, and some questioned the biological intepretation anyway, but NNs have a strong appeal for many in the machine learning community. Indeed, well-publicized large projects using *deep learning* have revitalized interest in NNs.

We'll present a picture for a specific dataset shortly, but here is an overview of how the method works:

- A neural network consists of a number of *layers*.
- In pictures describing a particular network, there is an input layer on the far left, and an output layer on the far right, with one or more *hidden* layers in between.
- The outputs of one layer are fed as inputs into the next layer.
- The output is typically a single number, for regression problems, and *c* numbers for a *c*-class classification problem; the latter numbers are the conditional probabilities of the classes.
- The inputs into a layer a fed through what amounts to a linear model. The outputs of a layer are fed through an *activation function*, which is analogous to kernel functions in SVM, to accommodate nonlinear relations.

11.1 Example: Vertebrae Data

Let us once again consider the vertebrae data. For this example, we'll use the **neuralnet** package, available from CRAN, and then switch to the more complex but more powerful **h2o** package.

```
> library(neuralnet)
> vert <- read.table('column_3C.dat',header=FALSE)
> library(dummies)  # R 'formula' doesn't run in neuralnet
> ys <- dummy(vert$V7)  # 3 columns of 1s and 0s, for 3 classes
> vert <- cbind(vert[,1:6],ys)  # replace original Y by dummies
> names(vert)[7:9] <- c('DH','NO','SL')
> set.seed(9999)
> nnout <- neuralnet(DH+NO+SL ~ V1+V2+V3+V4+V5+V6,
   data=vert,hidden=3,linear.output=FALSE)
> plot(nnout)
```

Note that we needed to create dummy variables for each of the three classes. Also, **neuralnet**()'s computations involve some randomness, so for the sake of reproducibility, we've called **set.seed()**.

As usual in this book, we are using the default values for the many possible arguments, including using the logistic function for activation,

$$g(t) = \frac{1}{1 + e^{-t}} \tag{11.1}$$

(It is just used as a mechanism for inducing nonlinearity, and does not have much connection to the logistic classification model.)

However, we have set a nondefault value of **hidden = 3**. That argument is a vector through which the user states the number of hidden layers and the number of neurons in each layer. Since the number 3, viewed as a vector, is a vector of length 1, we are saying we want just one hidden layer. The 3 then says we want three *neurons* or *units* in that layer.[1]

The network is shown in Figure 11-1.[2] function, as see above. In this dataset, we have $p = 6$ features, named **V1**, **V2** and so on. You can see those on the far left, with six circles forming the input layer. The rightmost three circles form the output layer, in this 3-class problem. In between we see the hidden layer, with three circles for the three units.

In our call to **neuralnet()**, we also had to specify **linear.output = FALSE**, to indicate that this is a classification problem rather than regression.

The several lines coming into a neuron from the left are labeled with numbers representing weights. The final value fed into that neuron is the weighted sum of the "ancestor" neurons on the left. So, that weighted sum is analogous to linear regression. The analog of the constant or intercept term in linear regression is represented by the circles labeled "1"; they too are multiplied by a wieght.

As mentioned above, the weighted sum coming out of a circle is then fed into the activation function, with the value of that function then being the final output value at the neuron.

Weights are computed from the data using a very complex process. Like linear regression, it works by attempting to minimize a sum of squared prediction errors (or other loss function).

11.1.1 Prediction

To predict a new case, the values of its p features are fed into the input layer, and the weights are applied as the case makes it way through the network. Eventually the class probabilities come out on the right side of the picture. The **neuralnet** package, as usual, includes generic **predict()** function for this purpose.

Let's use it to predict a modified version of row 3 in our data:

1. Neural networks were originally inspired by a desire to model the brain, hence the term *neurons*. Today they are not viewed that way, and the term *units* is more common.
2. This figure was produced by the package's generic **plot()**

```
> newx <- vert[3,1:6]
> newx[,5] <- 88.22
> predict(nnout,vert[3,1:6])
       [,1]     [,2]      [,3]
3 0.3721868 0.602857 0.0246287
```

These are the class probabilities. The second class, 'NO', is the likeliest, so we would predict this case to be in that class. Or, if we want to automate that conversion of probabilities to class, we could do

```
> which.max(predict(nnout,newx))
[1] 2  # class 2, 'NO'
```

or even

```
> classNum <- which.max(predict(nnout,newx))
> c('DH','NO','SL')[classNum]
```

11.1.2 Hyperparameters: neuralnet

Neural network libraries are notorious for having tons of hyperparameters. Here are some offered by **neuralnet**:

- Number of hidden layers.
- Number of units per layer (need not be constant across layers).
- Learning rate (similar to the quantity of the same name in Section 5.3.7). We can also set an *adaptive* learning rate, which starts out large but is reduced as we near the minimum point.
- Maximum number of iteration steps.

The latter two are designed to help the convergence process settle on a good set of weights.

11.2 Pitfall: Issues with Convergence, Platforms Etc.

NNs form the most complex of the major ML methods, by far. Users must be willing to deal with a number of issues that do not arise much with other methods.

11.2.1 Convergence

Here is a rough comparison of some major methods:

method	iterative? (no is good)	convex? (yes is good)
linear model	no	yes
SVM	yes	yes
NNs	yes	no

With a linear model, the solution is an explicit, closed-form function of the data, hence no iteratively guessing for the solution.

The term *convex* is technical, but the key property for us is that the solution is unique, no local minima like that at $x = 0.4$ in Figure 5-2. (The reader may benefit from reviewing Section 5.3.7 before continuing.)

Actually, we know *a priori* that with NNs even the global minimum point is not unique. To see this, look again at Figure 11-1. Say we swap the top circle in the hidden layer and and the bottom circle, with the corresponding input lines moving as well. It's just a relabeling, and we would get the same total prediction sum of squares. Since we can make lots of swaps like that, it is clear that there are many sets of weights that attain the same minimum sums of squares.

No wonder, the, that NNs are subject to convergence problems. A model may run for a long time without converging. Or worse, it may converge to a local minimum, a bad set of weights with poor predictive power for new cases in the future. One particular malady worth mentioning is the "broken clock problem," in which all predicted values are identical.

We will return to this issue shortly.

11.2.2 Software Complexity

Any powerful implementation of NNs — say **keras**, **MXNet** or the one featured here, **h2o** — will be quite complex. The user may thus encounter difficulties installing, configuring and executing such packages. They are powerful, yes, but that power may come at a price.

This calls for patience on the part of new users. In some cases, one may see a cryptic error message. The usual tacks of doing a Web search for the error message, posting queries on Web forums and so on will usually produce a quick solution. In some cases, restarting the software (in the case of **h2o**, this may include terminating the associated Java processes) will clear up the problem. You may also try adding more memory (e.g., via the **max_mem_size** argument in **h2o.init()**).

11.3 Activation Functions

Over the years, there has been some debate as to good choices for the activation function. In principle, any nonlinear function should work, but issues do arise, especially concerning convergence problems.

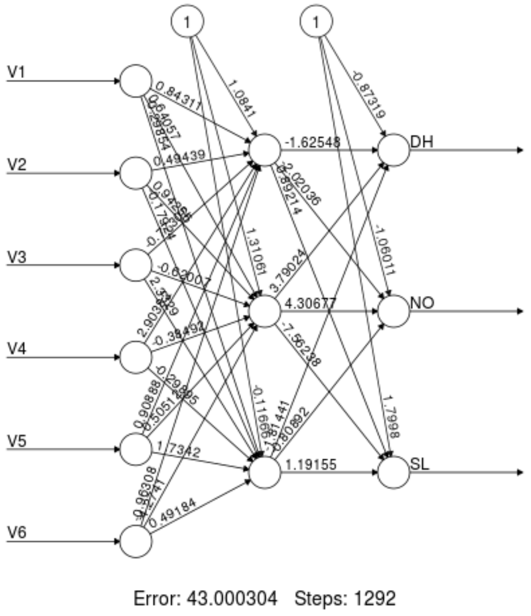

Figure 11-1: Verbebrae NN, 1 hidden layer (see color insert)

Consider once again Figure 5-2. The minimum around 2.2 comes at a rather sharp dip.[3] But what if the curve were to look like that in Figure 11-3? Here even a larger learning rate might have us spending many iterations with almost no progress. This is the *vanishing gradient problem*. And if the curve is very sharp near the minimum, we may have an *exploding gradient problem*.

The choice of activation function plays a big role in these things. There is a multiplicative effect across layers,[4], and quantities in (-1,1) become smaller and smaller as that effect grows, hence a vanishing gradient. If the gradient is large at each layer

After years of trial and error, the popular choice among NN users today is the *Rectified Linear Unit* (ReLU): $f(x)$ is 0 for $x < 0$, but is equal to x for $x \geq 0$.

11.4 Regularization

As noted, NNs have a tendency to overfit, with many having thousands of weights, some even millions. Remember, the weights are essentially linear regression coefficients, so the total number of weights is effectively the new value of p, our number of features. Some way must be found to reduce that value.

3. In calculus terms, large second derivative.
4. Again, for those who know calculus, this is the Chain Rule in action.

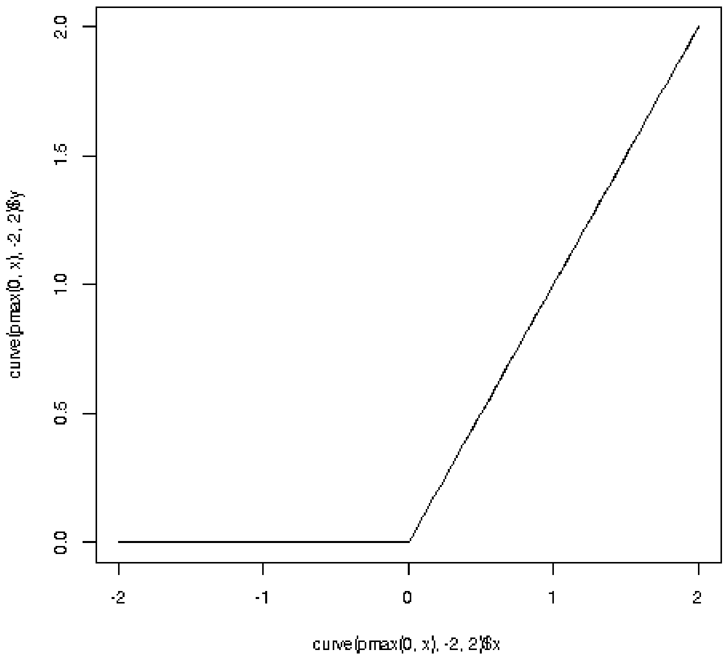

Figure 11-2: ReLU

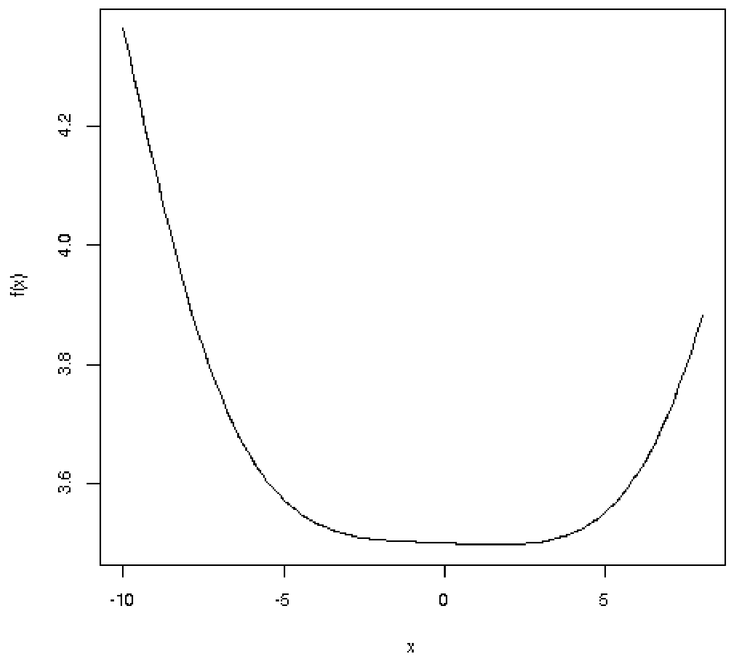

Figure 11-3: Shallow min region

11.4.1 ℓ_1, ℓ_2 Regularization

Since NNs (typically) minimize a sum of squares, one can apply a penalty term to reduce the size of the solution, just as in the cases of ridge, the LASSO and SVM. Recall also that in the LASSO, with the ℓ_1 penalty,this tends to produce a sparse solution, with most coefficients being 0s.

Well, that is exactly what we want here. We fear we have too many weights, and we hope that applying an ℓ_1 penalty will render most of them nil.

However, that may not happen with NNs, due to the use of nonlinear activation functions. The problem is that the contours in Figure 8-4 are no longer ellipses, and thus the "first contact point" will not likely be at a corner of the diamond.

Nevertheless, ℓ_1 will still shrink the weights, as will ℓ_2, so we should achieve dimension reduction in some sense.

11.4.2 Regularization by Dropout

If a weight is 0, then in a picture of the network such as Figure 11-1, the corresponding link is removed. So, if the goal is to remove some links, why not simply remove some links directly? That is exactly what *dropout* does.

For instance, if our dropout rate is 0.2, we randomly choose 20% of the links and remove them.

11.5 Convergence Tricks

As noted, it is often a challenge to configure NN analysis so that proper convergence to a good solution is attained. Below are a few tricks one can try, typically specified via one or more hyperparameters:

In some cases convergence problems may be solved by scaling the data, either using the R **scale()** function or by mapping to [0,1] (Section 3.1.3.3). It is recommended that one routinely scale one's data, and many NN implementations scale by default. That is the case with the **h2o** package featured in this chapter.

Here are some other tricks to try:

- *learning rate:* Discussed earlier.
- *early stopping:* In most algorithms, the more iterations the better, but in NNs, many analysts depart from conventional wisdom. They fear that running the algorithm for too long may result in convergence to a poor solution. This leads to the notion of *early stopping*, of which there are many variants. Basically we can stop if the accuracy in the validation set begins to deteriorate, even though we have not performed the planned number of epochs. This is the subject of some options in **h2o.deeplearning()**.
- *momentum:* The rough idea here is that "We're on a roll," with the last few epochs producing winning moves in the right direction, reducing validation error each time. So, instead of calculating the next step size individually, why not combine the last few step sizes?

The next step size will be set to a weighted average of the last few, with heavier weight on the more recent ones.

11.6 Fitting Neural Network with the h2o Package

The **h2o** package consists of a suite of various ML methods, with the important advantage that they all are capable of parallel computation, essential in large datasets.

The NNs implementation is highly sophisticated, with myriad hyperparameters. Some of them are rather arcane — and as usual in this book, we will cover only the basics — but the package equips us to take on very challenging ML problems. Again, parallel computation is an integral aspect of the software here.

11.6.1 Example: NYC Taxi Data

With same preprocessing as in Section 4.3, with our data frame **yell**, let's use NNs to predict trip time. Here is the prep work:

```
library(h2o)
h2o.init()  # required initialization
yellh2o <- as.h2o(yell)  # h2o has its special data structure
```

The **h2o** package adopts the "triple cross-validation" approach that we mentioned in Section 3.4, partitioning the data into training, validation and test sets. The first two are used for the package's internal cross-validation, and are thus essential. In our first example here, we will just fit the model directly to the data, not setting up a test set, with 80% of the data in the training set and 19% in the validation set. (It does not allow having a void test set, so we have 1% here.)

```
splits <- h2o.splitFrame(yellh2o, c(0.8,0.19), seed=99999)
```

OK, ready to run:

```
nnout <- h2o.deeplearning(
      training_frame = splits[[1]],
      validation_frame = splits[[2]],
      x = names(yellh2o)[-7],
      y = names(yellh2o)[7],
      hidden = c(50,50),
      epochs = 25
)
```

Nothing new here except **epochs**, which as noted before is the number of iterations to run. However, early stopping is implemented by default, so iterations may not fully meet this parameter.

To illustrate predicting in this setup, let's predict a modified version of one of the data points:

```
> z <- yellh2o[8,]
> z[1,2] <- 2
> z
  passenger_count trip_distance PULocationID
1               6             2           79
  DOLocationID PUweekday DOweekday tripTime
1          148         7         7      694

[1 row x 7 columns]
> h2o.predict(nnout,z)
   predict
1 688.3905
```

11.6.2 Hyperparameters: h2o

The package certainly gives the user lots of choices! Let's count them, using R's **formals()** function, a *metaprogramming* function that forms an R list of the named arguments of a function:

```
> length(formals(h2o.deeplearning))
[1] 89
```

Yes, the function actually has 89 arguments! Many of them are aimed at improving convergence behavior, again such as the learning rate and adaptive variants, as well as various methods for early stopping. The **hidden** argument works as in **neuralnet**. Here are a few more:

- **activation**: Quoted name of the desired activation function. The default is 'Rectifier', ReLU.
- **l1** and **l2**: These do ℓ_1 and ℓ_2 regularization. So we minimize the original sum of squares plus **l1** or **l2** times a penalty term consisting of the sum of the weights or their squares.
- **hidden_dropout_ratios**: The user sets one dropout probability per hidden layer.

11.6.3 Grid Search: Taxi Data

Let's search combinations of numbers of neuron per hidden layer, ℓ_1 penalty, and number of epochs.

```
nnCall <- function(dtrn,dtst,cmbi,...) {
   yNum <- list(...)$yNum
   dtrn <- as.h2o(dtrn)
   dtst <- as.h2o(dtst)
   splits <- h2o.splitFrame(dtrn, c(0.6,0.19), seed=99999)
   nnout <- h2o.deeplearning(
      training_frame = splits[[1]],
      validation_frame = splits[[2]],
      x = names(dtrn)[-yNum],
      y = names(dtrn)[yNum],
      hidden = c(cmbi$hid,cmbi$hid),
      epochs = cmbi$epochs,
      quiet_mode=TRUE
   )
   preds <- h2o.predict(nnout,dtst)
   mean(abs(preds - dtst[,yNum]))
}
ftout <- fineTuning(yell,
   pars=list(hid=c(10,50,100,200),l1=c(0,0.2),epochs=c(10,25)),
   regCall=nnCall,nXval=10,yNum=7))
```

Here are the results:

```
   hid  l1 epochs  meanAcc    seAcc   bonfAcc
1   10 0.2     25 186.4247 3.459003 10.221931
2   50 0.0     10 186.6181 3.068476  9.067858
3   50 0.2     25 187.2439 3.503460 10.353309
4  200 0.0     10 188.6969 2.682247  7.926488
5  100 0.2     25 189.1903 5.540058 16.371795
6  100 0.2     10 190.2891 2.665792  7.877860
7  200 0.2     10 190.3145 4.012173 11.856640
8   50 0.0     25 190.6382 3.353712  9.910778
9   50 0.2     10 190.8747 2.194415  6.484864
10 200 0.0     25 190.9686 3.541748 10.466455
11 200 0.2     25 191.2162 3.331480  9.845078
12 100 0.0     25 191.7996 2.464705  7.283616
13  10 0.0     25 192.1888 3.266617  9.653397
14  10 0.0     10 194.1112 2.358299  6.969167
15 100 0.0     10 194.1788 2.882786  8.519115
16  10 0.2     10 197.5511 3.135322  9.265401
```

Some plots are shown in Figure 11-4 and 11-5. Higher number of epochs and higher degree of regularization seem to have important effects. Note, however, the larger Bonferroni values. As usual, further investigation is warranted.

11.7 Example: StumbleUpon Data

This is another dataset from Kaggle. The goal is to predict whether a Web site will become "evergreen," i.e. long-lasting.

There were various problems with the data. For instance, the missing values were coded '?' rather than NA, which caused numeric variables to be stored in character form. After some data wrangling, here is what the first record looks like:

```
> su[1,]
```

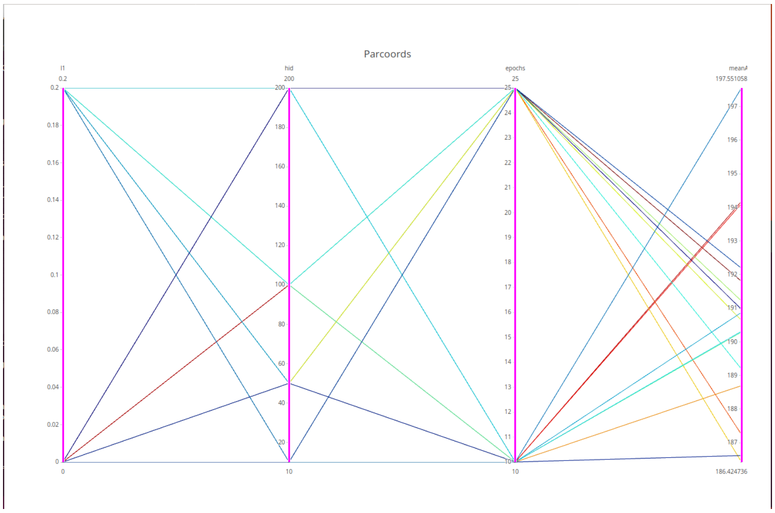

Figure 11-4: Taxi data, fineTuning() output, 1

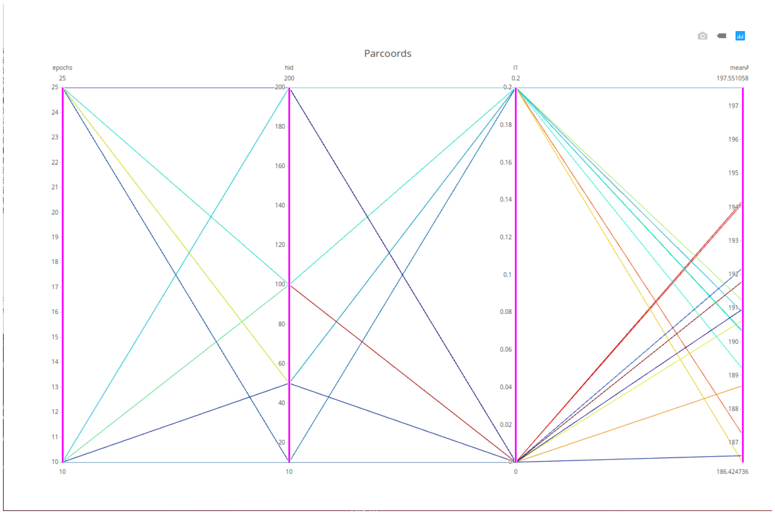

Figure 11-5: Taxi data, fineTuning() output, 2

```
  alchemy_category alchemy_category_score
1         business               0.789131
  avglinksize commonlinkratio_1 commonlinkratio_2
1    2.055556         0.6764706         0.2058824
  commonlinkratio_3 commonlinkratio_4
1        0.04705882        0.02352941
  compression_ratio embed_ratio framebased
1         0.4437832           0          0
  frameTagRatio hasDomainLink html_ratio
1    0.09077381             0  0.2458312
  image_ratio lengthyLinkDomain linkwordscore
1 0.003883495                 1            24
  news_front_page non_markup_alphanum_characters
1               0                           5424
  numberOfLinks numwords_in_url
1           170               8
  parametrizedLinkRatio spelling_errors_ratio
1             0.1529412            0.07912957
  label
1     0
```

And here is the prep and our call:

```
> suh2o <- as.h2o(su)
> splits <- h2o.splitFrame(suh2o, c(0.6,0.19), seed=99999)
> nnout <- h2o.deeplearning(
     training_frame = splits[[1]],
     validation_frame = splits[[2]],
     x = 1:21, y = 22, hidden = rep(50,3), epochs = 50)
```

Note that the function allows one to specify the features and outcome variable via column numbers in the dataset. And since the *Y* variable here is an R factor, the function knew to treat this as a classification problem.

To illustrate prediction, let's again predict a modified version of one of the data points:

```
> su[8,]
  alchemy_category alchemy_category_score
8             <NA>                     NA
  avglinksize commonlinkratio_1 commonlinkratio_2
8    1.883333          0.719697         0.2651515
  commonlinkratio_3 commonlinkratio_4
8         0.1136364        0.01515152
  compression_ratio embed_ratio framebased
8         0.4993481           0          0
  frameTagRatio hasDomainLink html_ratio
8    0.02661597             0  0.1737459
  image_ratio lengthyLinkDomain linkwordscore
8  0.02583026                 0             5
```

```
  news_front_page non_markup_alphanum_characters
8              NA                          27656
  numberOfLinks numwords_in_url
8           132               4
  parametrizedLinkRatio spelling_errors_ratio
8            0.06818182             0.1485507
  label
8     0
> z <- su[8,]
> z[,19] <- 56  # only 56 links, not 132
> h2o.predict(nnout,as.h2o(z))
  |========================================| 100%
  |========================================| 100%
  predict        p0         p1
1       0 0.9838046 0.0161954
```

The estimated conditional class probabilities are about 0.91 and 0.09 for classes 0 and 1, so we predict the former, seen as 0 in the 'predict' column.

11.7.1 Grid Search: StumbleUpon Data

Since this is a classification problem, our **regCall()** function changes slightly, in the last line:

```
nn01Call <- function(dtrn,dtst,cmbi,...) {
   yNum <- list(...)$yNum
   dtrn <- as.h2o(dtrn)
   dtst <- as.h2o(dtst)
   splits <- h2o.splitFrame(dtrn, c(0.6,0.19), seed=99999)
   nnout <- h2o.deeplearning(
      training_frame = splits[[1]],
      validation_frame = splits[[2]],
      x = names(dtrn)[-yNum],
      y = names(dtrn)[yNum],
      hidden = c(cmbi$hid,cmbi$hid),
      epochs = cmbi$epochs,
      quiet_mode=TRUE
   )
   preds <- h2o.predict(nnout,dtst)
   mean(preds[,1] == dtst[,yNum])
}
```

And here is our call, and the results:

```
> ftout <- fineTuning(su,
   pars=list(hid=c(50,100,200),l1=c(0.01,0.1,0.3)),
   regCall=nn01Call,nXval=5,yNum=22)
> ftout$outdf
  hid   l1 meanAcc       seAcc    bonfAcc
```

```
1  50 0.10  0.6288 0.006887670 0.01909897
2  50 0.01  0.6332 0.009200000 0.02551088
3 100 0.10  0.6344 0.009389356 0.02603594
4 200 0.10  0.6356 0.018977882 0.05262417
5  50 0.30  0.6404 0.018037738 0.05001723
6 200 0.30  0.6428 0.006711185 0.01860959
7 100 0.01  0.6488 0.010928861 0.03030487
8 100 0.30  0.6532 0.011182129 0.03100716
9 200 0.01  0.6604 0.019135308 0.05306070
```

One interesting result here is that the heaviest ℓ_1 penalty, 0.3, seemed to do the best.

11.8 Example: African Soil Data

As we noted in Section 5.2.3.1, this dataset is important because $p > n$. Thus there is high danger of overfitting, so we will try regularization in the forms of both ℓ_1 and dropout. Let's also try varying the learning rate.

```
> nnCall
function(dtrn,dtst,cmbi,...) {
   dtrn <- as.h2o(dtrn)
   dtst <- as.h2o(dtst)
   splits <- h2o.splitFrame(dtrn, c(0.8,0.19), seed=99999)
   nnout <- h2o.deeplearning(
      training_frame = splits[[1]],
      validation_frame = splits[[2]],
      x = names(dtrn)[1:3578],
      y = names(dtrn)[3597],
      hidden = c(cmbi$hid,cmbi$hid),
      epochs = cmbi$epochs,
      activation="RectifierWithDropout",
      hidden_dropout_ratios = c(cmbi$drop,cmbi$drop),
      rate = cmbi$lr
   )
   preds <- h2o.predict(nnout,dtst)
   mean(abs(preds - dtst[,3597]))
}

> ftout <-
   fineTuning(afrsoil,
   pars=list(hid=c(25,75,150),
     drop=c(0.1,0.3,0.5),l1=c(0.01,0.1,0.2)),
     lr=c(0.001,0.005,0.01),
     regCall=nnCall,nXval=5)
> ftout$outdf
   hid drop   l1   meanAcc       seAcc    bonfAcc
1   75  0.5 0.10 0.4617314 0.017069048 0.05313624
```

```
2    25  0.5 0.10 0.4621802 0.012717921 0.03959111
3    75  0.3 0.01 0.4654548 0.023220129 0.07228466
4    25  0.1 0.01 0.4681233 0.017372519 0.05408095
5    25  0.5 0.01 0.4691097 0.008060311 0.02509189
6   150  0.5 0.01 0.4744390 0.016670277 0.05189486
7    25  0.3 0.01 0.4768885 0.014992992 0.04667344
8   150  0.3 0.10 0.4784278 0.010950723 0.03408979
9   150  0.3 0.01 0.4804130 0.013055232 0.04064116
10   75  0.5 0.01 0.4928280 0.036239523 0.11281426
11   75  0.1 0.01 0.4985511 0.023642604 0.07359983
12   25  0.1 0.20 0.4997850 0.016150282 0.05027611
13 150  0.5 0.10 0.5101304 0.005873828 0.01828533
14   25  0.3 0.10 0.5102162 0.024840995 0.07733045
15   75  0.1 0.10 0.5145073 0.035657030 0.11100095
16 150  0.1 0.10 0.5253034 0.010456328 0.03255073
17   25  0.1 0.10 0.5283610 0.039062783 0.12160312
18   75  0.3 0.10 0.5378577 0.013458684 0.04189712
19 150  0.1 0.20 0.5535894 0.040257586 0.12532256
20 150  0.1 0.01 0.5656318 0.026745878 0.08326038
21   75  0.1 0.20 0.5895071 0.050759411 0.15801492
```

(One of the later cases resulted in convergence prblems, so the grid search was terminated.)

11.9 Close Relation to Polynomial Regression

In Section 7.8, we introduced *polynomial regression*, which is a linear model in which the features are in polynomial form. So for instance instead of just having people's heights and ages as features, in a quadratic model we now would also have the squares of heights and ages, as well as an height $\times$ age term.

Polynomials popped up again in SVM, with polynomial kernels. Again we might have for instance not just height and age, but also the squares of heights and ages, as well the height $\times$ age term. And we noted that even RBF, a nonpolynomial kernel, is approximately a polynomial, due to Taylor series expansion.

It turns out that NNs essentially do polynomial regression as well. To see this, let's look again at Figure 11-1. Suppose we take as our activation function the squaring function t^2. That is not a common choice, but we'll start with that and then extend the argument

So, in the hidden layer in the picture, a circle sums its inputs from the left, then outputs the square of the sum. That means the outputs of the hidden layer are second-degree polynomals in the inputs. If we were to have a second hidden layer, its outputs would be fourth-degree polynomials.

What if our activation function itself were to be a polynomial? Then again each successive layer would give us higher and higher-degree polynomials in the inputs.

Since NNs minimize the sum of squared prediction errors, just as in the linear model, you can see that the minimizing solution will be that of polynomial regression.

And what about the popular activation functions? One is the *hyperbolic tangent*, *tanh*(t), whose graph looks similar to the logistic function. But it too has a Taylor series expansion, so what we are doing is approximately polynomial regression.

ReLU does not have a Taylor expansion, but we can form a polynomial approximation there too.

In that case, why not just use polynomial regression in the first place? Why NNs? One answer is that, as we saw in Section 10.11 it just would be infeasible for large-p data. The kernel trick is very helpful here, and there is even the *kernel ridge regression* method that applies this to linear ridge models, but it turns out that this too is infeasible for large-n.

NNs have their own computational issues, as noted, but though trying many combinations of hyperparameters we may still have a good outcome. Also, if we manage to find a good NN fit on some classes of problems, sometimes we can tweak it to find a good NN fit on some related class (*transfer learning*).

12

THE VERDICT: APPROXIMATION BY LINES AND PLANES

PART III

OTHER ISSUES

13

NOT ALL THERE: WHAT TO DO WITH MISSING VALUES

PART IV

APPLICATIONSAPPLICATIONS

14

HANDLING SEQUENTIAL DATA

15

TEXTUAL DATA

16

IMAGE CLASSIFICATION

17

RECOMMENDER SYSTEMS

A

LIST OF ACRONYMS AND SYMBOLS

Some frequently-appearing cases.

k-NN: k-Nearest Neighbors
MAPE: mean absolute prediction error
ML: machine learning
PC: Principal Component
PCA: Principal Component Analysis
r(t): true regression function
$\widehat{r}(t)$: estimated regression function from data
SVM: Support vector machine.
$\widehat{\theta}$: estimate of *theta*

B

STATISTICS–MACHINE LEARNING TERMINOLOGY CORRESPONDENCE

- *predictor variables — features*
- *prediction — inference*
- *tuning parameter — hyperparameter*
- *observations — examples*
- *normal distribution — Gaussian distribution*
- *model fitting — learning*
- *class — label*
- *covariate — side information*
- *intercept term/constant term — bias*
- *dummy variable — one-hot coding*

UPDATES

Visit `http://borisv.lk.net/latex.html` for updates, errata, and other information.

COLOPHON

The fonts used in *The Art of Machine Learning* are New Baskerville, Futura, The Sans Mono Condensed and Dogma. The book was typeset with LaTeX 2_ε package `nostarch` by Boris Veytsman *()*.

The book was produced as an example of the package `nostarch`.