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

Library of Congress Cataloging-in-Publication Data
Uthor, A. U. and Respondent, C. O. R. and Iter, W. R.
  Pellentesque habitant morbi: Cum sociis natoque penatibus/
  A. U. Thor, C. O. R. Respondent and W. R. Iter
       p. cm.
  Includes index.
  ISBN-10: 1-23456-789-0
  ISBN-13: 123-4-56789-012-3
  1. Morbi (habitant). 2. Penatibus. I. Title.
XX2303.5.T324 2008

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 a number of 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. He was one of the four main developers of the CART method, and did pioneering work in the theory of k-Nearest Neighbor methodology. He was highly opinionated and blunt — but that is exactly the kind that makes one think, and question and defend one's assumptions.

In 2016 I was appointed to the Organizing Committee of the useR! conference, held that year at Stanford University. I started attending the weekly Statistics Department there, which I have done frequently ever since. In addition to the remarkable camaraderie I've seen 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've never met Sir David Cox, but 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 follows that philosophy, hopefully at least somewhat as effectively as Sir David.

I owe more direct thanks to Achim Zeileis, one of the authors of the **partykit** package of R routines that form excellent implementations of data par-

titioning 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.

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. The key issue is this:

> Some ML books focus on advanced mathematics, with applications playing an ancillary role. But even the applications-oriented books present ML in "cookbook" or "recipe" fashion, in a Step 1, Step 2,... format.
> In my view, this is a sad mistake. *ML is an art, not a science.* It is not a recipe. One does not have to be a math whiz to use ML effectively, but one does need to understand the concepts well.

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."

Topical Coverage and Style:

- This book is not aimed at mathematicians or computer scientists. **It's "ML for the rest of us,"** accessible to anyone who understands histograms and scatter plots. Thus, the book is not mathematical, and has virtually no formal equations.
  On the other hand, as noted, **this is not a "cookbook,"** taking a "Step 1, Step 2,..." approach, giving the reader the idea that she can simply plug her data into some neural network software and, bam!, out comes a great predictive machine. *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. Other books mention this, of course, but in a vague, unsatisfying manner. Here we explain what bias and variance really mean in any given application.
- Readers want to get started with the topic 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.

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.
- **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.

Backround 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, 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.** R plays a major supporting role and we use prominent R packages for ML throughout the book, yes. But in order to be able to *use* ML well, the reader should focus on the structure and interpretation of the ML models themselves.

Indeed, a reader who uses R packages other than those used here, or 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.

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.

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

So, make sure you have R and the **regtools** package installed on your computer, and let's proceed.[1]

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

1.1 Reminder to the Reader

Just one more point before getting into the thick of things. A quote from our Preface:

> This book is not aimed at mathematicians or computer scientists. **It's "ML for the rest of us."** Thus, book is not mathematical, and has virtually no equations.
> Equally important, though, **this is not a "cookbook."** Many presentations give the reader the idea that she can simply plug her data into some neural network software and, bam!, out comes a great predictive machine. **Those dazzling ML successes you've heard about come only after careful, lengthy tuning on the analyst's part.**
> The proper tuning of an ML algorithm in turn requires a precise, nuanced understanding of the underlying principles of ML. **Imparting this insight without confusing, distracting math is a central goal of the book.**

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.

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)
```

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 ex-

ample 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**.[2]

In R, a categorical variable has a formal class, **factor**. Some R packages automatically generate dummies from factors, but others do not, so it's important to be able to generate them 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.

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[3] — 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,

2. 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.

3. The word *predictor* is also used, and occasionally one still hears the old-fashioned term *independent variable*.

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.[4] Here the cancer type is categorical, and we have four classes, which we'd represent by four dummies.[5] So, if a person in a dataset of breast cancer patients, is of 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 of-

4. 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.

5. 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 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.

ten 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*.[6] 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 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?

6. 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.

```
> 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 Key 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,[7] 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.

7. 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."

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 geography 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. 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.

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.[8]

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?[9]

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."

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

9. 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.

- 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:

- 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**.[10] The call form, without options (they'll come in shortly), is

10. 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

```
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
- **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:

faster nearest-neighbor algorithm, and thus are preferable for large datasets. But in the interest of simplicity, we stick to the basic function.

```
> 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[11] 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.[12]

11. Actually this is the Pythagorean Theorem from high-school geometry, but we need not go into that.

12. 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.

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 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

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

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.

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 j^{th} 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 get perfect prediction by using $k = 1$.

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 k^{th}." 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.)

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.

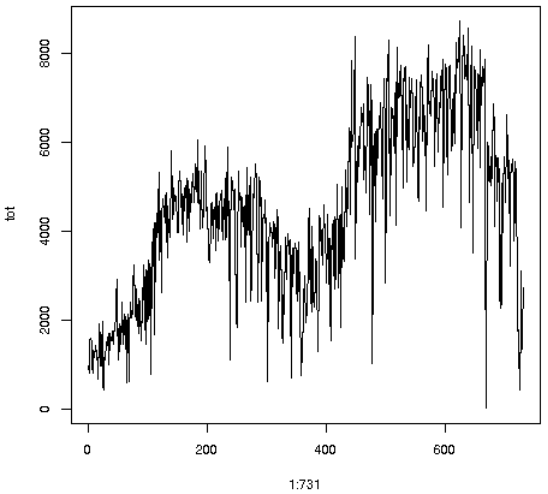

Figure 1-1: Time trends, bike data

1.11.2 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 4. 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')
```

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.

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.

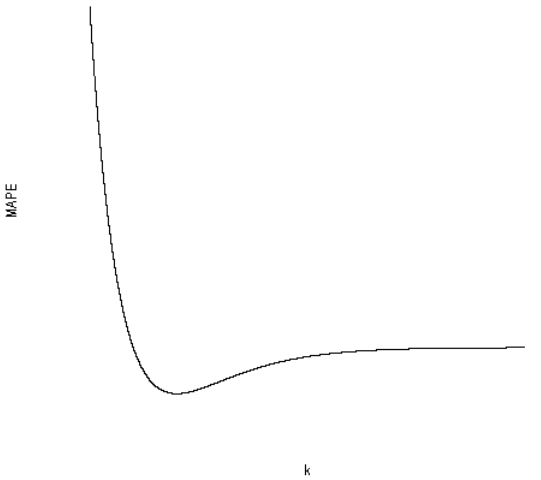

Figure 1-2: Typical mean loss vs. k

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."[13] 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.

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

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."

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.[14]

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.

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

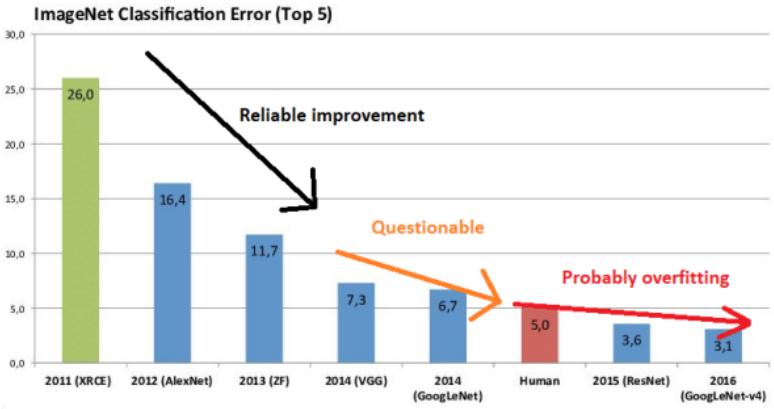

Figure 1-3: AI p-hacking

1.16 Pitfall: Dirty Data

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

```
> head(day1)
  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.

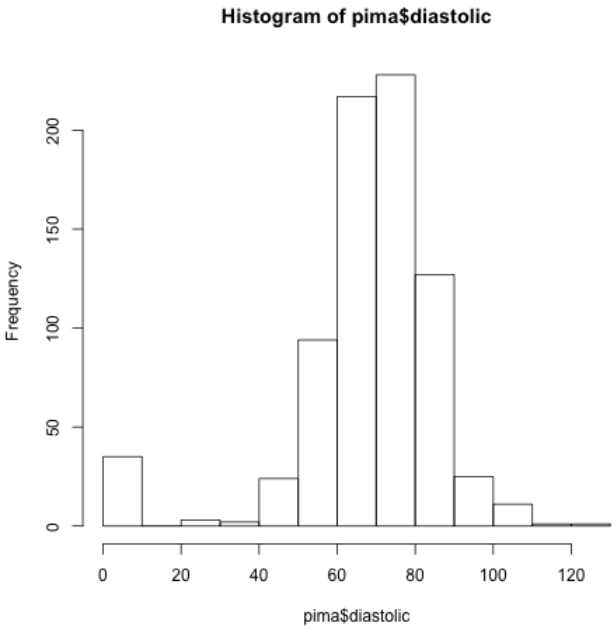

Figure 1-4: Diastolic blood pressure

Another example is the New York City Taxi Data.[15] 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?

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

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

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.

1.18.1 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

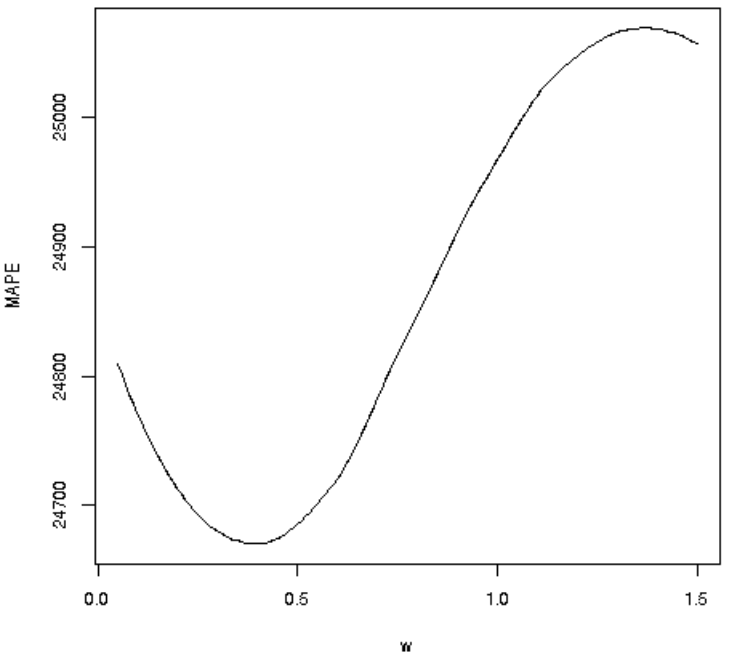

Figure 1-5: Effects of reweighting age

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.

1.19 Notes for the Mathematically Curious

In some chapters, we will include some brief notes on the mathematical aspects of the material, with references, for those with some math stat background. These are strictly supplementary in nature, and can be safely skipped.

1.19.1 More and the Bias-Variance Tradeoff

The Mean Squared Error of an estimator can easily be shown to be

$$MSE(\widehat{\theta}) = \text{bias}^2 + Var(\widehat{\theta})$$

This is shown in any math stat book, or in my text on regression and classification[16]

16. N. Matloff, *Statistical Regression and Classification: from Linear Models to Machine Learning*, CRC, 2017, sEc. 9.1.

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.

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"
```

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:

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

```
> 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 the last chapter, 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, indicated in our **factorsToDummies()** argument **omitLast=TRUE**. An exception is for categorical Y, where we will 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 are **Contract.Month-to-month** and **Contract.One year**. Numeric variables, e.g. **MonthlyCharges**, are unchanged.

But 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,]
```

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? 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
```

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.

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:

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

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.

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

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
```

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 Hypeparameters

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.

4. *https://www.datacamp.com/community/tutorials/diving-deep-imbalanced-data*
5. *https://www.fharrell.com/post/classification/*

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)
```

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"
```

6. Advocates of rebalancing sometimes imbalance can cause a parametric model, say logistic regression (Chapter 6) 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*

```
[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 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 both 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.

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 prob-

lematic when we try to use the "leaving one out" method discussed in earlier chapters.

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 fraciton 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 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

Even if $p < n$, many of the tools we'll use here will be aimed at reducing p, i.e. *dimension reduction*.

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[1] (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.

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

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

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.[2]

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.)

2. 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*
.

Leaving-one-out is in that context called n-fold cross validation, for reasons we'll see shortly.

3.2.1 Problems with Leaving-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 $n = 500,000$, then n^2 is 250 billion.

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.

Another issue to deal with is that there is mathematical theory that suggests that leave-one-out may not work well even if we have time to do the full computation with $q = n$. 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, 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 Data

Recall the dataset in Section 1.18.1, 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
```

Now let's randomly choose the training and holdout sets:[3] 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? As discussed in more detail below, 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.

(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.

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

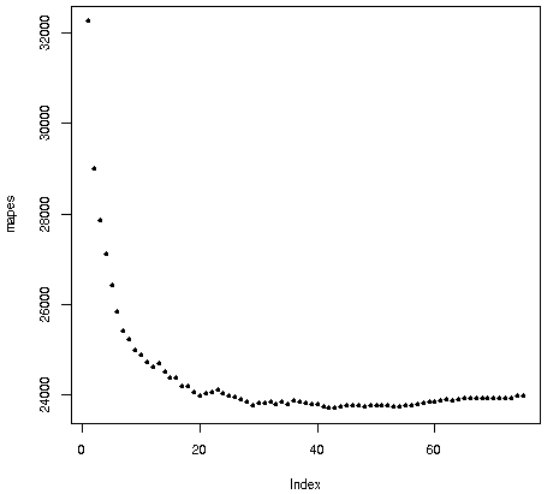

Figure 3-1: Cross-Validation, k = 1,...,75

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, abput 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.

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*.

Ironically, this process, in addition to reducing variance, also reduces bias, as follows. Some splits will underestimate the MAPE values for the full set, while others will overestimate it. By computing the MAPE estimate for all splits, and then averaging them, the biases largely cancel.

For "ordinary" applications, a single holdout split should be enough, and we will adopt this approach in this book.

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.

If we are cross-validating on three hyperparameters, the potential for problems is even higher, and the more hyperparameters, the worse the problem. Thus it makes sense to smooth the values before finding the minimizing (k, m) pair or whatever.

In the simple one-hyperparameter example in Figure 3-1, we might, for instance, replace each point by the average of itself and its left and right neighbors. In fact, that's 3-nearest neighbor regression! And moreover, since we are worried about outliers in this context here, it would be better to use the median of the neighboring points.

The **regtools** package has a function **fineTuning** for doing all this — generating all the parameter combinations, running the model on each one, and smoothing the results using k-NN. We will use this function later, but for now it would be useful to go through the steps individually.

3.3.1 Example: Million Song Dataset

As noted, with over a half million data points and only 90 features, there is probably no real need for dimension reduction here. So, in order in order to illustrate the principle, I ran an analysis on a set of 1000 randomly-chosen rows of the data frame. After splitting the data into training and holdout sets of sizes 800 and 200, **kNN()** was run with $k = 1, 2, ..., 75$ and the number of principal components s equal to 5,10,15,...,85:

```
kout <- data.frame(
   i=vector(length=340),j=vector(length=340),MAPE=vector(length=340))
nrkout <- 0
for(k in seq(5,100,5))
   for(pcs in seq(5,85,5)) {
      knnout <- kNN(xtrn[,1:pcs],ytrn,xtst[,1:pcs],k,scaleX=F)
```

```
    i <- k / 5
    j <- pcs / 5
    nrkout <- nrkout + 1
    kout[nrkout,1:2] <- c(i,j)
    kout[nrkout,3] <- mean(abs(knnout$regests - ytst))
    cat(i,j,kout[nrkout,3],'\n',sep = ' ')
  }
```

Here **xtrn** and **ytrn** were the X and Y portions of the training set, with **xtst** and **ytst** serving as X and Y in the holdout set.

Smoothing of the output was then done using 5 neighbors. Here is the result:

```
> koutSmooth <- kNN(kout[,1:2],kout[,3],NULL,5,
+    smoothingFtn=median,leave1out=TRUE)$regests
> bestK <- which.min(koutSmooth)
> kout[bestK,]
   i j  MAPE
91 6 6 8.272
```

As expected, this was somewhat less optimistic than the unsmoothed output:

```
> bk <- which.min(kout[,3])
> kout[bk,]
   i j  MAPE
36 3 2 8.224
```

Again, as stated earlier, if we had many parameters to search over, not just the two here (k and s), the difference between smoothed and unsmoothed output might be much larger, with the unsmoothed version being misleading.

3.4 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. This bizarre situation is discussed briefly on a mathematical level in Section 3.6, but what are the implications for k-NN?

It follows that distance is nearly constant, and thus k-NN, which relies on distance, is thought to fare poorly in high dimensions. If so, this conceivably could be remedied by using a weighted distance function.

At any rate, there are so many other serious issues in high dimensions that the CoD fades into the background. Many ML methods, such as the linear/logistic models and neural networks, depend on matrix inversion, and in very high dimensions numerical issues (roundoff error) can substantially reduce the accuracy. Stepwise methods, in which we add features to

our model sequentially, at each step bringing in the "best" of those that remain, can suffer from p-hacking in high dimensions, and so on.

3.5 Going Further Computationally

For very large datasets, the **data.table** package is highly recommended for large data frame-types of operations. The **bigmemory** package is for specialists who understand multicore operation. Also, for those who know SQL databases, there are several packages that interface to such data, such as **RSQLite** and **dplyr**.

3.6 For the Mathematically Curious

It was stated above that in high dimensions, i.e. settings with many features, all the points are approximately equidistant. What does that really mean?

How does this bizarre result come about? Consider an i.i.d. sequence $X_1, X_2, X_3, ..., X_n$ and consider their sum T. The ratio of the standard deviation of T to the mean of T is $O(1/\sqrt{n})$. For large n, that quantity is small, so T is nearly constant. If we take the X_i to be terms in the sum of squares in a distance, it

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.

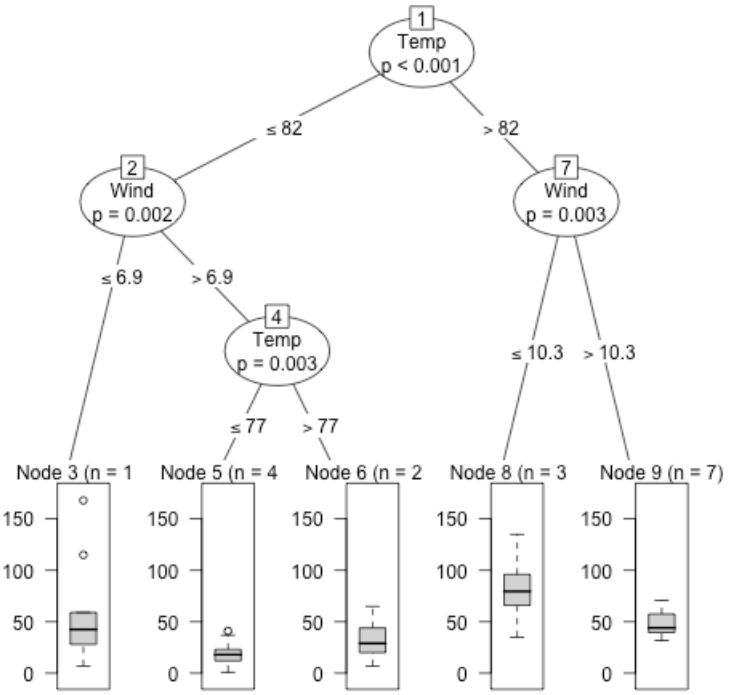

Figure 4-1: Sample plot from ***ctree***

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
\bigskip

\section{Example:  New York City Taxi Data}
\label{taxi}

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.

\subsection{Data Preparation}

First, let's take a look around:

\bigskip
\begin{lstlisting}
> 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.

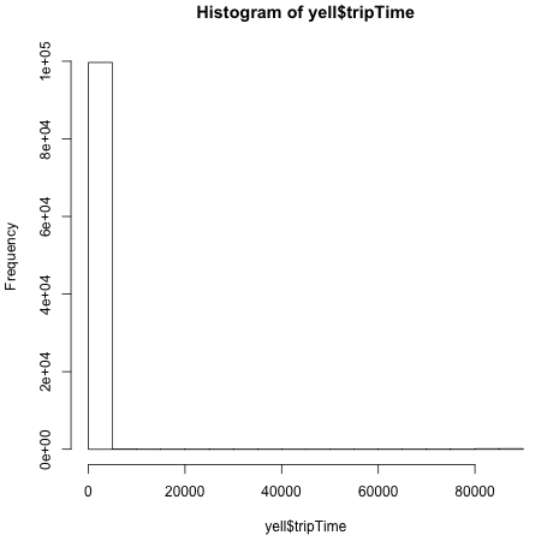

Figure 4-2: Taxi data, trip time

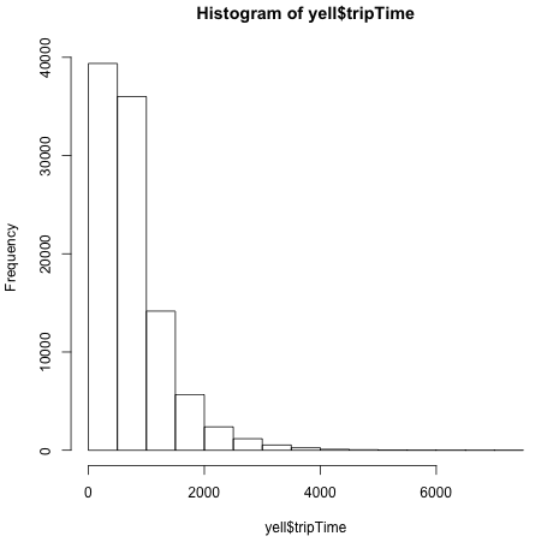

Figure 4-3: Taxi data, modified, trip time

4.2.9 Tree-Based Analysis

OK, ready to go. Let's set up for cross-validation, say with 5000 in our holdout set:[4]

```
> tstidxs <- sample(1:nrow(yell),5000)
> yelltst <- yell[tstidxs,]
> yelltrn <- yell[-tstidxs,]
> ctout <- ctree(tripTime ~ .,data=yelltrn)
> preds <- predict(ctout,yelltst)
> mean(abs(preds-yelltst$tripTime))
[1] 200.1927
```

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.

The resulting tree is too large and complex to call **plot()**, but let's get a sample of it, say node 559:

```
> nodes(ctout,559)
[[1]]
559) trip_distance <= 13.59; criterion = 1, statistic = 294.386
  560) PULocationID == {3, 13, 20, 25, 32, 37, 41, 42, 48, 52, 61, 62, 68,
  69, 81, 88, 90, 97, 100, 101, 107, 121, 125, 130, 142, 143, 144, 152,
  158, 161, 163, 174, 186, 188, 193, 209, 211, 215, 230, 231, 234, 236,
  238, 239, 244, 246, 249, 252, 256, 261, 264}; criterion = 1, ...
  560) PULocationID == {4, 10, 14, 18, 28, 33, 43, 45, 50, 55, 56, 65, 75,
  79, 87, 93, 113, 114, 132, 137, 138, 140, 141, 146, 148, 151, 162, 164,
  166, 170, 182, 196, 197, 203, 226, 229, 232, 233, 237, 241, 250, 258,
  260, 262, 263}
559) trip_distance > 13.59
  563) PULocationID == {14, 23, 38, 39, 43, 61, 76, 77, 116, 143, 144,
  146, 170, 206, 233, 239, 254}; criterion = 1, statistic = 223.767
```

So at that particular node, we go left or right, depending on whether the trip distance is below 13.59 or not. In either case, we then look at the pickup location. If we had gone left for trip distance, we now go left again if the pickup location has ID 3, 13, 20 and so on.

4. The run shown below was done using the **party** package, a predecessor of **partykit**. The documentation for the latter recommends using the former if the latter produces an error message "Too many levels." This will often be the case when working with categorical variables with a large number of categories, e.g. a large number of different pickup and dropoff locations.

4.3 Example: Forest Cover Data

Another UCI dataset, Covertype, aims to "[predict] forest cover type from cartographic variables only."[5] 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.

4.3.1 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. Easily, fixed, of course:

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

4.3.2 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. To keep things simple, then, let's dispense with cross-validation.

```
> ctout <- ctree(V55 ~ .,data=cvr,controls=ctree_control(maxdepth=6))
> preds <- predict(ctout,cvr)
> mean(preds != cvr$V55)
[1] 0.2974
```

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

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

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.

4.3.3 Diagnostic: the Confusion Matrix

Let's take a closer look:

```
> table(preds,cvrtst$V55)
preds    1    2    3    4    5    6    7
    1 1338  639    0    0    0    0   78
    2  412 1790   53    0   73   42    1
    3    0   26  265   37    3   86    0
    4    0    0    2    1    0    0    0
    5    0    0    0    0    2    0    0
    6    0    0    0    0    0    1    0
    7   33    2    0    0    0    0  116
```

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 1 of the 129 instances of class 6 was correctly identified, though again we may be able to improve on this with better values of the hyperparameters, or for that matter, a method called *boosting* that we'll bring in later in this chapter.

4.4 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 criterion: In forming the tree, and some node is created, should we split that node and thus make a further subtree? And if so, how should we do the split? Other hyperparameters may control the maximum depth of the tree, the minimum node size, and so on.

Before continuing, let's get an overview of the process. The splitting process in **partykit** works by performing nominal significance tests.[6] The feature, if any, most "significantly" related to Y for the data within a node will be used for the basis of splitting this node. If the relation with Y is not "significant," we do not split the node.

4.4.1 Hyperparameters in the party Package

The full call form of **ctree** is

```
ctree(formula, data = list(), subset = NULL, weights = NULL,
    controls = 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 **controls**. 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)
```

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.[7]
- **alpha:** The significance level for the tests.

4.4.2 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. Many statisticians these days frown on the use of significance testing. But these are not formal tests, just mechanisms to decide on which feature to split.
7. 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.

4.4.3 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?

```
> 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.

4.5 Diagnostics: Checking Per-Node Accuracy Levelsl

4.6 The Function regtools::fineTuning

Say we decide to use the 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.

It will be nice to have a systematic way to try all those combinations, and to tally the results. The function **fineTuning()** in the **regtools** packages does this in more:[8]

- It will automate our cross-validation process.
- It will do fit-and-predict within the cross-validation process.
- It will record the results, displaying them at the end.
- It will also smooth those results.

4.6.0.1 Example: Wisconsin Breast Cancer Data

Our main call will be

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

```
fineTuning(dataset=wpbc,
   pars=list(minsp=c(5,10,20,50),maxd=c(2,5,8),alpha=c(0.01,0.05,0.20)),
   regCall=theCall,nCombs=9,nTst=50,nXval=5,k=3)
```

The first few arguments are clear. What about **regCall**?

In orer 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. The function will call whatever function we give it, which is:

```
theCall <- function(dtrn,dtst,cmbi) {
   ctout <- ctree(status ~ .,dtrn,
      control=ctree_control(minsplit=cmbi[1],maxdepth=cmbi[2],alpha=cmbi[3]))
   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 cases **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 number 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.

So, here we go:

```
> fineTuning(dataset=wpbc,
   pars=list(minsp=c(5,10,20,50),maxd=c(2,5,8),alpha=c(0.01,0.05,0.20)),
   regCall=theCall,nCombs=9,nTst=50,nXval=5,k=3)
$outdf
  minsp maxd alpha meanLoss  smoothed
1    20    5  0.01    0.772 0.7266667
2    50    5  0.05    0.752 0.7280000
3    50    5  0.20    0.740 0.7306667
4    50    8  0.05    0.816 0.7306667
5     5    8  0.20    0.732 0.7320000
6    20    8  0.20    0.732 0.7320000
7    10    8  0.20    0.752 0.7320000
8    10    8  0.01    0.756 0.7333333
9    50    2  0.05    0.736 0.7346667
```

```
10    10    8  0.05    0.716 0.7373333
11    50    5  0.01    0.792 0.7373333
12    10    5  0.05    0.728 0.7386667
13    20    2  0.05    0.776 0.7386667
14    10    2  0.01    0.728 0.7400000
15    20    2  0.01    0.744 0.7400000
16     5    2  0.05    0.692 0.7413333
17     5    8  0.01    0.756 0.7413333
18    50    2  0.20    0.720 0.7426667
19     5    2  0.01    0.748 0.7426667
20    20    5  0.05    0.752 0.7426667
21    20    8  0.01    0.764 0.7426667
22    50    8  0.20    0.744 0.7453333
23    50    8  0.01    0.752 0.7480000
24     5    8  0.05    0.752 0.7520000
25    50    2  0.01    0.784 0.7520000
26    10    2  0.05    0.808 0.7520000
27    20    8  0.05    0.780 0.7533333
28     5    5  0.01    0.760 0.7546667
29    10    5  0.01    0.736 0.7680000
30     5    5  0.05    0.700 0.7706667
31    20    5  0.20    0.704 0.7746667
32    10    5  0.20    0.732 0.7746667
33     5    5  0.20    0.768 0.7746667
34    20    2  0.20    0.708 0.7773333
35     5    2  0.20    0.720 0.7773333
36    10    2  0.20    0.760 0.7773333
...
```

The unsmoothed accuracies are given in the **meanLoss** column. These are proportions of correct predictions, so larger is better.

Note that the unsmoothed numbers do seem to be overly extreme, either optimistically or pessimistically. There is a hyperparameter combinatins with reported accuracy 0.816, , but the last 3 rows are best after smoothing. They have only 2 levels in the tree and a liberal 0.20 value for **alpha**; once those are set, the minimum node size doesn't matter.

The smoothing helps compensate for our small n here.

4.7 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 day 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. Those changes may in turn affect lower 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 high, so let's generate more trees!" But how?

4.7.1 Bagging: Generating New Trees by Resampling

A handy tool from modern statistics is the *bootstrap*. Starting with our original data, once again considered a sample from some population, we'll generate many new samples from the original one. We do this by randomly sampling n of our n data points — *with* replacement. Typically we'll get a few duplicates.

4.7.1.1 The Method

For each of the new samples, we will apply whatever analytical tool we're interested in. In this case, that is decision trees. By resampling, then, we'll get n new trees, thus decreasing variance, as desired.

Say we have a new case to be predicted. We will then *aggregate* the 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 **??**, 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.[9]

4.7.1.2 The cforest Function in partykit

The **partykit** package includes an implementation of bagging, **cforest**

4.7.2 Boosting: Tweaking a Tree

AdaBoost is the best off-the-shelf classifier in the world — CART co-inventor (and not the inventor of AdaBoost) Leo Breiman, 1996
There is no such thing as a free lunch — old economics saying

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

We will cover this vital topic in Chapter 7, but giving an overview here will deepen insight into the bias vs. variance issue with regard to trees.

Imagine a classification problem, just two classes, so $Y = 1, 0$, 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 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.

So the idea of boosting is to tweak the original tree, then tweak that new tree and so on. After generating m trees — m is a hyperparameter — then make a "supertree" from a weighted average of the individual trees. In a regression situation, to predict a new case with a certain X value, we plug that value into all the trees, yielding m 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.)

5

THE VERDICT: NEIGHBORHOOD-BASED MODELS

- Introduce some general issues that will arise throughout the book.
- Introduce the k-NN and tree methods, while at the same time using this method as an example of those general issues.

In light of that second goal, it's time to take stock of the k-NN method. Advantages:

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

PART II

METHODS BASED ON LINES AND PLANES

6

PARAMETRIC METHODS: LINEAR AND GENERALIZED LINEAR MODELS

6.1 The Local-Linear k-Nearest Neighbors Method

7

TWEAK THE MODEL: BOOSTING METHODS

REPRISE THE BRIEF OVERIVIEW FROM THE TREES CHAPTER

7.0.0.1 Implementation: AdaBoost

The first proposal made for boosting was *AdaBoost*. The tweaking involved assigning weights to the points in our training set, updating them with each new tree. 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,m) {
   # uniform weights to begin
   wts <- rep(1/n,n)
   trees <- list()
   alpha <- vector(length=m)  # weights for the trees
   for(treeNum in 1:m) {
      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:m) {
      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 the later trees are given more weight.

7.0.0.2 Gradient Boosting

7.0.0.3 The gbm Package

USE AND PLOT THE X-VAL OPTION

7.0.0.4 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 m is set too large, producing too many trees, we may overfit.

In other words, though Breiman had a point (especially in saying "off the shelf," meaning usable without a lot of hyperparameters to worry about), that old saying about "no free lunch" applies as well. As always, cross-validation is one measure we can consider for avoiding this problem.

7.1 Diagnostics: Not All Leaves Are Created Equal

USE MY IDEA OF c() TO GET ALL THE OBS IN A NODE, FINDING THE ERROR ON EACH, THEN OVERALL RATE; MAKE THIS A SUPPLEMENTAL UTILITY FOR partykit; FIND THE ERROR RATES, WHICH MAY DIFFER WIDELY FROM NODE TO NODE; WHAT TO DO?

8

THE VERDICT: PARAMETRIC MODELS

9

LINEAR MODELS ON STEROIDS: NEURAL NETWORKS

10

A BOUNDARY APPROACH: SUPPORT VECTOR MACHINES

11

THE VERDICT: APPROXIMATION BY LINES AND PLANES

PART III

OTHER ISSUES

12

CUTTING THINGS DOWN TO SIZE: REGULARIZATION

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
$\widehat{\theta}$: 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 *(2008/06/06 v1.3 Typesetting books for No Starch Press)*.

The book was produced as an example of the package `nostarch`.