

Figure FM.1 cover

FM.1 Practical Data Science with R

Techniques to ensure the success of data science projects

Nina Zumel and John Mount

FM.2 Dedication

To our parents: Olive and Paul Zumel; Peggy and David Mount.

FM.3 Foreword

This is the book for you if you want to work as a data scientist or already do. This book will demonstrate the methods, habits and interactions of successful data scientists and data science projects. For speed and brevity we will limit ourselves to tools and packages from the R statistical programming environment. We have learned a lot from the many different people and want to distill down and share the best practices we have seen. Some chapters are elementary and some are advanced, but all chapters contain things we wish we had learned a lot earlier in our careers.

Throughout this book we are going to emphasize scientific principles such as repeatability of experiments. We will also emphasize software engineering principles such as automation of steps. We see scientific principles and software engineering principles as being co-equal ways to think about data science projects. You automate steps because you will have to repeat them and you can repeat steps because of your version control and automation.

FM.4 Preface

This is the book we wish we had available to hand out to clients and peers. Its purpose is to explain the best parts of statistics, computer science and machine learning that are relevant to data science. Most data scientists have arrived recently from some other field, so can still benefit in being reminded of some of the best tools from the many fields that contribute to data science. Our challenge became: could we write on so many topics in the style of a concrete and useful manual? We feel by concentrating on fully worked exercises on real data (in the end this book works completely through over 10 significant datasets) we are able to both illustrate what we are really wanted to teach and include all the preparatory steps necessary in any real world project.

FM.5 Acknowledgements

Thank you to the many reviewers and others who contributed comments and corrections. An especial thanks to our development editor Cynthia Kane.

FM.6 About this book

FM.6.1 What is data science?

The statistician William S. Cleveland defined data science as an interdisciplinary field larger than statistics itself. We define data science as managing the process that can transform hypotheses and data into actionable predictions. Typical predictive analytic goals include: predicting who will win an election, what products will sell well together, which loans will default, or which advertisements will be clicked on. The data scientist is responsible for all of: acquiring the data, managing the data, choosing the modeling technique, writing the code and verifying the results.

Data science draws on tools from the empirical sciences, statistics, reporting, analytics, visualization, business intelligence, expert systems, machine learning, databases, data warehousing, data mining, and big data. It is because we have so many tools that we need a discipline that covers them all. What distinguishes data science itself from the tools and techniques is the central goal of deploying effective decision-making models to a production environment.

Data science is often a "second calling." Many of the best data scientists we meet started as programmers, statisticians, business intelligence analysts or scientists. By adding a few more techniques to their repertoire they became excellent data scientists. That observation drives this book: we will introduce the practical skills needed by the data scientist by concretely working through all of the common project steps on real data. Some steps you will know better than we do, some you will pick up quickly, and some you may need to research further.

Much of the theoretical basis of data science comes from statistics. However, data science as we know it is very much influenced by technology and software engineering methodologies, and has largely evolved in heavily computer science and information technology driven groups. We can call out some of the engineering flavor of data science by listing some famous examples:

- Amazon's product recommendation systems.
- Google's advertisement valuation systems.
- LinkedIn's contact recommendation system.
- Twitter's trending topics.
- Walmart's consumer demand projection systems.

These systems share a lot of features:

- All of these systems are *built off large data sets*. That is not to say they are all in the realm of "big data." But none of them could have been accomplished from only small data sets. To manage the data, these systems require concepts from computer science: database theory, parallel programming theory, streaming data techniques, and data warehousing.
- Most of these systems are *online or live*. Rather than producing a single report or analysis, the data science team deploys a decision procedure or scoring procedure to either directly make decisions or directly show results to a large number of end users. The production deployment is the last chance to get things right, as the data scientist can not always be around to explain defects.
- All of these systems are *allowed to make mistakes* at some non-negotiable rate.
- None of these systems are concerned with *cause*. They are successful when they find useful correlations and are not held to correctly sorting cause from effect.

This book will teach the principles and tools needed to build systems like these. We want to teach the common tasks, steps and tools used to successfully deliver such projects. Our emphasis is on the whole process, project management, working with others and presenting results to non-specialists.

FM.6.2 Chapter roadmap

This book will cover the following.

- Managing the data science process itself. The data scientist must have the ability to measure and track their own project.
- Applying many of the most powerful statistical and machine learning techniques used in data science projects. Think of this book as a series of explicitly worked exercises in using the programming language "R" to perform actual data science work.
- Preparing presentations for the various stakeholders: management, users, deployment team and so on. You must be able to explain your work in concrete terms to mixed audiences using words in their common meanings, *not in whatever technical definition is insisted on in a given field*. You can't get away with just throwing data science project results over the fence.

This book is example driven. We supply prepared example data (

<https://github.com/WinVector/zmPDSwR> with R code and links back to original sources) and the code to produce all results and almost all graphs found in the book (<https://github.com/WinVector/zmPDSwR/raw/master/CodeExamples.zip>, a "warts and all" distribution to ensure all steps and workarounds are demonstrated). Work through the examples with us and you will have worked through at least 10 significant data science projects.

The material is organized as follows:

- *Chapter 1 The data science process*: how to work as a data scientist.
- *Chapter 2 Loading data into R*: how to start working with data in R.
- *Chapter 3 Exploring Data*: what to first look for in data.
- *Chapter 4 Managing Data*: how to prepare data for analysis.
- *Chapter 5 Choosing and Evaluating Models*: a dictionary of business needs to evaluation and modeling techniques.
- *Chapter 6 Using Memorization Methods*: building simple models.
- *Chapter 7 Using Linear and Logistic Regression*: building models with additive structure.
- *Chapter 8 Using Unsupervised Methods*: what to do in projects where there is no labeled training data available.
- *Chapter 9 Exploring Advanced Methods*: fixing modeling problems.
- *Chapter 10 Documentation and Deployment*: how to document and deploy your models.
- *Chapter 11 Presentation*: how to present to different audiences.
- *Chapter 12 Further reading*: topics for further study.

The material is organized in terms of goals and task, bringing in tools as they are needed. Over ten substantial projects are worked through in this book.

FM.6.3 Audience

PREREQUISITES

Good prerequisites for this book are both the statmethods website <http://www.statmethods.net> and the book "R in Action" by Robert Kabacoff (now available in a second edition: <http://www.manning.com/kabacoff2/>). For statistics we recommend "Statistics, 4th Edition" by David Freedman, Robert Pisani, and Roger Purves. We don't expect our readers to be experts in statistics or R, but they may want access to these references to brush up on important topics or read in parallel with this book. In general what we expect from our ideal reader is:

- An interest in working examples. By working through the examples you will learn at least one way to perform all steps of a project. You must be willing to attempt simple

scripting and programming to get the full value of this book. For each example we work you should try variations and expect both some failures (where your variations do not work) and some successes (where your variations outperform our example analyses).

- Some familiarity with the R statistical system and the will to write short scripts and programs in R. We recommend a few good books in our appendices. We will work specific problems in R, to understand what is going on you will need to run the examples and read additional documentation to understand variations of the commands we did not demonstrate.
- Some experience with basic statistical concepts such as probabilities, means, standard deviations and significance. We will introduce these concepts as they are needed, but you may need to read additional references as we work through examples. We will define some terms and refer to some topic references and blogs where appropriate. However, we expect you will have to perform some of your own internet searches on some topics.
- A computer (OSX, Linux or Windows) to install R and other tools on. Internet access to download tools and data sets. We strongly suggest working through the examples, examining the R `help()` on various methods and following up some of the additional references.

WHAT IS NOT IN THIS BOOK?

- *This book is not an R manual.* We are going to use R to concretely demonstrate the important steps of data science projects. We will teach enough R for you to work through the examples, but a reader unfamiliar with R will want to refer to Appendix XRF:appendix_A:xAppendixR as well as to the many excellent R books and tutorials already available.
- *This book is not a set of case studies.* We are going to emphasize methodology and technique. Example data and code is given only to make sure we are giving concrete usable advice.
- *This book is not a "big data" book.* We feel most significant data science occurs at a database or file manageable scale (often larger than memory, but still small enough to be easy to manage). Valuable data that maps measured conditions to dependent outcomes tends to be expensive to produce and that tends to bound its size. For some report generation, data mining and natural language processing you will have to move into the big data regime.
- *This is not a theoretical book.* We are not going to emphasize the absolute rigorous theory of any one technique. The goal of data science is to be flexible, have a number of good techniques at hand and be willing to research a technique more deeply if it appears to apply to the problem at hand. We are going to prefer R-code notation even in our text for concepts over beautifully typeset equations as the R-code can be directly used.
- *This is not a machine learning tinker's book.* We are going to emphasize methods that are *already implemented in R*. We will discuss the theory of operation of these methods: what they are trying to do and what are the critical diagnostics. We will not discuss how to implement them (even when implementation is easy) as they are already implemented.

FM.6.4 Code conventions and downloads

Throughout we are going to write about concepts (both statistics and machine learning), include concrete code and explore partnering with and presenting to non-specialists. We hope when you don't find one of these topics novel that we are able to share a wrinkle on one or two of the other topics that you may not have thought about recently. We encourage you to try the example R code as you read the text; even when we are discussing fairly abstract aspects of data science we will illustrate examples with concrete data and code. We are arranging topics in book in an order that we feel increases understanding.

In our examples any prompt such as > or \$ are to be ignored. Inline results are usually prefixed by R's comment character #.

All the examples (with informal notes) are publicly available from <https://github.com/WinVector/zmPDSwR>. You can explore this repository online or clone it onto your own machine. All code extracts from the book are also in the zip file <https://github.com/WinVector/zmPDSwR/raw/master/CodeExamples.zip> (copying code from the zip file can be easier than copying and pasting from the book).

FM.6.5 Software or hardware requirements

We suggest the reader be prepared to work through our examples. To do so they will need some sort of computer (Linux, OSX or Windows) and to install the following software (installation described in Appendix XRF:appendix_A:xAppendixR. All of the software we recommend is fully cross platform (Linux, OSX or Windows), freely available and usually open source.

We suggest at least installing:

- R itself <http://cran.r-project.org>.
- Various packages from cran (installed by R itself using the `install.packages()` command and activated using the `library()` command).
- git for version control <http://git-scm.com>.
- RStudio for an integrated editor, execution and graphing environment <http://www.rstudio.com>.
- A bash shell for system commands. This is built in for Linux and OSX and can be added to Windows by installing Cygwin <http://www.cygwin.com>. We do not write any scripts, so an experienced Windows shell user can skip installing Cygwin if they are able to translate commands into the appropriate windows.

FM.7 About the authors

FM.7.1 Nina Zumel

The first author, Nina Zumel, has worked as a scientist at one the largest independent, nonprofit research institutes. She has worked as chief scientist of a price optimization company and founded a contract research company. Nina Zumel is now a principal consultant at Win-Vector LLC. She can be reached at nzumel@win-vector.com.



Figure FM.2 Nina Zumel

FM.7.2 John Mount

The second author, John Mount, has worked as a computational scientist in biotechnology, a stock trading algorithm designer and managed a research team for a major online shopping site. John Mount is now a principal consultant at Win-Vector LLC. He can be reached at jmount@win-vector.com.



Figure FM.3 John Mount

FM.8 External links section

FM.9 front figures

FM.1 FM.2 FM.3

Part 1

In Part 1 we concentrate on the most essential tasks in data science: working with your partners, defining your problem, and examining your data.

The data science process



This chapter will cover

- The roles in a data science project
- The stages of a data science project
- Setting expectations for a new data science project

The data scientist is responsible for guiding a data science project from start to finish. Success in a data science project comes not from access to any one exotic tool, but from having quantifiable goals, good methodology, cross-discipline interactions, and a repeatable workflow.

This chapter will walk through what a typical data science project looks like: the kinds of problems you encounter, the types of goals you should have, the tasks that are likely, and what sort of results are expected.

1.1 The roles in a data science project

Data science is not performed in a vacuum. It's a collaborative effort that draws on a number of roles, skills, and tools. Before we talk about the process itself, let's look at the roles that must be filled in a successful project. Project management has been a central concern of software engineering for a long time, so we can look there for guidance. In defining the roles here, we've taken some ideas from Fred Brooks's "surgery team" ideas of software development and also from the agile software development paradigm.

1.1.1 Project roles

Let's look at a few recurring roles in a data science project:

Table 1.1 Data science project roles and responsibilities

Role	Responsibilities
The project sponsor	Represents the business interests; champions the project.
The client	Represents end users' interests; domain expert.
The data scientist	Sets and executes analytic strategy; communicates with sponsor and client
The data architect	Manages data and data storage; sometimes manages data collection.
Operations	Manages infrastructure; deploys final project results.

Sometimes, these roles may overlap. Some roles—in particular the client, data architect, and operations—are often filled by people who are not on the data science project team, but are key collaborators.

THE PROJECT SPONSOR

The most important role in a data science project is the project sponsor. The sponsor is the person who wants the data science result; generally they represent the business interests. The sponsor is responsible for deciding whether the project is a success or failure. The data scientist may fill the sponsor role for their own project if they feel they know and can represent the business needs, but that's not the optimal arrangement. The ideal sponsor meets the following condition: if they're satisfied with the project outcome, then the project is by definition a success. *Getting sponsor sign-off becomes the central organizing goal of a data science project.*

TIP**Keep the sponsor informed and involved**

It's critical to keep the sponsor informed and involved. Show them plans, progress, and intermediate successes or failures in terms they can understand. A good way to guarantee project failure is to keep the sponsor in the dark.

To ensure sponsor sign-off, you must get clear goals from them through directed interviews. You attempt to capture the sponsor's expressed goals as quantitative statements. An example goal might be: "identify 90% of accounts that will go into default at least two months before the first missed payment with a false positive rate of no more than 25%." This is a precise goal that allows you to check in parallel if meeting the goal is actually going to make business sense and whether you have good enough data and tools to achieve the goal.

THE CLIENT

While the sponsor is the role who represents the business interest, the client is the role who represents the model's end users' interests. Sometimes, the sponsor and the client may be the same person. Again, the data scientist may fill the client role if they can weight business trade-offs, but this isn't ideal.

The client's role in the project is more hands-on than the sponsor; they are the interface between the technical details of building a good model, and the day-to-day work process into which the model will be deployed. They aren't necessarily mathematically or statistically sophisticated, but are familiar with the relevant business processes, and serve as the domain expert on the team. In the loan application example that we discuss later in this chapter, the client may be a loan officer, or someone who represents the interests of loan officers.

As with the sponsor, you should keep the client informed and involved. Ideally you'd like to have regular meetings with them, to keep your efforts aligned with the needs of the end users. Generally, the client belongs to a different group in the organization, and has other responsibilities beyond your project. Keep meetings focused; present results and progress in terms they can understand, and take their critiques to heart. If the end users can't or won't use your model, then the project is not a success, in the long run.

THE DATA SCIENTIST

The next role in a data science project is the data scientist, who is responsible for taking all steps to make the project succeed, including setting the project strategy and keeping the client informed. They design the project steps, pick the data sources, and pick the tools to be used. Since they pick the techniques that will be tried, they have to know a lot about statistics and machine learning. They're also responsible for the project planning and tracking, though they may do this with a project management partner.

At a more technical level, the data scientist also looks at the data, performs statistical tests and procedures, applies machine learning models, and evaluates results—the science portion of data science.

THE DATA ARCHITECT

The data architect is responsible for all of the data and its storage. Often this role is outside of the data science group and is a database administrator or architect. The data architect is often managing a data warehouse for many different projects, and they may only be available for quick consultation.

OPERATIONS

The operations role is critical both in acquiring data and delivering the final results. The person filling this role usually has operational responsibilities outside of the data science group. For example, if you're deploying a data science result that affects how products are sorted on an online shopping site, then the person responsible for running the site will have a lot to say on how such a thing can be deployed. This person will likely have constraints on response time, programming language, or data size that you need to respect in deployment. The person in the operations role may already be supporting your sponsor or your client, so they are often easy to find (though their time may be already very much in demand).

1.2 The stages of a data science project

The ideal data science environment is one that encourages feedback and iteration between the data scientist and all other stakeholders. This is reflected in the life cycle of a data science project. Even though this book, like any other discussion of the data science process, breaks up the cycle into distinct stages, in reality the boundaries between the stages are fluid, and the activities of one stage will often overlap those of other stages. Often, you will loop back and forth between two or more stages before moving forward in the overall process. This is shown in figure 1.1.

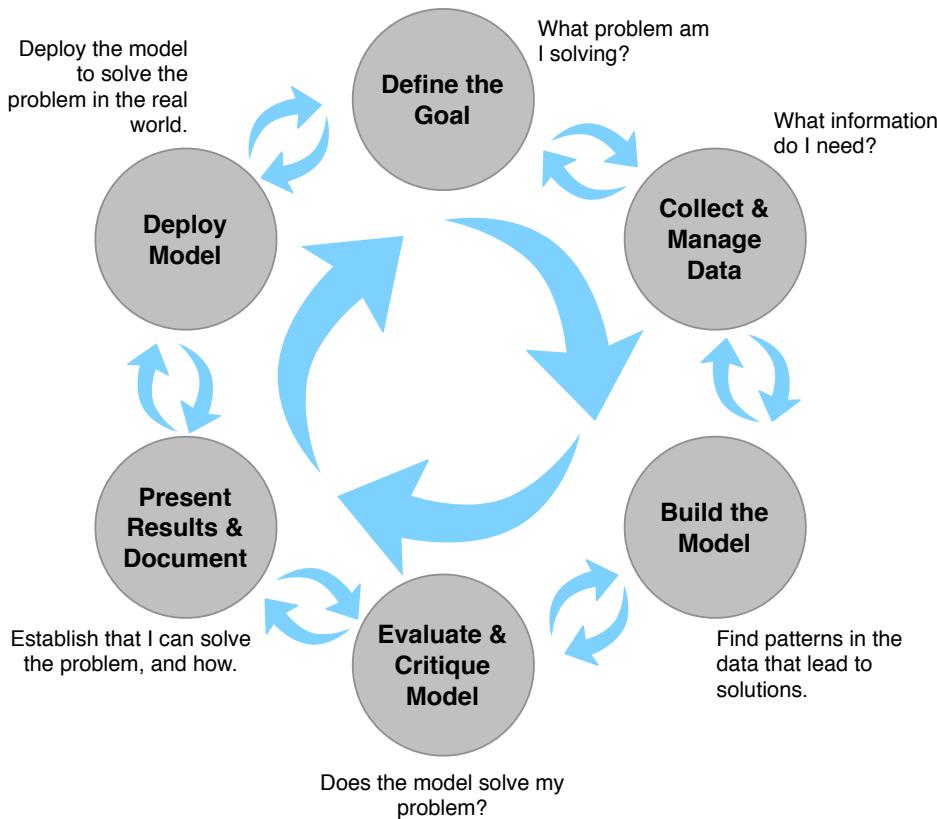


Figure 1.1 The lifecycle of a data science project: loops within loops.

Even after you complete a project and deploy a model, new issues and questions can arise from seeing that model in action. The end of one project may lead forward to another follow-up project.

Let's talk about the different stages shown in figure 1.1. To make things concrete, suppose you're working for a German bank.¹ The bank feels that it is losing too much money to bad loans, and wants to reduce its losses. This is where your data science team comes in.

Footnote 1 For this chapter, we will use a credit dataset donated by Professor Dr. Hans Hofmann to the UCI Machine Learning Repository in 1994. We've simplified some of the column names for clarity. The dataset can be found at [http://archive.ics.uci.edu/ml/datasets/Statlog+\(German+Credit+Data\)](http://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data)). We will show how to load this data and prepare it for analysis in Chapter XRF:chapter_2:chStartingWithRandData. Note that the German currency at the time of data collection was the Deutsch mark (DM).

1.2.1 Defining the goal

The first task in a data science project is to define a measurable and quantifiable goal. At this stage, learn all that you can about the context of your project.

- Why do the project sponsors want the project in the first place? What do they lack, and what do they need?

- What are they doing to solve the problem now, and why isn't that good enough?
- What resources will you need: what kind of data, how much staff, will you have domain experts to collaborate with, what are the computational resources?
- How do the project sponsors plan to deploy your results? What are the constraints that have to be met for successful deployment?

Let's come back to our loan application example. The ultimate business goal is to reduce the bank's losses from bad loans. Your project sponsor envisions a tool to help loan officers more accurately score loan applicants, and so cut down the number of bad loans given out. At the same time, it's important that the loan officers feel that they have final discretion on the loan approvals.

Once you and the project sponsor and other stakeholders have established preliminary answers to these questions, you and they can start defining the precise goal of the project. The goal should be specific and measurable: not

We want to get better at finding bad loans.

but instead

We want to reduce our rate of loan charge-offs by at least 10%, using a model that predicts which loan applicants are likely to default.

A concrete goal begets concrete stopping conditions and concrete acceptance criteria. The less specific the goal, the likelier that the project will go unbounded, because no result will be "good enough." If you don't know what you want to achieve, you don't know when to stop trying—or even what to try. When the project eventually terminates—because either time or resources run out—no one will be happy with the outcome.

This doesn't mean that more exploratory projects aren't needed at times: "Is there something in the data that correlates to higher defaults?" or "Should we think about reducing the kinds of loans we give out? Which types might we eliminate?" In this situation, you can still scope the project with concrete stopping conditions, such as a time limit. The goal is then to come up with candidate hypotheses. These hypotheses can then be turned into concrete questions or goals for a full-scale modeling project.

Once you have a good idea of the project's goals, then you can focus on collecting data to meet those goals.

1.2.2 Data collection and management

This step encompasses identifying the data you need, exploring it, and conditioning it to be suitable for analysis. This stage is often the most time-consuming step in the process. It's also one of the most important.

- What data is available to me?
- Will it help me solve the problem?
- Is it enough?
- Is it of good enough quality?

Imagine that for your loan application problem, you've collected a sample of representative loans from the last decade (excluding home loans). Some of the loans have defaulted; most of them (about 70%) not. You've collected a variety of attributes about each loan application, as listed in table 1.2.

Table 1.2 Loan data attributes

- Status.of.existing.checking.account (*at time of application*)
- Duration.in.month (*loan length*)
- Credit.history
- Purpose (*car loan, student loan, etc.*)
- Credit.amount (*loan amount*)
- Savings.Account.or.bonds (*balance/amount*)
- Present.employment.since
- Installment.rate.in.percentage.of.disposable.income
- Personal.status.and.sex
- Cosigners
- Present.residence.since
- Collateral (*car, property, etc.*)
- Age.in.years
- Other.installment.plans (*Other loans/lines of credit—the type*)
- Housing (*own, rent, etc.*)
- Number.of.existing.credits.at.this.bank
- Job (*employment type*)
- Number.of.dependents
- Telephone (*do they have one*)
- Good.Loan (*dependent variable*)

In our data Good.Loan takes on two possible values: *GoodLoan* and *BadLoan*.

For the purposes of this discussion, we assume that a *GoodLoan* was paid off, and a *BadLoan* defaulted.

As much as possible, try to use information that can be directly measured, rather than information that is inferred from another measurement. For example, you might be tempted to use income as a variable, under the reasoning that a lower income implies more difficulty paying off a loan. The ability to pay off a loan is more directly measured by considering the size of the loan payments relative to the borrower's disposable income. This information is more useful than income alone; we have it in our data as the variable *Installment.rate.in.percentage.of.disposable.income*.

This is the stage where you conduct initial exploration and visualization of the data. You'll also be cleaning the data: repairing data errors and transforming variables, as needed. In the process of exploring and cleaning the data, you may discover that it isn't suitable for your problem, or that you need other types of information as well. You may discover things in the data that raise issues more important than the one you originally planned to address. For example, the data in figure 1.2 seems counter-intuitive.

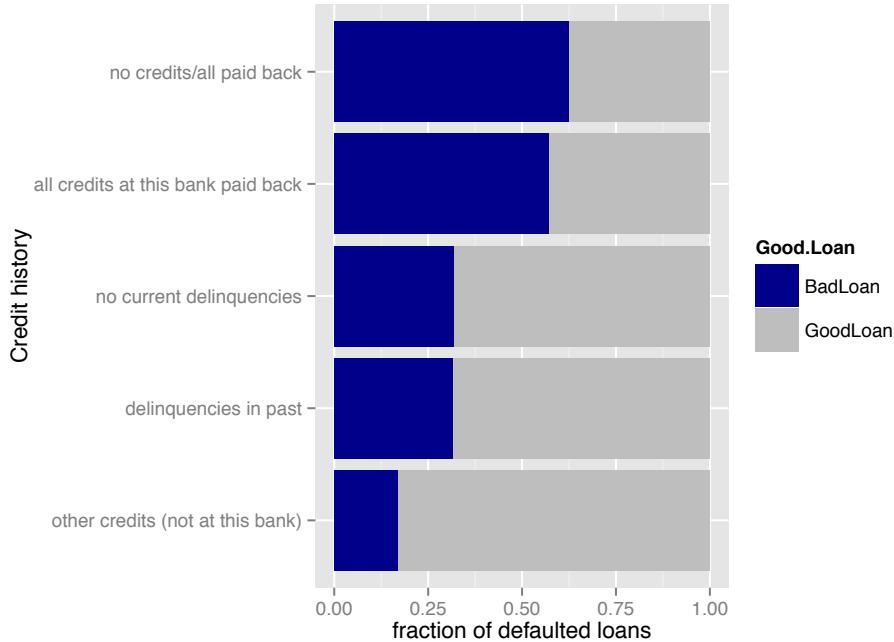


Figure 1.2 The fraction of defaulting loans by credit history category. The dark blue region of each bar represents the fraction of loans in that category that defaulted.

Why would some of the seemingly safe applicants ("all credits paid back") default at a higher rate than seemingly riskier ones ("delinquencies in past")?

After looking more carefully at the data, and sharing puzzling findings with other stakeholders and domain experts, you realize that this sample is inherently biased: *we only have loans that were actually made (and therefore already accepted)*. Overall, there are fewer risky-looking loans than safe-looking ones in the data. The probable story is that risky-looking loans were approved after a much stricter vetting process, a process that perhaps the “safe” loan applicants could bypass. This suggests that if your model is to be used downstream of the current application approval process, credit history is no longer a useful variable. It also suggests that even “safe” loan applications should be more carefully scrutinized.

Discoveries like this may lead you and other stakeholders to change or refine the project goals. In this case, you may decide to concentrate on the seemingly safe loan applications. It’s common to cycle back and forth between this stage and the previous one, as well as between this stage and the modeling stage, as you discover things in the data. We’ll cover data exploration and management in depth in chapters XRF:chapter_3:chExploringData and XRF:chapter_4:chManagingData.

1.2.3 Modeling

We finally get to statistics and machine learning during the modeling, or analysis stage. Here is where you try to extract useful insights from the data in order to achieve your goals. Since many modeling procedures make specific assumptions about data distribution and relationships, there will be overlap and back-and-forth between the modeling stage and the data cleaning stage as you try to find the best way to represent the data, and the best form in which to model it.

The most common data science modeling tasks are:

- Classification—*Deciding* if something belongs to one category or another.
- Scoring—Predicting or *estimating* a numeric value such as a price or probability.
- Ranking—Learning to *order items* by preferences.
- Clustering—*Grouping items* into most similar groups.
- Finding relations—*Finding correlations* or potential causes of effects seen in the data.
- Characterization—*Very general* plotting and report generation from data.

For each of these tasks, there are several different possible approaches. We’ll cover some of the most common approaches to the different tasks in this book.

The loan application problem is a classification problem: you want to identify loan applicants who are likely to default. Three common approaches in such cases are logistic regression, Naive Bayes classifiers, and decision trees (we’ll cover these methods in-depth in future chapters). You have been in conversation with

loan officers and others who would be using your model in the field, so you know that they want to be able to understand the chain of reasoning behind the model's classification, and they want an indication of how confident the model is in its decision: is this applicant highly likely to default, or only somewhat likely? Given the preceding desiderata, you decide that a decision tree is most suitable. We'll cover decision trees more extensively in a future chapter, but for now the call in R is as follows:²

Footnote 2 In this chapter, we are deliberately fitting a small and shallow tree, for clarity of illustration.

```
model <- rpart(Good.Loan ~
                 Duration.in.month +
                 Installment.rate.in.percentage.of.disposable.income +
                 Credit.amount +
                 Other.installment.plans
                 , data=creditdata,
                 control=rpart.control(maxdepth=4),
                 method="class")
```

Let's suppose that you discover the model shown in figure 1.3.

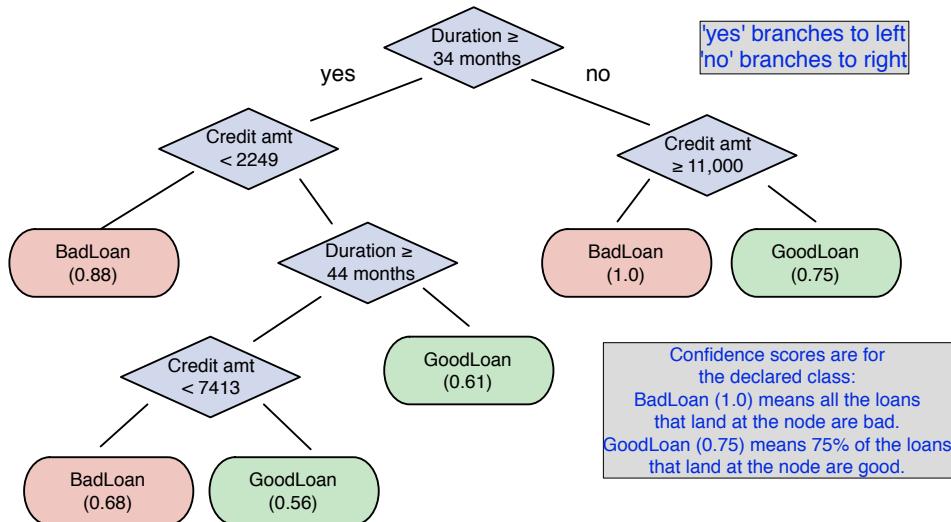


Figure 1.3 A decision tree model for finding bad loan applications, with confidence scores.

We'll discuss general modeling strategies in chapter XRF:chapter_5:chCritiquingModels and go into details of specific modeling algorithms in part XRF:part_2: Modeling Methods:bkptModelingMethods.

1.2.4 Model evaluation and critique

Once you have a model, you need to determine if it meets your goals.

- Is it accurate enough for your needs? Does it generalize well?
- Does it perform better than “the obvious guess”? Better than whatever estimate you currently use?
- Do the results of the model (coefficients, clusters, rules) make sense in the context of the problem domain?

If you’ve answered “no” to any of these questions, it’s time to loop back to the modeling step—or decide that the data doesn’t support the goal you’re trying to achieve. No one likes negative results, but understanding when you can’t meet your success criteria with current resources will save you fruitless effort. Your energy will be better spent on crafting success. This might mean defining more realistic goals, or gathering the additional data or other resources that you need to achieve your original goals.

Returning to the loan application example, the first thing to check is that the rules that the model discovered make sense. Looking at figure 1.3, you don’t notice any obvious strange rules, so you can go ahead and evaluate the model’s accuracy. A good summary of classifier accuracy is the *confusion matrix*, which tabulates actual classifications against predicted ones.³

Footnote 3 Normally, we would evaluate the model against a test set (data that was not used to build the model). In this example, for simplicity, we evaluate the model against the training data (data that was used to build the model).

```
> resultframe <- data.frame(Good.Loan=creditdata$Good.Loan,
                               pred=predict(model, type="class"))
> rtab <- table(resultframe) ①
> rtab
  pred
Good.Loan BadLoan GoodLoan
BadLoan      41     259
GoodLoan     13     687

> sum(diag(rtab))/sum(rtab) ②
[1] 0.728
> sum(rtab[1,1])/sum(rtab[,1]) ③
[1] 0.7592593
> sum(rtab[1,1])/sum(rtab[1,]) ④
[1] 0.1366667
> sum(rtab[2,1])/sum(rtab[2,]) ⑤
[1] 0.01857143
```

- ① Create the confusion matrix. Rows represent actual loan status; columns represent predicted loan status. The diagonal entries represent correct predictions.
- ② Overall model accuracy. 73% of the predictions were correct.
- ③ Model precision: 76% of the applicants predicted as bad really did default.

- ④ Model recall: The model found 14% of the defaulting loans.
- ⑤ False positive rate: 2% of the good applicants were mistakenly identified as bad.

The model predicted loan status correctly 73% of the time—better than chance (50%). In the original dataset, 30% of the loans were bad, so guessing GoodLoan all the time would be 70% accurate (though not very useful). So you know that the model does better than random and somewhat better than the obvious guess.

Overall accuracy is not enough. You want to know what kind of mistakes are being made. Is the model missing too many bad loans, or is it marking too many good loans as bad? *Recall* measures how many of the bad loans the model can actually find. *Precision* measures how many of the loans identified as bad really are bad. *False positive rate* measures how many of the good loans are mistakenly identified as bad. Ideally, you want the recall and the precision to be high, and the false positive rate to be low. What constitutes “high enough” and “low enough” is a decision that you make together with the other stakeholders. Often, the right balance requires some trade-off between recall and precision.

There are other measures of accuracy, and other measures of the goodness of a model, as well. We’ll talk about model evaluation in chapter XRF:chapter_5:chCritiquingModels.

1.2.5 Presentation and documentation

Once you have a model that meets your success criteria, you’ll present your results to your project sponsor and other stakeholders. You must also document the model for those in the organization who are responsible for using, running, and maintaining the model once it has been deployed.

Different audiences require different kinds of information. The business-oriented audiences want to understand the impact of your findings in terms of business metrics. In our loan example, the most important thing to present to business audiences is how your loan application model will reduce charge-offs (the money that the bank loses to bad loans). Suppose your model identified a set of bad loans in your data that amounted to 22% of the total money lost to defaults. Then your presentation or executive summary should emphasize that the model can potentially reduce the bank’s losses by that amount, as shown in figure 1.4.

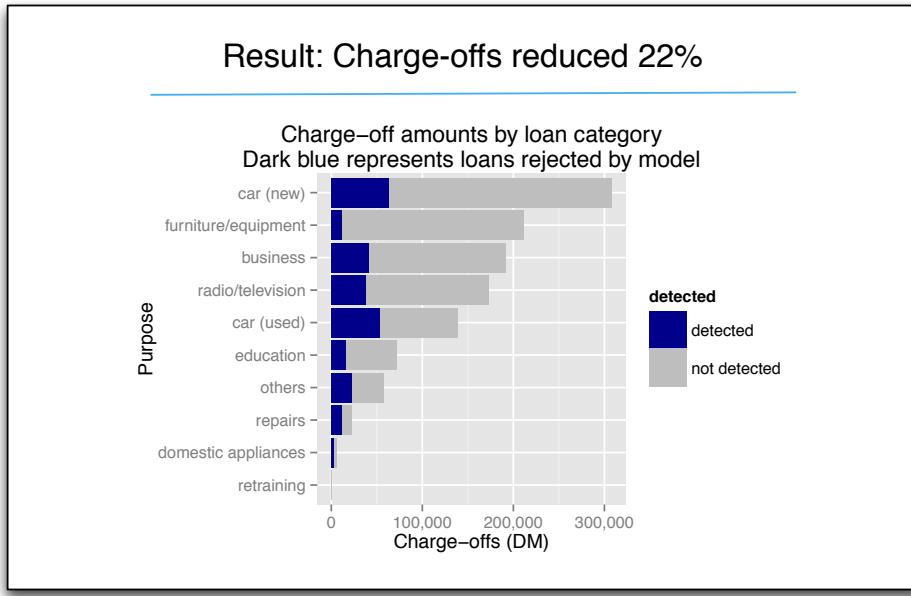


Figure 1.4 Notional slide from an executive presentation

You also want to give this audience your most interesting findings or recommendations, for instance that new car loans are much riskier than used car loans, or that most money is lost to bad car loans and bad equipment loans (assuming that the audience didn't already know these facts). Technical details of the model won't be as interesting to this audience, and you should skip them or present them at only a high level.

A presentation for the model's end users (the loan officers) would instead emphasize how the model will help them do their job better.

- How should they interpret the model?
- What does the model output look like?
- If the model provides a trace of which rules in the decision tree executed, how do they read that?
- If the model provides a confidence score in addition to a classification, how should they use the confidence score?
- When might they potentially overrule the model?

Presentations or documentation for operations staff should emphasize the impact of your model on the resources that they're responsible for.

We'll talk about the structure of presentations and documentation for various audiences in part XRF:part_3: Delivering Results:bkptResults.

1.2.6 Model deployment and maintenance

Finally, the model is put into operation. In many organizations this means the data scientist no longer has primary responsibility for the day-to-day operation of the model. But you still should ensure that the model will run smoothly and won't make disastrous unsupervised decisions. You also want to make sure that the model can be updated as its environment changes. And in many situations, the model will initially be deployed in a small pilot program. The test might bring out issues that you didn't anticipate, and you may have to adjust the model appropriately.

For example: you may find that loan officers frequently override the model in certain situations, because it contradicts their intuition. Is their intuition wrong? Or is your model incomplete? Or in a more positive outcome, your model may perform so successfully that the bank wants you to extend it to home loans as well.

We'll discuss these considerations in chapter XRF:chapter_10:chDocumentation.

Before we dive deeper into the stages of the data science life cycle in the following chapters, we'll discuss an important aspect of the initial project design stage: setting expectations.

1.3 Setting expectations

Setting expectations is a crucial part of defining the project goals and success criteria. The business-facing members of your team (in particular, the project sponsor) probably already have an idea of the performance required to meet business goals: for example, the bank wants to reduce their losses from bad loans by at least 10%. Before you get too deep into a project, you should make sure that the resources you have are enough for you to meet the business goals.

In this section, we discuss ways to estimate whether the data you have available is good enough to potentially meet desired accuracy goals. This is an example of the fluidity of the project life cycle stages. You get to know the data better during the exploration and cleaning phase; after you have a sense of the data, you can get a sense of whether the data is good enough to meet desired performance thresholds. If it's not, then you'll have to revisit the project design and goal-setting stage.

1.3.1 Determining lower and upper bounds on model performance

Understanding how well a model *should* do for acceptable performance, and how well it *can* do given the available data, are both important when defining acceptance criteria.

THE NULL MODEL: A LOWER BOUND ON PERFORMANCE

You can think of the *null model* as being “the obvious guess” that your model must do better than. In situations where there’s a working model or solution already in place that you’re trying to improve, the null model is the existing solution. In situations where there’s no existing model or solution, the null model is the simplest possible model (for example, always guessing GoodLoan, or always predicting the mean value of the output, when you’re trying to predict a numerical value). The null model represents the lower bound on model performance that you should strive for.

In our loan application example, 70% of the loan applications in the dataset turned out to be good loans. A “model that labels all loans as GoodLoan (in effect, using only the existing process to classify loans) would be correct 70% of the time. So you know that any actual model that you fit to the data should be better than 70% accurate to be useful. Since this is the simplest possible model, its error rate is called the *base error rate*.

How much better than 70% should you be? In statistics there’s a procedure called *hypothesis testing*, or *significance testing*, that tests whether your model is equivalent to a null model (in this case, whether a new model is basically only as accurate as guessing GoodLoan all the time). You want your model’s accuracy to be “significantly better”—in statistical terms—than 70%. We will cover the details of significance testing in chapter XRF:chapter_5:chCritiquingModels.

Accuracy is not the only (or even the best) performance metric. In our example, the null model would have zero recall in identifying bad loans, which obviously is not what you want. Generally if there is an existing model or process in place, you’d like to have an idea of its precision, recall, and false positive rates; if the purpose of your project is to improve the existing process, then the current model must be unsatisfactory for at least one of these metrics. This also helps you determine lower bounds on desired performance.

THE BAYES RATE: AN UPPER BOUND ON MODEL PERFORMANCE

The business-dictated performance goals will of course be higher than the lower bounds discussed here. You should try to make sure as early as possible that you have the data to meet your goals.

One thing to look at is what statisticians call the *unexplainable variance*: how much of the variation in your output can’t be explained by your input variables. Let’s take a very simple example: suppose you want to use the rule of thumb that

loans that are more than 15% of the borrower's disposable income will default; otherwise, the loan is good. You want to know if this rule alone will meet your goal of predicting bad loans with at least 85% accuracy. Let's consider the two populations next.

```
> tab1
          loan.quality.pop1 ①
loan.as.pct.disposable.income goodloan badloan
    LT.15pct      50      0
    GT.15pct       6     44
> sum(diag(tab))/sum(tab) ②
[1] 0.94
>
> tab2
          loan.quality.pop2 ③
loan.as.pct.disposable.income goodloan badloan
    LT.15pct      34     16
    GT.15pct      18     32
> sum(diag(tab))/sum(tab) ④
[1] 0.66
```

- ➊ The count of correct predictions is on the diagonal of tab1. In this first population, all the loans that were less than 15% of disposable income were good loans, and all but six of the loans that were greater than 15% of disposable income defaulted. So we know that `loan.as.pct.disposable.income` models loan quality well in this population. Or as statisticians might say, `loan.as.pct.disposable.income` “explains” the output (loan quality).
- ➋ In fact, it is 94% accurate.
- ➌ In the second population, about a third of the loans that were less than 15% of disposable income were defaulted, and over half of the loans that were greater than 15% of disposable income were good. So we know that `loan.as.pct.disposable.income` does not model loan quality well in this population.
- ➍ The rule of thumb is only 66% accurate.

For the second population, you know that you can't meet your goals using only `loan.as.pct.disposable.income`. To build a more accurate model, you'll need additional input variables.

The limit on prediction accuracy due to unexplainable variance is known as the *Bayes rate*. You can think of the Bayes rate as describing the best accuracy you can achieve given your data. If the Bayes rate doesn't meet your business-dictated performance goals, then you shouldn't start the project without revisiting your goals, or finding additional data to improve your model.⁴

Footnote 4 The Bayes rate gives the best possible accuracy, but the most accurate model doesn't always have the best possible precision or recall (though it may represent the best trade-off of the two).

Exactly finding the Bayes rate is not always possible—if you could always find the best possible model, then your job would already be done. If all your variables are discrete (and you have a lot of data), you can find the Bayes rate by building a lookup table for all possible variable combinations. In other situations, a nearest-neighbor classifier (we'll discuss them in chapter XRF:chapter_8:chGroupingMethods) can give you a good estimate of the Bayes rate, even though a nearest-neighbor classifier may not be practical to deploy as an actual production model. In any case, you should try to get some idea of the limitations of your data early in the process, so you know whether it's adequate to meet your goals.

1.4 Summary

The data science process involves a lot of back-and-forth: back-and-forth between the data scientist and other project stakeholders, and back-and-forth between the different stages of the process. Along the way, you'll encounter surprises and stumbling blocks; this book will teach you procedures for overcoming some of these hurdles. It's important to keep all the stakeholders informed and involved; when the project ends, no one connected with it should be surprised by the final results.

SIDE BAR Key takeaways

- A successful data science project involves more than just statistics. It also requires a variety of roles to represent business and client interests, as well as operational concerns.
- Make sure you have a clear and verifiable goal.
- Make sure you have set realistic expectations for all stakeholders.

In the next chapters, we'll look at the stages that follow project design: loading, exploring, and managing the data. Chapter XRF:chapter_2:chStartingWithRandData will cover a few basic ways to load the data into R, into a format that's convenient for analysis.

1.5 External links section



Loading data into R

This chapter will cover:

- R's data frame structure
- Loading data into R from files and from relational databases
- Transforming data for analysis

If your experience has been like ours, many of your data science projects start when someone points you toward a bunch of data and you're left to make sense of it. Your first thought may be to use shell tools or spreadsheets to sort through it, but you quickly realize that you're taking more time tinkering with the tools than actually analyzing the data. Luckily, there's a better way. In this chapter we'll demonstrate how to quickly load and start working with data using R. Using R to transform data is easy because R's main data type (the data frame) is ideal for working with structured data, and R has adapters that read data from many common data formats. In this chapter, we'll start with small example datasets found in files and then move to datasets from relational databases. By the end of the chapter you'll be able to confidently use R to extract, transform, and load data for analysis.¹

Footnote 1 We'll demonstrate and comment on the R commands necessary to prepare the data, but if you're unfamiliar with programming in R, we recommend at least skimming appendix XRF:appendix_A:xAppendixR or consulting a good book on R such as *R in Action*. All the tools we need are freely available and we have instructions how to download and start working with them in appendix XRF:appendix_A:xAppendixR.

For our first example let's start with some example datasets from files.

2.1 Working with data from files

The most common “ready-to-go” data format is a family of tabular formats called *structured values*. Most of the data you find will be in (or nearly in) one of these formats. When you can read such files into R, you can analyze data from an incredible range of public and private data sources. In this section we’ll work together on two examples of loading data from structured files, and one example of loading data directly from a relational database. The point is to get data quickly into R so we can then use R to perform interesting analyses.

2.1.1 Working with well-structured data from files or URLs

The easiest data format to read is table-structured data with headers. As shown in figure 2.1, this data is arranged in rows and columns where the first row gives the column names. Each column represents a different fact or measurement; each row represents an instance or datum about which we know the set of facts. A lot of public data is in this format, so being able to read it opens up a lot of opportunities.

◆	A	B	C	D	E	F	G
1	buying	maint	doors	persons	lug_boot	safety	rating
2	vhigh	vhigh		2	2 small	low	unacc
3	vhigh	vhigh		2	2 small	med	unacc
4	vhigh	vhigh		2	2 small	high	unacc
5	vhigh	vhigh		2	2 med	low	unacc
6	vhigh	vhigh		2	2 med	med	unacc

Figure 2.1 Car data viewed as a table

Before we load the German credit data that we used in the previous chapter, let’s demonstrate the basic loading commands with a simple data file from the University of California Irvine Machine Learning Repository (<http://archive.ics.uci.edu/ml/>). The UCI data files tend not to come with headers, so to save steps (and to keep it very basic, to start) we’ve pre-prepared our first data example from the UCI car dataset: <http://archive.ics.uci.edu/ml/machine-learning-databases/car/>. Our pre-prepared file is <http://www.win-vector.com/dfiles/car.data.csv> and looks like the following (details found at <https://github.com/WinVector/zmPDSwR/tree/master/UCICar>).

```
buying,maint,doors,persons,lug_boot,safety, rating ①  
vhigh,vhigh,2,2,small,low,unacc ②  
vhigh,vhigh,2,2,small,med,unacc  
vhigh,vhigh,2,2,small,high,unacc  
vhigh,vhigh,2,2,med,low,unacc  
...  
...
```

- ① The header row contains the names of the data columns, in this case separated by commas. When the separators are commas the format is called comma separated values or .csv
- ② The data rows are in the same format as the header row, but each row contains actual data values. In this case the first row represents the set of name/value pairs: buying=vhigh, maintenance=vhigh, doors=2, persons=2, and so on.

TIP

Avoid “by hand” steps

We strongly encourage you to avoid performing any steps “by hand” when importing data. It’s tempting to use an editor to add a header line to a file, as we did in our example. A better strategy is to write a script either outside R (using shell tools) or inside R to perform any necessary reformatting. Automating these steps greatly cuts down the amount of trauma and work during the inevitable data refresh.

Notice this file is already structured like a spreadsheet with easy-to-identify rows and columns. The data shown here are claimed to be the details about recommendations on cars, but are in fact made-up examples used to test some machine-learning theories. Each (non-header) row represents review of a different model of car. The columns represent facts about each car model. Most of the columns are objective measurements (purchase cost, maintenance cost, number of doors and so on) and the final column “rating” is marked with the overall rating (vgood, good, acc, and unacc). These sort of explanations can’t be found in the data but must be extracted from the documentation found with the original data.

LOADING WELL-STRUCTURED DATA FROM FILES OR URLs

Loading data of this type into R is a one-liner: we use the R command `read.table()` and we’re done. If data were always in this format, we’d meet all of the goals of this section and be ready to move on to modeling with just the following code.

```
uciCar <- read.table( ①
  'http://www.win-vector.com/dfiles/car.data.csv', ②
  sep=',', ③
  header=T ④
)
```

- ① Command to read from a file or URL and store the result in a new data frame object called `uciCar`.
- ② Filename or URL to get the data from.

- 3 Specify the column or field separator as a comma
- 4 Tell R to expect a header line that defines the data column names

This loads the data and stores it in a new R data frame object called `uciCar`. *Data frames* are R's primary way of representing data and are well worth learning to work with (as we discuss in our appendices). The `read.table()` command is powerful and flexible; it can accept many different types of data separators (commas, tabs, spaces, pipes, and others) and it has many options for controlling quoting and escaping data. `read.table()` can read from local files or remote URLs. If a resource name ends with the `.gz` suffix, `read.table()` assumes the file has been compressed in gzip style and will automatically decompress it while reading.

EXAMINING OUR DATA

Once we have loaded the data into R, we'll want to examine it. The commands to always try first are:

- `class()` which tells you what type of R object you have. In our case, `class(uciCar)` tells us the object `uciCar` is of class `data.frame`.
- `help()` which gives you the documentation for a class. In particular try `help(class(uciCar))` or `help("data.frame")`.
- `summary()` which gives you a summary of almost any R object. `summary(uciCar)` shows us a lot about the distribution of the UCI car data.

For data frames, the command `dim()` is also important, as it shows us how many rows and columns are in the data. We show the results of a few of these steps next (steps are prefixed by `>` and R results are shown after each step).

```
> class(uciCar)
[1] "data.frame"
> summary(uciCar)
  buying      maint       doors 
high :432    high :432    2     :432
low  :432    low  :432    3     :432
med  :432    med  :432    4     :432
vhigh:432   vhigh:432   5more:432

  persons      lug_boot      safety 
2      :576    big    :576    high:576
4      :576    med    :576    low :576
more:576   small:576   med  :576

  rating
acc  : 384
good :  69
```

1

```
unacc:1210  
vgood: 65  
  
> dim(uciCar)  
[1] 1728 7
```

2

- 1 The loaded object uciCar is of type data.frame.
- 2 The [1] is just an output sequence marker; the actual information is: uciCar has 1728 rows and 7 columns. Always try to confirm you got a good parse by at least checking that the number of rows is exactly one fewer than the number of lines of text in the original file. The difference of one is because the column header counts as a line, but not as a data row.

The `summary()` command shows us the distribution of each variable in the dataset. For example: we know each car in the dataset was declared to seat 2, 4 or more persons, and we know there were 576 two-seater cars in the dataset. Already we've learned a lot about our data, without having to spend a lot of time setting pivot tables as we would have to in a spreadsheet.

WORKING WITH OTHER DATA FORMATS

.csv is not the only common data file format you will encounter. Other formats include .tsv (tab-separated values), pipe-separated files, Microsoft Excel workbooks, JSON data, and XML. R's built in `read.table()` command can be made to read most separated value formats. Many of the deeper data formats have corresponding R packages:

- *XLS/XLSX*—
<http://cran.r-project.org/doc/manuals/R-data.html#Reading-Excel-spreadsheets>
- *JSON*—<http://cran.r-project.org/web/packages/rjson/index.html>
- *XML*—<http://cran.r-project.org/web/packages/XML/index.html>
- *MongoDB*—<http://cran.r-project.org/web/packages/rmongodb/index.html>
- *SQL*—<http://cran.r-project.org/web/packages/DBI/index.html>

2.1.2 Using R on less structured data

Data isn't always available in a ready-to-go format. Data curators often stop just short of producing a ready-to-go machine readable format. The German bank credit dataset discussed in chapter XRF:chapter_1:chProjectLifeCycle is an example of this. This data is stored as tabular data without headers; it uses a cryptic encoding of values that requires the dataset's accompanying documentation to untangle. This is not uncommon and is often due to habits or limitations of other tools that commonly work with the data. Instead of reformatting the data before we bring it into R, as we did in the last example, we'll now show how to reformat the data using R. This is a much better practice, as we can save and reuse the R commands needed to prepare the data.

Details of the German bank credit dataset can be found at [http://archive.ics.uci.edu/ml/datasets/Statlog+\(German+Credit+Data\)](http://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data)). We're going to show how to transform this data into something meaningful using R. After these steps, you can perform the analysis already demonstrated in chapter XRF:chapter_1:chProjectLifeCycle. As we can see in our file excerpt, the data is an incomprehensible block of codes with no meaningful explanations.

```
A11 6 A34 A43 1169 A65 A75 4 A93 A101 4 ...
A12 48 A32 A43 5951 A61 A73 2 A92 A101 2 ...
A14 12 A34 A46 2096 A61 A74 2 A93 A101 3 ...
...
```

TRANSFORMING DATA IN R

Data often needs a bit of transformation before it makes any sense. In order to decrypt troublesome data, you need what's called the *schema documentation* or a *data dictionary*. In this case the included dataset description says the data is 20 input columns followed by one result column. In this example there's no header in the data file. The column definitions and the meaning of the cryptic A-codes are all in the accompanying data documentation. Let's start by loading the raw data into R. We can either save the data to a file or let R load the data directly from the URL. Start a copy of R or RStudio and type in the following commands.

```
d <- read.table('http://archive.ics.uci.edu/ml/\n  machine-learning-databases/statlog/german/german.data',
  stringsAsFactors=F,header=F)
print(d[1:3,])
```

Notice this prints out the exact same three rows we saw in the raw file with the addition of column names V1 through V21. We can change the column names to something meaningful with the following command:

```
colnames(d) <- c('Status.of.existing.checking.account',
  'Duration.in.month', 'Credit.history', 'Purpose',
  'Credit.amount', 'Savings account/bonds',
  'Present.employment.since',
  'Installment.rate.in.percentage.of.disposable.income',
  'Personal.status.and.sex', 'Other.debtors/guarantors',
  'Present.residence.since', 'Property', 'Age.in.years',
  'Other.installment.plans', 'Housing',
  'Number.of.existing.credits.at.this.bank', 'Job',
  'Number.of.people.being.liable.to.provide.maintenance.for',
  'Telephone', 'foreign.worker', 'Good.Loan')
print(d[1:3,])
```

The `c()` command is R's method to construct a vector. We copied the names directly from the dataset documentation. By assigning our vector of names into the data frame's `colnames()` slot, we've reset the data frame's column names to something sensible. We can find what slots and commands our data frame `d` has available by typing `help(class(d))`.

The data documentation further tells us the column names, and also has a dictionary of the meanings of all of the cryptic A-* codes. For example it says in column 4 (now called *Purpose*, meaning the purpose of the loan) the code A40 is a new car loan, A41 is used car loan and so on. We copied 56 such codes into an R list that looks like the following:

```
mapping <- list(
  'A40'='car (new)',
  'A41'='car (used)',
  'A42'='furniture/equipment',
  'A43'='radio/television',
  'A44'='domestic appliances',
  ...
)
```

TIP

Lists are R's map structures

Lists are R's map structures. They can map strings to arbitrary objects. The important list operations `[]` and `%in%` are *vectorized*. This means when applied to a vector of values, they return a vector of results by performing one lookup per entry.

With the mapping list defined, we can then use the following for-loop to convert each column that was of character type (skipping numeric columns) from the cryptic A-* codes into short descriptions taken directly from the data documentation.

```
for(i in 1:(dim(d))[2]) {  
  if(class(d[,i])=='character') {  
    d[,i] <- as.factor(as.character(mapping[d[,i]]))  
  }  
}
```

- ➊ (dim(d))[2] is the number of columns in the data frame d
- ➋ Note that the indexing operator [] is vectorized. Each step in the for loop remaps an entire column of data through our list.

We share the complete set of column preparations for this dataset here: <https://github.com/WinVector/zmPDSwR/blob/master/Statlog/> We encourage the reader to download the data and try these steps themselves.

EXAMINING OUR NEW DATA

We can now easily examine the purpose of the first three loans with the command `print(d[1:3, 'Purpose'])`. We can look at the distribution of loan purpose with `summary(d$Purpose)` and even start to investigate the relation of loan type to loan outcome as shown in listing 2.1.

Listing 2.1 Summary of loan and purpose

You should now be able to load data from files. But a lot of data you want to work with isn't in files, but in databases. So it's important that we work through how to load data from databases directly into R.

2.2 Working with relational databases

In many production environments, the data you want lives in a relational or SQL database, not in files. Public data is often in files (as they are easier to share), but your most important client data is often in databases. Relational databases scale easily to the millions of records, and supply important production features such as parallelism, consistency, transactions, logging, and audits. Relational databases are designed to support online transaction processing (OLTP), so they're likely where transactions you need to know about already are.

Often you can export the data into a structured file and use the methods of our previous sections to then transfer the data into R. But this is generally not the right way to do things. Exporting from databases to files is often unreliable and idiosyncratic due to variations in database tools and the typically poor job done quoting and escaping characters that are confused with field separators. Data in a database is often stored in what is called a *normalized form*, which requires relational preparations called *joins* before the data is ready for analysis. Also, you often don't want a dump of the entire database, but instead wish to freely specify which columns and aggregations you need during analysis.

The right way to work with data found in databases is to connect R directly to the database, which is what we'll demonstrate in this section.

As a step of the demonstration, we'll show how to load data into a database. Knowing how to load data into a database is useful for problems that need more sophisticated preparation that we've so far discussed. Relational databases are the right place for transformations such as joins or sampling. Let's start working with data in a database for our next example.

2.2.1 A production-sized example

For our production-size example we will use the United States Census 2011 national PUMS American Community Survey data found at http://www.census.gov/acs/www/data_documentation/pums_data/. This is a remarkable dataset recording facts about around 3 million individuals and 1.5 million households. Each row contains over 200 facts about each individual or household (income, employment, education, number of rooms, and so on). The data has household cross-reference IDs so individuals can be joined to the household they were in. The dataset size is interesting: a few gigabytes when zipped up. So it's small enough to move on a good network or thumb-drive, but larger than is convenient to work with on a laptop with R (which is more comfortable in the hundreds of thousands of rows range).

This millions of rows size is the sweet-spot for relational database or SQL-assisted analysis on a single machine. We're not yet forced to move into a MapReduce or database cluster to do our work, but we do want to use a database for some of the initial data handling. We'll work through all of the steps of acquiring this data and preparing it for analysis in R.

CURATING THE DATA

A hard rule of data science is that you must be able to reproduce your results. At the very least, be able to repeat your own successful work through your recorded steps and without depending on any stash of intermediate results. Everything must either have directions how to produce it or clear documentation where it came from. We call this the “no alien artifacts” discipline. For example when we said we’re using PUMS American Community Survey data, this isn’t a precise enough statement for anybody to know what data we actually mean. Our actual notebook entry (which we keep online, so we can search it) on the PUMS data is as shown in listing 2.2.

Listing 2.2 PUMS data provenance documentation

```
3-12-2013
PUMS Data set from:
http://www.census.gov/acs/www/data_documentation/pums_data/ ①
select "2011 ACS 1-year PUMS" ②
select "2011 ACS 1-year Public Use Microdata Samples\
(PUMS) - CSV format"
download "United States Population Records" and
"United States Housing Unit Records"
http://www2.census.gov/acs2011_1yr/pums/csv_pus.zip ③
http://www2.census.gov/acs2011_1yr/pums/csv_hus.zip
downloaded file details:
$ ls -lh *.zip ④
239M Oct 15 13:17 csv_hus.zip
580M Mar  4  06:31 csv_pus.zip
$ shasum *.zip ⑤
cdfdfb326956e202fdb560ee34471339ac8abd6c  csv_hus.zip
aa0f4add21e327b96d9898b850e618aec10f6d0  csv_pus.zip
```

- ① Where we found the data documentation. This is important to record as many data files don't contain links back to the documentation. Census PUMS does in fact contain embedded documentation, but not every source is so careful.
- ② How we navigated from the documentation site to the actual data files. It may be necessary to record this if the data supplier requires any sort of click-through license to get to the actual data.
- ③ The actual files we downloaded.
- ④ The sizes of the files after we downloaded them.
- ⑤ Cryptographic hashes of the file contents we downloaded. These are very short summaries (called hashes) that are very unlikely to have the same value for different files. These summaries can later help us determine if another researcher in our organization is in fact using the same data distribution or not.

TIP**Keep notes**

A big part of being a data scientist is being able to defend your results and repeat your work. We strongly advise keeping a notebook. We also strongly advise keeping all of your scripts and code under version control as we discuss in appendix XRF:appendix_A:xAppendixR. You absolutely want to be able to answer exactly what code and which data were used to build results you presented last week.

STAGING THE DATA INTO A DATABASE

Structured data at the millions of rows scale is best handled in a database. You can try to work with text-processing tools, but a database is much better at representing the fact that your data is arranged in both rows and columns (not just lines of text).

We will use three database tools in this example: the server-less database engine H2, the database loading tool SQLScrewdriver, and the database browser SQuirreLSQL. All of these are Java-based, run on many platforms, and are open source. We describe how to download and start working with all of them in appendix XRF:appendix_A:xAppendixR.

If you have a database such as MySQL or PostgreSQL already available, we recommend using one of them instead of using H2.² To use your own database you'll need to know enough of your database driver and connection information to build a JDBC connection. If using H2, you'll only need to download the H2 driver as described in appendix XRF:appendix_A:xAppendixR, pick a file path to store your results, and pick a user name and password (both are set on first use, so there are no administrative steps). H2 is a serverless zero-install relational database that supports queries in SQL. It's powerful enough to work on PUMS data and easy enough to use you get it running quickly using our instructions in appendix XRF:appendix_A:xAppendixR.

Footnote 2 If you have access to a parallelized SQL database such as Greenplum, we strongly suggest using it to perform aggregation and preparation steps on your big data. Being able to write standard SQL queries and have them finish quickly at big data scale can be game-changing.

We're going to use the Java based tool SQLScrewdriver to load the PUMS data into our database. We first copy our database credentials into a Java properties XML file like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
```

```

<properties>
    <comment>testdb</comment>
    <entry key="user">u</entry> ①
    <entry key="password">u</entry> ②
    <entry key="driver">org.h2.Driver</entry>
    <entry key="url">jdbc:h2:H2DB \
        ;LOG=0;CACHE_SIZE=65536;LOCK_MODE=0;UNDO_LOG=0</entry> ④
</properties>

```

- ① User name to use for database connection.
- ② Password to use for database connection.
- ③ Java classname of the database driver. SQLScrewdriver used JDBC which is a broad database application programming interface layer. You could use another database such as PostgreSQL by specifying a different driver name such as org.postgresql.Driver.
- ④ URL specifying database. For H2 it is just jdbc:h2: followed by the file prefix you wish to use to store data. The items after the semicolon are performance options. For Postgresql it would be something more like jdbc:postgresql://host:5432/db. The descriptions of the URL format and drivers should be part of your database documentation, and you can use SquirrelSQL to confirm you have them right.

We then use Java at the command line to load the data. To load the four files containing the two tables, we ran the following commands.

```

java -classpath SQLScrewdriver.jar:h2-1.3.170.jar \ ①
    com.winvector.db.LoadFiles \ ②
    file:dbDef.xml \ ③
    , \ ④
    hus \ ⑤
    file:csv_hus/ss11husa.csv file:csv_hus/ss11husb.csv ⑥
java -classpath SQLScrewdriver.jar:h2-1.3.170.jar \ ⑦
    com.winvector.db.LoadFiles \
    file:dbDef.xml , pus \
    file:csv_pus/ss11pusa.csv file:csv_pus/ss11pushb.csv

```

- ① Java command and required jars. The jars in this case are SQLScrewdriver and the required database driver.
- ② Class to run: LoadFiles, the meat of SQLScrewdriver.
- ③ URL pointing to database credentials.
- ④ Separator to expect in input file (use t for tab).
- ⑤ Name of table to create.
- ⑥ List of comma separated files to load into table.
- ⑦ Same load pattern for personal information table.

SQLScrewdriver infers data types by scanning the file and then creates new

tables in your database. It then populates these tables with the data. SQLScrewdriver also adds four additional “provenance” columns when loading your data. These columns are ORIGININSERTTIME, ORIGFILENAME, ORIGFILEROWNUMBER, and ORIGINRANDEGROUP. The first three fields record when you ran the data load, what file name the row came from, and what line the row came from. The ORIGINRANDEGROUP is a pseudo-random integer distributed uniformly from 0 through 999, designed to make repeatable sampling plans easy to implement. You should get in the habit of having annotations and keeping notes at each step of the process.

We can now use a database browser like SquirreLSQL to examine this data. We start up SquirreLSQL, and copy the connection details from our XML file into a database alias as shown in appendix XRF:appendix_A:xAppendixR. We’re then ready to type SQL commands into the execution window. A couple of commands you can try are `SELECT COUNT(1) FROM hus` and `SELECT COUNT(1) FROM pus`, which will tell you that the `hus` table has 1,485,292 rows and the `pus` table has 3,112,017 rows. Each of the tables has over 200 columns and there are over a billion cells of data in these two tables. We can actually do a lot more. In addition to the SQL execution panel, SquirreLSQL has an Objects panel that allows graphical exploration of database table definitions. Figure 2.2 shows some of the columns in the `hus` table.

Database table we are examining.

Connect to: h2PUMS Active Session: 1 - h2PUMS (H2DB) as u

Catalog: H2DB Objects SQL

COLUMN_NAME	TYPE_NAME
ORIGFILEROWNUMBER	BIGINT
ORIGFILENAME	VARCHAR
ORIGINERTTIME	TIMESTAMP
ORIGINRNGROUP	BIGINT
KI	VARCHAR
SERIALNO	BIGINT
DIVISION	BIGINT
PUMA	BIGINT
REGION	BIGINT
ST	BIGINT
ADJHSG	BIGINT
ADIINC	BIGINT

Query 1 of 1, Rows read: 1, Elapsed time (seconds) – Total: 100.325, SQL query: 100.296, Reading results 0.029

Logs: Errors 0, Warnings 0, Info: 11 73 of 164 MB 0 9:14:42 AM PST

Data provenance columns added by SQLScrewdriver (row number, file name, insert time)

Random grouping column by SQLScrewdriver: an integer from 0 to 999, useful for sampling.

Data columns supplied by US Census.

Figure 2.2 SQuirreLSQL table explorer

Now we can view our data as a table (as we would in a spreadsheet). We can now examine, aggregate, and summarize our data using the SQuirreLSQL database browser. Figure 2.3 shows a few example rows and columns from the household data table.

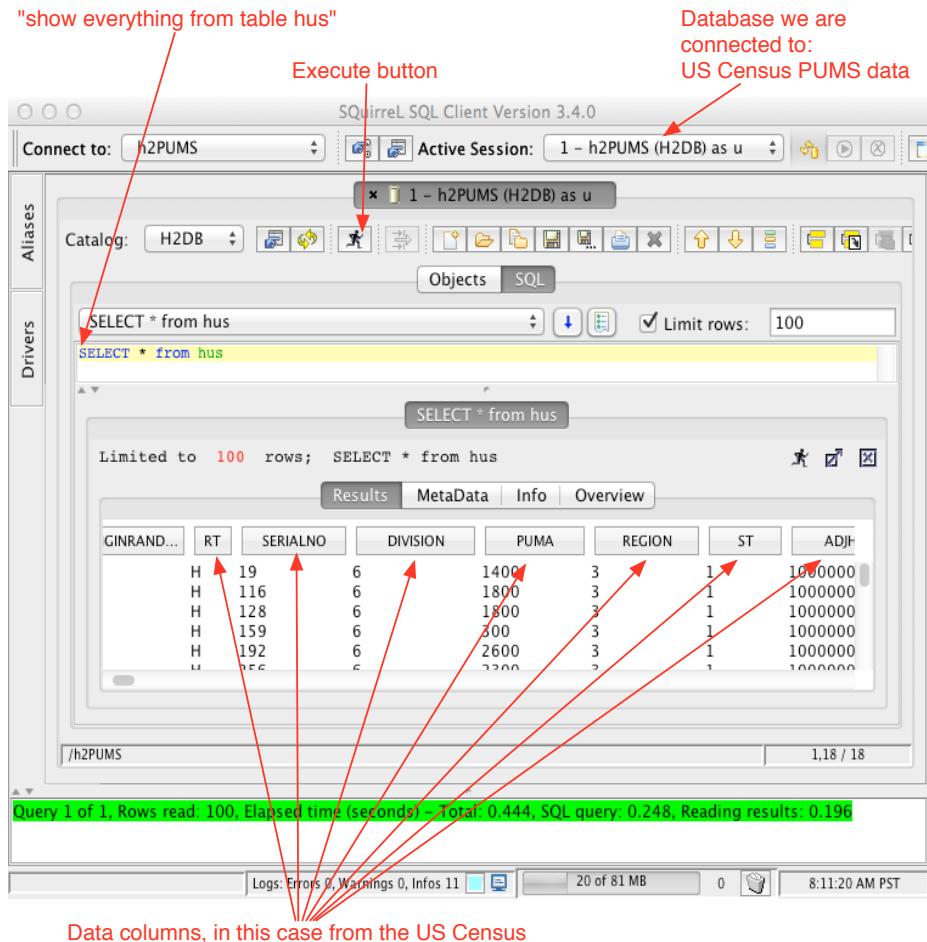


Figure 2.3 Browsing PUMS data using SQuirreLSQL

2.2.2 Loading data from a database into R

To load data from a database we use a database connector. Then we can directly issue SQL queries from R. SQL is the most common database query language and allows us to specify arbitrary joins and aggregations. SQL is called a *declarative language* (as opposed to a procedural language) because in SQL we specify what relations we would like our data sample to have, not how to compute them. For our example we load a sample of the household data from the `hus` table and the rows from the person table `pus` that are associated with those households. This is what is meant by a *join*: a join is a set of records that are produced by matching records from one or more tables. In this case the interesting join³ is a set of personal records that match a given set of household records.

Footnote 3 For convenience we actually use what is called *sub-select* to specify it in this case.

```

options( java.parameters = "-Xmx2g" ) ①
library(RJDBC)
drv <- JDBC("org.h2.Driver", ②
  "h2-1.3.170.jar", ③
  identifier.quote='') ④
options<-";LOG=0;CACHE_SIZE=65536;LOCK_MODE=0;UNDO_LOG=0"
conn <- dbConnect(drv,paste("jdbc:h2:H2DB",options,sep=' '),"u","u")
dhus <- dbGetQuery(conn,"SELECT * FROM hus WHERE ORIGINRANNGROUP<=1") ⑤
dpus <- dbGetQuery(conn,"SELECT pus.* FROM pus WHERE pus.SERIALNO IN \
  (SELECT DISTINCT hus.SERIALNO FROM hus \
  WHERE hus.ORIGINRANNGROUP<=1)") ⑥
dbDisconnect(conn) ⑦
save(dhus,dpus,file='phsample.RData') ⑧

```

- ① Set Java option for extra memory before DB drivers are loaded.
- ② Specify the name of the database driver, same as in our XML database configuration.
- ③ Specify where to find the implementation of the specified database driver.
- ④ SQL column names with mixed case capitalization, special characters, or that collide with reserved words must be quoted. We're specifying single-quote as the quote we'll use when quoting column names, which may be a different quote than the one we're using for SQL literals.
- ⑤ Create a data frame called dhus from * (everything) from the d database table hus, taking only rows where ORGINRANGGROUP <= 1. The ORGINRANGROUP column is a random integer from 0 through 999 that SQLScrewdriver adds to the rows during data load to facilitate sampling. In this case we're taking 2/1000ths of the data rows to get small sample.
- ⑥ Create a data frame called dpus from the database table pus, taking only records that have a household ID in the set of household IDs we selected from households table hus.
- ⑦ Disconnect for the database.
- ⑧ Save the two data frames into a file named phsample.RData, can be read in with load(). Try help("save") or help("load") for more details.

And we're in business; the data has been unpacked from the Census supplied .csv files into our database and a useful sample has been loaded into R for analysis. We have actually accomplished a lot. Generating, as we have, a uniform sample of households and matching people would be tedious using shell tools. It's exactly what SQL databases are designed to do well.

TIP**Don't be too proud to sample**

Many data scientists spend too much time adapting algorithms to work directly with “big data.” Often this is wasted effort, as for many model types you would get nearly the exact same results on a reasonably sized data sample. You only need to work with “all of your data” when what you’re modeling isn’t well served by sampling, for example when characterizing rare events or performing bulk calculations over social networks.

Note that this data is still in some sense large (out of the range where using spreadsheets is actually reasonable). Using `dim(dhus)` and `dim(dpus)`, we see that our household sample has 2,982 rows and 210 columns, and the people sample has 6,279 rows and 288 columns. All of these columns are defined in the Census documentation.

2.2.3 Working with the PUMS data

We must remember the whole point of loading data (even from a database) into R is to facilitate modeling and analysis. The data analyst should always have their “hands in the data” and always take a quick look at their data after loading it. If you’re not willing to work with the data, you shouldn’t bother loading it into R. To emphasize analysis we’ll demonstrate how to perform quick examination of the PUMS data.

LOADING AND CONDITIONING THE CENSUS PUMS DATA

Each row of PUMS data represents a single anonymized person or household. Personal data recorded includes occupation, level of education, personal income, and many other demographics variables. To load our prepared data frame,

`download phsample.Rdata from`

<https://github.com/WinVector/zmPDSwR/tree/master/PUMS> and run the following command in R:

```
load('phsample.RData')
```

Our example problem will be to predict income (represent in U.S. dollars in the field PINCP) using the following variables:

- *Age*—An integer found in the column `AGEP`.
- *Employment class*—Examples: for profit company, non-profit company, ... found in

column `COW`.

- *Education level*—Examples: no high school diploma, high school, college, and so on, found in column `SCHL`.
- *Sex of worker*—Found in column `SEX`.

We don't want to concentrate too much on this data; our goal is only to illustrate modeling procedure. Conclusions are very dependent on choices of data conditioning (what subset of the data you use), and data coding (how you map records to informative symbols). This is why empirical scientific papers have a mandatory “materials and methods” section describing how data was chosen and prepared. Our data treatment is to select a subset of “typical full time workers” by restricting to data that meets all of the following conditions:

- Workers self-described as full time employees.
- Workers reporting at least 40 hours a week of activity.
- Workers of 20 to 50 years of age.
- Workers with an annual income between \$1,000 and \$250,000 dollars.

The code to limit to our desired subset of the data is given next.

```
psub = subset(dpus,with(dpus,(PINCP>1000)&(ESR==1)&
(PINCP<=250000)&(PERNP>1000)&(PERNP<=250000)&
(WKHP>=40)&(AGEP>=20)&(AGEP<=50)&
(PWGTP1>0)&(COW %in% (1:7))&(SCHL %in% (1:24))))
```

①

① Subset of data rows matching detailed employment conditions

RECODING THE DATA

Before we work with the data recode some of the variable for readability. In particular we want to recode variables that are enumerated integers into meaningful factor level names, but for readability and to prevent accidentally treating such variables as mere numeric values. Listing 2.3 show the typical steps needed to perform a useful recoding.

List 2.3 Recoding variables

```
psub$SEX = as.factor(ifelse(psub$SEX==1,'M','F')) ①  
psub$SEX = relevel(psub$SEX,'M') ②  
cowmap <- c("Employee of a private for-profit",  
          "Private not-for-profit employee",  
          "Local government employee",  
          "State government employee",  
          "Federal government employee",  
          "Self-employed not incorporated",  
          "Self-employed incorporated")  
  
psub$COW = as.factor(cowmap[psub$COW]) ③  
psub$COW = relevel(psub$COW,cowmap[1])  
  
schlmap = c( ④  
            rep("no high school diploma",15),  
            "Regular high school diploma",  
            "GED or alternative credential",  
            "some college credit, no degree",  
            "some college credit, no degree",  
            "Associate's degree",  
            "Bachelor's degree",  
            "Master's degree",  
            "Professional degree",  
            "Doctorate degree")  
psub$SCHL = as.factor(schlmap[psub$SCHL])  
psub$SCHL = relevel(psub$SCHL,schlmap[1])  
  
dtrain = subset(psub,ORIGINRANDECODE >= 500) ⑤  
dtest = subset(psub,ORIGINRANDECODE < 500) ⑥
```

- ① Reencode sex from 1/2 to M/F.
- ② Make the reference sex M, so F encodes a difference from M in models.
- ③ Reencode class of worker info into a more readable form.
- ④ Reencode education info into a more readable form and fewer levels (merge all levels below high school into same encoding).
- ⑤ Subset of data rows used for model training.
- ⑥ Subset of data rows used for model testing.

The data preparation is making use of R’s vectorized lookup operator []. For details on this or any other R commands, we suggest using the R `help()` command and appendix XRF:appendix_A:xAppendixR (for help with [], type `help('[')`).

The standard trick to work with variables that take on a small number of string values is to reencode them into what’s called a *factor* as we’ve done with the `as.factor()` command. A factor is a list of all possible values of the variable (possible values are called *levels*) and each level works (under the covers) as

what's called an *indicator variable*. An indicator is a variable that is one when a condition we're interested in is true, and zero otherwise. Indicators are a useful encoding trick. For example SCHL is reencoded as 8 indicators with the names as shown in figure XRF:figure_7.1.4:bachcoef plus the undisplaced level "no high school diploma." Each indicator takes a value of zero, except when the SCHL variable has a value equal to the indicator's name. When the SCHL variable matches the indicator name, the indicator is set to 1 to indicate the match. Figure 2.4 illustrates the process. SEX and COW underwent similar transformations.

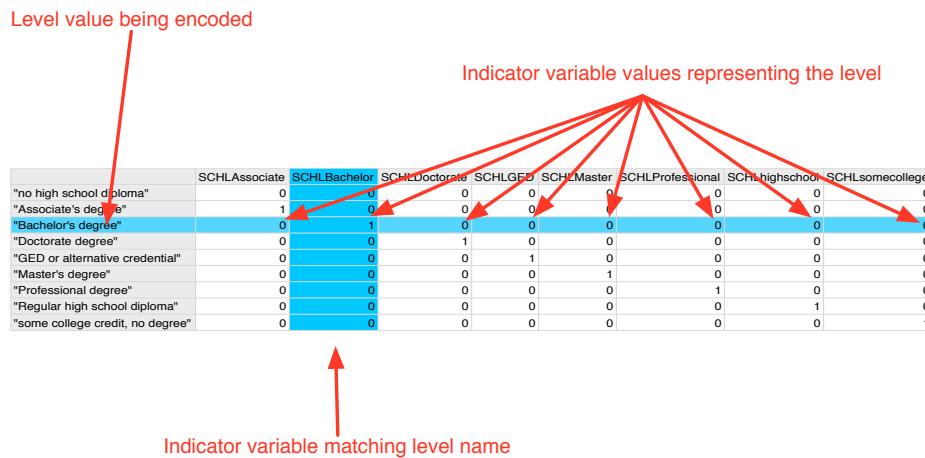


Figure 2.4 String encoded as indicators

EXAMINING THE PUMS DATA

At this point we're ready to do some science, or at least start looking at the data. For example we can quickly tabulate the distribution of category of work:

```
> summary(dtrain$COW)
Employee of a private for-profit          Federal government employee
                           423                               21
Local government employee                 Private not-for-profit employee
                           39                               55
Self-employed incorporated               Self-employed not incorporated
                           17                               16
State government employee
                           24
```

WARNING**Watch out for NAs**

R's representation for blank or missing data is called *NA*. Unfortunately a lot of R commands quietly skip NAs without warning. The command `table(dpus$COW,useNA='always')` will show NAs much like `summary(dpus$COW)` does.

We will return to the Census example and demonstrate more sophisticated modeling techniques in chapter XRF:chapter_7:chFunctionalModels.

2.3 Summary

In this chapter we have shown how to extract, transform, and load data for analysis. For smaller datasets we perform the transformations in R, and for larger datasets we advise using a SQL database. In either case we save *all* of the transformation steps as code (either in SQL or in R) that can be reused in the event of a data refresh. The whole purpose of this chapter is to prepare for the actual interesting work in our next chapters: exploring, managing, and correcting data.

SIDE BAR**Key takeaways**

- Data frames are your friend.
- Use `read_table()` to load small structured datasets into R.
- You can use a package like RJDBC to load data into R from relational databases, and to transform or aggregate the data before loading using SQL.
- Always document data provenance.

2.4 External links section

3

Exploring data

This chapter will cover

- Exploring your data using summary statistics
- Exploring your data using visualization
- Typical problems and issues to look for during data exploration

In the last two chapters, you learned how to set the scope and goal of a data science project, and to load your data into R. In this chapter, we'll start to get our hands into the data.

Suppose your goal is to build a model to predict which of your customers don't have health insurance; perhaps you want to market inexpensive health insurance packages to them. You've collected a data set of customers whose health insurance status you know. You've also identified some customer properties that you believe help predict insurance coverage: age, employment status, income, information about residence and vehicles, and so on. You've put all your data into a single data frame called *custdata* that you've input into R.¹ Now you're ready to start building the model to identify the customers you're interested in.

Footnote 1 We have a copy of this synthetic data set available for download from <https://github.com/WinVector/zmPDSwR/tree/master/Custdata>, and once saved, you can load it into R with the command `custdata = read.table('custdata.tsv', header=T, sep='\\t')`.

It's tempting to dive right into the modeling step without looking very hard at the dataset first, especially when you have a lot of data. Resist the temptation. No dataset is perfect: you'll be missing information about some of your customers, and you'll have incorrect data about others. Some data fields will be dirty and inconsistent. If you don't take the time to examine the data before you start to model, then you may find yourself redoing your work repeatedly, as you discover

bad data fields, or variables that need to be transformed before modeling. In the worst case, you'll build a model that returns incorrect predictions—and you won't be sure why. By addressing data issues early, you can save yourself some unnecessary work, and a lot of headaches!

You'd also like to get a sense of who your customers are: are they young, middle-aged, or seniors? How affluent are they? Where do they live? Knowing the answers to these questions can help you build a better model, because you'll have a more specific idea of what information predicts insurance coverage more accurately.

In this chapter we'll demonstrate some ways to get to know your data, and discuss some of the potential issues that you're looking for as you explore. Data exploration uses a combination of *summary statistics*—means and medians, variances, and counts—and *visualization*, or graphs of the data. You can spot some problems just using summary statistics; other problems are easier to find visually.

SIDE BAR Organizing data for analysis

For most of this book, we'll assume that the data you're analyzing is in a single data frame. This is not usually how that data is stored. In a database, for example, data is usually stored in *normalized form* to reduce redundancy: information about a single customer is spread across many small tables. In log data, data about a single customer can be spread across many log entries, or sessions. These formats make it easy to add (or in the case of a database, modify) data, but are not optimal for analysis. You can often join all the data you need into a single table in the database using SQL, but in appendix XRF:appendix_A:xAppendixR we'll discuss commands like `merge` that you can use within R to further consolidate data.

3.1 Using summary statistics to spot problems

In R, you will typically use the `summary` command to take your first look at the data.

Listing 3.1 The `summary()` command

```
> summary(custdata)
custid      sex
Min. : 2068 F:440
1st Qu.: 345667 M:560
Median : 693403
Mean   : 698500
3rd Qu.:1044606
Max.   :1414286

is.employed      income ①
Mode :logical    Min.   : -8700
FALSE:73        1st Qu.: 14600
TRUE :599       Median : 35000
NA's :328       Mean   : 53505
                  3rd Qu.: 67000
                  Max.   :615000

marital.stat
Divorced/Separated:155
Married           :516
Never Married    :233
Widowed          : 96

health.ins ②
Mode :logical
FALSE:159
TRUE :841
NA's :0

housing.type ③
Homeowner free and clear     :157
Homeowner with mortgage/loan:412
Occupied with no rent       : 11
Rented                   :364
NA's                     : 56

recent.move      num.vehicles
Mode :logical    Min.   :0.000
FALSE:820        1st Qu.:1.000
TRUE :124        Median :2.000
NA's :56         Mean   :1.916
                  3rd Qu.:2.000
                  Max.   :6.000
                  NA's   :56

age            state.of.res ④
Min.   : 0.0   California :100
1st Qu.: 38.0  New York   : 71
Median : 50.0  Pennsylvania: 70
Mean   : 51.7  Texas      : 56
```

```
3rd Qu.: 64.0    Michigan    : 52
Max.     :146.7    Ohio        : 51
                  (Other)      :600
```

- 1 The variable `is.employed` is missing for about a third of the data. The variable `income` has negative values, which are potentially invalid.
- 2 About 84% of the customers have health insurance.
- 3 The variables `housing.type`, `recent.move`, and `num.vehicles` are each missing 56 values.
- 4 The average value of the variable `age` seems plausible, but the minimum and maximum values seem unlikely. The variable `state.of.res` is a categorical variable; `summary()` reports how many customers are in each state (for the first few states).

The `summary` command on a data frame reports back a variety of summary statistics on the numerical columns of the data frame, and count statistics on any categorical columns (if the categorical columns have already been read in as factors²). You can also ask for summary statistics on specific numerical columns by using the commands `mean`, `variance`, `median`, `min`, `max`, and `quantile` (which will return the quartiles of the data by default).

Footnote 2 Categorical variables are of class `factor` in R. They can be represented as strings (class `character`), and some analytical functions will automatically convert strings variables to factor variables. To get a summary of the variable, it needs to be a factor.

As you see from listing 3.1, the summary of the data helps you quickly spot potential problems, like missing data or unlikely values. You also get a rough idea of how categorical data is distributed. Let's go into more detail about the typical problems that you can spot using the summary.

3.1.1 Typical problems that can be spotted from data summaries

At this stage, you're looking for several common issues:

- Missing values
- Invalid values and outliers
- Data range that is too wide or too narrow
- Units

MISSING VALUES

A few missing values may not really be a problem, but if a particular data field is largely unpopulated, it shouldn't be used as an input without some repair (as we'll discuss in chapter XRF:chapter_4:chManagingData section XRF:sect2_4.1.1:sec_missingvalues). In R, for example, many modeling algorithms will quietly drop rows with missing values by default. As you see in listing 3.2, all the missing values in the *is.employed* variable could cause R to quietly ignore nearly a third of the data.

Listing 3.2 Will the variable *is.employed* be useful for modeling?

```
is.employed      1
Mode :logical
FALSE:73
TRUE :599
NA's :328

          housing.type      2
Homeowner free and clear    :157
Homeowner with mortgage/loan:412
Occupied with no rent       : 11
Rented                      :364
NA's                        : 56

recent.move      num.vehicles
Mode :logical   Min.    :0.000
FALSE:820        1st Qu.:1.000
TRUE :124        Median  :2.000
NA's :56         Mean    :1.916
                  3rd Qu.:2.000
                  Max.    :6.000
                  NA's    :56
```

- ➊ The variable *is.employed* is missing for about a third of the data. Why? Is employment status unknown? Did the company start collecting employment data only recently? Does NA mean “not in the active workforce” (for example, students or stay-at-home parents)?
- ➋ The variables *housing.type*, *recent.move*, and *num.vehicles* are only missing a few values. It’s probably safe to just drop the rows that are missing values—especially if the missing values are all the same 56 rows.

If a particular data field is largely unpopulated, it’s worth trying to determine why; sometimes the fact that a value is missing is informative in and of itself. For example, why is the *is.employed* variable missing so many values? There are many possible reasons, as we discussed in listing 3.2.

Whatever the reason for missing data, you must decide on the most appropriate

action. Do you include a variable with missing values in your model, or not? If you decide to include it, do you drop all the rows where this field is missing, or do you convert the missing values to zero or to an additional category? We'll discuss ways to treat missing data in chapter XRF:chapter_4:chManagingData. In this example, you might decide to drop the data rows where you're missing data about housing or vehicles, since there aren't many of them. You probably don't want to throw out the data where you're missing employment information, but instead treat the *NA* as a third employment category. You will likely encounter missing values when model scoring, so you should deal with them during model training.

INVALID VALUES AND OUTLIERS

Even when a column or variable isn't missing any values, you still want to check that the values that you do have make sense. Do you have any invalid values or outliers? Examples of invalid values include negative values in what should be a non-negative numeric data field (like age or income), or text where you expect numbers. Outliers are data points that fall well out of the range where you expect the data to be. Can you spot the outliers and invalid values in listing 3.3?

Listing 3.3 Examples of invalid values and outliers

```
> summary(custdata$income)
   Min. 1st Qu. Median     Mean 3rd Qu.      ①
 -8700    14600   35000    53500   67000
   Max.
 615000

> summary(custdata$age)
   Min. 1st Qu. Median     Mean 3rd Qu.      ②
    0.0    38.0   50.0    51.7    64.0
   Max.
 146.7
```

- ① Negative values for income could indicate bad data. They might also have a special meaning, like "amount of debt." Either way, you should check how prevalent the issue is, and decide what to do: Do you drop the data with negative income? Do you convert negative values to zero?
- ② Customers of age zero, or customers with age greater than about 110 are outliers. They fall out of the range of expected customer values. Outliers could be data input errors. They could be special sentinel values: zero might mean "age unknown" or "refuse to state." And some of your customers might be especially long-lived.

Often, invalid values are simply bad data input. Negative numbers in a field like *age*, however, could be a *sentinel value* to designate "unknown". Outliers might

also be data errors, or sentinel values. Or they might be valid but unusual data points—people do occasionally live past 100.

As with missing values, you must decide the most appropriate action: drop the data field, drop the data points where this field is bad, or convert the bad data to a useful value. Even if you feel certain outliers are valid data, you might still want to omit them from model construction (and also collar allowed prediction range), since the usual achievable goal of modeling is to predict the typical case correctly.

DATA RANGE

You also want to pay attention to how much the values in the data vary. If you believe that age or income help to predict health insurance coverage, then you should make sure there is enough variation in the age and income of your customers for you to see the relationships. Let's look at income again, in listing 3.4. Is the data range wide? Is it narrow?

Listing 3.4 Looking at the data range of a variable

```
> summary(custdata$income)
   Min. 1st Qu. Median      Mean 3rd Qu.
 -8700    14600   35000    53500    67000
   Max.
 615000
```

①

- ① Income ranges from zero to over half a million dollars; a very wide range.

Even ignoring negative income, the *income* variable in listing 3.4 ranges from zero to over half a million. That's pretty wide (though typical for income). Data that ranges over several orders of magnitude like this can be a problem for some modeling methods. We'll talk about mitigating data range issues when we talk about logarithmic transformations in chapter XRF:chapter_4:chManagingData.

Data can be too narrow, too. Suppose all your customers were between the ages of 50 and 55. It's a good bet that age range wouldn't be a very good predictor of health insurance coverage for that population, since it doesn't vary much at all.

SIDE BAR**How narrow is “too narrow” a data range?**

Of course, the term *narrow* is relative. If we were predicting the ability to read for children between ages of 5 and 10, then age probably is a useful variable as-is. For data including adult ages, you may want to transform or bin ages in some way, as you don’t expect a significant change in reading ability between ages 40 and 50. You should rely on information about the problem domain to judge if the data range is narrow, but a rough rule of thumb is the ratio of the standard deviation to the mean. If that ratio is very small, then the data isn’t varying much.

We’ll revisit data range in section 3.2, when we talk about examining data graphically.

One factor that determines apparent data range is the unit of measurement. To take a non-technical example, we measure the ages of babies and toddlers in weeks or in months, because developmental changes happen at that time scale for very young children. Suppose we measured baby ages in years. It might appear numerically that there isn’t much difference between a one-year-old and a two-year-old. In reality, there is a dramatic difference, as any parent can tell you! Units can present potential issues in a dataset for another reason, as well.

UNITS

Does the income data in figure 3.5 represent hourly wages, or yearly wages in units of \$1000? As a matter of fact, it’s the latter, but what if you thought it was the former? You might not notice the error during the modeling stage, but down the line, someone will start inputting hourly wage data into the model, and get back bad predictions in return.

Listing 3.5 Checking units sounds silly, but is easy to overlook, and can lead to spectacular errors if not caught

```
> summary(Income)
   Min. 1st Qu. Median     Mean 3rd Qu.    Max.
 -8.7    14.6   35.0    53.5   67.0   615.0
```

1

- 1 The variable Income is defined as “Income = custdata\$income/1000”. But suppose you didn’t know that. Looking only at the summary, the values could plausibly be interpreted to mean either “hourly wage” or “yearly income in units of \$1000.”

Are time intervals measured in days, hours, minutes, or milliseconds? Are

speeds in kilometers per second, miles per hour or knots? Are monetary amounts in dollars, thousands of dollars, or 1/100th of a penny (a customary practice in finance, where calculations are often done in fixed-point arithmetic)? This is actually something that you'll catch by checking data definitions in data dictionaries or documentation, rather than in the summary statistics; the difference between hourly wage data and annual salary in units of \$1000 may not look that obvious at a casual glance. But it's still something to keep in mind while looking over the value ranges of your variables, because often you can spot when measurements are in unexpected units. Automobile speeds in knots look a lot different than they do in miles per hour.

3.2 Spotting problems using graphics and visualization

As you've seen, you can spot plenty of problems just by looking over the data summaries. For other properties of the data, pictures are better than text.

We cannot expect a small number of numerical values [summary statistics] to consistently convey the wealth of information that exists in data. Numerical reduction methods do not retain the information in the data.

-- William Cleveland, *The Elements of Graphing Data*

Figure 3.1 shows a plot of how customer ages are distributed. We'll talk about what the y-axis of the graph means later; for right now, just know that the height of the graph corresponds to how many customers in the population are of that age. As you can see, information like the peak age of the distribution, the existence of subpopulations, and the presence of outliers are easier to absorb visually than they are to determine textually.

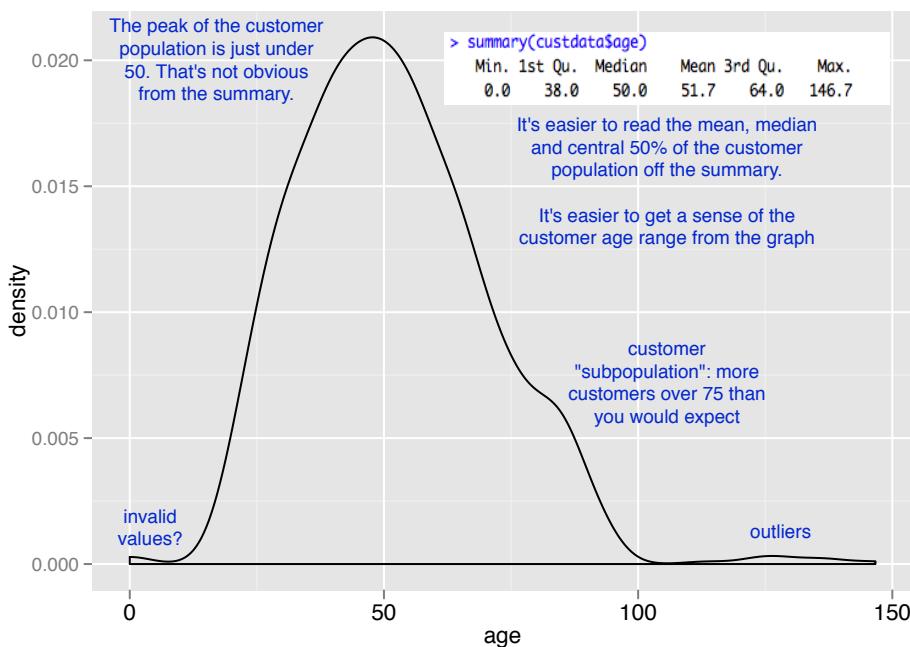


Figure 3.1 Some information is easier to read from a graph, and some from a summary.

The use of graphics to examine data is called *visualization*. We try to follow William Cleveland's principles for scientific visualization. Details of specific plots aside, the key points of Cleveland's philosophy are:

- A graphic should display as much information as it can, with the lowest possible cognitive strain to the viewer.
- Strive for clarity. Make the data stand out. Specific tips for increasing clarity include:
 - Avoid too many superimposed elements, for example too many curves in the same graphing space.
 - Find the right aspect ratio and scaling to properly bring out the details of the data.
 - Avoid having the data all skewed to one side or other of your graph.
- Visualization is an iterative process. Its purpose is to answer questions about the data.

During the visualization stage, you graph the data, learn what you can, and then regraph the data to answer the questions that arise from your previous graphic. Different graphics are best suited for answering different questions. We'll look at some of them in this section.

In this book, we use `ggplot2` to demonstrate the visualizations and graphics; of course, other R visualization packages can produce similar graphics.

SIDE BAR**A note on ggplot2**

The theme of this section is how to use visualization to explore your data, not how to use ggplot2. We chose ggplot2 because it excels at superimposing multiple graphical elements together, but its syntax can take some getting used to. The key points to understand when looking at our code snippets are:

- Graphs in ggplot2 can only be defined on data frames. The variables in a graph—the *x* variable, the *y* variable, the variables that define the color or the size of the points—are called *aesthetics*, and are declared by using the `aes` function.
- The `ggplot()` function declares the graph object. The arguments to `ggplot()` can include the data frame of interest, and the aesthetics. The `ggplot()` function doesn't of itself produce a visualization; visualizations are produced by *layers*.
- Layers produce the plots and plot transformations, and are added to a given graph object using the `+` operator. Each layer can also take a data frame and aesthetics as arguments, in addition to plot-specific parameters. Examples of layers are `geom_point` (for a scatter plot) or `geom_line` (for a line plot).

This syntax will become clearer in the examples that follow. For more information, we recommend Hadley Wickham's reference site <http://ggplot2.org>, which has pointers to online documentation, as well as to Dr. Wickham's ggplot2 reference book.

In the next two sections we'll see how to use pictures and graphs to identify data characteristics and issues. In section 3.2.1 we'll look at visualizations for single variables. In section 3.2.2 we'll look at visualizations for two variables. But let's start by looking at a single variable.

3.2.1 Visually checking distributional assumptions for a single variable

The visualizations in this section help you answer questions like:

- What is peak value of the distribution?
- How many peaks are there in the distribution (Unimodality vs. Bimodality)?
- How normal (or lognormal) is the data? We discuss normal and lognormal distributions in appendix XRF:appendix_B:xAppendixStat.
- How much does the data vary? Is it concentrated in a certain interval, or in a certain category?

One of the things that is easier to see visually is the shape of the data distribution. Except for the blip to the right, the graph in figure 3.1 (which we have reproduced as the gray curve in figure 3.2) is almost shaped like the normal distribution that we saw in appendix XRF:appendix_B:xAppendixStat. As we mention in that section, many analysis methods (linear and logistic regression, in particular) assume that the input data is approximately normal in distribution (at least for continuous variables), so you want to verify whether this is the case.

You can also see that the gray curve in figure 3.2 has only one peak, or that it's *unimodal*. This is another property that you want to check in your data.

Why? Because (roughly speaking), a unimodal distribution corresponds to one population of subjects. For the gray curve in figure 3.2, the mean customer age is about 52, and 50% of the customers are between 38 and 64 (the first and third quartiles). So you can say that a “typical” customer is middle-aged, and probably possesses many of the demographic qualities of a middle-aged person—though of course you have to verify that with your actual customer information.

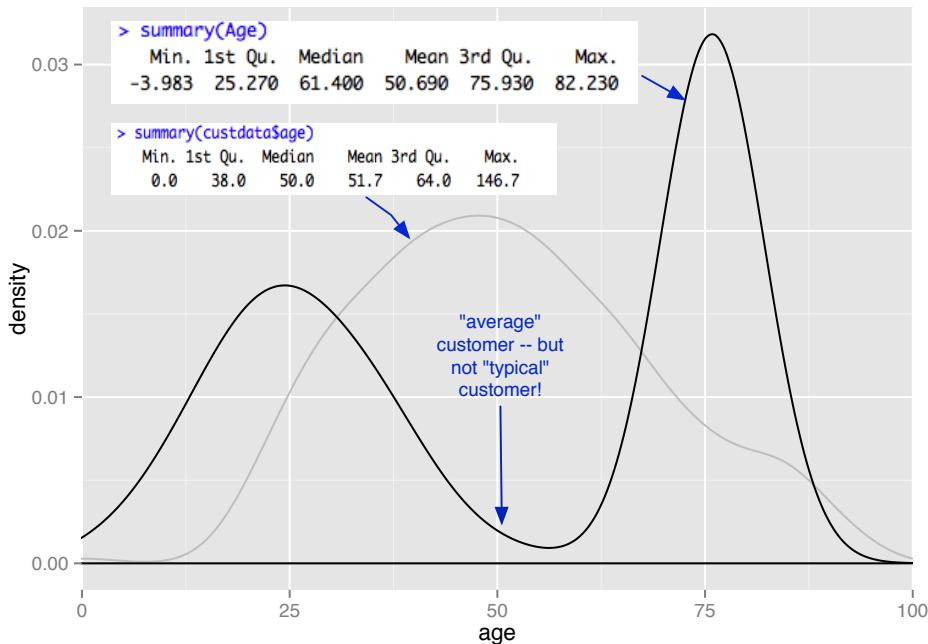


Figure 3.2 A unimodal distribution (in gray) can usually be modelled as coming from a single population of users. With a bimodal distribution (in black), your data often comes from two populations of users.

The black curve in figure 3.2 shows what can happen when you have two peaks, or a *bimodal distribution*. (A distribution with more than two peaks is

simply called *multimodal*.) This set of customers has about the same mean age as the customers represented by the gray curve—but a 50-year old is hardly a “typical” customer! This (admittedly exaggerated) example corresponds to two populations of customers: a fairly young population mostly in their 20s and 30s, and an older population mostly in their 70s. These two populations probably have very different behavior patterns, and if you want to model whether a customer has health insurance or not, it wouldn’t be a bad idea to model the two populations separately—especially if you’re using linear or logistic regression.

The histogram and the density plot are two visualizations that help you quickly examine the distribution of a numerical variable. Figures 3.1 and 3.2 are density plots. Whether you use histograms or density plots is largely a matter of taste. We tend to prefer density plots, but histograms are easier to explain to less quantitatively-minded audiences.

HISTOGRAMS

A basic histogram bins a variable into fixed-width buckets and returns the number of data points that falls into each bucket. For example, you could group your customers by age range, in intervals of five years: 20-25, 25-30, 30-35, and so on. Customers at a boundary age would go into the higher bucket: 25-year-olds go into the 25-30 bucket. For each bucket, you then count how many customers are in that bucket. The resulting histogram is shown in figure 3.3.

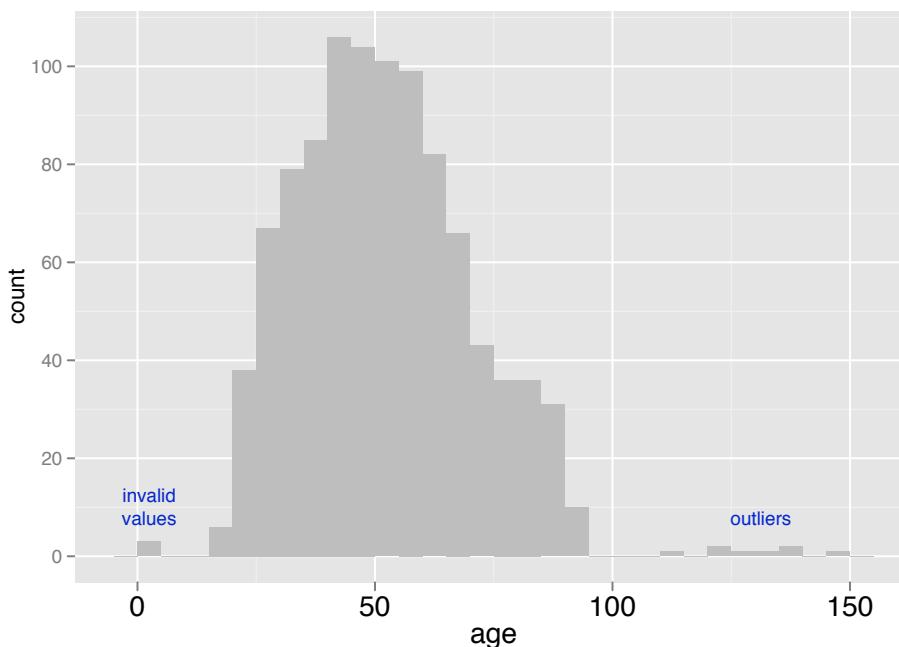


Figure 3.3 A histogram tells you where your data is concentrated. It also visually highlights outliers and anomalies.

You create the histogram in figure 3.3 in `ggplot2` with the `geom_histogram` layer.

```
library(ggplot2)          ①
ggplot(custdata) +
  geom_histogram(aes(x=age),
    binwidth=5, fill="gray") ②
```

- ① Load the `ggplot2` library, if you haven't already done so.
- ② The `binwidth` parameter tells the `geom_histogram` call how to make bins of five-year intervals (default is `datarange/30`). The `fill` parameter specifies the color of the histogram bars (default: black).

The primary disadvantage of histograms is that you must decide ahead of time how wide the buckets are. If the buckets are too wide, you can lose information about the shape of the distribution. If the buckets are too narrow, the histogram can look too noisy to read easily. An alternative visualization is the density plot.

DENSITY PLOTS

You can think of a *density plot* as a “continuous histogram” of a variable, except the area under the density plot is equal to one. A point on a density plot corresponds to the fraction of data (or the percentage of data, divided by 100) that takes on a particular value. This fraction is usually very small. When you look at a density plot, you’re more interested in the overall shape of the curve than in the actual values on the y-axis. You’ve seen the density plot of age; figure 3.4 shows the density plot of income.

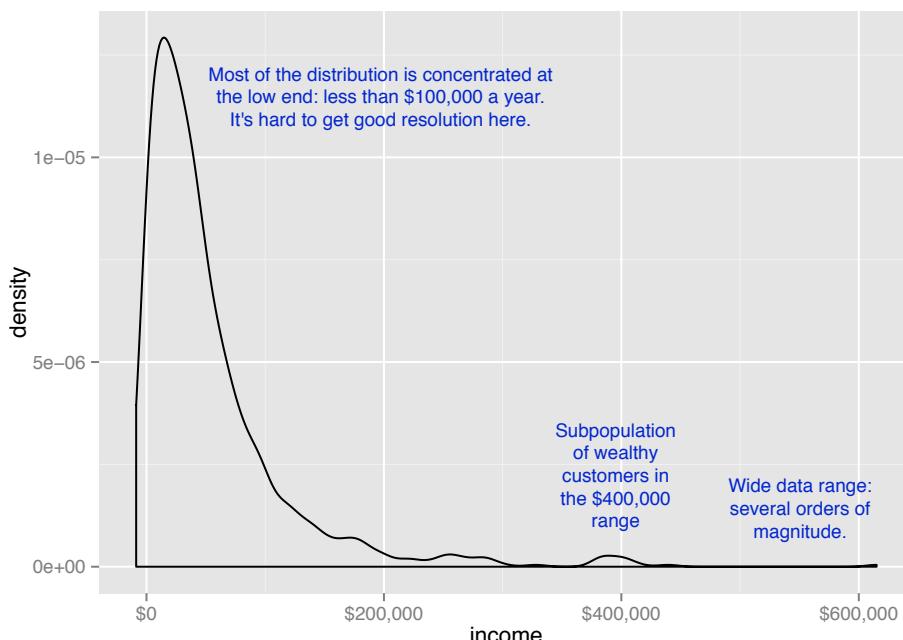


Figure 3.4 Density plots also show where data is concentrated. This plot also highlights a population of higher-income customers.

You produce figure 3.4 with the `geom_density` layer.

```
library(scales) ①  
ggplot(custdata) + geom_density(aes(x=income)) +  
  scale_x_continuous(labels=dollar) ②
```

- ① The scales package brings in the dollar scale notation.
- ② Set the x-axis labels to dollars.

When the data range is very wide, and the mass of the distribution is heavily concentrated to one side, like the distribution in figure 3.4, it’s difficult to see the

details of its shape. For instance, it's hard to tell the exact value where the income distribution has its peak. If the data is non-negative, then one way to bring out more detail is to plot the distribution on a logarithmic scale, as shown in figure 3.5. This is equivalent to plotting the density plot of $\log_{10}(\text{income})$.

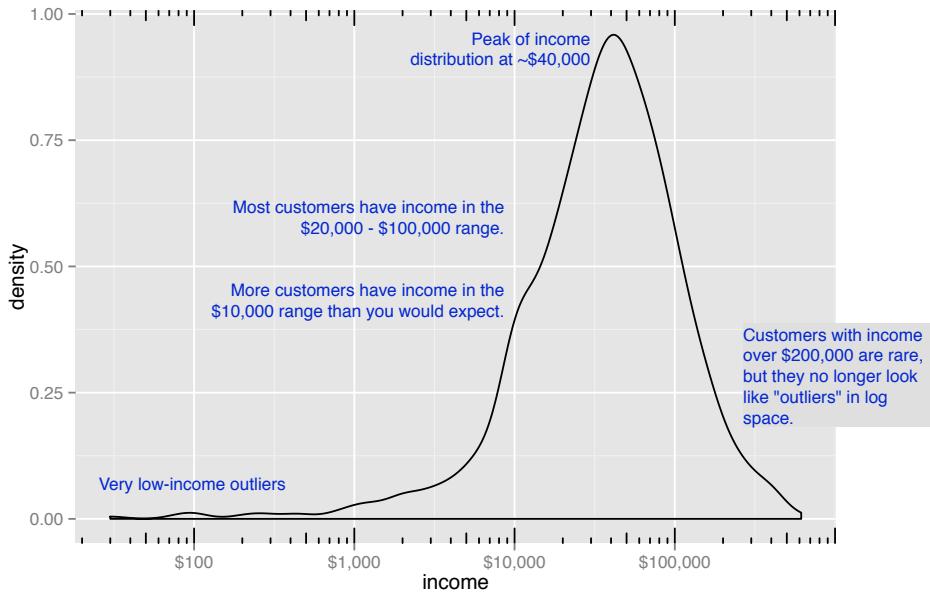


Figure 3.5 The density plot of income on a \log_{10} scale highlights details of the income distribution that are harder to see in a regular density plot.

In `ggplot2`, you can plot figure 3.5 with the `geom_density` and `scale_x_log10` layers. See, for example, listing 3.6.

Listing 3.6 Creating a log-scaled density plot

```
ggplot(custdata) + geom_density(aes(x=income)) +
  scale_x_log10(breaks=c(100,1000,10000,100000), labels=dollar) + ①
  annotation_logticks(sides="bt") ②
```

- ① Set the x-axis to be in \log_{10} scale, with manually set tick points and labels as dollars
- ② Add log-scaled tick marks to the top and bottom of the graph

When you issued the preceding command, you also got back a warning message:

```
Warning messages:
1: In scale$trans$trans(x) : NaNs produced
```

```
2: Removed 79 rows containing non-finite values (stat_density).
```

This tells you that `ggplot2` ignored the zero and negative valued rows (since $\log(0)$ is Infinity), and that there were 79 such rows. Keep that in mind when evaluating the graph.

TIP

When should you use a logarithmic scale?

You should use a logarithmic scale when percent change, or change in orders of magnitude, is more important than changes in absolute units. You should also use a log scale to better visualize data that is heavily skewed.

For example, in income data, a difference in income of five thousand dollars means something very different in a population where the incomes tend to fall in the tens of thousands of dollars than it does in populations where income falls in the hundreds of thousands or millions of dollars. In other word, what constitutes a “significant difference” depends on the order of magnitude of the incomes you’re looking at. Similarly, in a population like that in figure 3.5, a few people with very high income will cause the majority of the data to be compressed into a relatively small area of the graph. For both those reasons, plotting the income distribution on a logarithmic scale is a good idea.

In log space, income is distributed as something that looks like a “normalish” distribution, as you would expect from our discussion in appendix XRF:appendix_B:xAppendStat. It’s not exactly a normal (in fact, it appears to be at least two normal distributions mixed together), but it’s close enough to normal for analysis methods like linear or logistic regression.

BAR CHARTS

A *bar chart* is a histogram for discrete data: it records the frequency of every value of a categorical variable. Figure 3.6 shows the distribution of marital status in your customer dataset. If you believe that marital status helps predict health insurance coverage, then you want to check that you have enough customers with different marital statuses to help you discover the relationship between being married (or not) and having health insurance.

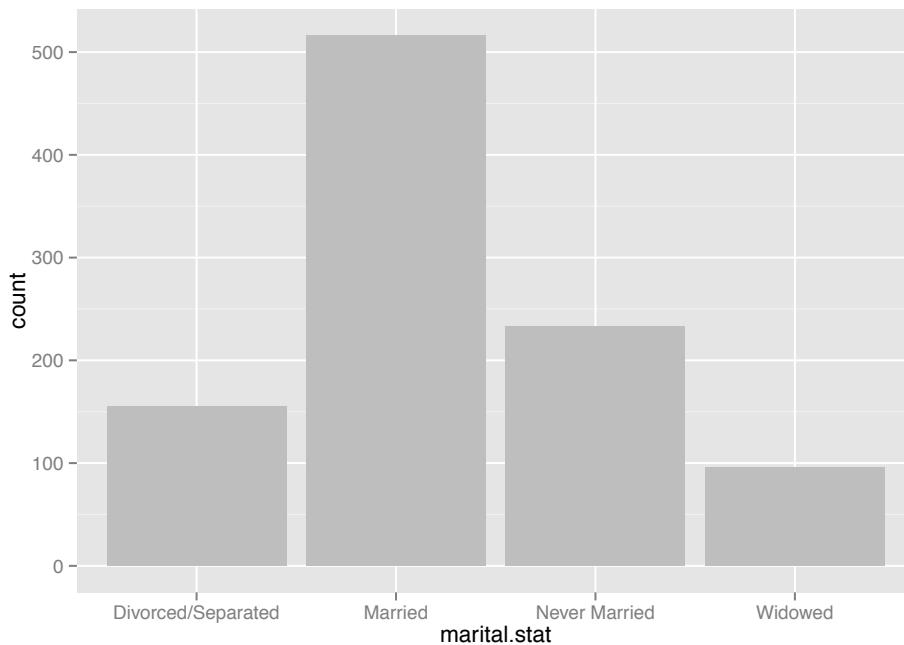


Figure 3.6 Bar charts show the distribution of categorical variables.

The `ggplot2` command to produce figure 3.6 uses `geom_bar`.

```
ggplot(custdata) + geom_bar(aes(x=marital.stat), fill="gray")
```

This graph doesn't really show any more information than `summary(custdata$marital.stat)` would show, although some people find the graph easier to absorb than the text. Bar charts are most useful when the number of possible values is fairly large, like state of residence. In this situation, we often find that a sideways graph is more legible than an upright graph.

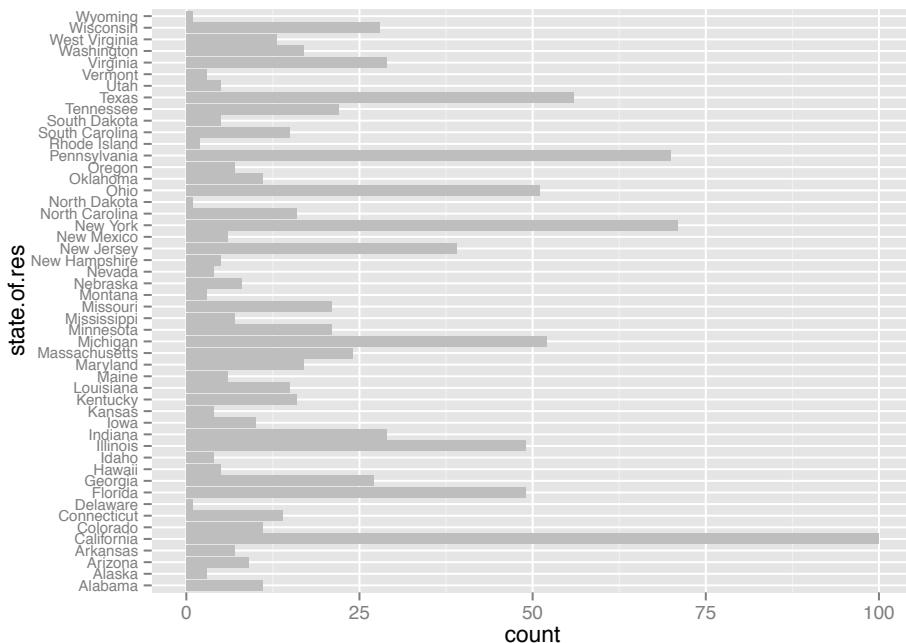


Figure 3.7 A sideways bar chart can be easier to read when there are several categories with long names.

The `ggplot2` command to produce figure 3.7 is

```
ggplot(custdata) +
  geom_bar(aes(x=state.of.res), fill="gray") + ①
  coord_flip() + ②
  theme(axis.text.y=element_text(size=rel(0.8))) ③
```

- ① Plot bar chart as before: `state.of.res` is on x axis, `count` is on y-axis
- ② Flip the x and y axes: `state.of.res` is now on the y-axis
- ③ Reduce the size of the y-axis tick labels to 80% of default size for legibility

Cleveland³ recommends that the data in a bar chart (or in a *dot plot*, Cleveland's preferred visualization in this instance) be sorted, to more efficiently extract insight from the data. This is shown in figure 3.8.

Footnote 3 Cleveland, William S. *The Elements of Graphing Data*, Hobart Press, 1994.

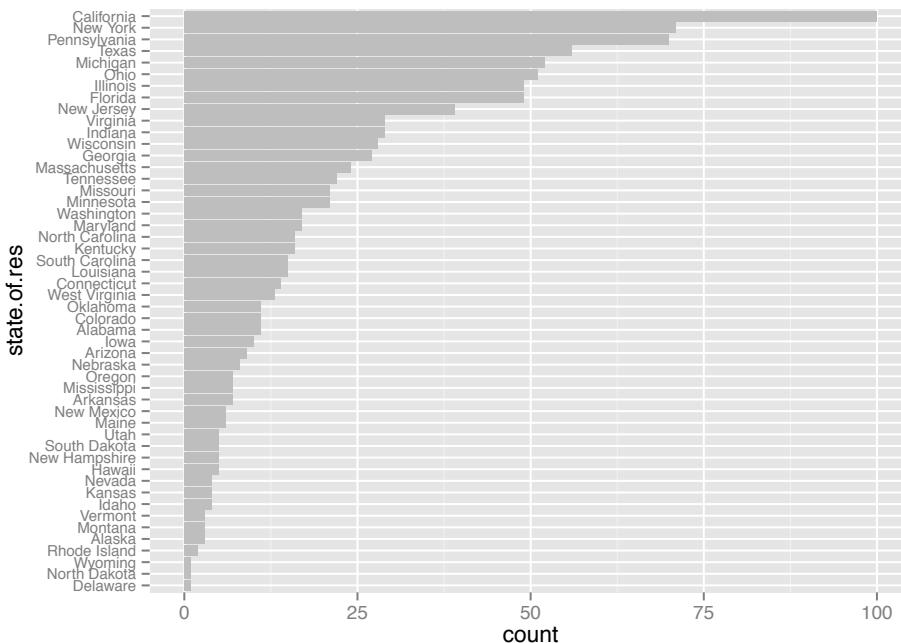


Figure 3.8 Sorting the bar chart by count makes it even easier to read

This visualization requires a bit more manipulation, at least in `ggplot2`, because by default, `ggplot2` will plot the categories of a factor variable in alphabetical order. To change this, we have to manually specify the order of the categories—in the factor variable, not in `ggplot2`.

```

> statesums <- table(custdata$state.of.res)      ①
> statef <- as.data.frame(statesums)            ②
> colnames(statef)<-c("state.of.res", "count")  ③
> summary(statef)                                ④
state.of.res      count
Alabama     : 1    Min.   :  1.00
Alaska      : 1    1st Qu.:  5.00
Arizona     : 1    Median : 12.00
Arkansas    : 1    Mean    : 20.00
California: 1    3rd Qu.: 26.25
Colorado    : 1    Max.    :100.00
(Other)     :44
> statef <- transform(statef,
  state.of.res=reorder(state.of.res, count))       ⑤
> summary(statef)                                ⑥
state.of.res      count
Delaware     : 1    Min.   :  1.00
North Dakota: 1    1st Qu.:  5.00
Wyoming     : 1    Median : 12.00
Rhode Island: 1    Mean   : 20.00
Alaska       : 1    3rd Qu.: 26.25

```

```

Montana      : 1    Max.    :100.00
(Other)      :44
> ggplot(statef)+ geom_bar(aes(x=state.of.res,y=count),
  stat="identity",
  fill="gray") +
  coord_flip() +
  theme(axis.text.y=element_text(size=rel(0.8)))

```

- ➊ The table() command aggregates the data by state of residence—exactly the information that bar chart plots.
- ➋ Convert the table object to a data frame using as.data.frame(). The default column names are Var1 and Freq.
- ➌ Rename the columns for readability.
- ➍ Notice that the default ordering for the state.of.res variable is alphabetical
- ➎ Use the reorder() function to set the state.of.res variable to be count-ordered. Use the transform() function to apply the transformation to the statef data frame.
- ➏ The state.of.res variable is now count ordered.
- ➐ Since the data is being passed to geom_bar pre-aggregated, specify both the x and y variables, and use stat="identity" to plot the data exactly as given.
- ➑ Flip the axes and reduce the size of the label text as before.

Before we move on to visualizations for two variables, let's summarize the visualizations that we've reviewed in this section.

Table 3.1 Visualizations for one variable

Graph type	Uses
Histogram	<ul style="list-style-type: none"> • Examine data range • Check number of modes • Check if distribution is normal/lognormal • Check for anomalies and outliers
Density plot	<ul style="list-style-type: none"> • Examine data range • Check number of modes • Check if distribution is normal/lognormal • Check for anomalies and outliers
Bar chart	Compare relative or absolute frequencies of the values of a categorical variable.

3.2.2 Visually checking relationships between two variables

In addition to examining variables in isolation, you will often want to look at the relationship between two variables. For instance you want to answer questions like:

- Is there a relationship between the two inputs *age* and *income* in my data?
- What kind of relationship, and how strong?
- Is there a relationship between the input *marital status* and the output *health insurance*? How strong?

You will precisely quantify these relationships during the modeling phase, but exploring them now gives you a feel for the data, and helps you determine which variables are the best candidates to include in a model.

First, let's consider the relationship between two continuous variables. The most obvious way (though not always the best) is the line plot

LINE PLOTS

Line plots work best when the relationship between two variables is relatively clean: each x-value has a unique (or nearly unique) y-value, as in figure 3.9.

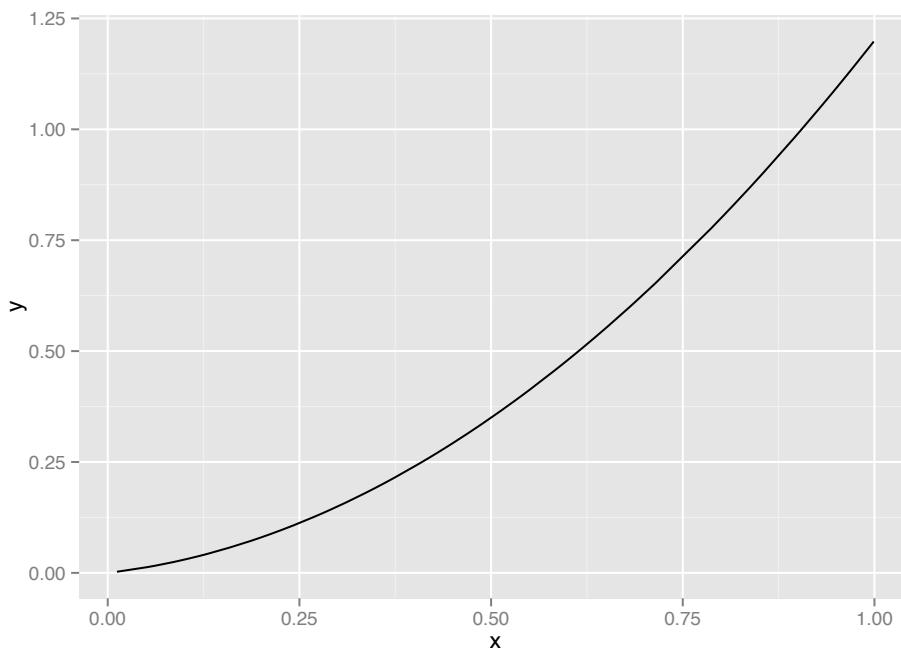


Figure 3.9 Example of a line plot.

You plot figure 3.9 with `geom_line`.

```
x <- runif(100) ①  
y <- x^2 + 0.2*x ②  
ggplot(data.frame(x=x,y=y), aes(x=x,y=y)) + geom_line() ③
```

- ① First, generate the data for this example. The x variable is uniformly randomly distributed between zero and one.
- ② The y variable is a quadratic function of x
- ③ Plot the line plot

When the data is not so cleanly related, line plots aren't as useful; you'll want to use the scatter plot instead, as we'll see in the next section.

SCATTER PLOTS AND SMOOTHING CURVES

You'd expect there to be a relationship between age and health insurance, and also a relationship between income and health insurance. But what is the relationship between age and income? If they track each other perfectly, then you might not want to use both variables in a model for health insurance. The appropriate summary statistic is the correlation, which we compute on a safe subset of our data:

```
custdata2 <- subset(custdata,  
  (custdata$age > 0 & custdata$age < 100  
  & custdata$income > 0)) ①  
  
cor(custdata2$age, custdata2$income) ②  
  
[1] -0.02240845 ③
```

- ① Only consider subset of data with reasonable age and income values
- ② Get correlation of age and income
- ③ Resulting correlation

The negative correlation is surprising, since you'd expect that income should increase as people get older. A visualization gives you more insight into what is going on than a single number can. We can try a scatter plot first (figure 3.10).

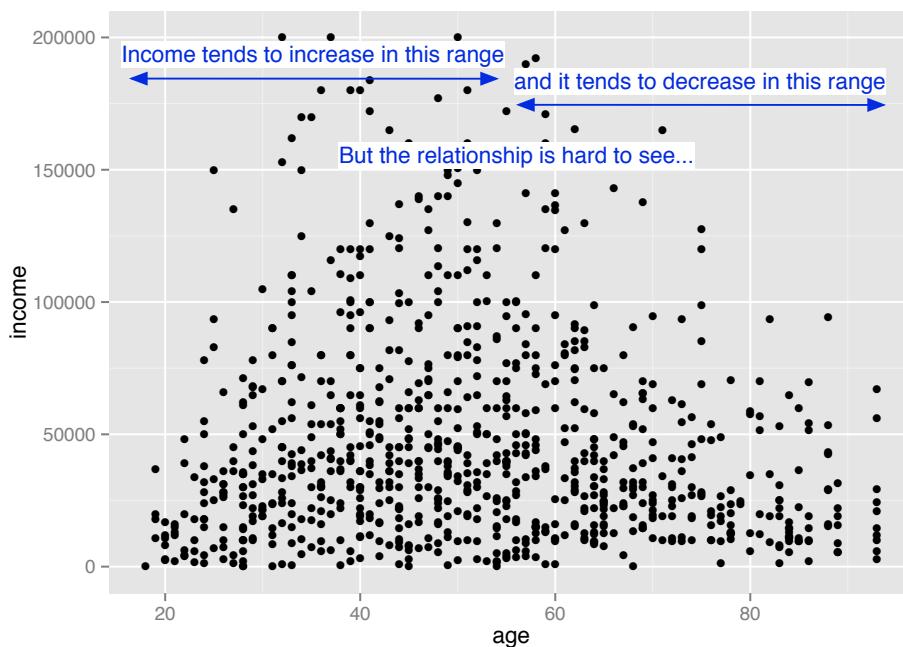


Figure 3.10 A scatter plot of income versus age.

You plot figure 3.10 with `geom_point`.

```
ggplot(custdata2, aes(x=age, y=income)) +  
  geom_point() + ylim(0, 200000)
```

The relationship between age and income isn't easy to see. You can try to make the relationship more clear by also plotting a linear fit through the data, as shown in figure 3.11.

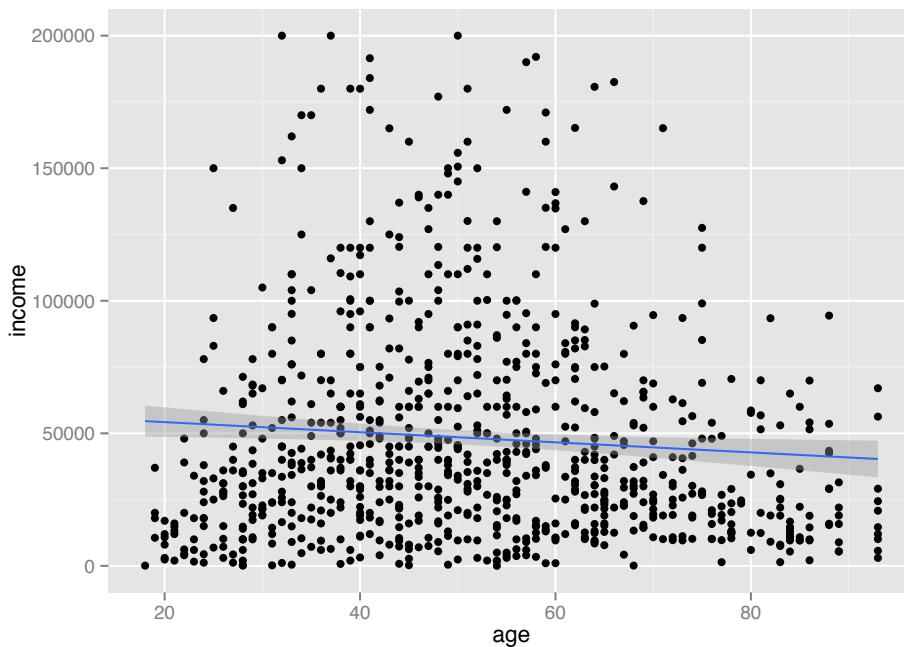


Figure 3.11 A scatter plot of income versus age, with a linear fit.

You plot figure 3.11 using the `stat_smooth` layer.⁴

Footnote 4 The `stat` layers in `ggplot2` are the layers that perform transformations on the data. They are usually called under the covers by the `geom` layers. Sometimes you have to call them directly, to access parameters that aren't accessible from the `geom` layers. In this case, the default smoothing curve used `geom_smooth` (which we'll see shortly) is a loess curve. To plot a linear fit we must call `stat_smooth` directly.

```
ggplot(custdata2, aes(x=age, y=income)) + geom_point() +
  stat_smooth(method="lm") +
  ylim(0, 200000)
```

In this case, the linear fit doesn't really capture the shape of the data. You can better capture the shape by instead plotting a smoothing curve through the data, as shown in figure 3.12.

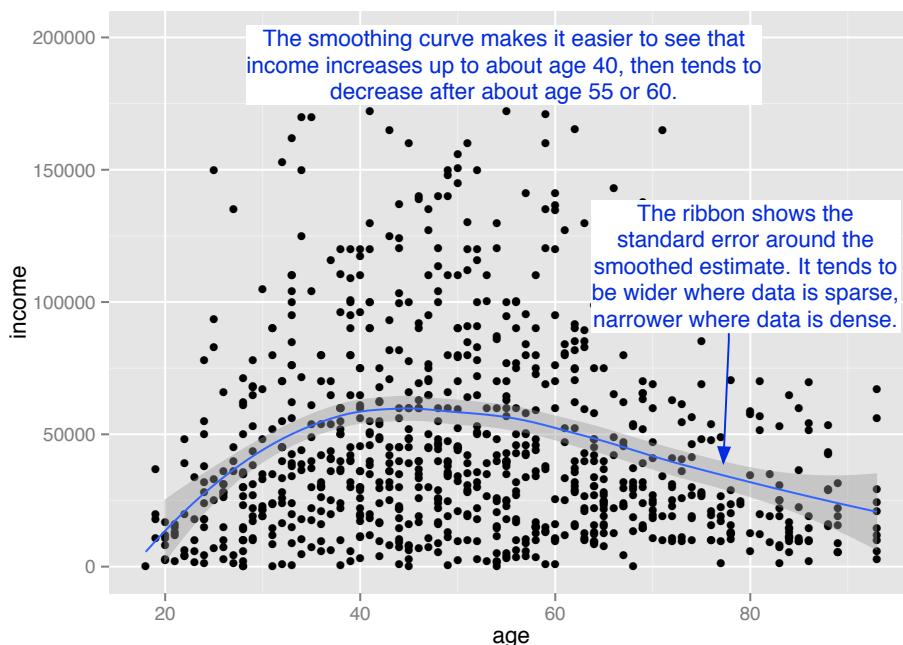


Figure 3.12 A scatter plot of income versus age, with a smoothing curve.

In R, smoothing curves are fit using the `loess` (or `lowess`) functions, which calculate smoothed local linear fits of the data. In `ggplot2`, you can plot a smoothing curve to the data by using `geom_smooth`.

```
ggplot(custdata2, aes(x=age, y=income)) +
  geom_point() + geom_smooth() +
  ylim(0, 200000)
```

A scatter plot with smoothing curve also makes a good visualization of the relationship between a continuous variable and a boolean. Suppose you're considering using age as an input to your health insurance model. You might want to plot health insurance coverage as a function of age, as shown in figure 3.13. This will show you that the probability of having health insurance increases as customer age increases.

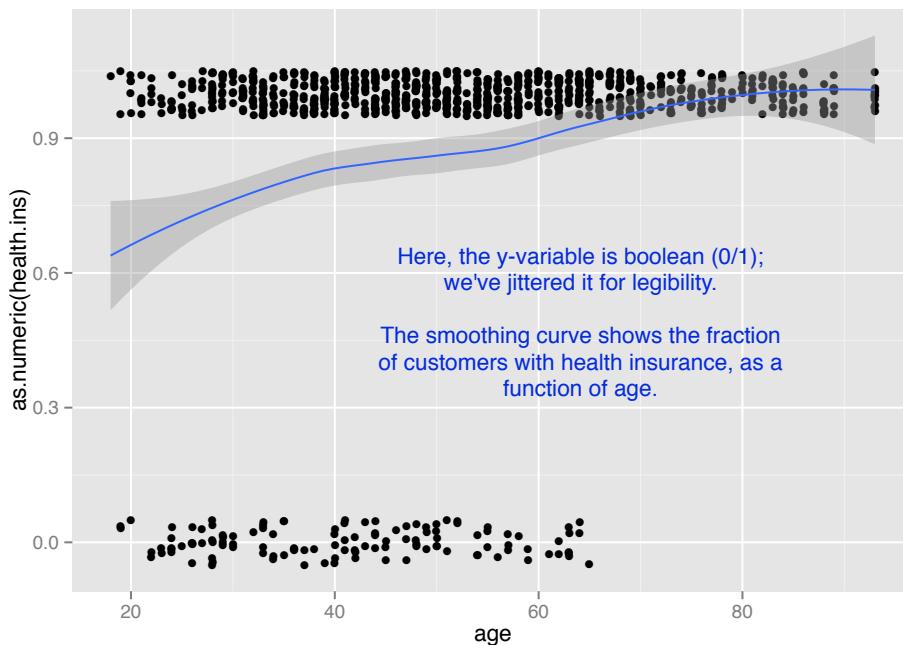


Figure 3.13 Fraction of customers with health insurance, as a function of age

You plot figure 3.13 with the command:

```
ggplot(custdata2, aes(x=age, y=as.numeric(health.ins))) + ①
  geom_point(position=position_jitter(w=0.05, h=0.05)) + ②
  geom_smooth() ③
```

- ① The boolean variable `health.ins` must be converted to a 0/1 variable using `as.numeric`
- ② Since y values can only be zero or one, add a small jitter to get a sense of data density
- ③ Add smoothing curve

In our health insurance examples, the dataset is small enough that the scatter plots that you've created are still legible. If the dataset were a hundred times bigger, there would be so many points that they would begin to plot on top of each other; the scatter plot would turn into an illegible smear. In high-volume situations like this, try an aggregated plot, like a hexbin plot.

HEXBIN PLOTS

A *hexbin plot* is like a two-dimensional histogram. The data is divided into bins, and the number of data points in each bin is represented by color or shading. Let's go back to the income versus age example. Figure 3.14 shows a hexbin plot of the data. Note how the smoothing curve traces out the shape formed by the densest region of data.

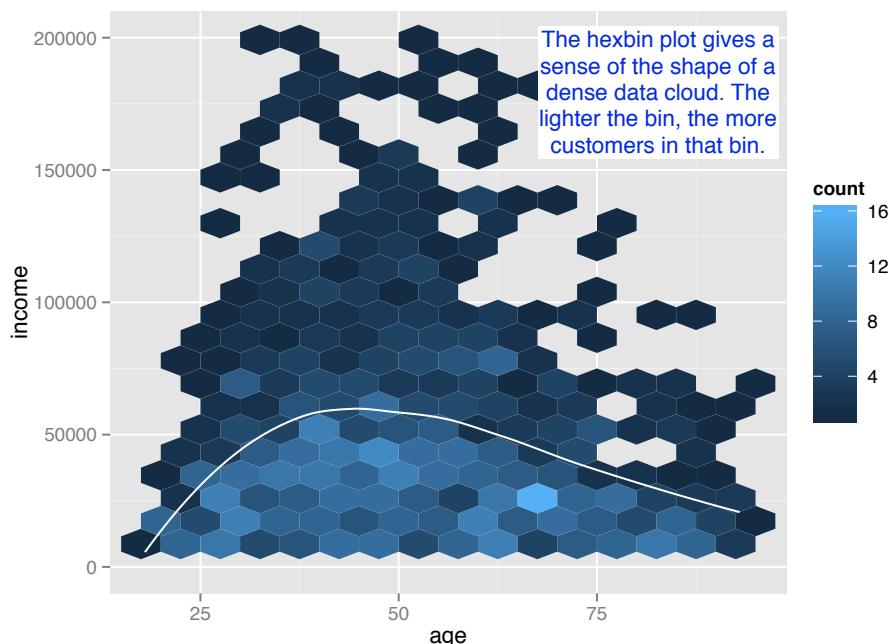


Figure 3.14 Hexbin plot of income versus age, with a smoothing curve superimposed in white.

To make a hexbin plot in R, you must have the `hexbin` package installed. We discuss how to install R packages in appendix XRF:appendix_A:xAppendixR. Once `hexbin` is installed and the library loaded, you create the plots using the `geom_hex` layer.

```
library(hexbin)    ①  
ggplot(custdata2, aes(x=age, y=income)) +  
  geom_hex(binwidth=c(5, 10000)) + ②  
  geom_smooth(color="white", se=F) + ③"  
  ylim(0,200000)
```

① Load hexbin library

②

- Create hexbin with age binned into 5-year increments, income in increments of \$10,000
- ③ Add smoothing curve in white, suppress standard error ribbon (se=F)

In this section and the previous section we've looked at plots where at least one of the variables is numerical. But in our health insurance example, the output is categorical, and so are many of the input variables. Next we'll look at ways to visualize the relationship between two categorical variables.

BAR CHARTS FOR TWO CATEGORICAL VARIABLES

Let's examine the relationship between marital status and health insurance. The most straightforward way to visualize this is with a *stacked bar chart*, as shown in figure 3.15.

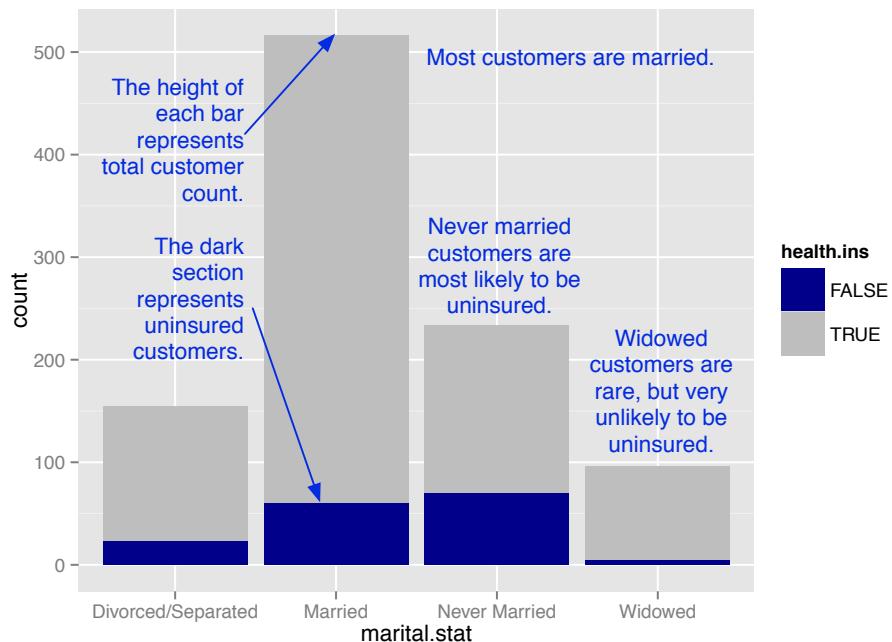


Figure 3.15 Health insurance versus marital status: stacked bar chart

Some people prefer the side-by-side bar chart, shown in figure 3.16, which makes it easier to compare the number of both insured and uninsured across categories.

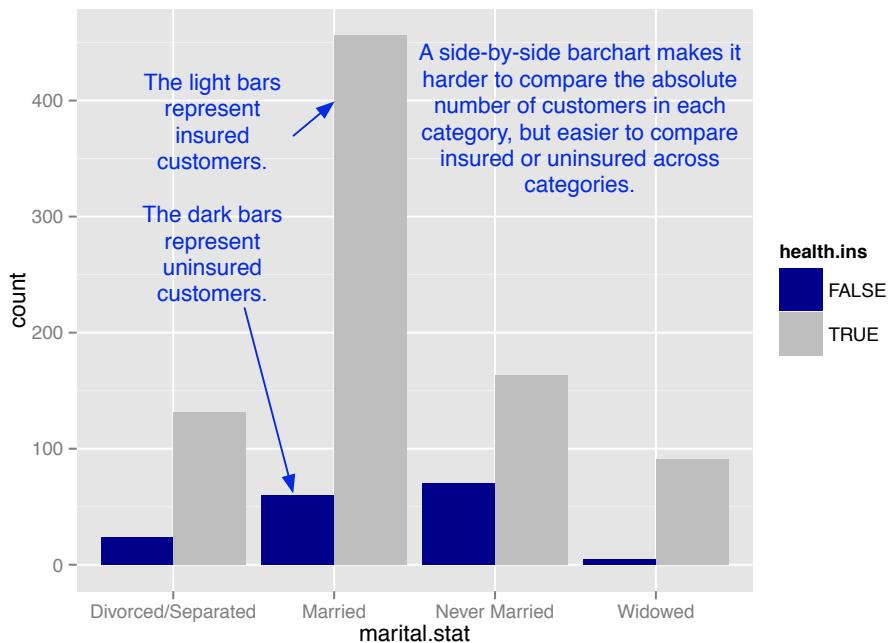


Figure 3.16 Health insurance versus marital status: side-by-side bar chart

The main shortcoming of both the stacked and side-by-side bar charts is that you can't easily compare the ratios of insured to uninsured across categories, especially for rare categories like *Widowed*. You can use what ggplot2 calls a *filled bar chart* to plot a visualization of the ratios directly, as in figure 3.17.

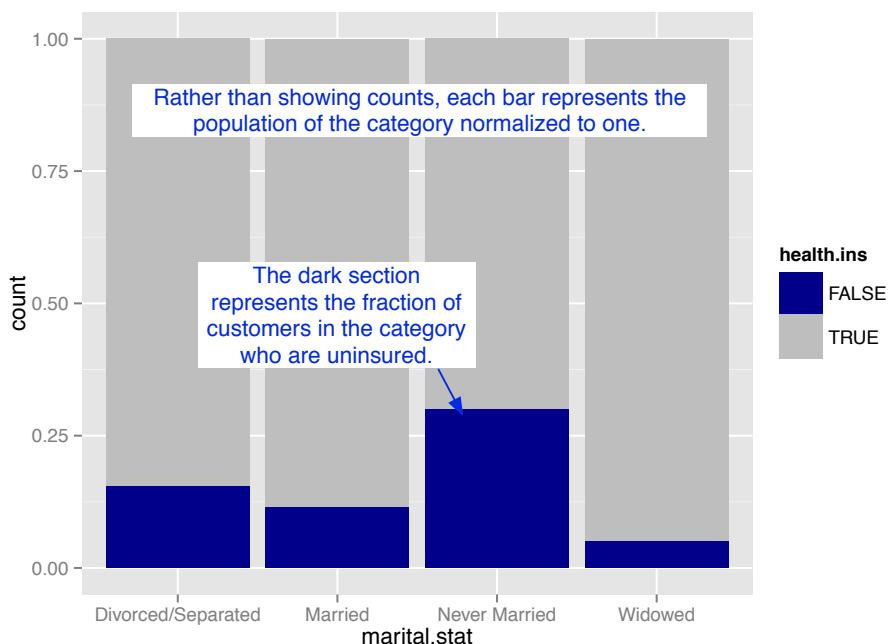


Figure 3.17 Health insurance versus marital status: filled bar chart

The filled bar chart makes it obvious that divorced customers are slightly more likely to be uninsured than married ones. But you've lost the information that being widowed, though highly predictive of insurance coverage, is a rare category.

Which bar chart you use depends on what information is most important for you to convey. The `ggplot2` commands for each of these plots are given next. Note the use of the `fill` aesthetic; this tells `ggplot2` to color (fill) the bars according to the value of the variable `health.ins`. The `position` argument to `geom_bar` specifies the bar chart style.

```
ggplot(custdata) + geom_bar(aes(x=marital.stat,  
    fill=health.ins))
```

① Stacked bar chart,
the default

```
ggplot(custdata) + geom_bar(aes(x=marital.stat,  
    fill=health.ins),  
    position="dodge")
```

② Side-by-side bar
chart

```
ggplot(custdata) + geom_bar(aes(x=marital.stat,  
    fill=health.ins),  
    position="fill")
```

③ Filled bar chart

To get a simultaneous sense of both the population in each category and the ratio of insured to uninsured, you can add what's called a *rug* to the filled bar chart. A rug is a series of ticks or points on the x-axis, one tick per datum. The rug is dense where you have a lot of data, and sparse where you have little data. This is shown in figure 3.18.

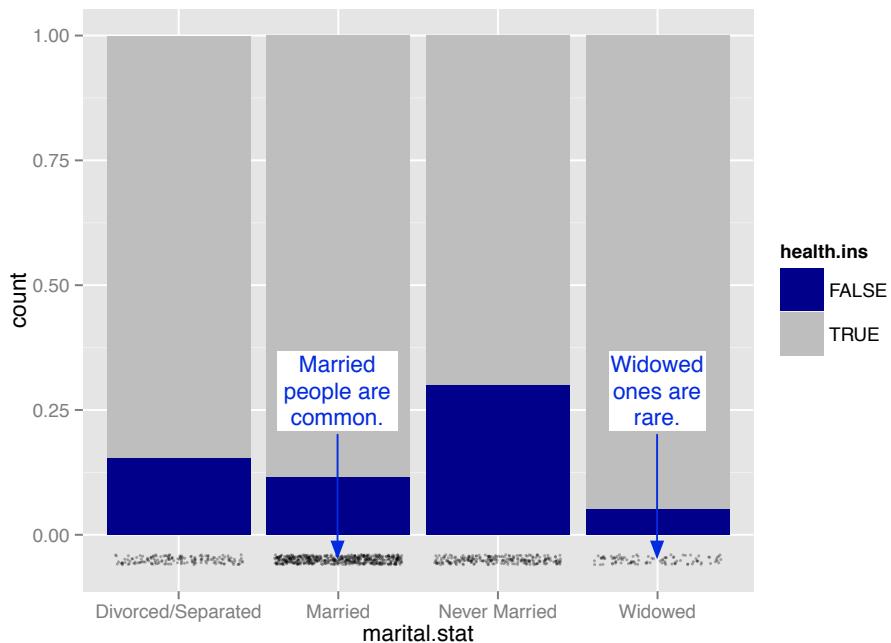


Figure 3.18 Health insurance versus marital status: filled bar chart with rug

You generate this graph by adding a `geom_point` layer to the graph.

```
ggplot(custdata, aes(x=marital.stat)) +
  geom_bar(aes(fill=health.ins), position="fill") +
  geom_point(aes(y=-0.05), size=0.75, alpha=0.3, ①
  position=position_jitter(h=0.01)) ②
```

- ① Set the points just under the y-axis, three-quarters of default size, and make them slightly transparent with the alpha parameter
- ② Jitter the points slightly for legibility

In the preceding examples, one of the variables was binary; the same plots can be applied to two variables that each have several categories, but the results are harder to read. Suppose you're interested in the distribution of marriage status across housing types. I find the side-by-side bar chart easiest to read in this situation, but it's not perfect, as you see in figure 3.19.

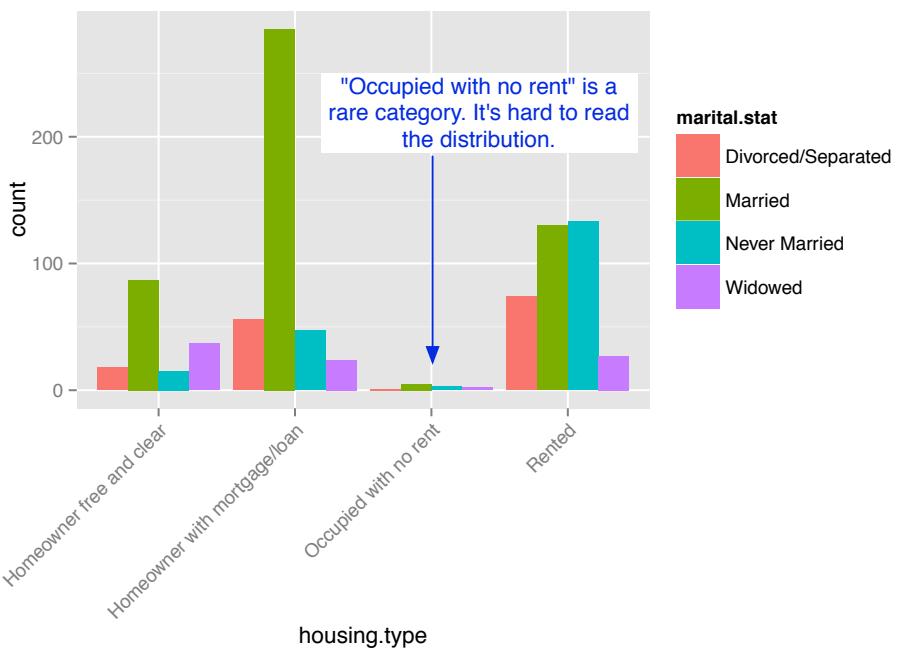


Figure 3.19 Distribution of marital status by housing type: side-by-side bar chart

A graph like figure 3.19 gets cluttered if either of the variables has a large number of categories. A better alternative is to break the distributions into different graphs, one for each housing type. In `ggplot2` this is called *faceting* the graph, and you use the `facet_wrap` layer. The result is in figure 3.20.

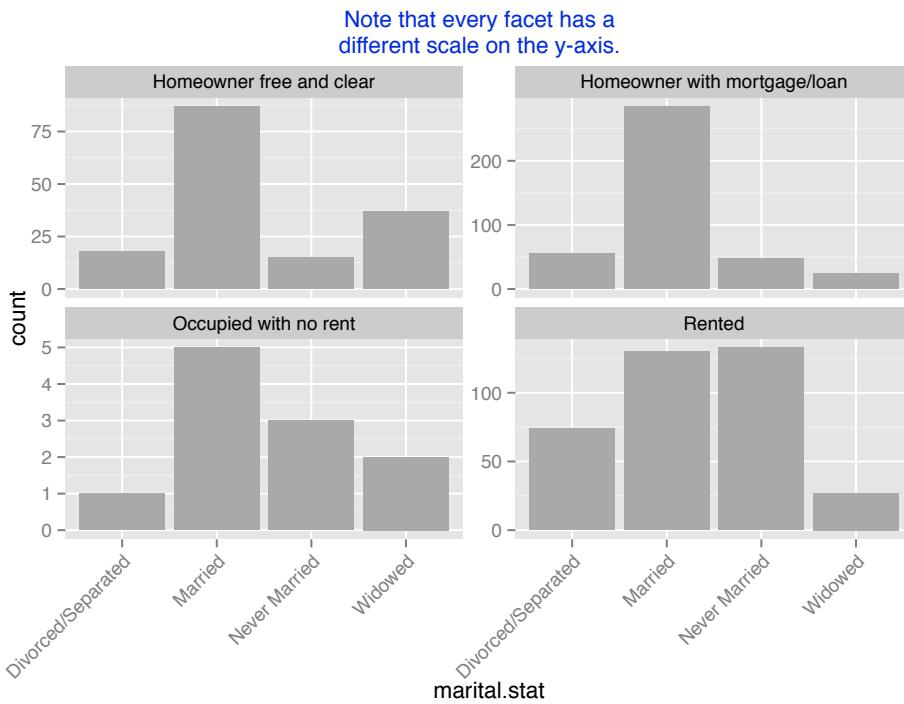


Figure 3.20 Distribution of marital status by housing type: faceted side-by-side bar chart

The code for figure 3.19 and figure 3.20:

```

ggplot(custdata2) +
  geom_bar(aes(x=housing.type, fill=marital.stat ),
  position="dodge") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) 1
2

ggplot(custdata2) +
  geom_bar(aes(x=marital.stat), position="dodge",
  fill="darkgray") +
  facet_wrap(~housing.type, scales="free_y") + 3
4
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) 5
5

```

- ➊ Side-by-side bar chart
- ➋ Tilt the x-axis labels so they don't overlap. You can also use `coord_flip()` to rotate the graph, as we saw previously. I prefer `coord_flip()` because the `theme()` layer is complicated to use.
- ➌ The faceted bar chart.
- ➍ Facet the graph by `housing.type`. The `scales="free_y"` argument specifies that each facet has an independently scaled y-axis (the default is that all facets have the same scales on both axes). The argument `free_x` would free the x-axis scaling, and the argument `free` frees both axes.
- ➎ As of this writing, `facet_wrap` is incompatible with `coord_flip`, so we have to tilt the x-axis labels.

Let's summarize the visualizations for two variables that we've covered.

Table 3.2 Visualizations for two variables

Graph type	Uses
Line plot	Shows the relationship between two continuous variables. Best when that relationship is functional, or nearly so.
Scatter plot	Shows the relationship between two continuous variables. Best when the relationship is looser or more “cloudlike” than can be easily seen on a line plot.
Smoothing curve	Shows underlying “average” relationship, or trend, between two continuous variables. Can also be used to show the relationship between a continuous and a binary or boolean variable: the fraction of <i>true</i> values of the discrete variable as a function of the continuous variable.
Hexbin plot	Shows the relationship between two continuous variables when the data is very dense.
Stacked bar chart	Shows the relationship between two categorical variables (<i>var1</i> and <i>var2</i>). Highlights the frequencies of each value of <i>var1</i> .
Side-by-side bar chart	Shows the relationship between two categorical variables (<i>var1</i> and <i>var2</i>). Good for comparing the frequencies of each value of <i>var2</i> across the values of <i>var1</i> . Works best when <i>var2</i> is binary.
Filled bar chart	Shows the relationship between two categorical variables (<i>var1</i> and <i>var2</i>). Good for comparing the relative frequencies of each value of <i>var2</i> within each value of <i>var1</i> . Works best when <i>var2</i> is binary.
Bar chart with faceting	Shows the relationship between two categorical variables (<i>var1</i> and <i>var2</i>). Best for comparing the relative frequencies of each value of <i>var2</i> within each value of <i>var1</i> when <i>var2</i> takes on more than two values.

There are many other variations and visualizations you could use to explore the data; the preceding set covers some of the most useful and basic graphs. You

should try different kinds of graphs to get different insights from the data. It's an interactive process. One graph will raise questions that you can try to answer by replotting the data again, with a different visualization.

Eventually, you have explored your data enough to get a sense of it, and to have spotted most major problems and issues. In the next chapter, we'll discuss some ways to address common problems that you may have discovered in the data.

3.3 Summary

At this point, you've gotten a feel for your data. You've explored it through summaries and visualizations; you now have a sense of the quality of your data, and of the relationships among your variables. You've caught and are ready to correct several kinds of data issues—although you will likely run into more issues as you progress.

Maybe some of the things you've discovered have led you to re-evaluate the question you're trying to answer, or to modify your goals. Maybe you've decided that you need more or different types of data to achieve your goals. This is all good. As we mentioned in the previous chapter, the data science process is made of loops within loops. The data exploration and data cleaning stages (we'll discuss cleaning in the next chapter) are two of the more time-consuming—and also the most important—stages of the process. Without good data, you can't build good models. Time you spend here is time you don't waste elsewhere.

SIDE BAR Key takeaways

- Take the time to examine your data before diving into the modeling.
- The `summary` command helps you spot issues with data range, units, data type, and missing or invalid values.
- Visualization additionally gives you a sense of data distribution and relationships among variables.
- Visualization is an iterative process and helps answer questions about the data. Time spent here is time not wasted during the modeling process.

In the next chapter, we'll talk about fixing the issues that you've discovered in the data.

3.4 External links section

Managing data



This chapter will cover

- Common data cleaning steps
- Organizing your data for the modeling process.

In chapter 3, you learned how to explore your data and to identify common data issues. In this chapter, we'll see how to fix the data issues that you've discovered. After that, we'll talk about organizing the data for the modeling process.

4.1 Cleaning data

In this section, we'll address issues that you discovered during the data exploration/visualization phase. This will include

- Treating missing values
- Common data transformations
 - Converting continuous variables to discrete
 - Normalization and rescaling
 - Logarithmic transformations

4.1.1 Treating missing values (NAs)

Let's take another look at some of the variables with missing values in our customer data set from the previous chapter. We've reproduced the summary in figure 4.1.

These variables are only missing a few values. It's probably safe to just drop the rows that are missing values -- especially if the missing values are all in the same 56 rows.

	housing.type	recent.move	num.vehicles
Homeowner free and clear	:157	Mode :logical	Min. :0.000
Homeowner with mortgage/loan	:412	FALSE:820	1st Qu.:1.000
Occupied with no rent	: 11	TRUE :124	Median :2.000
Rented	:364	NA's :56	Mean :1.916
NA's	: 56		3rd Qu.:2.000
			Max. :6.000
			NA's :56

is.employed	The <i>is.employed</i> variable is missing many values. Why?
Mode :logical	Is employment status unknown?
FALSE:73	Did the company only start collecting employment information recently?
TRUE :599	Does NA mean "not in the active workforce" (for example, students or stay-at-home parents)?
NA's :328	

Figure 4.1 Variables with missing values

Fundamentally, there are two things you can do with these variables: drop the rows with missing values, or convert the missing values to a meaningful value.

TO DROP OR NOT TO DROP?

Remember that we have a dataset of 1,000 customers; 56 missing values represents about 6% of the data. That's not trivial, but it's not huge, either. The fact that three variables are all missing exactly 56 values suggests that it's the same 56 customers in each case. That's easy enough to check.

```
> summary(custdata[is.na(custdata$housing.type),      1
                  c("recent.move", "num.vehicles")])    2

recent.move      num.vehicles      3
Mode:logical    Min.     : NA
NA's:56          1st Qu.: NA
                  Median : NA
                  Mean    :NaN
                  3rd Qu.: NA
                  Max.    : NA
                  NA's    :56
```

- ➊ Restrict to the rows where *housing.type* is NA
- ➋ Look only at the columns *recent.move* and *num.vehicles*
- ➌ The output: all NAs. All the missing data comes from the same rows.

Because the missing data represents a fairly small fraction of the dataset, it's probably safe just to drop these customers from your analysis. But what about the variable *is.employed*? Here you're missing data from a third of the customers. What do you do then?

MISSING DATA IN CATEGORICAL VARIABLES

The most straightforward solution is just to create a new category for the variable, called *missing*.

```
> custdata$is.employed.fix <- ifelse(is.na(custdata$is.employed),  
    "missing",  
    ifelse(custdata$is.employed==T,  
        "employed",  
        "not employed"))  
  
> summary(as.factor(custdata$is.employed.fix))  
  
employed      missing not employed  
      599          328            73
```

- 1 If *is.employed* value is missing...
- 2 assign the value "missing"
- 3 otherwise if *is.employed*==TRUE assign the value "employed"...
- 4 else assign the value "not employed"
- 5 The transformation has turned the variable type from factor to string. You can change it back with the *as.factor()* function.

Practically speaking, this is exactly equivalent to what we had before; but remember that most analysis functions in R (and in a great many other statistical languages and packages) will drop rows with missing data by default. Changing each NA (which is R's code for missing values) to the token *missing* (which is people-code for missing values) will prevent that.

The preceding fix will get the job done, but as a data scientist, you ought to be interested in *why* so many customers are missing this information. It could just be bad record-keeping, but it could be semantic, as well. In this case, the format of the data (using the same row type for all customers) hints that the NA actually encodes that the customer is not in the active workforce: they are a homemaker, a student, retired, or otherwise not seeking paid employment. Assuming that you don't want

to differentiate between retirees, students, and so on, then naming the category appropriately will make it easier to interpret the model that you build down the line—both for you and for others.

```
custdata$is.employed.fix <- ifelse(is.na(custdata$is.employed),  
    "not in active workforce",  
    ifelse(custdata$is.employed==T,  
        "employed",  
        "not employed"))
```

If you did want to differentiate retirees and students and so on, you'd need additional data to make the correct assignments.

TIP**Why a new variable?**

You'll notice that we created a new variable called *is.employed.fix*, rather than simply replacing *is.employed*. This is a matter of taste. I prefer to have the original variable on hand, in case I second-guess my data cleaning and want to redo it. This is mostly a problem when the data cleaning involves a complicated transformation, like determining which customers are retirees and which ones are students. On the other hand, having two variables about employment in my data frame leaves me open to accidentally using the wrong one. Both choices have advantages and disadvantages.

Missing values in categorical variables are a relatively straightforward case to work through. What about numeric data?

MISSING VALUES IN NUMERIC DATA

Suppose your income variable is missing substantial data:

```
> summary(custdata$Income)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	NA's
0	25000	45000	66200	82000	615000	328

You believe that income is still an important predictor of health insurance coverage, so you still want to use the variable. What do you do?

THE CASE WHERE VALUES ARE MISSING AT RANDOM

You might believe that the data is missing because of a *faulty sensor*—in other words, the data collection failed at random. In this case, you can replace the missing values with the expected, or mean income.

```
> meanIncome <- mean(custdata$Income, na.rm=T) ①
> Income.fix <- ifelse(is.na(custdata$Income),
  meanIncome,
  custdata$Income)
> summary(Income.fix)

Min. 1st Qu. Median Mean 3rd Qu. Max.
0     35000   66200 66200 66200 615000
```

- ① Don't forget the argument "na.rm=T"! Otherwise, the mean() function will include the NAs by default, and meanIncome will be NA.

Assuming that the customers with missing income are distributed the same way as the others, then this estimate will be correct on average, and you'll be about as likely to have overestimated customer income as underestimated it. It's also an easy fix to implement.

This estimate can be improved when you remember that income is related to other variables in your data—for instance, you know from your data exploration in the previous chapter that there's a relationship between age and income. There might be a relationship between state of residence or marital status and income, as well. If you have this information, you can use it.

Note that the method of imputing a missing value of an input variable based on the other input variables can be applied to categorical data, as well. The text *R in Action* (Kabakoff, 2011) includes an extensive discussion of several methods available in R.

It's important to remember that replacing missing values by the mean, as well as many more sophisticated methods for imputing missing values, assume that the customers with missing income are in some sense random (the “faulty sensor” situation). It's possible that the customers with missing income data are *systematically* different from the others. For instance, it could be the case that the customers with missing income information truly have no income—because they're not in the active workforce. If this is so, then “filling in” their income information by using one of the preceding methods is the wrong thing to do. In this situation, there are two transformations you can try.

THE CASE WHERE VALUES ARE MISSING SYSTEMATICALLY

One thing you can do is to convert the numeric data into categorical data, and then use the methods that we discussed previously. In this case, you would divide the income into some income categories of interest, such as “below \$10,000,” or “from \$100,000 to \$250,000” using the `cut` function, and then treat the NAs as we did in when working with missing categorical values.

```
> breaks <-c(0, 10000, 50000, 100000, 250000, 1000000) ①

> Income.groups <- cut(custdata$Income,
+                         breaks=breaks, include.lowest=T) ②

> summary(Income.groups) ③

[0,1e+04] (1e+04,5e+04] (5e+04,1e+05] (1e+05,2.5e+05] (2.5e+05,1e+06]
       63           312          178           98            21
NA's
328

> Income.groups <- as.character(Income.groups) ④

> Income.groups <- ifelse(is.na(Income.groups),
+                           "no income", Income.groups) ⑤

> summary(as.factor(Income.groups))

(1e+04,5e+04] (1e+05,2.5e+05] (2.5e+05,1e+06] (5e+04,1e+05] [0,1e+04]
       312           98            21           178            63
no income
328
```

- ① Select some income ranges of interest. To use the `cut()` function, the upper and lower bounds should encompass the full income range of the data.
- ② Cut the data into income ranges. The `include.lowest=T` argument makes sure that zero income data is included in the lowest income range category. By default it would be excluded.
- ③ The `cut()` function produces factor variables. Note the NAs are preserved.
- ④ To preserve the category names before adding a new category, convert the variable to strings.
- ⑤ Add the “no income” category to replace the NAs.

This grouping approach can work well, especially if the relationship between income and insurance is non-monotonic (the likelihood of insurance doesn’t strictly increase or decrease with income). It does require that you select good cuts, and it’s a less concise representation of income than a numeric variable.

You could also replace all the NAs with zero income—but the data already has customers with zero income. Those zeros could be from the same mechanism as the NAs (customers not in the active workforce), or they could come from another mechanism—for example customers who have been unemployed the entire year. A trick that has worked well for us is to replace the NAs with zeros and add an additional variable (we call it a *masking variable*) to keep track of which data points have been altered.

```
missingIncome <- is.na(custdata$Income) ①  
Income.fix <- ifelse(is.na(custdata$Income), 0, custdata$Income) ②
```

- ① The missingIncome variable lets you differentiate the two kinds of zeros in the data: the ones that you are about to add, and the ones that were already there.
- ② Replace the NAs with zeros.

You give both variables, *missingIncome* and *Income.fix*, to the modeling algorithm, and it can determine how to best use the information to make predictions. Note that if the missing values really are missing at random, then the masking variable will basically pick up the variable's mean value (at least in regression models).

In summary, to properly handle missing data you need to know why the data is missing in the first place. If you don't know whether the missing values are random or systematic, we'd recommend assuming that the difference is systematic, rather than trying to impute values to the variables based on the faulty sensor assumption.

In addition to fixing missing data, there are other ways that you can transform the data to address issues that you found during the exploration phase. In the next section, we examine some common transformations.

4.1.2 Data transformations

The purpose of data transformation is to make data easier to model—and easier to understand. For example, the cost of living will vary from state to state, so what would be a high salary in one region could be barely enough to scrape by in another. If you want to use income as an input to your insurance model, it might be more meaningful to normalize a customer's income by the typical income in the area where they live. This is an example of a relatively simple (and common) transformation.

```

> summary(medianincome) ①

      State      Median.Income
      : 1      Min.    :37427
Alabama : 1      1st Qu.:47483
Alaska  : 1      Median  :52274
Arizona : 1      Mean    :52655
Arkansas: 1      3rd Qu.:57195
California: 1      Max.    :68187
(Other)   :46

> custdata <- merge(custdata, medianincome,
                     by.x="state.of.res", by.y="State") ②

> summary(custdata[,c("state.of.res", "income", "Median.Income")]) ③

      state.of.res      income      Median.Income
California :100      Min.    :-8700      Min.    :37427
New York   : 71      1st Qu.:14600     1st Qu.:44819
Pennsylvania: 70      Median  :35000     Median  :50977
Texas      : 56      Mean    :53505     Mean    :51161
Michigan   : 52      3rd Qu.:67000     3rd Qu.:55559
Ohio       : 51      Max.    :615000    Max.    :68187
(Other)    :600

> custdata$income.norm <- with(custdata, income/Median.Income) ④

> summary(custdata$income.norm)

  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
-0.1791  0.2729  0.6992  1.0820  1.3120 11.6600

```

- ① Suppose medianincome is a data frame of median income by state
- ② Merge median income information into the custdata data frame by matching the column custdata\$state.of.res to the column medianincome\$State
- ③ Median.Income is now part of custdata
- ④ Normalize income by Median.Income

The need for data transformation can also depend on which modeling method you plan to use. For linear and logistic regression, for example, you ideally want to make sure that the relationship between input variables and output variables is approximately linear, that the input variables are approximately normal in distribution, and that the output variable is constant variance (the variance of the output variable is independent of the input variables). You may need to transform some of your input variables to better meet these assumptions.

In this section we'll look at some useful data transformations and when to use them.

- Converting continuous variables to discrete
- Normalization
- Log transformations

CONVERTING CONTINUOUS VARIABLES TO DISCRETE

For some continuous variables, the exact value of that variable matters less than whether it falls into a certain range. For example, you may notice that customers with income less than \$20,000 have different health insurance patterns than customers with higher income. Or you may notice that customers under 25 and over 65 have high probabilities of insurance coverage, because they tend to be on their parents' coverage or on a retirement plan, respectively, whereas customers between those ages have a different pattern.

In these cases, you might want to convert the continuous *age* and *income* variables into ranges, or discrete variables. Discretizing continuous variables is useful when the relationship between input and output isn't linear, but you're using a modeling technique that assumes it is, like regression (see figure 4.2).

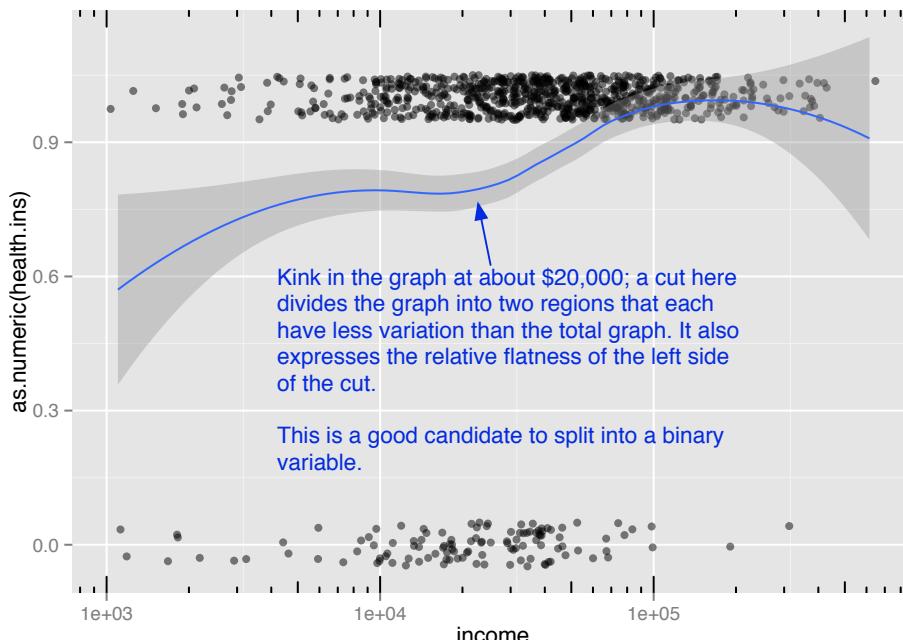


Figure 4.2 Health insurance coverage versus income (log10 scale)

Looking at figure 4.2, you see that you can replace the *income* variable with a boolean variable that indicates whether income is less than \$20,000.

```
> custdata$income.lt.20K <- custdata$income < 20000
```

```

> summary(custdata$income.lt.20K)
   Mode    FALSE     TRUE     NA's
logical      678      322       0 </programlisting>
                                         </informalexample>
                                         <para>If you want more than a simple threshold (as in
                                         the <code>cut</code> function, as we saw in
                                         are missing systematically."</para>
                                         <informalexample annotations="below">
                                         <programlisting>> brks <= c(0, 25, 65, In
①
                                         > custdata$age.range<= cut(custdata$age, breaks=brks, include.lowest=T) ②
                                         > summary(custdata$age.range) ③
                                         [0,25] (25,65] (65,Inf]
                                         56        732      212

```

- ① Select the age ranges of interest. The upper and lower bounds should encompass the full range of the data.
- ② Cut the data into age ranges. The `include.lowest=T` argument makes sure that zero age data is included in the lowest age range category. By default it would be excluded.
- ③ The output of `cut()` is a factor variable.

Even when you do decide not to discretize a numeric variable, you may still need to transform it to better bring out the relationship between it and other variables. We saw this in the example that introduced this section, where we normalized income by the regional median income. In the next section we'll talk about normalization and rescaling.

NORMALIZATION AND RESCALING

Normalization is useful when absolute quantities are less meaningful than relative ones. We've already seen an example of normalizing income relative to another meaningful quantity (median income). In that case, the meaningful quantity was external (came from the analyst's domain knowledge); but it can also be internal (derived from the data itself).

For example, you might be less interested in a customer's absolute age than you are in how old or young they are relative to a "typical" customer. Let's take the mean age of your customers to be the typical age. You can normalize by that.

```

> summary(custdata$age)
   Min. 1st Qu. Median Mean 3rd Qu. Max.
0.0    38.0   50.0  51.7   64.0  146.7
> meanage <- mean(custdata$age)
> custdata$age.normalized <- custdata$age/meanage

```

```
> summary(custdata$age.normalized)
  Min. 1st Qu. Median     Mean 3rd Qu.    Max.
0.0000  0.7350  0.9671  1.0000  1.2380  2.8370
```

A value for *age.normalized* that is much less than one signifies an unusually young customer; much greater than one signifies an unusually old customer. But what constitutes “much less” or “much greater” than one? That depends on how wide an age spread your customers tend to have. See figure 4.3 for an example.

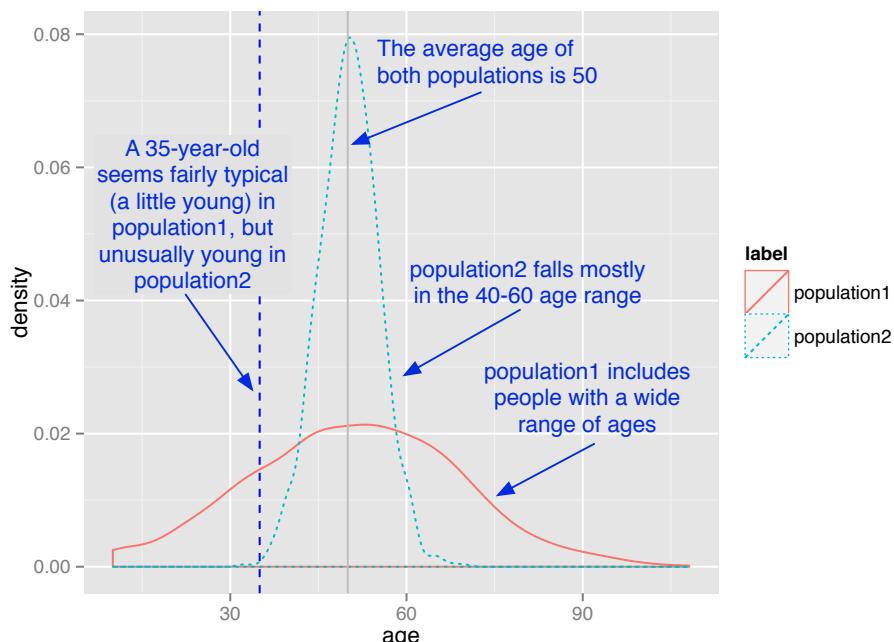


Figure 4.3 Is a 35-year-old young?

The typical age spread of your customers is summarized in the standard deviation. You can *rescale* your data by using the standard deviation as a unit of distance. A customer who is within one standard deviation of the mean is not much older or younger than typical. A customer who is more than one or two standard deviations from the mean can be considered much older, or much younger.

```
> summary(custdata$age)
  Min. 1st Qu. Median     Mean 3rd Qu.    Max.
0.0    38.0   50.0    51.7    64.0   146.7
> meanage <- mean(custdata$age)      ①
> stdage <- sd(custdata$age)        ②
> meanage
[1] 51.69981
> stdage
[1] 18.86343
```

```

> custdata$age.normalized <- (custdata$age-meanage)/stdage
> summary(custdata$age.normalized)
   Min. 1st Qu. Median      Mean 3rd Qu.      Max.
-2.74100 -0.72630 -0.09011  0.00000  0.65210  5.03500

```

3

- 1 Take the mean.
- 2 Take the standard deviation.
- 3 Use the mean value as the origin (or reference point) and rescale the distance from the mean by the standard deviation.

Now, values less than -1 signify customers younger than typical; values greater than 1 signify customers older than typical.

SIDE BAR A technicality

The common interpretation of standard deviation as a unit of distance implicitly assumes that the data is distributed normally. For a normal distribution, roughly two-thirds of the data (about 68%) is within plus/minus one standard deviation from the mean. About 95% of the data is within plus/minus two standard deviations from the mean. In figure 4.3, a 35-year-old is (just barely) within one standard deviation from the mean in population1, but more than two standard deviations from the mean in population2.

You can still use this transformation if the data isn't normally distributed, but the standard deviation is most meaningful as a unit of distance if the data is unimodal and roughly symmetric around the mean.

As we mentioned in the sidebar, normalizing by mean and standard deviation is most meaningful when the data distribution is roughly symmetric. Next, we'll look at a transformation that can make some distributions more symmetric.

LOG TRANSFORMATIONS FOR SKEWED AND WIDE DISTRIBUTIONS

Monetary amounts—incomes, customer value, account, or purchase sizes—are some of the most commonly encountered sources of skewed distributions in data science applications. In fact, as we discuss in appendix XRF:appendix_B:xAppendixStat, monetary amounts are often lognormally distributed—the log of the data is normally distributed. This leads us to the idea that taking the log of the data can restore symmetry to it. We demonstrate this in figure 4.4.

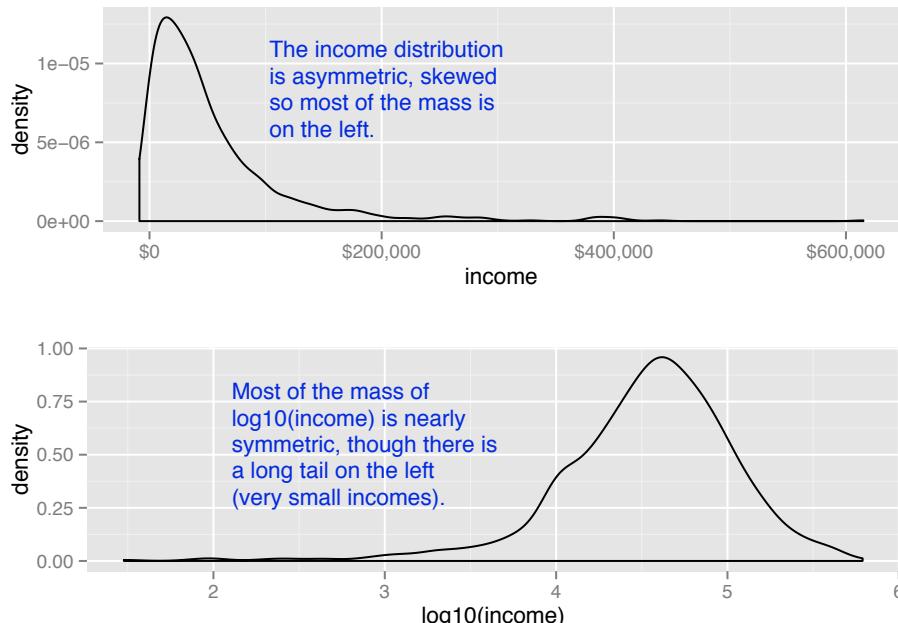


Figure 4.4 A nearly lognormal distribution, and its log

For the purposes of modeling, *which* logarithm you use—natural logarithm, log base 10, or log base 2—is generally not critical. In regression, for example, the choice of logarithm affects the magnitude of the coefficient that corresponds to the logged variable, but it doesn’t affect the value of the outcome. I like to use log base 10 for monetary amounts, because orders of ten seem natural for money: \$100, \$1000, \$10,000, and so on. The transformed data is easy to read.

NOTE

An aside on graphing

Notice that the bottom panel of figure 4.4 has the same shape as figure XRF:figure:3_3.2.1:fig_logincomedensity. The difference between using the `ggplot` layer `scale_x_log10` on a density plot of `income` and plotting a density plot of `log10(income)` is primarily axis labeling. Using `scale_x_log10` will label the x-axis in dollars amounts, rather than in logs.

It’s also generally a good idea to log transform data with values that range over several orders of magnitude. First, because modeling techniques often have a difficult time with very wide data ranges, and second, because such data often comes from multiplicative processes, so log units are in some sense more natural.

For example, when you’re studying weight loss, the natural unit is often pounds or kilograms. If I weigh 150 pounds, and my friend weighs 200, we’re both equally

active, and we both go on the exact same restricted-calorie diet, then we'll probably both lose about the same number of pounds—in other words, how much weight we lose doesn't (to first order) depend on how much we weighed in the first place, only on calorie intake. This is an *additive* process.

On the other hand, if management gives everyone in the department a raise, it probably isn't giving everyone \$5,000 extra. Instead, everyone gets a 2% raise: how much extra money ends up in my paycheck depends on my initial salary. This is a *multiplicative* process, and the natural unit of measurement is percentage, not absolute dollars. Other examples of multiplicative processes: a change to an online retail site increases conversion (purchases) for each item by 2% (not by exactly two purchases); a change to a restaurant menu increases patronage every night by 5% (not by exactly five customers every night). When the process is multiplicative, log-transforming the process data can make modeling easier.

Of course, taking the logarithm only works if the data is non-negative. There are other transforms, such as *arcsinh*, that you can use to decrease data range if you have zero or negative values. I don't like to use *arcsinh*, because I don't find the values of the transformed data to be meaningful. In applications where the skewed data is monetary (like account balances or customer value), I instead use what I call a *signed logarithm*. A signed logarithm takes the logarithm of the absolute value of the variable and multiplies by the appropriate sign. Values strictly between -1 and 1 are mapped to zero. The difference between log and signed log are shown in figure 4.5.

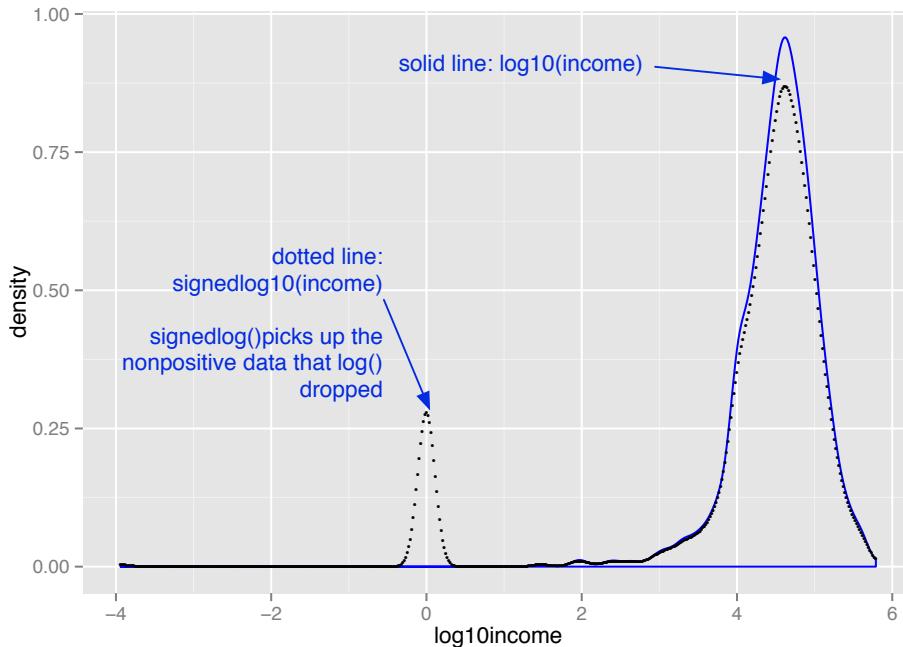


Figure 4.5 Signed log lets you visualize non-positive data on a logarithmic scale

Here's how to calculate signed log base 10, in R:

```
signedlog10 <- function(x) {
  ifelse(abs(x) <= 1, 0, sign(x)*log10(abs(x)))
}
```

Clearly this isn't useful if values below unit magnitude are important. But with many monetary variables (in U.S. currency), values less than a dollar aren't much different from zero (or one), for all practical purposes. So, for example, mapping account balances that are less than a dollar to \$1 (the equivalent every account always having a minimum balance of one dollar) is probably okay.¹

Footnote 1 There are methods other than capping to deal with signed logarithms, such as the arcsinh function (see <http://www.win-vector.com/blog/2012/03/modeling-trick-the-signed-pseudo-logarithm/>, but they also distort data near zero and make almost any data appear to be bi-modal.

Once you've got the data suitably cleaned and transformed, you're almost ready to start the modeling stage. Before we get there, we have one more step.

4.2 Sampling for modeling and validation

Sampling is the process of selecting a subset of a population to represent the whole, during analysis and modeling. In the current era of big datasets, some people argue that computational power and modern algorithms let us analyze the entire large dataset without the need to sample.

We can certainly analyze larger data sets than we could before, but sampling is a necessary task for other reasons. When you’re in the middle of developing or refining a modeling procedure, it’s easier to test and debug the code on small sub-samples before training the model on the entire dataset. Visualization can be easier with a sub-sample of the data; `ggplot` runs faster on smaller datasets, and too much data will often obscure the patterns in a graph, as we mentioned in the previous chapter. And often, it’s not feasible to use your entire customer base to train a model.

It’s important that the dataset that you do use is an accurate representation of your population as a whole. For example, your customers might come from all over the United States. When you collect your `custdata` dataset, it might be tempting to use all the customers from one state, say Connecticut, to train the model. But if you plan to use the model to make predictions about customers all over the country, it’s a good idea to pick customers randomly from all the states, because what predicts health insurance for Texas customers might be different from what predicts health insurance in Connecticut. This might not always be possible (perhaps only your Connecticut and Massachusetts branches currently collect the customer health insurance information), but the shortcomings of using a non-representative dataset should be kept in mind.

The other reason to sample your data is to create test and training splits.

4.2.1 Test and training splits

When you are building a model to make predictions, like our model to predict health insurance coverage, you need data to build the model. You also need data to test whether the model makes correct predictions on new data. The first set is called the *training set*, and the second set is called the *test (or hold-out) set*.

The training set is the data that you feed to the model-building algorithm—regression, decision trees, and so forth—so that the algorithm can set the correct parameters to best predict the outcome variable. The test set is the data that you feed into the resulting model, to verify that the model’s predictions are accurate. We’ll go into detail about the kinds of modeling issues that you can

detect by using hold-out data in chapter XRF:chapter_5:chCritiquingModels. For now, we'll just get our data ready for doing hold-out experiments at a later stage.

4.2.2 Creating a sample group column

A convenient way to manage random sampling is to add a sample group column to the data frame. The sample group column contains a number generated uniformly from zero to one, using the `runif` function. You can draw a random sample of arbitrary size from the data frame by using the appropriate threshold on the sample group column.

For example, once you've labeled all the rows of your data frame with your sample group column (let's call it `gp`), then the set of all rows such that $gp < 0.4$ will be about four-tenths, or 40%, of the data. The set of all rows where `gp` is between 0.55 and 0.70 is about 15% of the data ($0.7 - 0.55 = 0.15$). So you can repeatably generate a random sample of the data of any size by using `gp`.

```
> custdata$gp <- runif(dim(custdata)[1])      ①
> testSet <- subset(custdata, custdata$gp <= 0.1) ②
> trainingSet <- subset(custdata, custdata$gp > 0.1) ③
> dim(testSet)[1]
[1] 93
> dim(trainingSet)[1]
[1] 907
```

- ① `dim(custdata)` returns the number of rows and columns of the data frame as a vector, so `dim(custdata)[1]` returns the number of rows.
- ② Here we generate a test set of about 10% of the data (93 customers—a little over 9%, actually) and train on the remaining 90%.
- ③ Here we generate a training using the remaining data.

R also has a function called `sample` that draws a random sample (a uniform random sample, by default) from a data frame. Why not just use `sample` to draw training and test sets? You could, but using a sample group column guarantees that you'll draw the same sample group every time. This reproducible sampling is convenient when you're debugging code. In many cases, code will crash because of a corner case that you forgot to guard against. This corner case might show up in your random sample. If you're using a different random input sample every time you run the code, you won't know if you will tickle the bug again. This makes it hard to track down and fix errors.

You also want repeatable input samples for what software engineers call

regression testing (not to be confused with statistical regression). In other words, when you make changes to a model or to your data treatment, you want to make sure you don't break what was already working. If model version 1 was giving "the right answer" for a certain input set, you want to make sure that model version 2 does so also.

TIP

Reproducible sampling is not just a trick for R

If your data is in a database or other external store, and you only want to pull a subset of the data into R for analysis, you can draw a reproducible random sample by generating a sample group column in an appropriate table in the database, using the SQL command `RAND`.

4.2.3 Record grouping

One caveat is that the preceding trick works if every object of interest (every customer, in this case) corresponds to a unique row. But what if you're interested less in which customers don't have health insurance, and more about which *households* have uninsured members? If you're modeling a question at the household level rather than the customer level, then *every member of a household should be in the same group (test or training)*. In other words, the random sampling also has to be at the household level.

Suppose your customers are marked both by a household ID and a customer ID (so the unique ID for a customer is the combination (`household_id, cust_id`)). This is shown in figure 4.6. We want to split the households into a training set and a test set.

	household_id	cust_id	income
household 1	hh1	cust1	30200
household 2	hh2	cust1	24800
	hh2	cust2	134800
household 3	hh3	cust1	299000
	hh3	cust2	65000
	hh3	cust3	95000
household 4	hh4	cust1	38800
	hh4	cust2	0
household 5	hh5	cust1	100300
	hh5	cust2	27000

Figure 4.6 Example of dataset with customers and households

Here's one way to generate an appropriate sample group column:

```
hh <- unique(hhdata$household_id) ①
households <- data.frame(household_id = hh, gp = runif(length(hh))) ②
hhdata <- merge(hhdata, households, by="household_id") ③
```

- ① Get all unique household IDs from your data frame.
- ② Create a temporary data frame of household IDs and a uniformly random number from zero to one.
- ③ Merge new random sample group column back into original data frame.

The resulting sample group column is shown in figure 4.7. Now we can generate the test and training sets as before. This time, however, the threshold 0.1 doesn't represent 10% of the data rows, but 10% of the households (which may be more or less than 10% of the data, depending on the sizes of the households).

	household_id	cust_id	income	gp
household 1	hh1	cust1	30200	0.8625189
household 2	hh2	cust1	24800	0.8880607
	hh2	cust2	134800	0.8880607
	hh3	cust1	299000	0.9130094
household 3	hh3	cust2	65000	0.9130094
	hh3	cust3	95000	0.9130094
household 4	hh4	cust1	38800	0.5244124
	hh4	cust2	0	0.5244124
household 5	hh5	cust1	100300	0.5388283
	hh5	cust2	27000	0.5388283

Figure 4.7 Example of dataset with customers and households

4.2.4 Data provenance

You'll also want to add a column (or columns) to record data provenance: when your dataset was collected, perhaps what version of your data cleaning procedure was used on the data before modeling. This is akin to version control for data. It's handy information to have, to make sure that you're comparing apples to apples when you're in the process of improving your model, or comparing different models or different versions of a model. We'll talk more about data provenance and model versioning in chapter XRF:chapter_10:chDocumentation.

4.3 Summary

At some point, you'll have data that is as good quality as you can make it. You've fixed problems with missing data, and performed any needed transformations. You're ready to go on the modeling stage.

Remember, though, that data science is an iterative process. You may discover during the modeling process that you have to do additional data cleaning or transformation. You may have to go back even further and collect different types of data. That is why we recommend adding columns for sample groups and data provenance to your data sets (and later, to the models and model output), so you can keep track of the data management steps as the data and the models evolve.

In the next part of the book, we'll talk about the process of building and evaluating models to meet your stated objectives.

4.4 External links section

Part 2

In Part 2 we work with powerful modeling methods from statistics and machine learning.

5

Chapter 5: Choosing and Evaluating Models

This chapter will cover:

- How to choose modeling methods.
- How to evaluate model quality.
- What overfit models are and how to avoid them.

As a data scientist, your ultimate goal is to solve a concrete business problem: increase look-to-buy ratio, identify fraudulent transactions, predict and manage the losses of a loan portfolio. Many different statistical modeling methods can be used to solve any given problem. And each statistical method will have its advantages and disadvantages for a given business goal and business constraints. This chapter will present an outline of the most common machine learning and statistical methods used in data science.

To make progress you must be able to measure model quality during training and also ensure that your model will work as well in the production environment as it did on your training data. In general we will call these two tasks model *evaluation* and model *validation*. To prepare for these statistical tests we always split our data into training data and test data as illustrated in Figure 5.1.

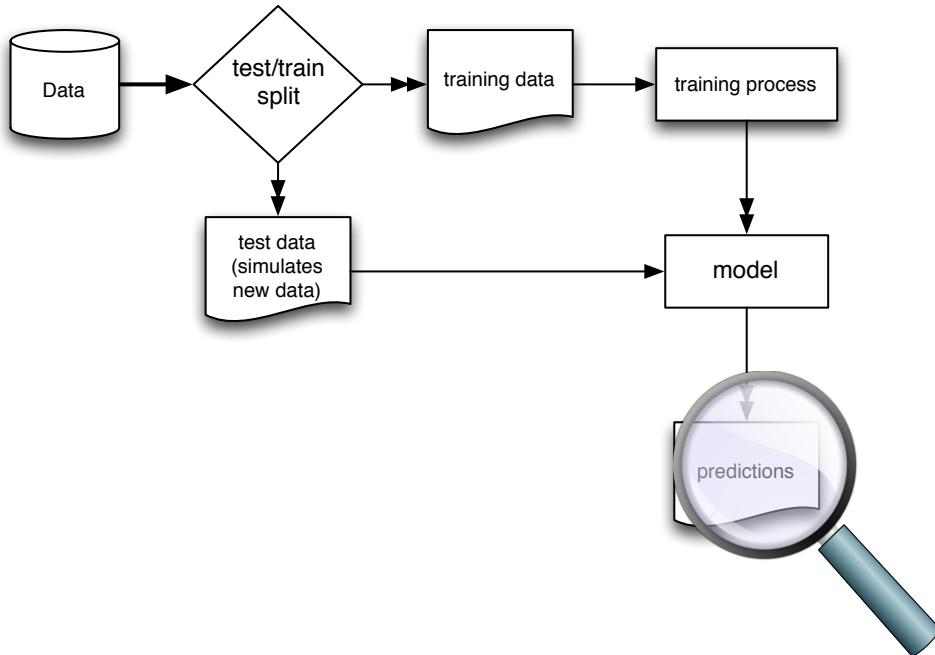


Figure 5.1 Schematic model construction and evaluation

Model Evaluation

We define model evaluation as quantifying the performance of a model. To do this we must find a measure of model performance that is appropriate to both the original business goal and the chosen modeling technique. For example: if we predicting who would default on loans we have a classification task and measures like precision and recall are appropriate. If we were instead predicting revenue lost to defaulting loans we have a scoring task and measures like root mean square error are appropriate. The point is: there are a number of measures the data scientist should be familiar with.

Model Validation

We define model validation as the generation of an assurance that the model will work in production as well as it worked during training. It is a disaster to build a model that works great on the original training data and then performs poorly when used in production. The biggest cause of model validation failures is not having enough training data to represent the variety of what may later be encountered in production. For example: training a loan default model on only people who repaid their loans might score well on a simple evaluation ("predicts no defaults and is 100% accurate!") but would obviously not be a good model to put into production. Validation techniques attempt to quantify this type of risk before you put a model into production.

In this chapter we are going to discuss the principles of method selection, evaluation, and validation before getting into the details of model fitting.¹

Footnote 1 For some help see: <http://www.win-vector.com/blog/category/statistics-to-english-translation/>.

5.1 Mapping Problems to Machine Learning Tasks

Our task is to map a business problem to a good machine learning method. To make things concrete, let's suppose that you are a data scientist at an online retail company. There are a number of business problems that your team might be called to address:

- Predicting what customers might buy, based on past transactions.
- Identifying fraudulent transactions.
- Determining price elasticity (the rate at which a price increase will decrease sales, and vice versa) of various products or product classes.
- Determining the best way to present product listings when a customer searches for an item.
- Customer segmentation: grouping customers with similar purchasing behavior
- Adword valuation: How much should the company be willing to spend to buy certain adwords on search engines?
- Evaluation of marketing campaigns.
- Organizing new products into a product catalog.

Your intended uses of the model have a big influence on what methods you should use. If you want to know how small variations in input variables affect outcome then you likely want to use a regression method. If you want to know what single variable drives most of a categorization then decision trees are likely a good choice. Also, each business problem suggests a statistical approach to try. If you are trying to predict scores some sort of regression is likely a good choice, if you are trying to predict categories then something like random forests is likely a good choice.

5.1.1 Solving Classification Problems

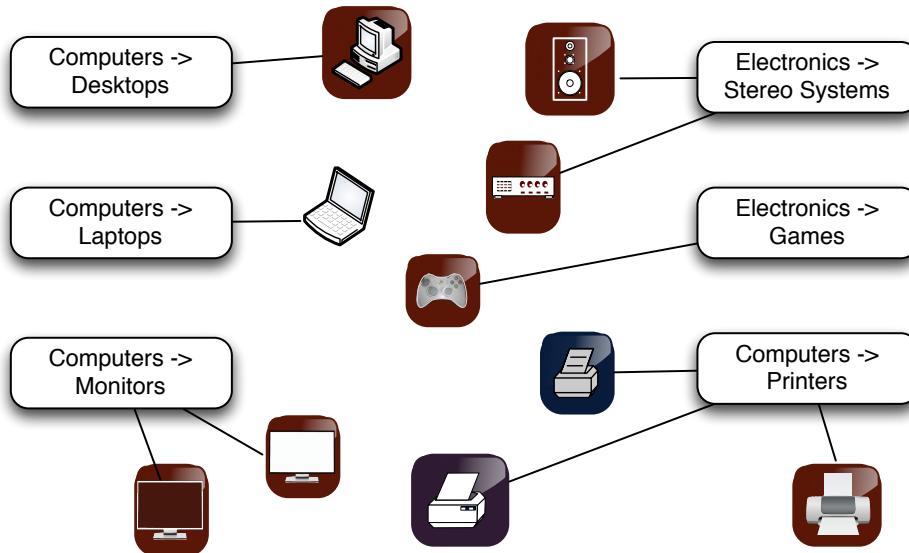


Figure 5.2 Assigning Products to Product Categories

Suppose your task is to automate the assignment of new products to your company's product categories, as shown in Figure 5.2. This can be more complicated than it sounds. Products that come from different sources may have their own product classification that does not coincide with the one that you use on your retail site, or they may not be pre-classified at all. Many large online retailers use teams of human taggers to hand-categorize their products. This is not only labor-intensive, but inconsistent and error-prone. Automation is an attractive option; it is labor-saving, and can improve the quality of the retail site.

Product categorization based on product attributes and/or text descriptions of the product is an example of *classification*: deciding how to assign (known) labels to an object. Classification itself is an example of what is called *supervised learning*: in order to learn how to classify objects, you need a dataset of objects that have already been classified (called the *training set*). Building training data is the major expense for most classification tasks, especially text-related ones.

SIDE BAR**Multi category versus two category classification**

Product classification is an example of *multi category* or *multinomial* classification. Most classification problems and most classification algorithms are specialized for two category classification or binomial classification. There are tricks to use binary classifiers to solve multi category problems (for example: building one classifier for each category, called a "one versus rest" classifier). However in most cases it is worth the effort to find a suitable multiple category implementation as they tend to work better than multiple binary classifiers (for example: using the package `mlogit` instead of the base method `glm()` for logistic regression).

COMMON CLASSIFICATION METHODS

We list some of the most common classification methods here. We'll cover them (and others) in more detail in later chapters.

Naive Bayes

Naive Bayes classifiers are especially useful for problems with many input variables, categorical input variables with a very large number of possible values, and text classification. Naive Bayes would be a good first attempt at solving the product categorization problem.

Decision Trees

Decision Trees are useful when input variables interact with the output in "if-then" kinds of ways (IF age > 65 THEN has.health.insurance=T). They are also suitable when inputs have an AND relationship to each other (IF age < 25 AND student=T THEN...) or when input variables are redundant or correlated. The decision rules that come from a decision tree are in principle easier for non-technical users to understand than the decision processes that come from other classifiers. We saw an example use of decision trees for predicting health insurance coverage in Chapter XRF:chapter_1:chProjectLifeCycle. We will also discuss an important extension of decision trees: Random Forests.

Logistic Regression

Logistic Regression is appropriate when you want to estimate class probabilities (the probability that an object is in a given class) in addition to class assignments.² An example use of a logistic regression-based classifier is estimating the probability of fraud in credit card purchases. Logistic regression is also a good

choice when you want an idea of the relative impact of different input variables on the output: for example, you might find out that a \$100 increase in transaction size increases the odds that the transaction is fraud by 2%, all else being equal.

Footnote 2 Strictly speaking, logistic regression is scoring (covered in the next section). To turn a scoring algorithm into a classifier requires a threshold. For scores higher than the threshold, assign one label, for lower scores, assign an alternative label.

Support Vector Machines

Support Vector Machines are useful when there are very many input variables or when input variables interact with the outcome or with each other in complicated (non-linear) ways. SVMs make fewer assumptions about variable distribution than many other methods, which makes them especially useful when the training data is not completely representative of the way the data is distributed in production.

5.1.2 Solving Scoring Problems

For a scoring example: suppose that your task is to help evaluate how different marketing campaigns can increase valuable traffic to the website. The goal is not only to bring more people to the site, but to bring more people who buy. You are looking at a number of different factors: the communication channel (ads on websites, YouTube videos, print media, email, and so on); the traffic source (Facebook, Google, radio stations...); the demographic targeted, the time of year, and so on.

Predicting the increase in sales from a particular marketing campaign is an example of *regression*, or *scoring*. Fraud detection can be considered scoring, too, if you are trying to estimate the probability that a given transaction is a fraudulent one (rather than just returning a yes/no answer). This is shown in Figure 5.3. Scoring is also an instance of supervised learning.

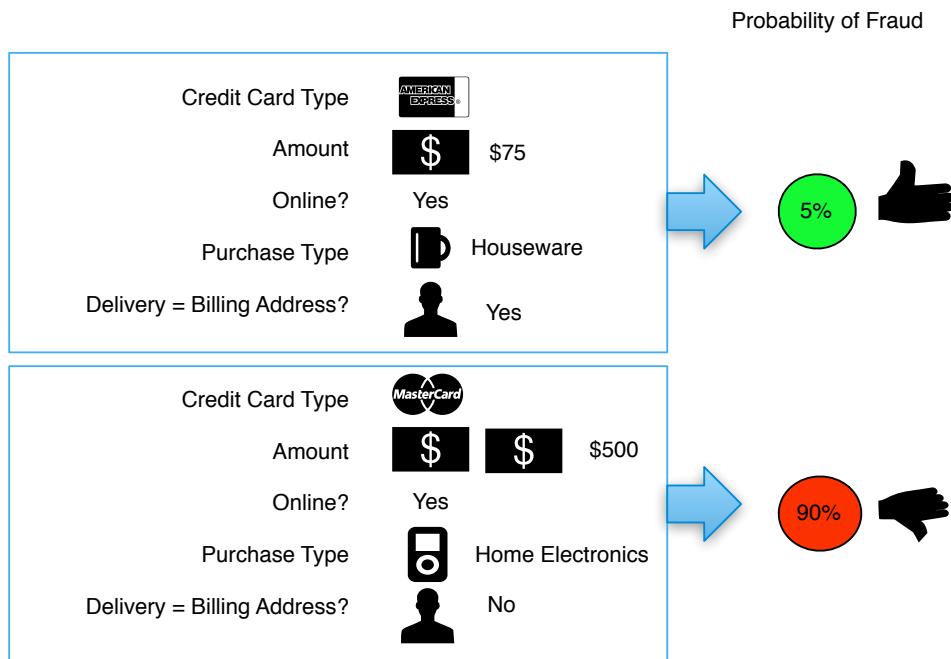


Figure 5.3 Notional Example of Determining the Probability that a Transaction is Fraudulent

COMMON SCORING METHODS

We'll cover the following two general scoring methods in more detail, in later chapters.

Linear Regression

Linear Regression builds a model such that the predicted numerical output is a linear additive function of the inputs. This can be a very effective approximation, even when the underlying situation is in fact non-linear. The resulting model also gives an indication of the relative impact of each input variable on the output. Linear Regression is often a good first model to try when trying to predict a numeric value.

Logistic Regression

Logistic Regression always predicts a value between zero and one, making it suitable for predicting probabilities (when the observed outcome is a categorical value) and rates (when the observed outcome is a rate or ratio). As we mentioned above, logistic regression is an appropriate approach to the fraud detection problem, if what you want to estimate is the probability that a given transaction is fraudulent or legitimate.

5.1.3 Working Without Known Targets

The above methods require that you have a training dataset of situations with known outcomes. In some situations, there is not (yet) a specific outcome that you want to predict. Instead, you may be looking for patterns and relationships in the data that will help you understand your customers or your business better.

These situations correspond to a class of approaches called *unsupervised learning*: rather than trying to predict outputs based on inputs, unsupervised learning tries to discover similarities and relationships in the data. We cover a few algorithms that can be suitable for problems like these:

- K-Means Clustering.
- Apriori Algorithm for finding Association Rules.
- Nearest Neighbor.

COMMON UNSUPERVISED APPROACHES

Each of these approaches looks for a different type of pattern, and so is suitable for a particular type of problem.

Clustering

Suppose you want to segment your customers into general categories of people with similar buying patterns. You might not know in advance what these groups should be. This problem is an example of *clustering*. The goal of clustering is to sort the data into groups such that members of a cluster are more similar to each other than they are to members of other clusters.

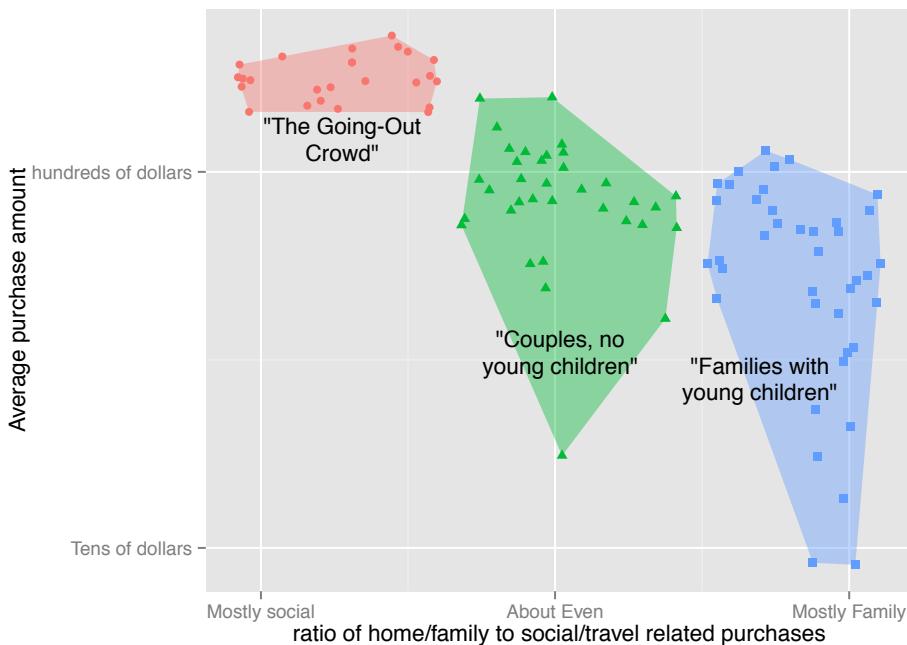


Figure 5.4 Notional Example of Clustering your Customers by Purchase Pattern and Purchase Amount

Suppose that you find (as in Figure 5.4) that your customers cluster into those with young children, who make more family-oriented purchases, and those with no children or with adult children, who make more leisure and social-activity related purchases. Once you have assigned a customer into one of those clusters, you can make general statements about their behavior. For example, a customer with young children is likely to respond more favorably to a promotion on attractive but durable glassware than to a promotion on fine crystal wine glasses.

Discovered clusters might also be used as the categories for a classification algorithm.

Association Rules

You might be interested in finding out which products tend to be purchased together. For example, you might find that bathing suits and sunglasses are frequently purchased at the same time, or that people who purchase certain cult movies, like *Repo Man*, will often buy the movie soundtrack at the same time. You can use this knowledge to make product recommendations: whenever you observe that someone has put a bathing suit into their shopping cart, you can recommend suntan lotion, as well. This is shown in Figure 5.5.

	bikini, sunglasses, sunblock, flip-flops
	swim trunks, sunblock
	tankini, sunblock, sandals
	bikini, sunglasses, sunblock
	one-piece, beach towel

Figure 5.5 Notional Example of Finding Purchase Patterns in your Data

Association rules are a popular technique for such finding co-occurrences in data. We will cover the Apriori algorithm for discovering association rules later.

Nearest Neighbor

Another way to make product recommendations is to find similarities in people (Figure 5.6). For example, to make a movie recommendation to customer *JaneB*, you might look for the three customers with the most similar movie rental history to hers. Any movies that those three people rented but *JaneB* has not are potentially useful recommendations for her.

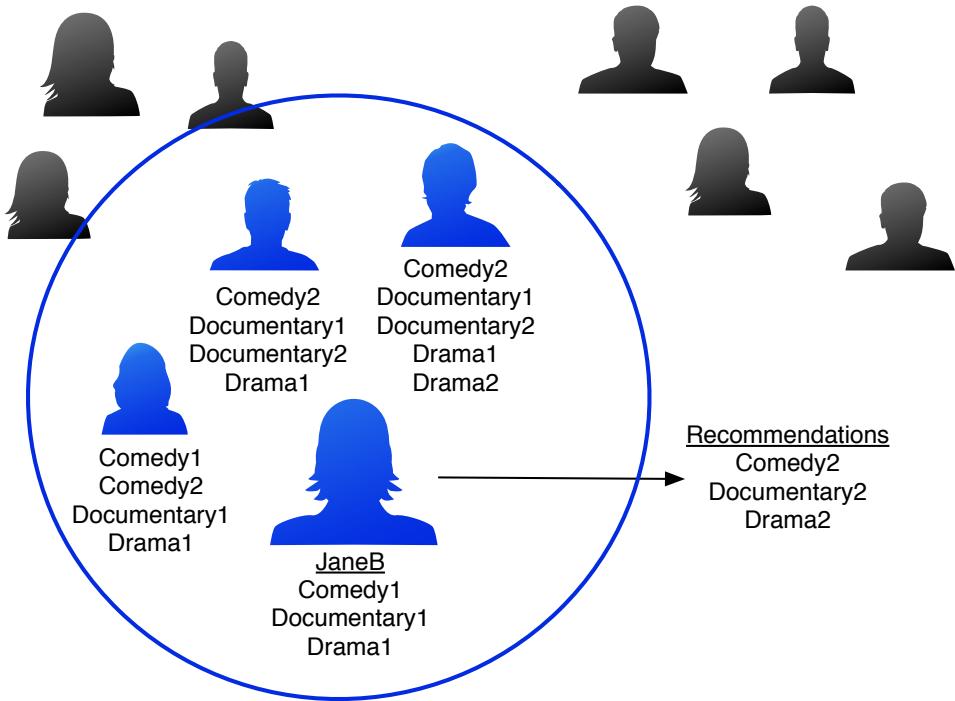


Figure 5.6 Look to the Customers With Similar Movie-Watching patterns as *JaneB* for her Movie Recommendations

This can be solved with *Nearest Neighbor* (or K-nearest neighbor methods, where in this case $K = 3$). Nearest neighbor algorithms predict something about a data point p (like a customer's future purchases) based on the data point or points that are most similar to p . We will cover the nearest neighbor approach later.

5.1.4 An Example Problem to Method Mapping Table

Table 5.1 maps some typical business problems to their corresponding machine learning task, and to some typical algorithms to tackle each task.

Table 5.1 From Problem to Approach

Example tasks	Machine Learning Task	Typical Algorithms
<ul style="list-style-type: none"> Identifying spam email Sorting products in a product catalog Identifying loans that are about to default Assigning customers to customer segments 	Classification: assigning known labels to objects	<ul style="list-style-type: none"> Decision Trees Naive Bayes Logistic Regression (with a threshold) Support Vector Machines
<ul style="list-style-type: none"> Predicting the value of adwords Estimating the probability that a loan will default. Predicting how much a marketing campaign will increase traffic or sales 	Regression: predicting or forecasting numerical values	<ul style="list-style-type: none"> Linear Regression Logistic Regression
<ul style="list-style-type: none"> Finding products that are purchased together Identifying web pages that are often visited in the same session Identifying successful (much-clicked) combinations of web page and adword 	Association Rules: finding objects that tend to appear in the data together	<ul style="list-style-type: none"> Apriori
<ul style="list-style-type: none"> Identifying groups of customers with the same buying patterns Identifying groups of products that are popular in the same regions or the same customer segments Identifying news items that are all discussing similar events. 	Clustering: finding groups of objects that are more similar to each other than to objects from other groups	<ul style="list-style-type: none"> K-Means

<ul style="list-style-type: none"> Making product recommendations for a customer based on the purchases of other customers like him Predicting the final price of an auction item based on the final prices of similar products that have been auctioned in the past 	<p>Nearest Neighbor: predicting a property of a datum based on the datum or datums that are most similar to it</p>	<ul style="list-style-type: none"> Nearest Neighbor
--	--	--

Notice that some problems show up multiple times in the table. Our mapping isn't hard-and-fast; any problem can be approached through a variety of mindsets, with a variety of algorithms. We are merely listing some common mappings and approaches to typical business problems. Generally, these should be among the first approaches to consider for a given problem; if they don't perform well, then you will want to research other approaches, or get creative with data representation and with variations of common algorithms.

SIDE BAR **Prediction versus Forecasting**

In everyday language, we tend to use the terms "prediction" and "forecasting" interchangeably. Technically to predict is to pick an outcome such as "it will rain tomorrow" and to forecast is to assign a probability "there is a 80% chance it will rain tomorrow." For unbalanced class applications (such as predicting credit default) the difference is very important. Consider the case of modeling loan defaults, and assume the overall default rate is 5%. Identifying a group that has a 30% default rate is an inaccurate prediction (you don't know who in the group will default, and most people in the group will not default), but potentially a very useful forecast (this group defaults at six times the overall rate).

5.2 Evaluating Models

When building a model the first thing to check is if the model even works on the data it was trained from. In this section we do this by introducing quantitative measures of model performance. From an evaluation point of view we group model types into:

- Classification
- Scoring
- Probability Estimation

- Ranking
- Clustering

For most model evaluations we just want to compute one or two summary scores that tell us if the model is effective. To decide if a given score is high or low we have to appeal to two ideal models: a null model (which tells us what low performance looks like) and a Bayes rate model (which tells us what high performance looks like).

A Null Model

A null model is the best model of a very simple form you are trying to out-perform. The two most typical null model choices are a model that is a single constant (returns the same answer for all situations) or a model that is independent (doesn't record any important relation or interaction between inputs and outputs). We use null models to lower-bound desired performance, so we usually compare ourselves to a best null model. For example: in a categorical problem the null model would always return the most popular category (as this is the easy guess that is least often wrong), for a score model the null model is often the average of all the outcomes (as this has the least square deviation from all of the outcomes) and so on. The idea is: if you are not out-performing the null model, you are not delivering value. Note that it can in fact be hard to do as good as the best null model, because even though the null model is simple it is privileged to know the overall distribution of the items it will be quizzed on. We will always assume the null model we are comparing to is the best of all possible null models.

Bayes Rate Model

A Bayes rate model (also sometimes called a saturated model) is a best possible model given the data at hand. The Bayes rate model is the perfect model and it only makes mistakes when there are multiple examples with the exact same set of known facts (same \mathbf{x} s) but different outcomes (different \mathbf{y} s). It isn't always practical to construct the Bayes rate model, but we invoke it as an upper bound on a model evaluation score.

If we feel our model is performing significantly above the null model rate and is approaching the Bayes rate, then we can stop tuning. When we have a lot of data and very few modeling features we can estimate the Bayes error rate. Another way

to estimate the Bayes rate is to hand-classify a small subset of your data among multiple people, the found inconsistencies can be thought of as an estimate of the Bayes rate.³

Footnote 3 There are a few "machine learning" magic methods that can introduce new synthetic features and in fact alter the Bayes rate. Typically this is done by adding "higher order" terms, interaction terms or kernelizing.

We also suggest comparing any complicated model against the best single variable you have available (see section XRF:sect1_6.2:sectBuildSingleVariableModel for how to convert single variables into single variable models).

We will present the standard measures of model quality, which are very useful in model construction. In all cases we suggest in addition to the standard model quality assessments you try to design your own custom "business oriented loss function" with your project sponsor or client. Usually this is as simple as assigning a notional dollar value to each outcome and then seeing how your model performs under that criterion. Let's start with how to evaluate classification models and then continue from there.

5.2.1 Evaluating Classification Models

A classification model places examples into two or more categories. The most common measure of classifier quality is called "accuracy." For measuring classifier performance we will first introduce the incredibly useful tool called the "confusion matrix" and show how it can be used to calculate many important evaluation scores. The first score we discuss will be accuracy, and then move on to better and more detailed measures such as precision and recall.

For our example let's use the example of classifying email into spam (email we in no way wanted) and non-spam (email we wanted). A ready to go example (with a good description) is the "Spambase" data set.⁴ Each row of this data set is a set of features measured for a specific email and an additional column telling if the mail was spam (unwanted) or non-spam (wanted). We will quickly build a spam classification model so we have results to evaluate. To do this we download the file Spambase/spamD.tsv from our book's GitHub site⁵ and then perform the steps shown in listing 5.1.

Footnote 4 <http://archive.ics.uci.edu/ml/datasets/Spambase>

Footnote 5 <https://github.com/WinVector/zmPDSwR/tree/master/Spambase>

Listing 5.1 Building and applying a logistic regression spam model

```
spamD <- read.table('spamD.tsv', header=T, sep='\t')
spamTrain <- subset(spamD, spamD$rgroup>=10)
spamTest <- subset(spamD, spamD$rgroup<10)
spamVars <- setdiff(colnames(spamD), list('rgroup', 'spam'))
spamFormula <- as.formula(paste('spam=="spam"',
  paste(spamVars, collapse=' + '), sep=' ~ '))
spamModel <- glm(spamFormula, family=binomial(link='logit'),
  data=spamTrain)
spamTrain$pred <- predict(spamModel, newdata=spamTrain,
  type='response')
spamTest$pred <- predict(spamModel, newdata=spamTest,
  type='response')
```

A sample of the results of our simple spam classifier are shown below:

```
> sample <- spamTest[c(7,35,224,327),c('spam','pred')]
> print(sample)
      spam      pred
115    spam 0.9903246227
361    spam 0.4800498077
2300 non-spam 0.0006846551
3428 non-spam 0.0001434345
```

THE CONFUSION MATRIX

The absolute most interesting summary of classifier performance is called the confusion matrix. This matrix is just a table that summarizes the classifier's predictions against the actual known data categories.

The confusion matrix is a table counting how often each combination of known outcome (the truth) occurred in combination with each prediction type. For our example spam problem the confusion matrix is given by the following R command:

```
> cM <- table(truth=spamTest$spam, prediction=spamTest$pred>0.5)
> print(cM)
      prediction
truth      FALSE TRUE
```

non-spam	264	14
spam	22	158

From this summary we can now start to judge the performance of the model. In a two by two confusion matrix every cell has a special name which we illustrate in table 5.2.

TIP

Changing a score to a classification

Notice we converted the numerical prediction score into a decision by checking if the score was above or below 0.5. This threshold can be arbitrary and picking thresholds other than 0.5 can allow the data scientist "trade precision for recall" (two terms which we define below). Always start at 0.5 but consider trying other thresholds.

Table 5.2 Standard 2 by 2 confusion matrix

	prediction=NEGATIVE	prediction=POSITIVE
truth mark=NOT IN CATEGORY	True Negatives (TN) cM[1,1]=264	False Positives (FP) cM[1,2]=14
truth mark=IN CATEGORY	False Negatives (FN) cM[2,1]=22	True Positives (TP) cM[2,2]=158

Most of the performance measures of a classifier can be read off the entries of this confusion matrix. We start with the most common measure: accuracy.

ACCURACY

Accuracy is by far the most widely known measure of classifier accuracy. For a classifier accuracy is defined as the number of items categorized correctly divided by the total number of items. It is simply what fraction of the time the classifier is correct. At the very least you want a classifier to be accurate. In terms of our confusion matrix accuracy is $(TP+TN) / (TP+FP+TN+FN)$ $= (cM[1,1]+cM[2,2]) / \text{sum}(cM)$ or 92% accurate. The error of around 8% is unacceptably high for a spam filter, but good for illustrating different sorts model evaluation criteria.

WARNING Categorization accuracy is not the same as numeric accuracy

It is very important to not confuse this use of the word "accuracy" used in a classification sense with "accuracy" used in a numeric sense (as in the ISO 5725 defining score based accuracy as a numeric quantity that can be decomposed into numeric versions of "trueness" and "precision."). These are, unfortunately, two different meanings of the word.

Before we move on we would like to share the confusion matrix of a good spam filter. Below we create the confusion matrix for the Akismet comment spam filter from the Win-Vector blog:

```
> t <- as.table(matrix(data=c(288-1,17,1,13882-17),nrow=2,ncol=2))
> rownames(t) <- rownames(cM)
> colnames(t) <- colnames(cM)
> print(t)
      FALSE   TRUE
non-spam    287     1
spam        17 13865
```

Because the Akismet filter uses link destination clues and determination from other web sites (in addition to text features) it achieves a more acceptable accuracy of $(t[1,1]+t[2,2])/sum(t)$ or over 99.87%. More importantly Akismet seems to have suppressed fewer good comments. Our next section on precision and recall will help us quantify this distinction.

WARNING Accuracy is an inappropriate measure for unbalanced classes

Suppose we have a situation where we have a rare event (say severe complications during child birth). If the event we are trying to predict is rate (say around 1% of the population) the null model of saying "the rare event never happens" is very accurate. The null model is in fact more accurate than a useful (but not perfect model) that identifies 5% of the population at being "at risk" and captures all of the bad events in its 5%. This is not any sort of "paradox," it is just that accuracy is not a good measure for events that have unbalanced distribution or unbalanced costs (different costs of "type 1" and "type 2" errors).

PRECISION AND RECALL

Another evaluation measure used by machine learning researchers is a pair of numbers called precision and recall. These terms come from the field of information retrieval and are defined as follows. Precision is what fraction of the items the classifier flags as being in the class are actually in the class. So precision = $TP / (TP+FP) = cM[2,2] / (cM[2,2]+cM[1,2])$, or about 0.92 (it is only a coincidence that this is so close to the accuracy number we reported earlier). Again, precision is how often a positive indication turns out to be correct. It is important to remember that the precision is a function of both the classifier and the data set. It doesn't make sense to ask how precise a classifier is in isolation, it is only sensible to ask how precise a classifier is for a given data set.

In our email spam example 93% precision means 7% of what was flagged as spam was in fact not spam. This is an unacceptable rate to lose possibly important messages. Akismet on the other hand had a precision of $t[2,2] / (t[2,2]+t[1,2])$ or over 99.99%, so in addition to having a high accuracy Akismet has even higher precision (very important in a spam filtering application).

The companion score to precision is recall. Recall is what fraction of the things that are in the class are detected by the classifier, or: $TP / (TP+FN) = cM[2,2] / (cM[2,2]+cM[2,1])$. For our email spam example this is 88% and for the Akismet example it is 99.87%. In both cases most spam is in fact tagged (we have high recall) and precision is emphasized over recall (which is appropriate for a spam filtering application).

What to remember is: precision is a measure of confirmation (how often when the classifier says "positive" is it in fact correct) and recall is a measure of utility (how much of what there was to find does the classifier in fact find). Precision and recall tend to be relevant to business needs, and are good measures to discuss with your project sponsor and client.

F1

F1 is a useful combination of precision and recall. If either of precision or recall is very small then F1 is also very small. F1 is defined as $2 * \text{precision} * \text{recall} / (\text{precision} + \text{recall})$. The idea is that a classifier that improves precision or recall by sacrificing a lot of the complementary measure will have a lower F1.

SENSITIVITY AND SPECIFICITY

Scientists and doctors tend to use a pair of measures called sensitivity and specificity.

Sensitivity is also called the true positive rate and is exactly equal to recall. Specificity is also called the true negative rate and is equal to $TN / (TN + FP) = cM[1,1] / (cM[1,1] + cM[1,2])$ or about 95%. Both sensitivity and specificity are measures of effect: what fraction of class members are identified as positive and what fraction of non-class members are identified as negative.

An important property of sensitivity and specificity is: if you flip your labels (switch from "spam" being the class you are trying to identify to "non-spam" being the class you are trying to identify) you just switch sensitivity and specificity. Also, any of the so-called "null classifiers" (classifiers that always say positive or always say negative) always return a zero score on one of sensitivity or specificity. So useless classifiers always score poorly on at least one of these measures. Finally, unlike precision and accuracy, sensitivity and specificity each only involve entries from a single row of table 5.2. So they are independent of the population distribution (which means they are better for some applications and worse for others).

SUMMARY OF COMMON CLASSIFICATION PERFORMANCE MEASURES

We summarize the behavior of both the email spam example and the Akismet example under the common measures we have discussed in table 5.3.

Table 5.3 Example classifier performance measures

	Formula	Email Spam example	Akismet Spam example
Accuracy	$(TP+TN)/(TP+FP+TN+FN)$	0.9214	0.9987
Precision	$TP/(TP+FP)$	0.9187	0.9999
Recall	$TP/(TP+FN)$	0.8778	0.9988
Sensitivity	$TP/(TP+FN)$	0.8778	0.9988
Specificity	$TN/(TN+FP)$	0.9496	0.9965

All of these formulas can seem confusing, and the best way to think about them is to shade in various cells in figure 5.2. If your denominator cells shade in a column then you are measuring a confirmation of some sort (how often the classifier's decision is correct). If your denominator cells shade in a row then you are measuring effectiveness (how much of a given class is detected by a the classifier). The main idea is to use these standard scores and then work with your client and sponsor to see what most models their business needs. For each score you should ask them if they need that score to be high and then run a quick thought experiment with them to confirm you have gotten their business need. You should then be able to write a project goal usual in terms of a minimum bound on a pair of these measures. Table 5.4 shows for each score a typical business need and an example follow-up question.

Table 5.4 Classifier performance measures business stories

Measure	Typical Business Need	Follow up question
Accuracy	"We need most of our decisions to be correct."	"Can we tolerate being wrong 5% of the time? And do users see mistakes like spam marked as non-spam or non-spam marked as spam as being equivalent?"
Precision	"Most of what we marked as spam had darn well better be spam."	"That would guarantee most of what is in the spam folder is in fact spam, but it isn't the best way to measure what fraction of the user's legitimate email is lost. I could cheat on this goal by sending all our users a bunch of easy to identify spam that we get right. Maybe we really want good specificity."
Recall	"We want cut down the amount spam a user sees by a factor of 10 (eliminate 90% of the spam)."	"If 10% of the spam gets through, will the user see mostly good mail or mostly spam? Will this result in a good user experience?"
Sensitivity	"We have to cut a lot of spam, otherwise the user won't see a benefit."	"If we cut spam down to 1% of what it is, would that be a good user experience?"
Specificity	"We must be at least '3-nines' on legitimate email, the user must see at least 99.9% of their non-spam email."	"Will the user tolerate missing 0.1% of their legitimate email, and should we keep a spam folder the user can look at?"

One conclusion for this dialogue process on spam classification would be to recommend writing the business goals as maximizing sensitivity while maintain a specificity of at least 0.999.

5.2.2 Evaluating Scoring Models

Evaluating models that assign scores can be a somewhat visual task. The main concept is looking at what is called the residuals or the difference between our predictions $f(x[i,])$ and actual outcomes $y[i]$. Figure 5.7 illustrates the concept.

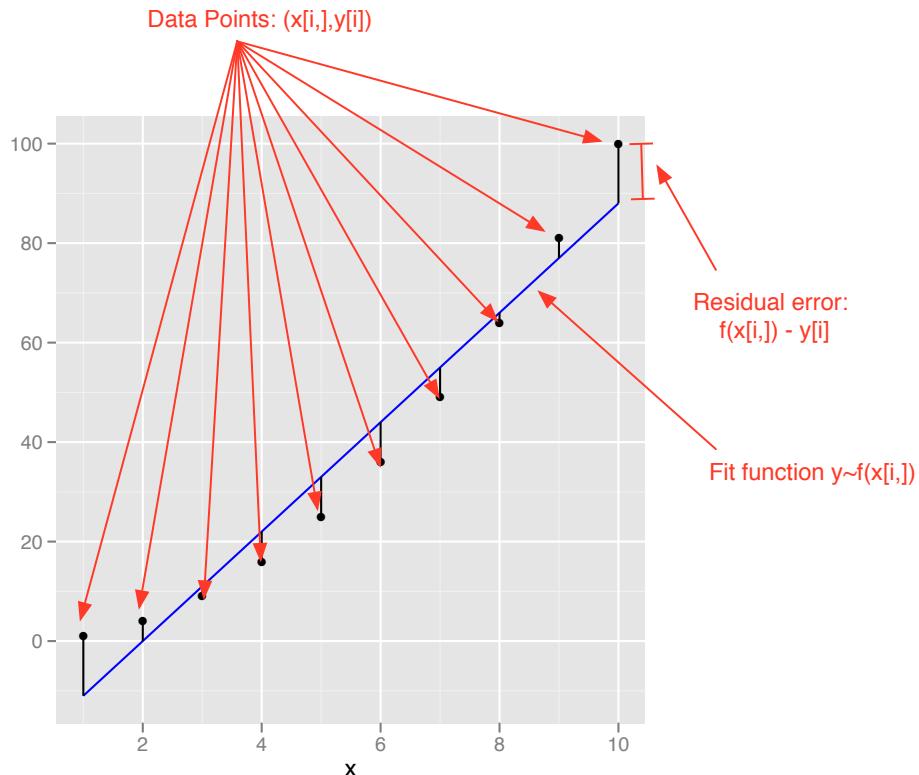


Figure 5.7 Scoring Residuals

The data and graph in Figure 5.7 were produced by the following R commands:

```
d <- data.frame(y=(1:10)^2,x=1:10)
model <- lm(y~x,data=d)
d$prediction <- predict(model,newdata=d)
library('ggplot2')
ggplot(data=d) + geom_point(aes(x=x,y=y)) +
  geom_line(aes(x=x,y=prediction),color='blue') +
  geom_segment(aes(x=x,y=prediction,yend=y,xend=x)) +
  scale_y_continuous('')
```

ROOT MEAN SQUARE ERROR

The most common measure of fit is called Root Mean Square Error (RMSE). This is the square root of the average square of the difference between our prediction and actual values. Think of it as being like a standard deviation: how much your prediction is typically off. In our case the RMSE is $\text{sqrt}(\text{mean}((\text{d\$prediction}-\text{d\$y})^2))$ or about 7.27. The RMSE is in the same units as your y-values are, so if your y-units are pounds your RMSE is in pounds. RMSE is a good measure, because it is often what the fitting algorithms you are using are explicitly trying to minimize. A good RMSE business goal would be: "we want the RMSE on account valuation to be under \$1000 per account."

R-SQUARED OR R²

Another important measure of fit is called R-squared (or R², or the coefficient of determination). It is defined as one minus how much unexplained variance your model leaves (measured relative to a null model of just using the average y as a prediction). In our case the R-squared is $1-\text{sum}((\text{d\$prediction}-\text{d\$y})^2)/\text{sum}((\text{mean}(\text{d\$y})-\text{d\$y})^2)$ or 0.95. R-squared is dimensionless (it is not the units of your y-s) and the best possible R-squared is 1.0 (with near zero or negative R-squared being horrible). R-squared can be thought of as what fraction of the y variation is explained by the model. Under certain circumstances R-squared is equal to the square of another measure called the correlation.⁶ R-squared can be derived from RMSE plus a few facts about the data (so R-squared can be thought of as normalized version of RMSE). A good R-squared business goal would be: "we want the model to explain 70% of account value."

Footnote 6 See <http://www.win-vector.com/blog/2011/11/correlation-and-r-squared/>

WARNING

Do not use correlation to evaluate model quality in production

It is tempting to use a score called correlation to measure model quality, but we actually advise against it. The problem is: correlation ignores shifts and scaling factors. So correlation is actually computing if there any shift and re-scaling of your predictor that is a good predictor. This is not a problem for training data (as these predictions tend to not have a systematic bias in shift or scaling by design) but can mask systematic errors that may arise when a model is used in production.

ABSOLUTE ERROR

For many applications (especially those involving predicting monetary amounts) measures such as absolute error `sum(abs(d$prediction-d$y))`, mean absolute error `sum(abs(d$prediction-d$y))/length(d$y)` and relative absolute error `sum(abs(d$prediction-d$y))/sum(abs(d$y))` are tempting measures. It does make sense to check and report these measures, however it is usually not advisable to make these measures the project goal or to attempt to directly optimize them. This is because absolute error measures tend not to "get aggregates right" or "roll up reasonably" as most of the squared errors do.

As an example consider an online advertising campaign with three advertisement purchases returning \$0,\$0 and \$25 respectively. Suppose our modeling task is as simple as picking a single summary value not too far from the original three prices. The price minimizing absolute error is the median, which is \$0 (yielding an absolute error of `sum(abs(c(0,0,25)-20))`) or \$25. The price minimizing square error is the mean, which is \$8.33 (which has a worse absolute error of \$33.33). However the median price of \$0 misleadingly values the entire campaign at \$0. One great advantage of the mean is: aggregating a mean prediction gives an unbiased prediction of the aggregate in question. It is *often* an unstated project need that various totals or roll-ups of predicted amount be close to the roll-ups of the unknown values to be predicted. For monetary applications predicting the totals or aggregates accurately is often more important than getting individual values right. In fact most statistical modeling techniques are designed for "regression" which is the unbiased prediction of means or expected values.

5.2.3 Evaluating Probability Models

A model type very related to both classification and scoring models are probability models. These are models that for each item in addition to deciding if it is in a given class or not also returns an estimated probability of confidence of being in the class. The modeling techniques of Logistic Regression and Decision Trees are fairly famous in being able to return good probability estimates. Such models can be evaluated on their final decisions as we have already shown in section 5.2.1, but they can also be evaluated in terms of their estimated probabilities. We will continue the example from section 5.2.1 in this vein. In our opinion most of the measures for probability models are very technical and very good and comparing the qualities of different models on the same data set. These criteria are however not easy to precisely translate into businesses needs. So we recommend tracking them, but not using them with your project sponsor or client.

When thinking about probability models it is useful to construct a double density plot like as shown in figure 5.8.

```
ggplot(data=spamTest) +  
  geom_density(aes(x=pred,color=spam,linetype=spam))
```

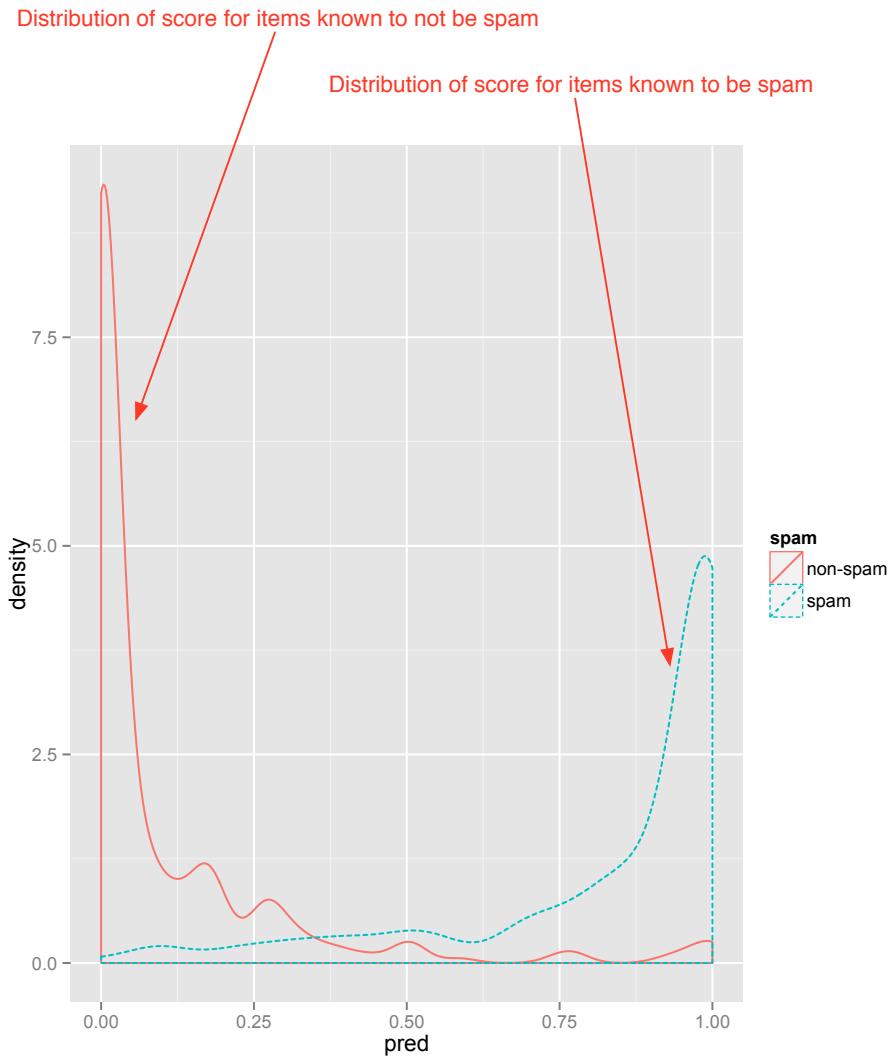


Figure 5.8 Distribution of score broken up by known classes

Figure 5.8 is particularly useful in picking and explaining classifier thresholds. It also illustrates what we are going to try to check when evaluating estimated probability models: examples in the class should mostly have high scores and examples not in the class should mostly have low scores.

THE RECEIVER OPERATING CHARACTERISTIC CURVE

The Receiver Operating Characteristic Curve (or ROC curve) is a popular alternative to the double density plot. For each different classifier we would get by picking a different score threshold between positive and negative determination we plot both the true positive rate and false positive rate. This curve represents every possible trade-off between sensitivity and specificity that is available for this classifier. The steps to produced the ROC plot in figure 5.9 are given below. In the last line we compute the AUC or "area under the curve" which is 1.0 for perfect classifiers and 0.5 for classifiers that do no better than random guesses.

```
library('ROCR')
eval <- prediction(spamTest$pred,spamTest$spam)
plot(performance(eval,"tpr","fpr"))
print(attributes(performance(eval,'auc'))$y.values[[1]])
[1] 0.9660072
```

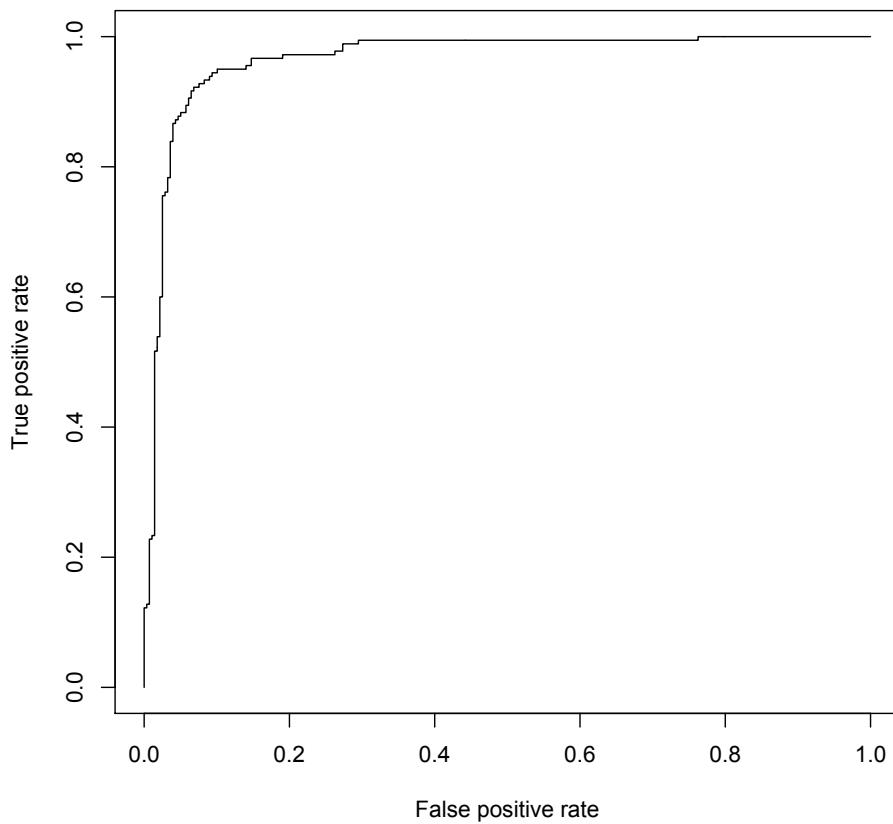


Figure 5.9 ROC curve for the spam scoring example

We are not big fans of the AUC, but working around the ROC curve with your project client is a good way to explore possible project goal trade-offs.

LOG LIKELIHOOD

An important evaluation of an estimated probability is the log likelihood. The log likelihood is the logarithm of the product of the probability the model assigned to each example.⁷ For spam email with a score of 0.9 of being spam the log likelihood is $\log(0.9)$, for a non-spam email the same score of 0.9 is a log likelihood of $\log(1-0.9)$ (or just the log of 0.1, which was the estimated probability of not being spam). The principle is if the model is a good explanation then the data should look likely under the model. The log likelihood of our example is given by:

Footnote 7 The traditional way of calculating the log likelihood is to compute the sum of the logarithms of the probabilities the model assigns to each example.

```

> sum(ifelse(spamTest$spam=='spam',
  log(spamTest$pred),
  log(1-spamTest$pred)))
[1] -134.9478
> sum(ifelse(spamTest$spam=='spam',
  log(spamTest$pred),
  log(1-spamTest$pred)))/dim(spamTest)[[1]]
[1] -0.2946458

```

The first term (**-134.9478**) is the model log likelihood the model assigns to the test data. This number is always going to be negative, and is better as we get closer to zero. The second expression is the log likelihood re-scaled by the number of data points to give us a rough average surprise per data-point. Now a good null-model in this case would be always returning the probability of $180/458$ (the number of known spam emails over the total number of emails as the best single number estimate of the chance of spam). This null model gives the following log likelihood.

```

> pNull <- sum(ifelse(spamTest$spam=='spam',1,0))/dim(spamTest)[[1]]
> sum(ifelse(spamTest$spam=='spam',1,0))*log(pNull) +
  sum(ifelse(spamTest$spam=='spam',0,1))*log(1-pNull)
[1] -306.8952

```

The spam model assigns a log likelihood of **-134.9478** which is much better than the null model's **-306.8952**.

DEVIANCE

Another common measure when fitting probability models is the deviance. The deviance is defined as $-2 * (\text{logLikelihood} - S)$ where S is a technical constant called "the log likelihood of the saturated model." The lower the residual deviance, the better the model. In most cases the saturated model is a perfect model that returns probability 1 for items in the class and probability 0 for items not in the class (so $S=0$). We are most concerned with differences of deviance: such as the difference between our the null deviance and the model deviance (and in this case the S cancels out). In our case this difference is $-2 * (-306.8952 - S) - -2 * (-134.9478 - S) = 344.9$. With $S=0$ the deviance can be used to calculate a pseudo R-squared⁸ Think of the null deviance as how much variation there is to explain, the and the model deviance has how much was left unexplained by the model. So in this case our pseudo R-squared is $1 - (-2 * (-134.9478 - S)) / (-2 * (-306.8952 - S)) = 0.56$ (good, but not great).

Footnote 8 <http://www.win-vector.com/blog/2011/09/the-simpler-derivation-of-logistic-regression/>

AIC

An important variant of deviance is the Akaike information criterion or AIC. This is equivalent deviance plus two times the number of parameters used in the model used to make the prediction. Thus AIC is deviance penalized for model complexity. A nice trick is to add to the deviance $2 * 2^{\text{entropy}}$, as 2^{entropy} can be thought of as the effective number of parameters. The AIC is useful for comparing models with different measures of complexity and variables with differing number of levels.⁹

Footnote 9 Rigorously balancing model quality and model complexity is a deep problem.

ENTROPY

Entropy is a fairly technical measure of information or surprise and is measured in a unit called "bits." If p is a vector containing the probability of each possible outcome then the entropy of the outcomes is calculated as: `sum(-p*log(p, 2))` where p (with the convention that $0*\log(0) = 0$). As entropy measures surprise you want what is called the conditional entropy of your model to be appreciably lower than the original entropy. The conditional entropy is a measure which gives a good measure of how good the prediction is on different categories, tempered by how often it predicts different categories. In terms of our confusion matrix `cM` we can calculate the original entropy and conditional (or residual) entropy as follows:

```
> entropy <- function(x) { ①
  xpos <- x[x>0]
  scaled <- xpos/sum(xpos)
  sum(-scaled*log(scaled,2))
}

> print(entropy(table(spamTest$spam))) ②
[1] 0.9667165

> conditionalEntropy <- function(t) { ③
  (sum(t[,1])*entropy(t[,1]) + sum(t[,2])*entropy(t[,2]))/sum(t)
}

> print(conditionalEntropy(cM)) ④
[1] 0.3971897
```

- ① Define a function that computes the entropy from a list of outcome counts.
- ② Calculate the entropy of the spam/non-spam distribution.
- ③ Function to calculate conditional or remaining entropy of the spam distribution (rows) given the prediction (columns).
- ④ Calculate the conditional or remaining entropy of the spam distribution given the prediction.

We see the initial entropy is 0.9667 bits per example (so a lot of surprise is present) and the conditional entropy is only 0.397 bits per example.

5.2.4 Evaluating Ranking Models

Ranking models are models that for a set of examples sort the rows or (equivalently) assign ranks to the rows. Ranking models are often trained by converting groups of examples into many pair-wise decisions (statements like "a is before b"). You can then apply the criteria for evaluating classifiers to quantify the quality of your ranking function. Two other standard measures of a ranking model are Spearman's rank correlation coefficient (treating assigned rank as a numeric score) and the data mining concept of "lift" (treating ranking as sorting)¹⁰. Ranking evaluation is well handled by business driven ad hoc methods, so we will not spend any more time on this issue.

Footnote 10 [http://en.wikipedia.org/wiki/Lift_\(data_mining\)](http://en.wikipedia.org/wiki/Lift_(data_mining))

5.2.5 Evaluating Clustering Models

Clustering models are hard to evaluate because they are unsupervised: the clusters items are assigned to are generated by the modeling procedure, not supplied in a series of annotated examples. Evaluation is largely checking observable summaries about the clustering. As a quick example we are going to demonstrate evaluating division of 100 random points in the plane into five clusters. We generate our example data and proposed k-means based clustering below.

```
set.seed(32297)
d <- data.frame(x=runif(100),y=runif(100))
clus <- kmeans(d,centers=5)
d$cluster <- clus$cluster
```

Because our example is two dimensional it is easy to visualize, so we can use the commands below to generate Figure 5.10 which we can refer to when thinking about clustering quality.

```
library('ggplot2'); library('grDevices')
h <- do.call(rbind,
  lapply(unique(clus$cluster),
    function(c) { f <- subset(d,cluster==c); f[chull(f),]}))
ggplot() +
  geom_text(data=d,aes(label=cluster,x=x,y=y,
    color=cluster),size=3) +
  geom_polygon(data=h,aes(x=x,y=y,group=cluster,fill=as.factor(cluster)),
    alpha=0.4,linetype=0) +
```

```
theme(legend.position = "none")
```

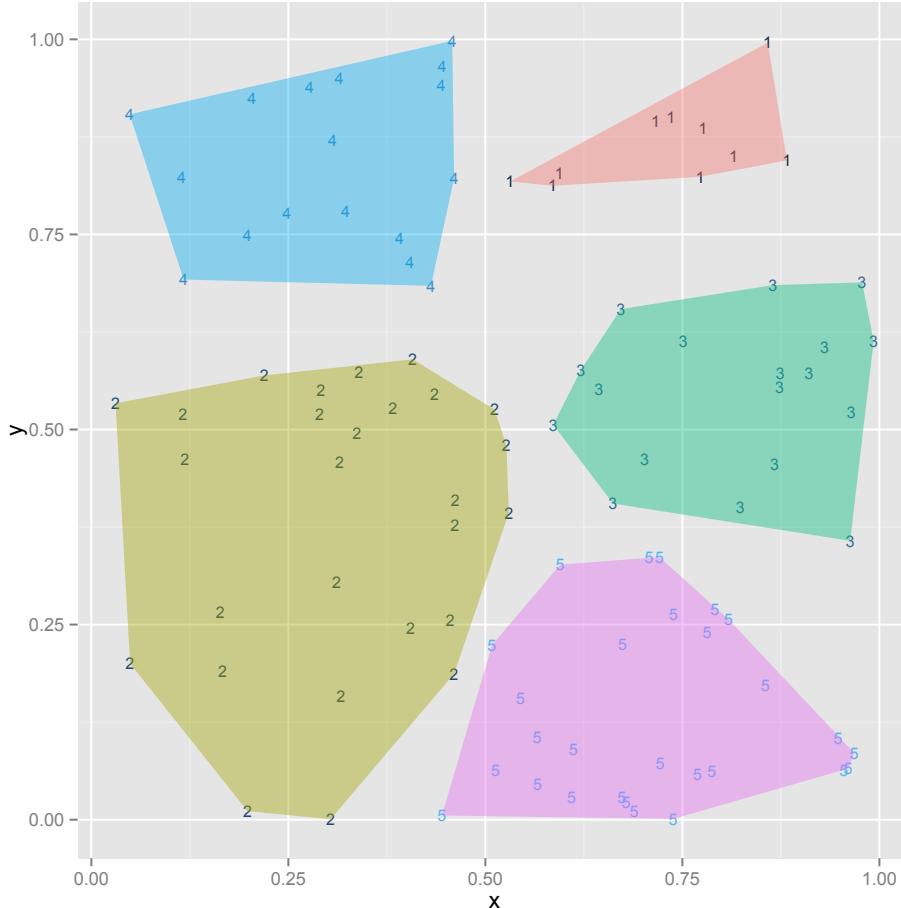


Figure 5.10 Example clustering

The first qualitative metrics are how many clusters you have (sometimes chosen by the user, sometimes chosen by the algorithm) and number of items in each cluster. This is quickly calculated by the table command:

```
> table(d$cluster)

 1  2  3  4  5 
10 27 18 17 28
```

We see we have five clusters each with 10 to 28 points. Two things to look out for are "hair clusters" (clusters with very few points) and "waste clusters" clusters

with a very large number of points. Both of these are usually not useful (hair clusters are essentially individual examples and huge clusters usually have very little in common between items in the cluster).

INTRA CLUSTER DISTANCES VERSUS CROSS CLUSTER DISTANCES

The most common goal for clusters is for them to be compact in what ever distance scheme you used to define them. The traditional measure of this is comparing the typical distance between two items in the same cluster to the typical distance between two items from different clusters. We can produce a table off all these distance facts as follows:

```
> library('reshape2')
> pairs <- data.frame(
+   ca = as.vector(outer(1:n,1:n,function(a,b) d[a,'cluster'])),
+   cb = as.vector(outer(1:n,1:n,function(a,b) d[b,'cluster'])),
+   dist = as.vector(outer(1:n,1:n,function(a,b)
+     sqrt((d[a,'x']-d[b,'x'])^2 + (d[a,'y']-d[b,'y'])^2)))
+ )
> #aggregate(dist ~ ca + cb,mean,data=pairs)
> dcast(pairs,ca~cb,value.var='dist',mean)
  ca      1       2       3       4       5
1  1 0.1478480 0.6524103 0.3780785 0.4404508 0.7544134
2  2 0.6524103 0.2794181 0.5551967 0.4990632 0.5165320
3  3 0.3780785 0.5551967 0.2031272 0.6122986 0.4656730
4  4 0.4404508 0.4990632 0.6122986 0.2048268 0.8365336
5  5 0.7544134 0.5165320 0.4656730 0.8365336 0.2221314
```

TREATING CLUSTERS AS CLASSIFICATIONS OR SCORES

Distance metrics are good for checking the performance of clustering algorithms, but they don't always translate to business needs. When sharing a clustering with your project sponsor or client we advise treating the cluster assignment as if it was a classification. For each cluster label generate an outcome assigned to cluster (such as all email in the cluster is marked as spam/non-spam or all accounts in the cluster are treated has having a revenue value equal to the mean revenue value in the cluster). Then use either the classifier or scoring model evaluation ideas to evaluate the value of the clustering. This scheme works best if the column you are considering outcome (such as spam/non-spam or revenue value of the account) was not used as one of the dimensions in constructing the clustering.

5.3 Validating Models

We have discussed how to choose a modeling technique and evaluate the performance of the model on training data. At this point your biggest worry should be the validity of your model: will it show similar quality on new data in production? We call the testing of a model on new data (or a simulation of new data from our test set) as model validation. The main things we are trying to defend against for are:

5.3.1 Identifying common model problems

Table 5.5 Common model problems

Problem	Description
bias	Systematic error in the model, such as always under predicting.
variance	Undesirable (but non systematic) distance between predictions and actual values. Often due to over sensitivity of the model training procedure to small variations in the distribution of training data.
over-fit	Features of the model that arise from relations that are in the training data, but not representative of the general population. Over-fit can usually be reduced by acquiring more training data and by techniques like regularization and bagging.
non-significance	A model that appears to show an important relation, when in fact the relation may not hold in the general population or equally good predictions can be made without the relation.

OVER FITTING

A lot of the modeling problems are related to over-fitting. Looking for signs of over-fit is a good first step in diagnosing models.

An over-fit model looks great on the training data and performs poorly on new data. A model's prediction error on the data that it trained from is called *training error*. A model's prediction error on new data is called *generalization error*. Usually, training error will be smaller than generalization error (no big surprise).

Ideally, though, the two error rates should be close. If generalization error is large, then your model has probably *overfit*: that is, it's memorized the training data instead of discovering generalizable rules or patterns. You want to avoid over fitting by preferring (as long as possible) simpler models which do in fact tend to generalize better.¹¹ In this section we are not just evaluating a single model, we are evaluating your data and work procedures. Figure 5.11 shows the typical appearance of a reasonable model and an overfit model.

Footnote 11 Other techniques to prevent overfitting include regularization (preferring small effects from model variables) and bagging (averaging different models to reduce variance).



Figure 5.11 A notional illustration of over fitting

An overly complicated and overfit model is bad for at least two reasons. First an overfit model may be much more complicated than anything useful. For example the extra wiggles in the over-fit part of figure 5.11 could make optimizing with respect to x needlessly difficult. Also, as we mentioned overfit models tend to not be as accurate in production as during training, which is very embarrassing.

5.3.2 Quantifying Model Soundness

It is important that you know, quantify and share how sound your model is. Evaluating a model only on the data used to construct it is favorably biased: models tend to look good on the data they were built from. Also, a single evaluation of model performance gives only a "point estimate" of performance. You need a good characterization of how much potential variation there is in your model production and measurement procedure, and how well your model is likely to perform on future data. We see these questions as being fundamental *frequentist* concern because they are questions about how model behavior changes under variations in data. The formal statistical term closest to these business questions is "significance" and we will abuse notation and call what we are doing significance testing.¹² In this section we will discuss some testing procedures, but leave off demonstrating implementation until later in this book.

Footnote 12 A lot of what we are doing is in fact significance testing, it is just we want to keep our testing choices based on measuring specific business relevant risks and the worry if we have gone slightly heterodox.

NOTE**Frequentist and Bayesian inference**

Following Efron there are at least two fundamental ways of thinking about inference: Frequentist and Bayesian. In Frequentist inference we assume a single unknown quantity (be it a parameter, model, or prediction) that we are trying to estimate. The Frequentist inference for a given data set is what is called a "point estimate" that varies as different possible data sets are proposed (yielding a distribution of estimates). In Bayesian inference we assume the unknown quantity to be estimated has many plausible values modeled by what is called a prior distribution. Bayesian inference is then using data (that is considered as unchanging) to build a tighter posterior distribution for the unknown quantity. There is a stylish snobbery that somehow Bayesian inference is newer and harder to do than Frequentist inference and therefore more sophisticated. In practice choosing your inference framework is not a matter of taste but a direct consequence of what sort of business question you are trying to answer.¹³ If you are worried about the sensitivity of your result to variation in data and modeling procedures you should work in the Frequentist framework. If you are worried about the sensitivity of your result to possible variation in the unknown quantity to be modeled you should work in the Bayesian framework.

Footnote 13 For more see:

<http://www.win-vector.com/blog/2013/05/bayesian-and-frequentist-approaches-ask-the-right-question/>

5.3.3 Ensuring model quality

The standards of scientific presentation are: you should always share how sensitive your conclusions are to variations in your data and procedures. You should never just show a model and a model quality statistics. You should also show the likely distribution of the statistic under variations in your modeling procedure or your data. This is why you would not say something like "we have an accuracy of 90% on our training data" but instead run additional experiments so you can say something like: "we see an accuracy of 85% on hold out data". Or even better: "we saw accuracies of at least 80% on all but 5 percent of our re-runs." These distributional statements tell you if you need more modeling features and/or more data.

TESTING ON HELD-OUT DATA

The data used to build a model is not the best data to test model performance. This is because there is an upward measurement *bias* in this data. Because this data was seen during model construction and model construction is optimising your performance measure (or at least something related to your performance measure) you tend to get exaggerated measures of performance on your training data. Most standard fitting procedures have some built in measure for this effect (for example the "adjusted R squared" report in linear regression) and the effect tends to diminish as your training data becomes large with respect to the complexity of your model.¹⁴

Footnote 14 See for example: "The Unreasonable Effectiveness of Data," Alon Halevy, Peter Norvig, and Fernando Pereira. IEEE Intelligent Systems, 2009.

The precaution for this optimistic bias we demonstrate throughout this book is: split your available data into test and training. Perform all of your clever work on the training data alone and delay measuring your performance with respect to your test data until as late as possible in your project (as all choice you make after seeing your test or hold-out performance introduce a modeling bias). The desire to "keep test secret" for as long as possible is why we often actually split data into training, calibration and test sets (as we demonstrate in section XRF:sect2_6.1.1:sectStartingwithKDD2009).

K-WAY CROSS VALIDATION

Testing on hold-out data while useful only gives a single "point estimate" of model performance. In practice we want both an unbiased estimate of our model's future performance on new data (simulated by test data) *and* an estimate of the distribution of this estimate under typical variations in data and training procedures. A good method to perform these estimates is "k-way cross validation" and the related ideas of empirical re-sampling and bootstrapping.

The idea behind k-way cross validation is to repeat the construction of the model on different subsets of the available training data and then evaluate the model only on data not seen during construction. This is an attempt to simulate the performance of the model on unseen future training data. The need to cross validate is one of the reasons it is critical that model construction be automatable, such as with a script in a language like R, and not depend on manual steps. Assuming you enough data to cross validate (i.e. not having to worry over-much about the "statistical efficiency" of techniques) is one of the differences between

the attitudes of data science and traditional statistics. Section XRF:sect2_6.2.3:sectCrossVal works through an example of automating k-way cross validation.

SIGNIFICANCE TESTING

Statisticians have a powerful idea related to cross validation called significance testing. Significance also goes under the name of "p-values" and you will actually be asked "what is your p-value" when presenting.

The idea behind significance is that we can believe our model's performance is good if it is very unlikely that a bad model construction technique could score such high performance. The deliberately bad models are called "null hypotheses" and the standard incantation is: "we can reject the null hypothesis" (meaning we can take our model as not coming from the bad process, and therefore likely to be good). The null models are always of a simple form: assuming two effects are independent when we are trying to model a relation, or assuming a variable has no effect when we are trying to measure an effect strength.

For example: suppose you've trained a model to predict how much a house will sell for, based on certain variables. You want to know if your model's predictions are better than simply guessing the average selling price of a house in the neighborhood (call this "the null model"). Your new model will mispredict a given house's selling price by a certain average amount, which we will call `err.model`. The null model will also mispredict a given house's selling price by a different amount, `err.null`. The null hypothesis is that $D = (err.null - err.model) == 0$ -- that is, that on average, the new model performs the same as the null model.

When you evaluate your model over a test set of houses, you will (hopefully) see that $D = (err.null - err.model) > 0$ (your model is more accurate). You want to make sure that this positive outcome is genuine, and not just something you observed by chance. The p-value is the probability that you would see a D as large as you observed if the two models actually perform the same.

Our advice is: to always think about p-values as estimates of how often you would find a relation (between the model and the data) when there is actually none. This is why low p-values are good, as they are estimates of the probabilities of

undetected disastrous situations.¹⁵ You might also think of the p-value as the probability of your whole modeling result being one big "false positive." So, clearly, you want the p-value (or the significance) to be small, say less than 0.05.

Footnote 15 See also:

<http://www.win-vector.com/blog/2013/04/worry-about-correctness-and-repeatability-not-p-values/>.

The traditional statistical method of computing significance or p-values is through Student's t-test or an f-test (depending on what you are testing). For classifiers there is a particularly good significance test that is run on the confusion matrix called the `fisher.test()`. These tests are built into most model fitters. They have a lot of math behind them that lets a statistician avoid fitting more than one model. These tests also rely on a few assumptions (to make the math work) that may or may not be true about your data and your modeling procedure.

One way to directly simulate a bad modeling situation is use a permutation test. This is when you permute the input (or independent) variables among example. In this case there is no real relation between the modeling features (which we have permuted among examples) and the quantity to be predicted because in our new data set the modeling features and the result come from different (unrelated) examples. Thus each re-run of the permuted procedure builds something very much like a null model. If our actual model is not much better than the population of permuted models then we should be very suspicious of our actual model. Notice that in this case, we are thinking about the uncertainty of our estimates as being a distribution drawn about the null model (or around coefficients being zero).

We could modify the code in section XRF:sect2_6.2.3:sectCrossVal to perform a (non-exact) permutation test by permuting the y-values each time we re-split the training data. Or we could try a package like that performs the work and/or brings in convenient formulas for the various probability and significance statements that come out of permutation experiments¹⁶

Footnote 16 For example: <http://cran.r-project.org/web/packages/coin/index.html>

CONFIDENCE INTERVALS

An very important and very technical frequentist statistical concept is the *confidence interval*. An example: a "95% confidence interval" is an interval from an estimation procedure such that the *procedure* is thought to have a 95% (or better) chance of catching the true unknown value to be estimated in an interval. It is *not* the case that there is a 95% chance that the unknown true value is actually in the interval at hand (thought it is often misstated as such). The Bayesian alternative to confidence intervals are *credible intervals* (which can be easier to understand, but do require the introduction of a prior distribution).

ABOUT "SLOPPY" USE OF STATISTICAL TERMINOLOGY

Statistics is the field that has spent the most time formally studying the issues of model correctness and model soundness (probability theory, operations research, theoretical computer science, econometrics, and a few other fields have of course also contributed). Because of their priority statisticians often insist that the checking of model performance and soundness be solely described in traditional statistical terms. However, a data-scientist must present to many non-statistical audiences, so the reasoning behind a given test is in fact best explicitly presented and discussed. It is just not practical to always allow the dictates of a single field to completely style a cross-disciplinary conversation.

5.4 Summary

You should now have some solid ideas on how to choose among modeling techniques. You should also know how to evaluate the quality data science work: be it your own or that of others. The rest of this part of the book will go into some more detail on how to build, test and deliver effective predictive models.

5.5 External links section

Chapter 6: Using Memorization Methods



This chapter covers basic models including:

- Using lookup table techniques
- Using nearest neighbor methods
- Using Naive Bayes methods
- Using decision trees

The simplest methods in data science are what we call "memorization methods." These are methods that generate answers by returning a majority category (in the case of classification) or average value (in the case of scoring) of a subset of the original training data. These methods can vary from models depending on a single variable (similar to the analyst's pivot table), to decision trees (similar to what is called "business rules"), to nearest neighbor and Naive Bayes methods.¹ In this chapter we will learn how to use these memorization methods to solve classification problems (though the same techniques also work for scoring problems).

Footnote 1 Be aware: "memorization methods" is a non-standard classification of technique that we are using to organize our discussion.

6.1 KDD and the 2009 KDD cup

For this chapter we will use the 2009 KDD cup as our example data set.

KDD is the "Conference on Knowledge Discovery and Data Mining" and is the premier conference on machine learning methods. Each year KDD hosts a data mining cup where teams analyze a data set and then are ranked against each other. The KDD cup is a very big deal and the inspiration for the famous Netflix Prize and even Kaggle competitions.

The 2009 KDD contest provided a data set about customer relationship modeling. The contest supplied 230 facts about 50,000 credit card accounts. From these features the goal was to predict account cancellation (called churn), the innate tendency to use new products and services (called appetency) and willingness to respond favourably to marketing pitches (called upselling).² As with many score based competitions this contest concentrates on machine learning and deliberately abstracts or skips over a number of important data science issues such as cooperatively defining goals, requesting new measurements, collecting data, and quantifying classifier performance in terms of business goals. In fact for this contest we don't have names or definitions for any of the independent (or input) variables and no real definition of the dependent (or outcome) variables. We get the advantage that the data is already in a ready to model format (all input variables and the results arranged in single row). But, we don't know the meaning of any variable (so we can't merge in outside data sources) and we can't use any method that treat time and repetition of events carefully (such as time series methods or survival analysis).

Footnote 2 Data available from: <http://www.sigkdd.org/kdd-cup-2009-customer-relationship-prediction>. We share the steps to prepare this data for modeling in R here:
<https://github.com/WinVector/zmPDSwR/tree/master/KDD2009>.

To simulate the data science processes we will assume that we can use any column we were given to make predictions (i.e. that all of these columns are known prior to needing a prediction³), the contest metric (AUC) is the correct one, and the AUC of the top contestant is a good Bayes rate estimate (telling us when to stop tuning).

Footnote 3 Checking if a column is actually going to be available during prediction conversely is some function of the unknown output is critical step in data science projects.

WARNING**The worst possible modeling outcome.**

The worst possible modeling outcome is not failing to find a good model. The worst possible modeling outcome is thinking you have a good model when you do not. One of the easiest ways to accidentally build such a deficient model is to have an instrumental or independent variable that is in fact a subtle function of the outcome. Such variables can easily leak into your training data, especially when you have no knowledge or control of variable meaning preparation. The point being, such variables will not actually be available in a real deployment and often are in training data packaged up by others.

6.1.1 Getting started with the KDD 2009 data

PREPARING THE DATA

For our example we will try to predict churn in the KDD data set. The KDD contest was judged in terms of AUC ("Area Under the Curve," a measure of prediction quality discussed in section XRF:sect2_5.2.2:sectEvalScoringModels) so we will also use AUC as our measure of performance.⁴ The winning team achieved an AUC of 0.76 on churn, so we will treat that as our upper bound on possible performance. Our lower bound on performance is an AUC of 0.5, as this is the performance of a useless model.

Footnote 4 Also, as is common for example problems, we have no project sponsor to discuss metrics with so our choice of evaluation is a bit arbitrary.

This problem has a large number variables many of which have a large number of possible levels. We are also using the AUC measure which is not particular resistant to over fitting (not having built in model complexity or chance corrections). Because of this concern we will split our data into three sets: training, calibration and test. The intent of the three way split is: we use the training set for most of our work, and we never look at the test set (we reserve it for our final report of model performance). The calibration set is used to simulate the unseen test set during modeling, that is we look at performance on the calibration set to estimate if we are over-fitting. This three way split procedure is recommended by many researchers. In this book we emphasize a two way training and test split and suggest in general steps like calibration and cross-validation estimates be performed by repeated re-splittings of the training portion of the data (allowing for more efficient estimation than a single split, and keeping the test data completely out of the modeling effort). For simplicity in this example we will split the training portion of our data into training and calibration only a single time in this example. Let's start work in listing 6.1 where we prepare the data for analysis and modeling.

Listing 6.1 Preparing the KDD data for analysis

```
d <- read.table('orange_small_train.data.gz',      ①
  header=T,
  sep='\t',
  na.strings=c('NA',''))
  ②
churn <- read.table('orange_small_train_churn.labels.txt',
  header=F,sep='\t') ③
d$churn <- churn$V1 ④
appetency <- read.table('orange_small_train_appetency.labels.txt',
  header=F,sep='\t')
d$appetency <- appetency$V1 ⑤
upselling <- read.table('orange_small_train_upselling.labels.txt',
  header=F,sep='\t')
d$upselling <- upselling$V1 ⑥
set.seed(729375)
d$rgroup <- runif(dim(d)[[1]])
dTrainAll <- subset(d,rgroup<=0.9)
dTest <- subset(d,rgroup>0.9) ⑦
vars <- setdiff(colnames(dTrainAll),
  c(outcomes,'rgroup'))
catVars <- vars[sapply(dTrainAll[,vars],class) %in%
  c('factor','character')] ⑧
numericVars <- vars[sapply(dTrainAll[,vars],class) %in%
  c('numeric','integer')] ⑨
rm(list=c('d','churn','appetency','upselling')) ⑩
outcome <- 'churn' ⑪
pos <- '1' ⑫
useForCal <- rbinom(n=dim(dTrainAll)[[1]],size=1,prob=0.1)>0 ⑬
dCal <- subset(dTrainAll,useForCal)
dTrain <- subset(dTrainAll,!useForCal)
```

- ① Read the file of independent variables. All data from
<https://github.com/WinVector/zmPDSwR/tree/master/KDD2009>
- ② Treat both "NA" and the empty string as missing data.
- ③ Read the "churn" dependent variable.
- ④ Add churn as a column.
- ⑤ Add appetency as a column.
- ⑥ Add upselling as a column.
- ⑦ Split data into train and test subsets.
- ⑧ Identify which features are categorical variables.
- ⑨ Identify which features are numeric variables.
- ⑩ Remove unneeded objects from workspace.
- ⑪ Choose which outcome to model (churn).

- ⑫ Choose which outcome is considered "positive."
- ⑬ Further split training data into training and calibration.

We are now ready to build some single variable models. Business analysts almost always build single variable models using categorical features, so we will start with these.

SIDE BAR Sub-sample to prototype quickly

Often the data scientist will be so engrossed with the business problem, math and data that they forget how much trial and error is needed. It is often a very good idea to first work on a small subset of your training data, so that it takes seconds to debug your code instead of minutes. Don't work with expensive data sizes until you have to.

6.2 Building Single Variable Models

Single variable models can be a very powerful tool. We are going to show how to build single variable models from both categorical and numeric variables. By the end of this section you should be able to build, evaluate and cross-validate single variable models with confidence.

6.2.1 Using categorical features

A single variable model based on a categorical feature is easiest to describe as a table. For this task business analysts use what is called a pivot table (which promotes values or levels of a feature to be new column names) and statisticians use what is called a contingency table (where each possibility is given a column name). In either case the R command to produce a table is called `table()`. To create a table comparing the levels of variable 218 against the labeled churn outcome we run the `table` command as follows:

```
table218 <- table(
  Var218=dTrain[, 'Var218'], ①
  churn=dTrain[, outcome], ②
  useNA='ifany') ③
print(table218)
      churn
Var218    -1      1
  cJvF 19101  1218
  UYBR 17599  1577
  <NA>   410   148
```

- ① Tabulate levels of Var28.
- ② Tabulate levels of churn outcome.
- ③ Include NA values in tabulation.

From this we see variable 218 takes on two values plus NA and we see the joint distribution of these value against the churn outcome. At this point it is very easy to write down a single variable model based on variable 218:

```
> print(table218[,2]/(table218[,1]+table218[,2]))
   CJvF      UYBR      <NA>
0.05994389 0.08223821 0.26523297
```

This summary tells us that when variable 218 takes on a value of "CJvF" then around 6% of the customers churn, when it is "UYBR" 8% of the customers churn and when it is not recorded ("NA") 27% of the customers churn. The utility of any variable level is a combination of how often the level occurs (rare levels are not very useful) and how extreme the distribution of the outcome is for records matching a given level. Variable 218 seems like a feature that is easy to use and helpful with prediction. In real work we would want to research out with our business partners why it has missing values and what the best thing to do when values are missing (this will depend on how the data was prepared). We also need to design a strategy for what to do if a new level not seen during training were to occur during model use. Since this is a contest problem with no available project partners we will build a function that converts NA to a level (as it seems to be pretty informative) and also treat novel values as uninformative. Our function to convert a categorical variable into a single model prediction is given in listing 6.2.

Listing 6.2 Function to build single variable models for categorical variables

```
mkPredC <- function(outCol, varCol, appCol) { ①
  pPos <- sum(outCol==pos)/length(outCol) ②
  naTab <- table(as.factor(outCol[is.na(varCol)]))
  pPosWna <- (naTab/sum(naTab))[pos] ③
  vTab <- table(as.factor(outCol),varCol)
  pPosWv <- (vTab[pos,]+1.0e-3*pPos)/(colSums(vTab)+1.0e-3) ④
  pred <- pPosWv[appCol] ⑤
  pred[is.na(appCol)] <- pPosWna ⑥
  pred[is.na(pred)] <- pPos ⑦
  pred ⑧
}
```

- ① Given a vector of training outcomes (outCol), a categorical training variable (varCol) and a prediction variable (appCol) use outCol and varCol to build a single variable model and then apply the model to appCol to get new predictions.
- ② Get stats on how often outcome is positive during training.
- ③ Get stats on how often outcome is positive for NA values of variable during training.
- ④ Get stats on how often outcome is positive, conditioned on levels of training variable.
- ⑤ Make predictions by looking up levels of appCol.
- ⑥ Add in predictions for NA levels of appCol.
- ⑦ Add in predictions for levels of appCol that were not known during training.
- ⑧ Return vector of predictions.

Listing 6.2 may seem like a lot of work, but placing all of the steps in a function lets us apply the technique to many variables quickly. The data set we are working with has 38 categorical variables, many of which are almost always NA and many of which have over 10,000 distinct levels. So we definitely want to automate working with these variables as we have. Our first automated step is to adjoin a prediction or forecast (in this case the predicted probability of churning) for each categorical variable as shown below:

```
for(v in catVars) {
  pi <- paste('pred',v,sep=' ')
  dTrain[,pi] <- mkPredC(dTrain[,outcome],dTrain[,v],dTrain[,v])
  dCal[,pi] <- mkPredC(dTrain[,outcome],dTrain[,v],dCal[,v])
  dTest[,pi] <- mkPredC(dTrain[,outcome],dTrain[,v],dTest[,v])
```

```
}
```

Notice in all cases we train with the training data frame and then apply to all three data frames `dTrain`, `dCal`, and `dTest`. We are using an extra calibration data frame (`dCal`) because we have so many categorical variables that have a very large number of levels and are very subject to over-fitting. We wish to have some chance of detecting this over-fitting before moving onto the test data (which we are using as our final check, so it is data we must not use during model construction and evaluation else we may have an exaggerated estimate of our model quality). Once we have the predictions we can find the categorical variables that have a good AUC both on the training data and on the calibration data not used during training. These are likely the more useful variables and are identified by the following loop:

```
library('ROCR')

> calcAUC <- function(predcol,outcol) {
  perf <- performance(prediction(predcol,outcol==pos),'auc')
  as.numeric(perf@y.values)
}

> for(v in catVars) {
  pi <- paste('pred',v,sep='')
  aucTrain <- calcAUC(dTrain[,pi],dTrain[,outcome])
  if(aucTrain>=0.8) {
    aucCal <- calcAUC(dCal[,pi],dCal[,outcome])
    print(sprintf("%s, trainAUC: %4.3f calibrationAUC: %4.3f",
      pi,aucTrain, aucCal))
  }
}
[1] "predVar200, trainAUC: 0.828 calibrationAUC: 0.527"
[1] "predVar202, trainAUC: 0.829 calibrationAUC: 0.522"
[1] "predVar214, trainAUC: 0.828 calibrationAUC: 0.527"
[1] "predVar217, trainAUC: 0.898 calibrationAUC: 0.553"
```

Notice how, as expected, each variables training AUC is inflated compared to its calibration AUC. This is because many of these variables have thousands of levels. For example `length(unique(dTrain$Var217))` is 12,434 indicating that variable 217 has 12,434 levels. A good trick to work around this is to sort the variables by their AUC score on the calibration set (not seen during training) which is a better estimate of the variable's true utility. In our case the

most promising variable is variable 206 which has both training and calibration AUCs of 0.59. The winning KDD entry, which was a model that combined evidence from multiple features, had a much larger AUC of 0.76.

6.2.2 Using numeric features

There are a number of ways to use a numeric feature to make predictions. A common method is to bin the numeric feature into a number of ranges and then use the range labels as a new categorical variable. R can do this quickly with its `quantile()` and `cut()` commands as we show below:

```
> mkPredN <- function(outCol,varCol,appCol) {
  cuts <- unique(as.numeric(quantile(varCol,
    probs=seq(0, 1, 0.1),na.rm=T)))
  varC <- cut(varCol,cuts)
  appC <- cut(appCol,cuts)
  mkPredC(outCol,varC,appC)
}
> for(v in numericVars) {
  pi <- paste('pred',v,sep='')
  dTrain[,pi] <- mkPredN(dTrain[,outcome],dTrain[,v],dTrain[,v])
  dTest[,pi] <- mkPredN(dTrain[,outcome],dTrain[,v],dTest[,v])
  dCal[,pi] <- mkPredN(dTrain[,outcome],dTrain[,v],dCal[,v])
  aucTrain <- calcAUC(dTrain[,pi],dTrain[,outcome])
  if(aucTrain>=0.55) {
    aucCal <- calcAUC(dCal[,pi],dCal[,outcome])
    print(sprintf("%s, trainAUC: %4.3f calibrationAUC: %4.3f",
      pi,aucTrain,aucCal))
  }
}
[1] "predVar6, trainAUC: 0.557 calibrationAUC: 0.554"
[1] "predVar7, trainAUC: 0.555 calibrationAUC: 0.565"
[1] "predVar13, trainAUC: 0.568 calibrationAUC: 0.553"
[1] "predVar73, trainAUC: 0.608 calibrationAUC: 0.616"
[1] "predVar74, trainAUC: 0.574 calibrationAUC: 0.566"
[1] "predVar81, trainAUC: 0.558 calibrationAUC: 0.542"
[1] "predVar113, trainAUC: 0.557 calibrationAUC: 0.567"
[1] "predVar126, trainAUC: 0.635 calibrationAUC: 0.629"
[1] "predVar140, trainAUC: 0.561 calibrationAUC: 0.560"
[1] "predVar189, trainAUC: 0.574 calibrationAUC: 0.599"
```

Notice in this case the numeric variables behave similarly on the training and calibration data. This is because our prediction method converts the numeric variable into a categorical variable with around ten well distributed levels, so our training estimate tends to be good and not over-fit. We could improve our numeric estimate by interpolating between quantiles. Other methods we could have used are

kernel based density estimation and parametric fitting. Both of these methods are usually available in the variable treatment steps of Naive Bayes classifiers.

A good way to visualize the predictive power of a numeric variable is the double density plot where we plot on the same graph the variable score distribution for positive examples and variable score distribution of negative examples as two groups. Figure 6.1 shows the performance of the single variable model built from the numeric feature `Var126`.

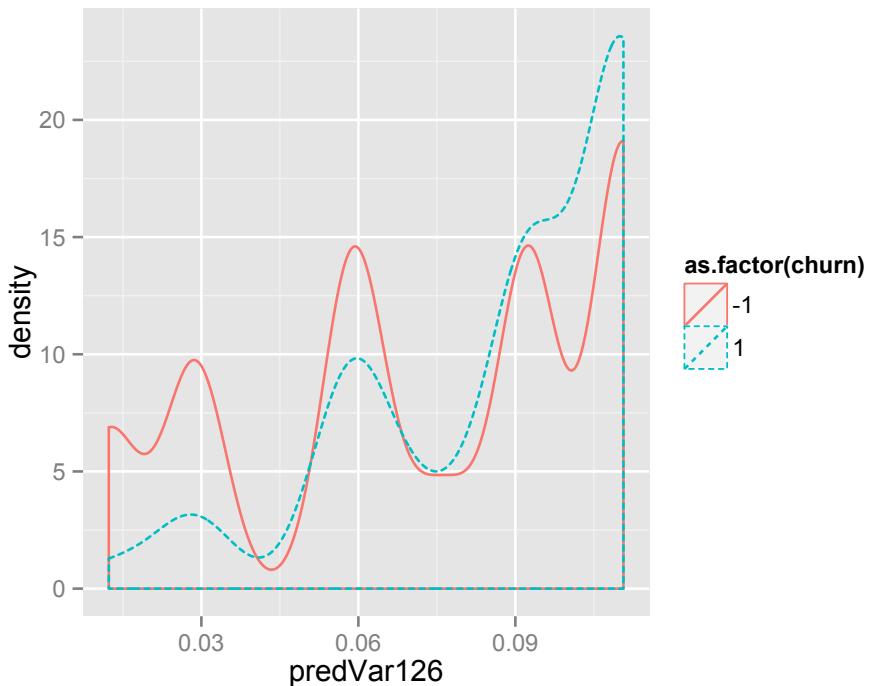


Figure 6.1 Performance of variable 126 on calibration data

The code to produce figure 6.1 is:

```
ggplot(data=dCal) +  
  geom_density(aes(x=predVar126,color=as.factor(churn)))
```

SIDE BAR**Dealing with missing values in numeric variables**

One of the best strategies we have seen for dealing with missing values in numeric variables is the following two step process. First for each numeric variable introduce a new advisory variable this is one when the original variable had a missing value and zero otherwise. Second replace all missing values of the original variable with zero. You now have removed all of the missing values and have recorded enough details so that missing values are not fully confused with actual zero values.

6.2.3 Using Cross-Validation to Estimate Effects of Overfitting

We now have enough experience fitting the KDD to try and estimate the degree of over fitting we are seeing in our models. We can use a procedure called "cross validation" to estimate the degree of over-fit we have hidden in our models. Cross validation applies in *all* modeling situations. This is the first opportunity we have to demonstrate it, so we will work through an example here.

In repeated cross validation a subset of the training data is used to build a model and a complementary subset of the training data is used to score the model. We can implement a cross-validated estimate of the AUC of the single variable model based on variable 217 with the following code:

```
> var <- 'Var217'  
> aucs <- rep(0,100)  
> for(rep in 1:length(aucs)) {  
  useForCalRep <- rbinom(n=dim(dTrainAll)[[1]],size=1,prob=0.1)>0 ①  
  predRep <- mkPredC(dTrainAll[!useForCalRep,outcome],  
    dTrainAll[!useForCalRep,var],  
    dTrainAll[useForCalRep,var]) ②  
  aucs[rep] <- calcAUC(predRep,dTrainAll[useForCalRep,outcome]) ③  
}  
> mean(aucs)  
[1] 0.5556656  
> sd(aucs)  
[1] 0.01569345
```

① For 100 iterations...

② Select a random subset of about 10% of the training data as a hold-out set

③ Use the random 90% of the training data to train a model, then evaluate that model on the hold-out set.

- ④ Calculate the resulting model's AUC using the hold-out set, store that value, and repeat.

This shows that the 100 fold replicated estimate of the AUC has a mean of 0.556 and a standard deviation of 0.016. So our original section 6.2 estimate of the AUC of this variable of 0.553 was very good. In some modeling circumstances training set estimations are often good enough (linear regression is often such an example). In many other circumstances estimations from a single calibration set are good enough. And in extreme cases (such as fitting models with very many variables or level values) you are well advised to use replicated cross validation estimates of variable utilities and model fits. Automatic cross validation is *extremely* useful in all modeling situations, so it is critical you automate your modeling steps so you can perform cross validation studies. We are demonstrating cross validation here as single variable models are among the simplest to work with.

ASIDE: CROSS VALIDATION IN FUNCTIONAL NOTATION

As a point of style `for () {}` loops are considered an undesirable crutch in R. We used a for-loop in our cross validation example as this is the style of programming that is likely to be most familiar to non-specialists. The point is for-loops over-specify what you want (they describe both what you want and the exact order of steps to achieve it) so they tend to be less re-usable and less composable with other computational. When you become proficient in R you look to eliminate for-loops from your code and use either vectorized or functional methods where appropriate. For example the cross validation we just demonstrated could be performed in a functional manner as follows.

```
> fCross <- function() {
  useForCalRep <- rbinom(n=dim(dTrainAll)[1],size=1,prob=0.1)>0
  predRep <- mkPredC(dTrainAll[!useForCalRep,outcome],
    dTrainAll[!useForCalRep,var],
    dTrainAll[useForCalRep,var])
  calcAUC(predRep,dTrainAll[useForCalRep,outcome])
}
> aucs <- replicate(100,fCross())
```

What we have done is wrap our cross reference work into a function instead of in a for-based code block. Advantages are: the function can be re-used, run in parallel and is shorter (as it avoid needless details about result storage and result

indices). The function is then called 100 times using the `replicate()` method (`replicate()` is a convenience method from the powerful `sapply()` family).

Notice we must write `replicate(100, fCross())` *not* the more natural `replicate(100, fCross)`. This is because R is expecting an expression (a sequence that implies execution) as the second argument and not the mere name of a function. The notation can be a bit confusing and the reason it works is that function arguments in R are *not* evaluated prior to being passed in to a function, but instead are evaluated inside the function.⁵ This is called "promised based" argument evaluation and is very powerful (allows user defined macros, lazy evaluation, placement of variable names on plots, user defined control structures, and user defined exceptions). This can also be very complicated, so it is best to think of R as having mostly using what is called "call by value semantics"⁶ where arguments are passed to functions as values evaluated prior to entering the function and alterations of these values are not seen outside of the function.

Footnote 5 For just a taste of the complexity this introduces try to read:

<http://developer.r-project.org/nonstandard-eval.pdf>

Footnote 6 http://en.wikipedia.org/wiki/Evaluation_strategy#Call_by_value

6.3 Building Models Using Many Variables

Models that combine the effects of many variable tend to be much more powerful than models that use only a single variable. In this section we will learn how to build some of the most fundamental multiple variable models: decision trees, nearest neighbor and Naive Bayes.

6.3.1 Variable Selection

A key part of building many variable models is selecting what variables⁷ to use and how the variables are to be transformed or treated. We have already discussed variable treatment in Chapter XRF:chapter_4:chManagingData so we will only discuss variable selection here.

Footnote 7 We will call variables used to build the model variously "variables," "independent variables," "input variables," "causal variables" and so on to try and distinguish them from the item to be predicted (which we will call "outcome" or "dependent").

When variables are available has a huge impact on model utility. For instance a variable that is available near (or even after) the time of the outcome to be modeled has happened may make a very accurate model with little utility. The analyst has to watch out for variables that are functions of or "contaminated by" the item to be

predicted. Which variables will actually be available in production is something you will want to discuss with your project sponsor. And sometimes you may want to improve model utility (at a possible cost of accuracy) by removing variables from the project design. An acceptable prediction one day before an event can be much more useful than a more accurate prediction one hour before the event.

Each variable we use represents a chance of explaining more of the outcome variation (a chance at building a better model) but also represents a possible source of noise and over-fitting. To control this effect we often pre-select which subset of variables we will use to fit. Variable selection can be an important defensive modeling step even for types of models that "don't need it" (as seen with decision trees in section 6.3.2). Listing 6.3 shows a hand-rolled variable selection loop where each variable is scored according to an AIC (Akaike information criterion) inspired score where a variable is scored with a bonus proportional to the scaled log-likelihood of the training data minus a penalty proportional to the complexity of the variable (which in this case is 2^{entropy}). The score is a bit ad-hoc but tends to work well in selecting variables. Notice we are using performance on the calibration set (not the training set) to pick variables.

Listing 6.3 Basic variable selection

```
entropy <- function(x) {
  xpos <- x[x>0 & !is.na(x)]
  scaled <- xpos/sum(xpos)
  sum(-scaled*log(scaled,2))
}

logLikelihood <- function(outCol,predCol) {
  sum(ifelse(outCol==pos,log(predCol),log(1-predCol)))
}

selVars <- c()
minStep <- 5
baseRateTrain <- logLikelihood(dTrain[,outcome],
  sum(dTrain[,outcome]==pos)/length(dTrain[,outcome]))
baseRateCheck <- logLikelihood(dCal[,outcome],
  sum(dCal[,outcome]==pos)/length(dCal[,outcome]))
for(v in catVars) {
  pi <- paste('pred',v,sep=' ')
  liCheck <- 2*((logLikelihood(dCal[,outcome],dCal[,pi]) -
    baseRateCheck) -
    2^entropy(table(dCal[,pi],useNA='ifany')))
  if(liCheck>minStep) {
    print(sprintf("%s, calibrationScore: %g",
      pi,liCheck))
    selVars <- c(selVars,pi)
  }
}
for(v in numericVars) {
  pi <- paste('pred',v,sep=' ')
  liCheck <- 2*((logLikelihood(dCal[,outcome],dCal[,pi]) -
    baseRateCheck) - 1)
  if(liCheck>=minStep) {
    print(sprintf("%s, calibrationScore: %g",
      pi,liCheck))
    selVars <- c(selVars,pi)
  }
}
```

In our case this picks 22 of the 212 possible variables. We will show in section 6.3.2 the performance of a multiple variable model with and without using variable selection.

6.3.2 Using Decision Trees

Decision trees are a very simple model type: they make a prediction that is piecewise constant. This is interesting because the null hypothesis that we are trying to outperform is often a single constant for the whole data set, so we can view a decision tree as a procedure to split the training data into pieces and use a simple memorized constant on each piece. Decision trees (especially a type called classification and regression trees, or CART) can be used to quickly predict either categorical or numeric outcomes. The best way to think about decision trees is they are machine generated business rules.

FITTING A DECISION TREE MODEL

Building a decision tree involves proposing many possible *data cuts* and then choosing best cuts based on simultaneous competing criteria of predictive power, cross validation strength and interaction with other chosen cuts. One of the advantages of using a canned package for decision tree work is not having to worry about tree construction details. Let's start by building a decision tree model for churn. The simplest way to call `rpart()` is to just give it a list of variables and see what happens (`rpart()`, unlike many R modeling techniques, has built-in code for dealing with missing values).

```
> library('rpart')
> fV <- paste(outcome,'>0 ~ ',
  paste(c(catVars,numericVars),collapse=' + '),sep=' ')
> tmodel <- rpart(fV,data=dTrain)
> print(calcAUC(predict(tmodel,newdata=dTrain),dTrain[,outcome]))
[1] 0.9241265
> print(calcAUC(predict(tmodel,newdata=dTest),dTest[,outcome]))
[1] 0.5266172
> print(calcAUC(predict(tmodel,newdata=dCal),dCal[,outcome]))
[1] 0.5126917
```

What we get is pretty much a disaster. The model looks way too good to believe on the training data (which it has memorized to a non-useful degree) and not as good as our best single variable models on withheld calibration and test data. A couple of possible sources of the failure are we have categorical variables with very many levels and we have a lot more NAs/missing-data than `rpart()`'s

surrogate value strategy was designed for. What we can do to work around this is fit on our re-processed variables which hide the categorical levels (replacing them with numeric predictions) and remove NAs (treating them as just another level).

```
> tVars <- paste('pred',c(catVars,numericVars),sep=' ')
> fV2 <- paste(outcome,'>0 ~ ',paste(tVars,collapse=' + '),sep=' ')
> tmodel <- rpart(fV2,data=dTrain)
> print(calcAUC(predict(tmodel,newdata=dTrain),dTrain[,outcome]))
[1] 0.928669
> print(calcAUC(predict(tmodel,newdata=dTest),dTest[,outcome]))
[1] 0.5390648
> print(calcAUC(predict(tmodel,newdata=dCal),dCal[,outcome]))
[1] 0.5384152
```

This result is about the same (also bad). So our next suspicion is that the overfitting is because our model is too complicated. To control `rpart()` model complexity we need to monkey a bit with the controls. We pass in an extra argument called `rpart.control` (use `help('rpart')` for some details on this control) that changes the decision tree selection strategy as we show below.

```
> tmodel <- rpart(fV2,data=dTrain,
  control=rpart.control(cp=0.001,minsplit=1000,
    minbucket=1000,maxdepth=5)
)
> print(calcAUC(predict(tmodel,newdata=dTrain),dTrain[,outcome]))
[1] 0.9421195
> print(calcAUC(predict(tmodel,newdata=dTest),dTest[,outcome]))
[1] 0.5794633
> print(calcAUC(predict(tmodel,newdata=dCal),dCal[,outcome]))
[1] 0.547967
```

This is very small improvement. We can waste a lot of time trying variations of the `rpart()` controls. The best guess is that this data set is unsuitable for decision trees and a method that deals better with over-fitting issues is needed: such as random forests which we will demonstrate in Chapter XRF:chapter_9:chAdvancedMethods. The best result we could get for this data set using decision trees was from using our selected variables (instead of all transformed variables) as we show below.

```

f <- paste(outcome,'>0 ~ ',paste(selVars,collapse=' + '),sep=' ')
> tmodel <- rpart(f,data=dTrain,
+ control=rpart.control(cp=0.001,minsplit=1000,
+ minbucket=1000,maxdepth=5)
)
> print(calcAUC(predict(tmodel,newdata=dTrain),dTrain[,outcome]))
[1] 0.6856049
> print(calcAUC(predict(tmodel,newdata=dTest),dTest[,outcome]))
[1] 0.6753746
> print(calcAUC(predict(tmodel,newdata=dCal),dCal[,outcome]))
[1] 0.6755194

```

These AUCs are not great (they are not near 1.0 or even particularly near the winning team's 0.76), but they are significantly better than any of the AUCs we saw from single variable models when checked on non training data. So we have finally built a legitimate multiple variable model.

To tune `rpart` we suggest in addition to trying variable selection (which is an odd thing to combine with decision tree methods) following the `rpart` documentation in trying different settings of the `method` argument. However, we quickly get better results with KNN and logistic regression; so it doesn't make sense to spend too long trying to tune decision trees for this particular data set.

HOW DECISION TREE MODELS WORK

At this point we can look at the model and use it to explain how decision tree models work.

Listing 6.4 Printing the decision tree

```
> print(tmodel)
n= 40518

node), split, n, deviance, yval
  * denotes terminal node

1) root 40518 2769.3550 0.07379436
  2) predVar126< 0.07366888 18188 726.4097 0.04167583 *
    4) predVar126< 0.04391312 8804 189.7251 0.02203544 *
      5) predVar126>=0.04391312 9384 530.1023 0.06010230
        10) predVar189< 0.08449448 8317 410.4571 0.05206204 *
          11) predVar189>=0.08449448 1067 114.9166 0.12277410 *
    3) predVar126>=0.07366888 22330 2008.9000 0.09995522
      6) predVar73< 0.05554501 5872 287.3650 0.05160082 *
      7) predVar73>=0.05554501 16458 1702.9070 0.11720740
        14) predVar205< 0.1047779 14771 1429.0370 0.10852350
          28) predVar218< 0.07134103 7919 643.0582 0.08915267
            56) predVar126< 0.1015407 4433 287.4613 0.06970449 *
              57) predVar126>=0.1015407 3486 351.7880 0.11388410 *
      29) predVar218>=0.07134103 6852 779.5731 0.13091070
        58) predVar74< 0.0797246 2785 234.9135 0.09299820 *
          59) predVar74>=0.0797246 4067 537.9154 0.15687240 *
    15) predVar205>=0.1047779 1687 263.0030 0.19324240 *
```

Each row in listing 6.4 that starts with a "#)" is called a node of the decision tree. This decision tree has 15 nodes. Node 1 is always called the root. Each node other than the root node has a parent and the parent of node k is node $\text{floor}(k/2)$. The indentation also indicates how deep in the tree a node is. Each node other than the root is named by what condition must be true to move from the parent to the node. You move from node 1 to node 2 if $\text{predVar126} < -0.002810871$ (and otherwise you move to node 3 which has the complementary condition). So to score a row of data we navigate from the root of the decision tree by the node conditions until we reach a node with no children, which is called a leaf node. Leaf node are marked with stars. The remaining three numbers reported for each node are: the number of training items that navigated to the node, the deviance of the set of training items that navigated to the node (a measure of how remains uncertain at this level of the tree) and the fraction of items that were in the positive class at the node (which is the prediction for leaf nodes).

6.3.3 Using Nearest Neighbor Methods

A k-nearest neighbor method (abbreviated as KNN) scores an example by finding the k training examples nearest to the example and then taking the average of their outcomes as the score.

One issue of KNN is its concept space. For example if we were to run 3-nearest neighbor on our data we have to understand that with three neighbors from the training data we always see either zero, one, two, or three examples of churn. So the estimated probability of churn is always going to be one of: 0%, 33%, 66%, or 100%. So the KNN model can only work well on an event as rare as churn (which has a rate of around 7% in our training data) if it is lucky enough to so thoroughly separate the positive and negative examples so that it is always (correctly) predicting 0% or 100% rates. For predicting event rates that don't have probabilities near 1/2 we suggest using a large k so KNN can write down enough different probabilities to be useful. For direct classification you may use small k, but if you are not able to precisely predict the actual individuals who churn and instead are settling on modeling variations in churn rate (as we are here) you must use a large k. We show a KNN run with k=200 in listing 6.5.

Listing 6.5 Running k nearest neighbors

```
> library('class')
> nK <- 200
> knnTrain <- dTrain[,selVars]
> knnC1 <- dTrain[,outcome]==pos
> knnPred <- function(df) {
+   knnDecision <- knn(knnTrain,df,knnC1,k=nK,prob=T)
+   ifelse(knnDecision==TRUE,
+         attributes(knnDecision)$prob,
+         1-(attributes(knnDecision)$prob))
+ }
> print(calcAUC(knnPred(dTrain[,selVars]),dTrain[,outcome]))
[1] 0.7418755
> print(calcAUC(knnPred(dCal[,selVars]),dCal[,outcome]))
[1] 0.7205581
> print(calcAUC(knnPred(dTest[,selVars]),dTest[,outcome]))
[1] 0.7090571
```

This is our best result yet. Notice how the double density performance plot in figure 6.2 is much better looking than figure 6.1. What we are looking for are for the two distributions to be unimodal⁸, and if not separated at least not completely

on top of each other.

Footnote 8 Distributions that are multi-modal are often evidence that there are significant effects we have not yet explained. Distributions that are unimodal or even look normal are consistent with the unexplained effects being simple noise.

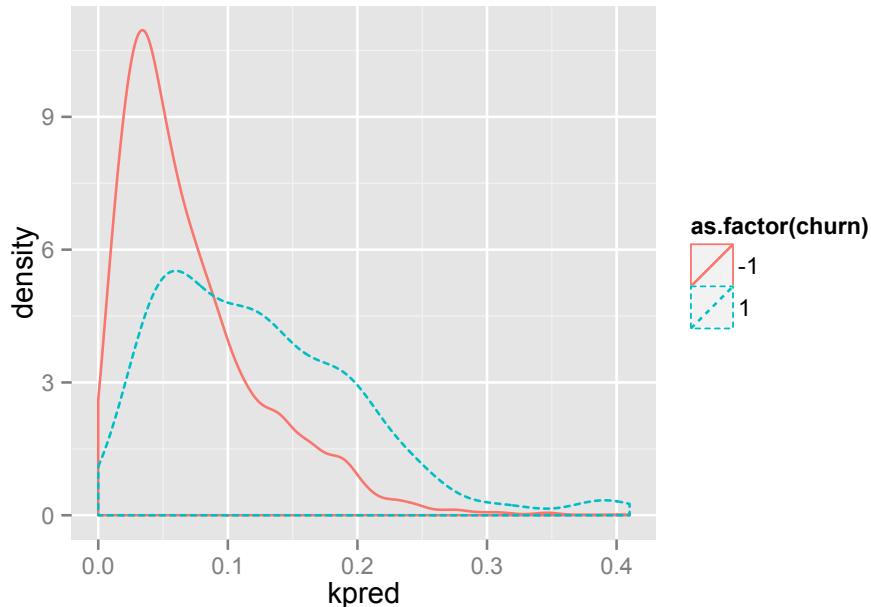


Figure 6.2 Performance of 200-nearest neighbors on calibration data

The code to produce figure 6.2 is given below:

```
dCal$kpred <- knnPred(dCal[,selvars])
ggplot(data=dCal) +
  geom_density(aes(x=kpred,
    color=as.factor(churn),linetype=as.factor(churn)))
```

This finally gives us a result good enough to bother plotting the ROC curve for. The code below produces figure 6.3

```
plotROC <- function(predcol,outcol) {
  perf <- performance(prediction(predcol,outcol==pos),'tpr','fpr')
  pf <- data.frame(
    FalsePositiveRate=perf@x.values[[1]],
    TruePositiveRate=perf@y.values[[1]])
  ggplot() +
    geom_line(data=pf,aes(x=FalsePositiveRate,y=TruePositiveRate)) +
    geom_line(aes(x=c(0,1),y=c(0,1)))
```

```

}
print(plotROC(knnPredP(dTest[,selvars]),dTest[,outcome]))

```

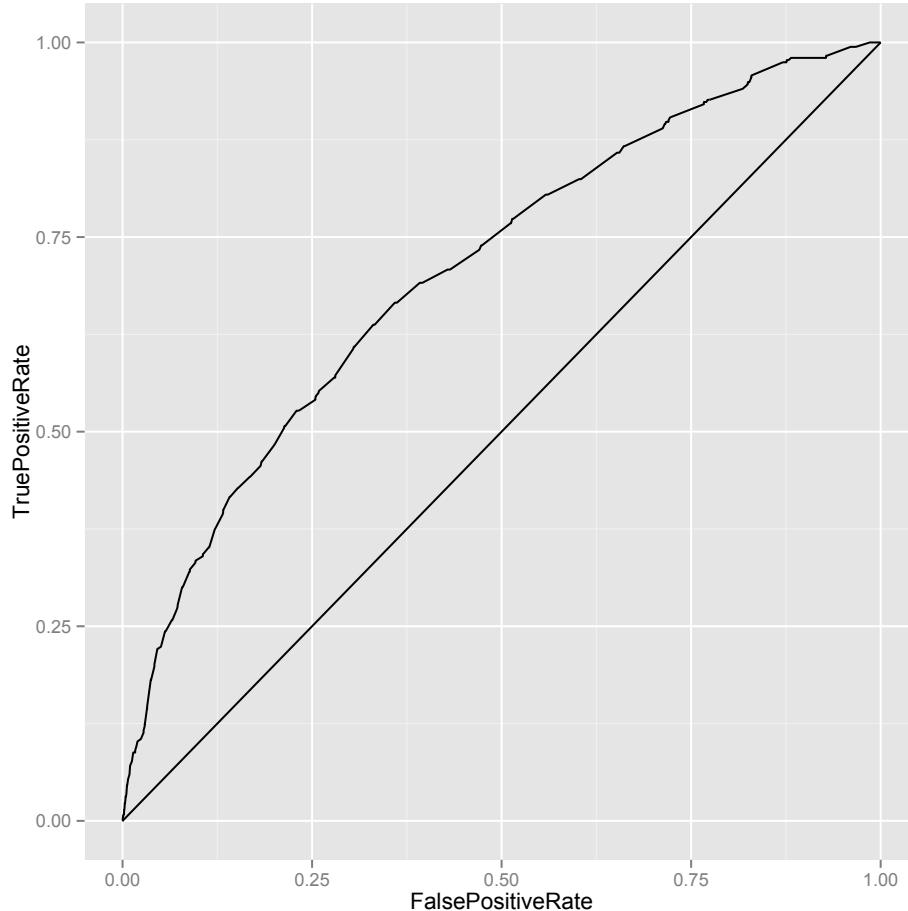


Figure 6.3 ROC of 200-nearest neighbors on calibration data

KNN is expensive both in time and space. Sometimes we can get similar results with more efficient methods such as logistic regression (which we will explain in Chapter XRF:chapter_7:chFunctionalModels). But as a quick example, logistic regression with our selected variables gives slightly better performance as we see below:

```

> gmodel <- glm(as.formula(f),data=dTrain,family=binomial(link='logit'))
> print(calcAUC(predict(gmodel,newdata=dTrain),dTrain[,outcome]))
[1] 0.7309537
> print(calcAUC(predict(gmodel,newdata=dTest),dTest[,outcome]))
[1] 0.7234645

```

```
> print(calcAUC(predict(gmodel,newdata=dCal),dCal[,outcome]))  
[1] 0.7170824
```

6.3.4 Using Naive Bayes

Naive Bayes is an interesting method that memorizes how each training variable is related to outcome, and then makes predictions by multiplying together the effects of each variable. Let's start with a brief description of the Naive Bayes method were we suppose we are trying to predict if somebody is employed based on their level of education, their geographic region, and other variables.

Let's call a specific variable `x_1` taking on a specific value `x_1` a piece of *evidence*, `ev_1`. For example, `education=="High School"` could be a piece of evidence. Let's call the outcome `y`. Then the fraction of all positive examples where `ev_1` is true is an approximation to the *conditional probability* of `ev_1`, given `y==T`. This is usually written as $P(ev_1 | y==T)$. Going back to our employment example, the conditional probability of having a high school education, given that you are employed, would be estimated by the fraction of all data subjects who are employed and have a high school education, over the total number of employed subjects.

What you are really interested in is the reverse: the conditional probability of a subject being employed, given that they have a high school education: $P(y==T | ev_1)$. This is easy to estimate for a single variable, but it is harder to estimate when you consider all the input variables and their values.

Let's call the input variables and their values the evidence that describes a data point: `c(ev1, ev2, ... evN)`. The Naive Bayes classifier maps evidence to a classification. This is shown in Figure 6.4 for the binary case.

1. Bayes' law tells us:

$$P(y == T|ev_1) = \frac{P(ev_1|y == T) \cdot P(y == T)}{P(ev_1)}$$

$$P(y == F|ev_1) = \frac{P(ev_1|y == F) \cdot P(y == F)}{P(ev_1)}$$

The left hand side is what you want; the right hand side can be estimated from the statistics of the training data.

2. The *Naive Bayes Assumption* lets us assume that all the evidence is conditionally independent of each other for a given outcome.

$$P(ev_1 \& ev_2 \dots \& ev_N | y == T) \approx P(ev_1|y == T) \cdot P(ev_2|y == T) \cdot \dots \cdot P(ev_N|y == T)$$

$$P(ev_1 \& ev_2 \dots \& ev_N | y == F) \approx P(ev_1|y == F) \cdot P(ev_2|y == F) \cdot \dots \cdot P(ev_N|y == F)$$

3. This gives us:

$$P(y == T|ev_1 \& \dots \& ev_N) \approx \frac{(P(ev_1|y == T) \cdot \dots \cdot P(ev_N|y == T)) \cdot P(y == T)}{P(ev_1 \& \dots \& ev_N)}$$

$$P(y == F|ev_1 \& \dots \& ev_N) \approx \frac{(P(ev_1|y == F) \cdot \dots \cdot P(ev_N|y == F)) \cdot P(y == F)}{P(ev_1 \& \dots \& ev_N)}$$

Figure 6.4 The Mathematics of Naive Bayes Classification

The numerator terms of the right hand sides of the final expressions can be calculated efficiently from the training data, while the left hand side can't. Notice that the denominators of the right hand sides are the same (so they can be canceled out, as the two numerators must sum to one). For a given observation of evidence `c(ev1, ev2 ... evN)`, you can then estimate the conditional probabilities of `y==T` and `y==F`, and assign the more probable outcome.

For numerical reasons, it's better to convert the products into sums, by taking the log of both sides. Since the denominator term is the same in both expressions, we can ignore it, since we only want to determine which expression is greater.

```
log(P(y==T|ev1...evN)) ~ log(P(ev1|y==T)) + ... + log(P(evN|y==T))
log(P(y==F|ev1...evN)) ~ log(P(ev1|y==F)) + ... + log(P(evN|y==F))
```

It's also a good idea to add a smoothing term, so that you are never taking the log of zero.

Naive Bayes is simple enough we can implement it in terms of our single variable models. We show one such implementation in listing 6.6.

Listing 6.6 Building, applying and evaluating a Naive Bayes model

```
nPos <- sum(dTrain[,outcome]==pos)
pPos <- nPos/length(dTrain[,outcome])  
①
nBayes <- function(pf) {
  eps <- 1.0e-5
  lpc <- rowSums(log(pf/pPos + eps)) ②
  lpo <- rowSums(log((1-pf)/(1-pPos) + eps)) ③
  m <- pmax(lpc,lpo)
  pc <- pPos*exp(lpc - m)
  po <- (1-pPos)*exp(lpo - m) ④
  pc/(pc+po) ⑤
}  

pVars <- paste('pred',c(numericVars,catVars),sep=' ')
dTrain$nbpred1 <- nBayes(dTrain[,pVars])
dCal$nbpred1 <- nBayes(dCal[,pVars])
dTest$nbpred1 <- nBayes(dTest[,pVars]) ⑥
print(calcAUC(dTrain$nbpred1,dTrain[,outcome]))
[1] 0.9757348
print(calcAUC(dCal$nbpred1,dCal[,outcome]))
[1] 0.5995206
print(calcAUC(dTest$nbpred1,dTest[,outcome])) ⑦
[1] 0.5956515
```

- ① Define a function that performs the Naive Bayes prediction
- ② For each row: compute (with a smoothing term) the sum of $\log(P[\text{positive} \& \text{evidence}_i]/P[\text{positive}])$ across all columns. This is equivalent to the log of the product of $P[\text{evidence}_i | \text{positive}]$ up to terms that don't depend on the positive/negative outcome.
- ③ For each row: compute (with a smoothing term) the sum of $\log(P[\text{negative} \& \text{evidence}_i]/P[\text{negative}])$ across all columns. This is equivalent to the log of the product of $P[\text{evidence}_i | \text{negative}]$ up to terms that don't depend on the positive/negative outcome.
- ④ Shift sums down and exponentiate to compute the $Z^*P[\text{positive}]*\text{product}_i P[\text{evidence}_i | \text{positive}]$ and $Z^*P[\text{negative}]*\text{product}_i P[\text{evidence}_i | \text{negative}]$ without numeric overflows where Z is some unknown constant of proportionality.
- ⑤ Use the fact that the predicted positive probability plus the predicted negative probability should sum to one to find and eliminate Z . Return the correctly scaled predicted odds of being positive as our forecast.
- ⑥ Apply the function to make the predictions.
- ⑦ Calculate the AUCs. Notice the overfitting- fantastic performance on the training set that is not repeated on the calibration or test sets.

Intuitively, what we have done is built a new column from each of our single

variable models. This column is the logarithm of the ratio of the single variable model's predicted churn rate over the overall churn rate. When the model predicts a rate near the overall churn rate this ratio is near 1 and therefore the logarithm is near zero. Similarly, for high predicted churn rates the column column is a positive number and for low predicted churn rates the column is negative.

Summing these signed columns is akin to taking a net-consensus vote across all of the columns variables. If all the evidence is conditionally independent given the outcome (this is the Naive Bayes Assumption -- and remember it is only an assumption), then this is exactly the right thing to do. The amazing thing about the Naive Bayes classifier is that it can perform quite well even when the conditional independence assumption is not true.

SIDE BAR Smoothing

The most important design parameter in Naive Bayes is how *smoothing* is handled. The idea of smoothing is attempt to obey *Cromwell's rule* that no probability estimate of zero should ever be used in probabilistic reasoning. This is because if you are combining probabilities by multiplication (the most common method of combining probability estimates) then once some term is zero the entire estimate will be zero *no matter what the values of the other terms are*. The most common form of smoothing is called *Laplace smoothing* which counts k successes out of n trials as a success ratio of $(k+1)/(n+1)$ and not as a ratio of k/n (defending against the $k=0$ case). There are, however, many other important smoothing strategies.

There are many write-ups of Bayes Law and Naive Bayes methods that cover the math in much more detail. The main things to remember is Naive Bayes doesn't perform any clever optimization, so it is often inferior to methods like logistic regression and support vector machines. Also variable selection is very important for Naive Bayes. Naive Bayes is most useful when you have a very large number of features that are rare and/or nearly independent.

SIDE BAR**Document Classification and Naive Bayes**

Naive Bayes is the workhorse method when classifying text documents (such as email spam detectors). This is because the standard model for text documents (usually called *bag of words* or *bag of k-grams*) can have an extreme number of possible features. In the bag of k-grams model we pick a small k (typically 2) and each possible consecutive sequence of k words is a possible feature. Each document is represented as "bag" which is a sparse vector indicating which k-grams are in the document. The number of possible features runs into the millions but each document only has non-zero value on a number of features proportional to k time the size of the document.

Of course we can call also a pre-packaged Naive Bayes implementation (that includes its own variable treatments) as show in listing 6.7

Listing 6.7 Using a Naive Bayes package

```
> library('e1071')
> ff <- paste('as.factor(',outcome,'>0) ~ ',
+   paste(lVars,collapse=' + '),sep='')
> nbmodel <- naiveBayes(as.formula(ff),data=dTrain)
> dTrain$nbpred <- predict(nbmodel,newdata=dTrain,type='raw')[,'TRUE']
> dCal$nbpred <- predict(nbmodel,newdata=dCal,type='raw')[,'TRUE']
> dTest$nbpred <- predict(nbmodel,newdata=dTest,type='raw')[,'TRUE']
> calcAUC(dTrain$nbpred,dTrain[,outcome])
[1] 0.8200033
> calcAUC(dCal$nbpred,dCal[,outcome])
[1] 0.5484643
> calcAUC(dTest$nbpred,dTest[,outcome])
[1] 0.5462362
```

The `e1071 naiveBayes()` is performing a bit worse, likely this is due to it having to re-process our supplied numeric inputs (using either density, quantization or parametric estimates) as it has no way of knowing the variable we gave it are supposed to already represent ready to go Bayes terms. We could call it again with something closer to the input variables, but this library is not designed for the density of missing values in this data set. So we would have to deal with missing values by hand much as we have during data treatment (and by that time you can implement Naive Bayes as simple summation).

6.4 Summary

The single variable and multiple variable memorization style models in this section are always worth trying first. This is especially true if most of your variables are categorical variables, as memorization is a good idea in this case. The techniques of this chapter are also a good repeat example of variable treatment and variable selection.

We have, at a bit of a stretch, called all of the modeling techniques of this chapter "memorization methods." The reason for this is having worked an example using all of these models all in the same place we now have enough experience to see the common memorization traits in these models: their predictions are all sums of summaries of the original training data.

- Single variable models can be thought of as being simple memorizations or summaries of the training data. This is especially true for categorical variables where the model is essentially a contingency table or pivot table where for every level of the variable we record the distribution of training outcomes (see section 6.2.1). Some sophisticated ideas (like smoothing, regularization or shrinkage) may be required to avoid overfitting and build good single variable models. But in the end single variable models essentially organize the training data into a number of subsets indexed by the predictive variable and then store a summary of the distribution of outcome as their future prediction. These models are atoms or sub-assemblies that we sum in different ways to get the rest of the models of this chapter.
- k-Nearest Neighbor predictions are based on summaries of the k pieces of training data that are closest to the example to be scored. kNN models usually store all of their original training data instead of an efficient summary. So they truly do memorize the training data.
- Decision tree model decisions are also sums of summaries over subsets of the training data. For each scoring example the model makes a prediction by choosing the summary of all training data that was placed in the same leaf node of the decision tree as the current example to be scored. There is some cleverness in the construction of the decision tree itself, but once we have the tree it is enough to store a single data summary per tree leaf.
- Naive Bayes models partially memorize training data through intermediate features. Roughly speaking Naive Bayes models form their decision by building a large collection of unrelated single variable models.⁹ The Naive Bayes prediction for a given example is just the product of all the applicable single variable model adjustments (or isomorphically: the sum of logarithms of the single variable contributions). Notice Naive Bayes models are constructed without any truly clever functional forms or optimization steps. This is why we stretch terms a bit and call them memorization: their predictions are just sums of appropriate summaries of the original training data.

Footnote 9 As we saw in section 6.3.4 these are slightly modified single variable models as they model feature driven change in outcome distribution or in Bayesian terms "have the priors pulled out."

For all their fascinating features, at some point you will have needs that push you away from memorization methods. For some problems you want models that capture more of the functional or additive structure of relationships. In particular you will want to try regression for value prediction and logistic regression for category prediction as we demonstrate in Chapter XRF:chapter_7:chFunctionalModels.

6.5 External links section

Chapter 7: Using Linear and Logistic Regression

This chapter covers

- Using linear regression to predict quantities and find relations
- Interpreting the diagnostics from R's linear regression call `lm()`
- Using logistic regression to predict probabilities or categories, and find relations
- Interpreting the diagnostics from R's logistic regression call `glm()`

In the last chapter, we talked about using memorization methods for classification. In this chapter, we will talk about a different class of methods that are used for scoring and for classification. This class of methods is especially useful when you don't just want to predict an outcome, but you also want to know the relationship between the input variables and the outcome. Knowing such relationships is useful because it can often be used as *advice* on how to get the outcome that you want.

In this chapter we will show how to use linear regression to predict customer income and logistic regression to predict the probability that a newborn baby will need extra medical attention. We also walk through the diagnostics that R produces when you fit a linear or logistic model.

7.1 Using Linear Regression

Linear regression is the bread and butter prediction method for statisticians and data scientists. If you are trying to predict a numerical quantity like profit, cost, or sales volume, you should always try linear regression first. If it works you are done; if it fails the detailed diagnostics produced give you a good clue as to what methods you should try next.

In this section, we will use a real example (predicting personal income) to work

through all of the steps of producing and using a linear regression model.

Before we get to the main example, let's take a quick overview of the method.

7.1.1 Understanding linear regression

Suppose you are trying to predict how many pounds a person will lose on their diet and exercise plan in a month. You will base that prediction on other facts about that person, like their average daily caloric intake over that month, and how many hours a day they exercise. In other words, for every person i , you want to predict `pounds.lost[i]` based on `daily.cals[i]` and `daily.exercise[i]`. Linear regression assumes that `pounds.lost[i]` is a linear combination of `daily.cals[i]` and `daily.exercise[i]`:

```
pounds.lost[i] = b.cals * daily.cals[i] + b.exercise * daily.exercise[i]
```

The goal is to find the values of `b.cals` and `b.exercise` so that the linear combination of `daily.cals[i]` and `daily.exercise[i]` comes very close to `pounds.lost[i]` for all persons i in the training data.

Let's put this in more general terms. Suppose that $y[i]$ is the numeric quantity you want to predict, and $x[i,]$ is a row of inputs that corresponds to output $y[i]$. Linear regression finds a function $f(x)$ such that:

```
f(x[i,]) = b[1] x[i,1] + b[2] x[i,2] + ... b[n] x[i,n] .
```

We want numbers $b[1], \dots, b[n]$ (called the *coefficients* or *betas*) such that $f(x[i,])$ is as near as possible to $y[i]$ for all $(x[i,], y[i])$ pairs in our training data. R supplies a one-line command to find these coefficients: `lm()`.

In the idealized theoretic situation linear regression is used to fit $y[i]$ *only* when the $y[i]$ are themselves given by:

```
y[i] = b[1] x[i,1] + b[2] x[i,2] + ... b[n] x[i,n] + e[i]
```

In particular, this means that y is linear in the values of x : a change in the value of $x[i, m]$ by one unit (while holding all the other $x[i, n]$ constant) will change the value of $y[i]$ by the amount $b[m]$ always, no matter what the starting

value of $x[i, m]$ was. This is easier to see in one dimension. If $y = 3 + 2*x$, then if we increase x by 1, then y will always increase by 2, no matter what the starting value of x is. This would not be true for, say $y = 3 + 2*(x^2)$.

The last term $e[i]$ represents what are called *unsystematic errors*, or noise. Unsystematic errors average to zero (so they don't represent a net upward or net downward bias) and are uncorrelated with the $y[i]$.

Under these assumptions, linear regression is absolutely relentless in finding the best coefficients. If there is some advantageous combination or cancellation of features it will find it. One thing that linear regression does not do is reshape variables to be linear. Oddly enough linear regression often does an excellent job, even with misshaped variables.

NOTE**Thinking about linear regression**

When working with linear regression you will flip between "adding is too simple to work" and "how is it even possible to estimate the coefficients?" This is natural and comes from the fact that the method is in fact both simple and powerful. Our friend Philip Apps sums it up with: "you have to get up pretty early in the morning to beat linear regression."

As a toy example consider trying to fit the squares of the first 10 integers using only a linear function plus the constant 1. We are asking for coefficients $b[0]$ and $b[1]$ such that:

```
x[i]^2 nearly equals b[0] + b[1] x[i] .
```

This is clearly not a fair thing to ask, but linear regression still does a great job. It picks the following fit:

```
x[i]^2 nearly equals -22 + 11 x[i] .
```

As the figure 7.1 shows, this is a good fit in the region of values we trained on.

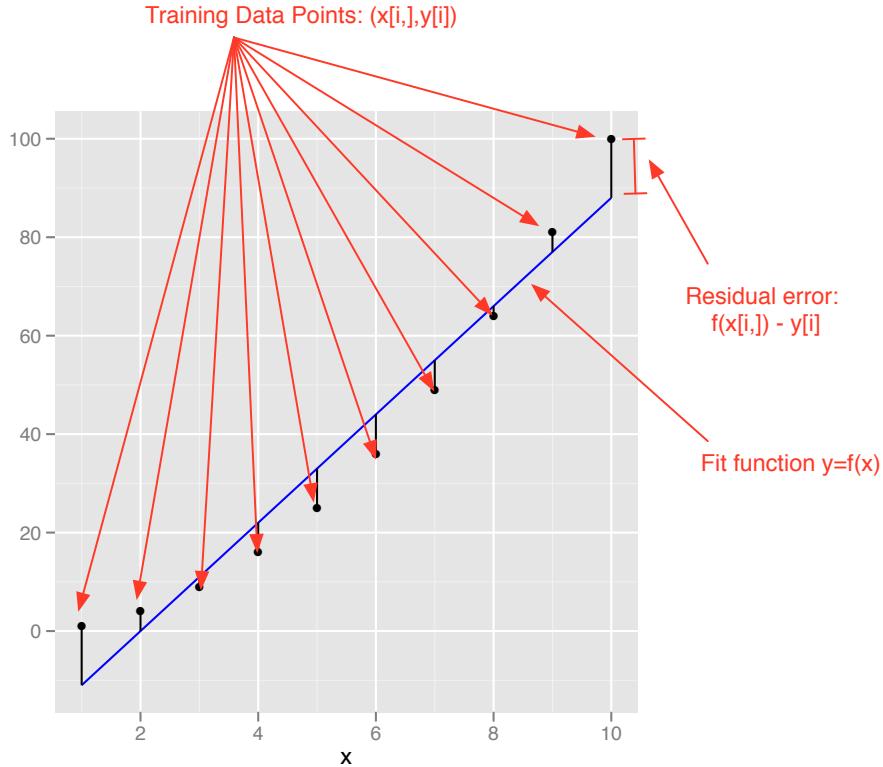


Figure 7.1 Fit versus actuals for $y=x^2$

The example in figure 7.1 is typical of how linear regression is "used in the field." That is, we are using a linear model to predict something that is itself not actually linear. Be aware that this is a bit of a sin: in particular, notice that the errors between the model's predictions and the true y are not unsystematic: the model underpredicts for specific ranges of x and overpredicts for others.

Next we will work through an example of how to apply linear regression on more interesting real data. Our actual example task will be to predict personal income from other demographic variables such as age and education from 2011 US Census PUMS data. In section XRF:sect2_2.2.3:sectLoadPUMS we prepared a small sample of PUMS data which we use here. As a reminder the data preparation steps included

- Restricting the data to full-time employees between 20 and 50 years of age, with an income between \$1000 and \$250,000.
- Dividing the data into a training set, `dtrain`, and a test set, `dtest`.

We can continue the example by loading `psub.RData`¹ and performing the following steps (which we will explain shortly):

Footnote 1 Copy from <https://github.com/WinVector/zmPDSwR/raw/master/PUMS/psub.RData>.

```
load("psub.RData")
dtrain <- subset(psub,ORIGINRANDBLOCK >= 500)
dtest <- subset(psub,ORIGINRANDBLOCK < 500)
model <- lm(log(PINCP,base=10) ~ AGEP + SEX + COW + SCHL,data=dtrain)
dtest$predLogPINCP <- predict(model,newdata=dtest)
dtrain$predLogPINCP <- predict(model,newdata=dtrain)
```

Each row of PUMS data represents a single anonymized person or household. Personal data recorded includes occupation, level of education, personal income and many other demographics variables.

For the analysis in this section, we will consider the input variables sex (SEX), class of worker (COW), and level of education (SCHL). The output variable is personal income (PINCP). We will also set the "reference level," or "default" sex to Male; the reference level of class of worker to "Employee of a private for-profit," and the reference level of education level to "no high school diploma." We will discuss reference levels below.

In addition to predicting income, you also have a secondary goal: to determine the effect of a bachelor's degree on income, relative to having no degree at all (the reference level "no high school diploma").

Now on to the model-building

7.1.2 Building a linear regression model

The first step in either prediction or finding relations or advice is to build the linear regression model.

The command to build the linear regression model in R is called `lm()`. The most important argument to `lm()` is a formula with "`~`" used in place of equals. The formula specifies what column of the data frame is the quantity to be predicted and what columns are to be used to make the predictions. Statisticians call the quantity to be predicted "the dependent variable" and the columns used to make "the independent variables." We find it is easier to call the quantity to be predicted "the y" and the variables used to make the predictions "the x's." Our formula is: "`log(PINCP,base=10) ~ AGEP + SEX + COW + SCHL`" which is read "predict the log base 10 of income as a function of age, sex, employment class and education."² The overall command is demonstrated in figure 7.2.

Footnote 2 Recall from the discussion of the lognormal distribution in Section XRF:sect2_4.1.2:sec_datatransformations that it is often useful to log-transform monetary quantities.

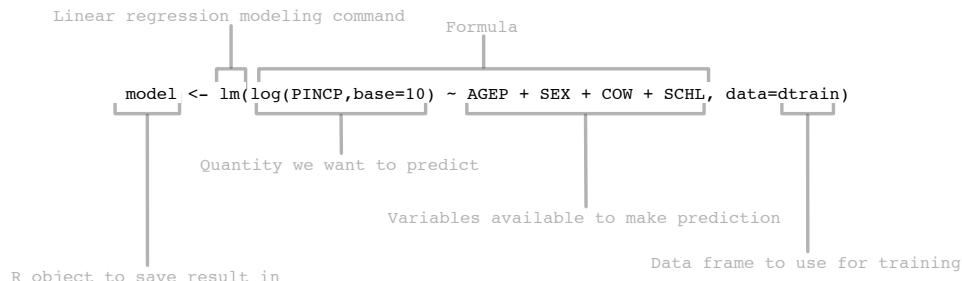


Figure 7.2 Building a linear model using the `lm()` command

The command in figure 7.2 builds the linear regression model and stores the results in the new object called "model." This model is both able to make predictions and extract important advice from the data.

WARNING

R stores training data in the model

R holds a copy of the training data in its model to supply the residual information seen in `summary(model)`. This holding a copy of the data is not strictly necessary and can needlessly run you out of memory. You can mitigate this problem somewhat by setting the parameters `model = F, x = F, y = F, qr = F` in the `lm()` call. If you are running low on memory (or swapping) you can dispose of R objects like `model` using the `rm()` command. In this case you would dispose of the `model` by running `rm("model")`.

7.1.3 Making predictions

Once you've called `lm()` to build the model, your first goal is to predict income. This is very easy to do in R. To predict you pass data into the `predict()` method. We demonstrate this using both the test and training data frames `dtest` and `dtrain` in Figure 7.3

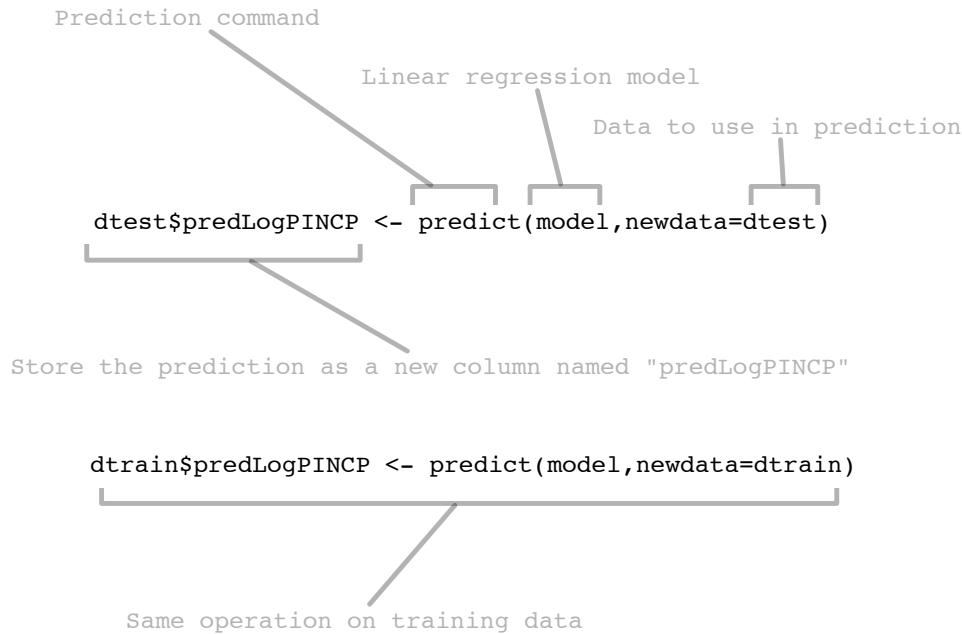


Figure 7.3 Making predictions with a linear regression model

The data frame columns `dtest$predLogPINCP` and `dtrain$predLogPINCP` now stores the predictions for the test and training sets, respectively. We have now both produced and applied a linear regression model.

CHARACTERIZING PREDICTION QUALITY

Before sharing predictions, you want to inspect both the predictions and model for quality.

We recommend plotting the actual `y` you are trying to predict as if it were a function of your prediction. In this case, plot `log(PINCP, base=10)` as if it were a function of `predLogPINCP`. If the predictions are very good then the plot will be dots arranged near the line `y=x`, which we call *the line of perfect prediction* (the phrase is not standard terminology; we use it to make talking about the graph easier). The commands to produce this for figure 7.4 are given below.

```
ggplot(data=dtest, aes(x=predLogPINCP, y=log(PINCP, base=10))) +
  geom_point(alpha=0.2, color="black") +
  geom_smooth(aes(x=predLogPINCP,
                  y=log(PINCP, base=10)), color="black") +
  geom_line(aes(x=log(PINCP, base=10),
                y=log(PINCP, base=10)), color="blue", linetype=2) +
  scale_x_continuous(limits=c(4,5)) +
  scale_y_continuous(limits=c(3.5,5.5))
```

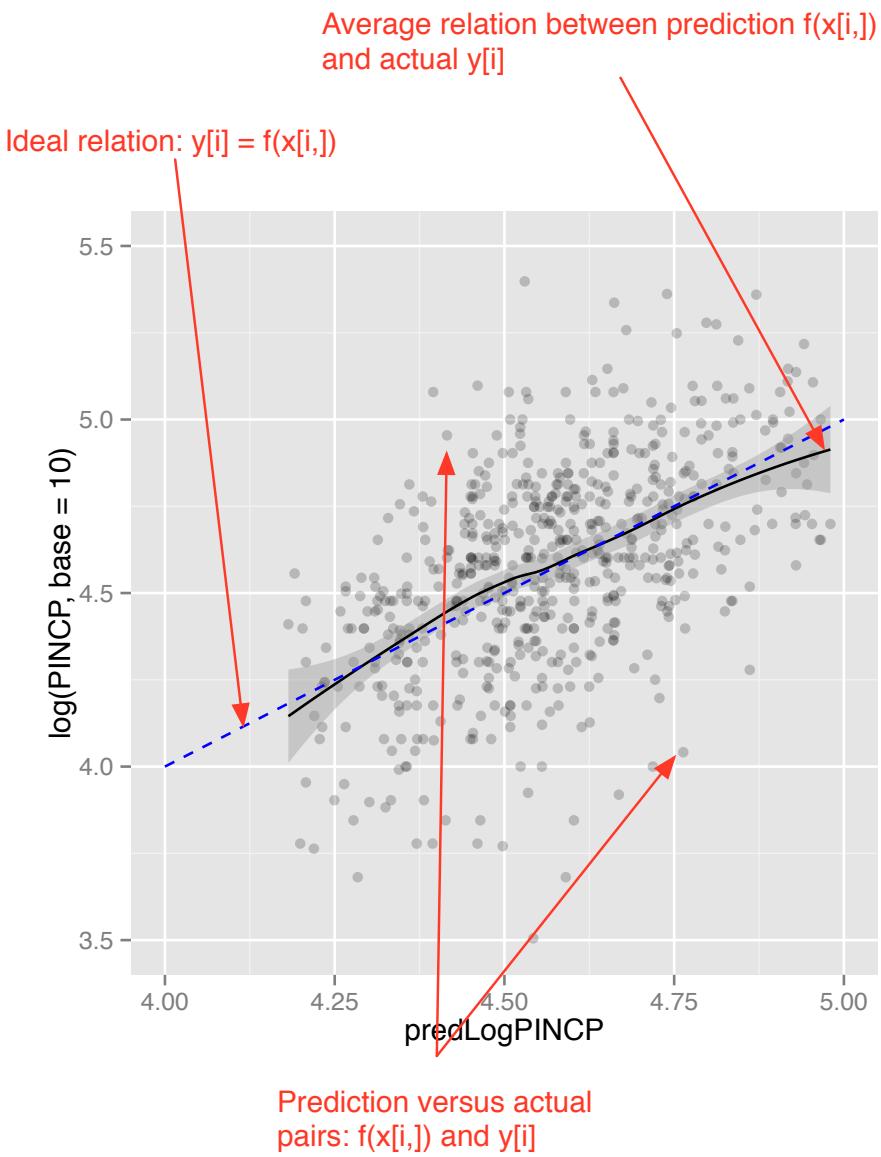
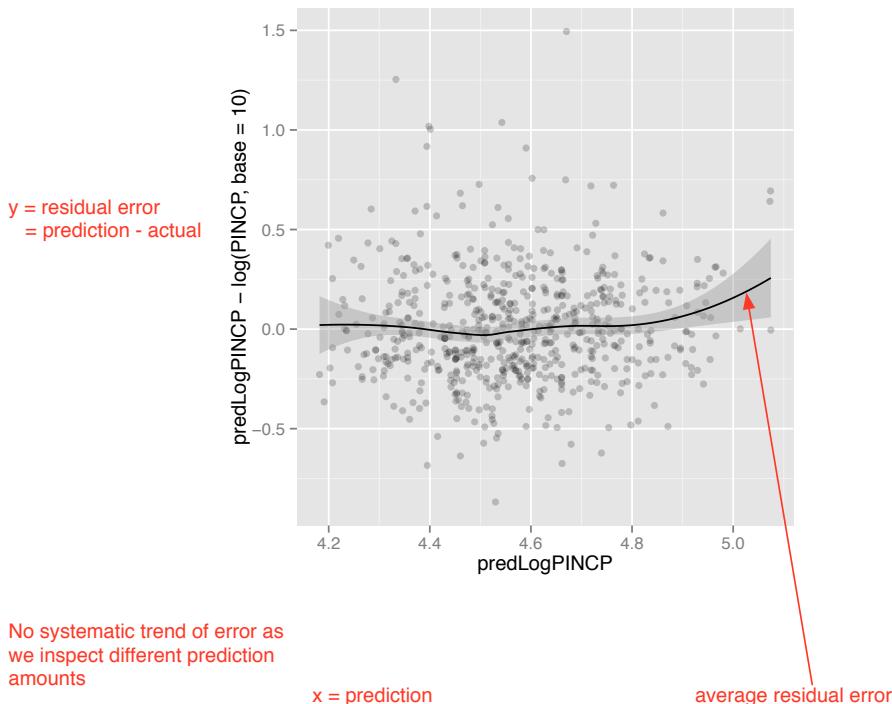


Figure 7.4 Plot of actual log income as a function of predicted log income

Statisticians prefer the residual plot shown in figure 7.5 where the predictions errors $\text{predLogPINCP}-\log(\text{PINCP, base}=10)$ are plotted as a function of predLogPINCP . In this case, the line of perfect prediction is the line $y=0$. The residual plot in figure 7.5 is prepared with the following R commands:

```
ggplot(data=dtest,aes(x=predLogPINCP,
                      y=predLogPINCP-log(PINCP,base=10))) +
  geom_point(alpha=0.2,color="black") +
  geom_smooth(aes(x=predLogPINCP,
```

```
y=predLogPINCP-log(PINCP,base=10)),
color="black")
```



SIDE BAR Why are the predictions on the x-axis, and not the true values?

The two graphs answer different questions. Statisticians tend to prefer the graph as given above, with predictions on the x-axis. One way to think about this is that, during deployment, you won't know the true outcome; only the prediction. The residual graph then gives you a sense of when the model may be over or under-predicting, based on the model's output.

A residual graph (or a fitted-versus-true graph) with the true outcome on the x-axis instead gives you a sense of where the model under- or over-predicts based on the actual outcome. This information is less helpful in deployment (since you don't know the actual outcome), but can be helpful when you are in the model development stage: you may need more or different input variables to predict well in different situations.

When you look at the true-versus-fitted or residual graphs, you are looking for some specific things:

On average, are the predictions correct?

In other words, does the smoothing curve lie more or less along the line of perfect prediction? Ideally, of course, the points will all lie very close to that line, but you may instead get a wider cloud of points (as we do in Figures 7.4 and 7.5) if your input variables don't explain the output too closely. But if the smoothing curve lies along the line of perfect prediction, then the model predicts correctly on average: it under-predicts about as much as it over-predicts.

Are there systematic errors?

If the smoothing curve veers off the line of perfect prediction too much, this is a sign of systematic under- or over-prediction in certain ranges: the error is correlated with $y[i]$. Many of the theoretical claims about linear regression depend on the observation error being uncorrelated with $y[i]$. Unstructured observation errors (the good case) are called *homoscedastic* and structured observation errors are called *heteroscedastic* and introduce prediction bias. For example, the toy fit in section 7.1 is heteroscedastic and is unsafe to use for values outside of its training range.

In addition to inspecting graphs you should produce quantitative summaries of the quality of the predictions and the residuals. One standard measure of quality of a prediction is called R^2 (pronounced "R squared"). You can compute the R^2 between the prediction and the actual y with the following R commands:

```
rsq <- function(y,f) { 1 - sum((y-f)^2)/sum((y-mean(y))^2) }
rsq(log(dtrain$PINCP,base=10),predict(model,newdata=dtrain))
rsq(log(dtest$PINCP,base=10),predict(model,newdata=dtest))
```

You want R^2 fairly large (1.0 is the largest we can achieve) and R^2 's that are similar on test and training. A significantly lower R^2 on test data is a symptom of an over fit model that looks good in training and will not work in production. In our case our R^2 's were 0.338 on training and 0.261 on test. Our model is of okay quality, and not over fit.

R^2 can be thought of as what fraction of the y variation is explained by the model. For well fit models R^2 is also equal to the square of the correlation between the predicted values and actual training values.

WARNING**R² can be overoptimistic**

In general, R² on training data will be higher for models with more input parameters, independently of whether the additional variables actually improve the model or not. That is why many people prefer the adjusted R² (which we will discuss more, below).

Also, R² is related to correlation, and the correlation can be artificially inflated if the model correctly predicts a few outliers (because the increased data range makes the overall data cloud appear "tighter" against the line of perfect prediction). Here's a toy example. Let `y <- c(1, 2, 3, 4, 5, 9, 10)` and `pred <- c(0.5, 0.5, 0.5, 0.5, 0.5, 9, 10)`. This corresponds to a model that is completely uncorrelated to the true outcome for the first five points, and perfectly predicts the last two points, which are somewhat far away from the first five. You can check for yourself that this obviously poor model has a correlation `cor(y, pred)` of about 0.926, with a corresponding R² of 0.858. So it's a very good idea to look at the true-versus-fitted graph in addition to checking R².

Another good measure to consider is root mean square error (RMSE):

```
rmse <- function(y, f) { sqrt(mean( (y-f)^2 )) }
rmse(log(dtrain$PINCP,base=10),predict(model,newdata=dtrain))
rmse(log(dtest$PINCP,base=10),predict(model,newdata=dtest))
```

You can think of the RMSE as a measure of the width of the data cloud around the line of perfect prediction.

7.1.4 Finding relations and extracting advice

Recall that your other goal, beyond predicting income, is to find the value of having a bachelor's degree. We will show how this value, and other relations in the data, can be read directly off a linear regression model.

All of the information in a linear regression model is stored in a block of numbers called the coefficients. The coefficients are available through the `coefficients(model)` command. The coefficients of our income model are shown in figure 7.6.

> coefficients(model)		
	(Intercept)	AGEP
	3.97328317	0.01171695
	SEXF	CWFederal government employee
	-0.09313262	0.05972743
	CWLocal government employee	CWPrivate not-for-profit employee
	-0.03441690	-0.13352439
	COWSelf-employed incorporated	COWSelf-employed not incorporated
	-0.07242727	-0.06060878
	COWState government employee	SCHLAssociate's degree
	-0.08128385	0.22143196
	SCHLBachelor's degree	SCHLDoctorate degree
	0.39383406	0.30655376
	SCHLGED or alternative credential	SCHLMaster's degree
	0.13776759	0.47728422
	SCHLProfessional degree	SCHLRegular high school diploma
	0.67877416	0.10174598
	SCHLsome college credit, no degree	
	0.16315219	

The modeled values of a bachelor's degree versus a high school diploma.

Figure 7.6 The model coefficients

REPORTED COEFFICIENTS

Our original modeling variables were only AGEPE, SEX, COW and SCHL; yet the model reports many more coefficients than these four. We are going to explain what all the reported coefficients are.

In figure 7.6 there are 8 coefficients that start with "SCHL." The original variable SCHL took on these 8 string values plus one more not shown: "no high school diploma". Each of these possible strings is called a *level*, and SCHL itself is called a *categorical variable* or a *factor variable*. The level that is not shown is called the *reference level*; the coefficients of the other levels are measured with respect to the reference level.

For example, SCHLBachelor's degree we find the coefficient 0.39 which is read as "the model gives a 0.39 bonus to log income for having a bachelor's degree, relative to not having a high school degree." This means that the income ratio between someone with a bachelor's degree and the equivalent person (same sex, age, and class of work) without a high school degree is about $10^{0.39}$, or 2.45 higher.

And under SCHLRegular high school diploma we find the coefficient 0.10. This is read as: the model believes that having a bachelor's degree tends to add 0.39 - 0.10 units to the predicted log income (relative to a high school degree). The modeled relation between the bachelor's degree's expected income and high school graduate's (all other variables being equal) is $10^{(0.39-0.10)}$ or about 1.8 times larger. The advice: college is worth it if you can find a job

(remember we limited to analyzing the fully employed, so this is assuming you can find a job).

`SEX` and `COW` are also discrete variables, and the coefficients that correspond to the different levels of `SEX` and `COW` can be interpreted in a similar manner. `AGEP` is a continuous variable with coefficient 0.0117. You can interpret this as a one year increase in age adding a 0.0117 bonus to log income; in other words an increase in age of one year corresponds to an increase of income of $10^{0.0117}$, or a factor of 1.027 -- about a 2.7% increase in income (all other variables being equal).

The coefficient "(Intercept)" is the coefficient corresponding to a variable that always has a value of 1, which is implicitly added to linear regression models unless use the special "0 +" notation in the formula during the call to `lm()`. This coefficient is a rough center for the model predictions.

SIDE BAR Indicator variables

Most modeling methods handle a string-valued (categorical) variable with n possible levels by converting it to n (or $n-1$) binary variables, or *indicator variables*. R has commands to explicitly control the conversion of string valued variables into well behaved indicators: `as.factor()` creates categorical variables from string variables; `relevel()` allows the user to specify the reference level.

However, beware of variables with a very large number of levels, like zip code. The runtime of linear (and logistic) regression increases as roughly the cube of the number of coefficients. Too many levels (or too many variables in general) will bog the algorithm down and require much more data for reliable inference.³

Footnote 3 For a trick dealing with factors with very many levels see:

<http://www.win-vector.com/blog/2012/07/modeling-trick-impact-coding-of-categorical-variables-with->

The above interpretations of the coefficients assume that the model has provided good estimates of the coefficients. We will see how to check that in the next section.

7.1.5 Reading the model summary and Characterizing coefficient quality

In section 7.1.3 we checked if our income predictions were to be trusted. We will now show how to check if model coefficients are reliable. This is especially urgent as we have been discussing showing coefficients relations to others as advice.

Most of what we need to know is already in the model summary, which is produced using the `summary()` command: `summary(model)`. This produces the output shown in figure 7.7.

Model call summary																																																																																																																	
<code>Call: lm(formula = log(PINCP, base = 10) ~ AGEP + SEX + COW + SCHL, data = dtrain)</code>																																																																																																																	
Residuals summary																																																																																																																	
<code>Residuals:</code> <table> <tr> <th>Min</th><th>1Q</th><th>Median</th><th>3Q</th><th>Max</th><th></th></tr> <tr> <td>-1.29220</td><td>-0.14153</td><td>0.02458</td><td>0.17632</td><td>0.62532</td><td></td></tr> </table>						Min	1Q	Median	3Q	Max		-1.29220	-0.14153	0.02458	0.17632	0.62532																																																																																																	
Min	1Q	Median	3Q	Max																																																																																																													
-1.29220	-0.14153	0.02458	0.17632	0.62532																																																																																																													
Coefficients table																																																																																																																	
<code>Coefficients:</code> <table> <thead> <tr> <th></th><th>Estimate</th><th>Std. Error</th><th>t value</th><th>Pr(> t)</th><th></th></tr> </thead> <tbody> <tr> <td>(Intercept)</td><td>3.973283</td><td>0.059343</td><td>66.954</td><td>< 2e-16 ***</td><td></td></tr> <tr> <td>AGEP</td><td>0.011717</td><td>0.001352</td><td>8.666</td><td>< 2e-16 ***</td><td></td></tr> <tr> <td>SEXF</td><td>-0.093133</td><td>0.023405</td><td>-3.979</td><td>7.80e-05 ***</td><td></td></tr> <tr> <td>COWFederal government employee</td><td>0.059727</td><td>0.060927</td><td>0.980</td><td>0.327343</td><td></td></tr> <tr> <td>COWLocal government employee</td><td>-0.034417</td><td>0.048030</td><td>-0.717</td><td>0.473928</td><td></td></tr> <tr> <td>COWPrivate not-for-profit employee</td><td>-0.133524</td><td>0.039223</td><td>-3.404</td><td>0.000709 ***</td><td></td></tr> <tr> <td>COWSelf-employed incorporated</td><td>-0.072427</td><td>0.068093</td><td>-1.064</td><td>0.287928</td><td></td></tr> <tr> <td>COWSelf-employed not incorporated</td><td>-0.060609</td><td>0.069244</td><td>-0.875</td><td>0.381779</td><td></td></tr> <tr> <td>COWState government employee</td><td>-0.081284</td><td>0.057796</td><td>-1.406</td><td>0.160146</td><td></td></tr> <tr> <td>SCHLAssociate's degree</td><td>0.221432</td><td>0.052094</td><td>4.251</td><td>2.49e-05 ***</td><td></td></tr> <tr> <td>SCHLBachelor's degree</td><td>0.393834</td><td>0.043249</td><td>9.106</td><td>< 2e-16 ***</td><td></td></tr> <tr> <td>SCHLDoctorate degree</td><td>0.306554</td><td>0.160127</td><td>1.914</td><td>0.056058 .</td><td></td></tr> <tr> <td>SCHLGED or alternative credential</td><td>0.137768</td><td>0.078192</td><td>1.762</td><td>0.078612 .</td><td></td></tr> <tr> <td>SCHLMaster's degree</td><td>0.477284</td><td>0.050895</td><td>9.378</td><td>< 2e-16 ***</td><td></td></tr> <tr> <td>SCHLProfessional degree</td><td>0.678774</td><td>0.087321</td><td>7.773</td><td>3.52e-14 ***</td><td></td></tr> <tr> <td>SCHLRegular high school diploma</td><td>0.101746</td><td>0.042628</td><td>2.387</td><td>0.017316 *</td><td></td></tr> <tr> <td>SCHLSome college credit, no degree</td><td>0.163152</td><td>0.042729</td><td>3.818</td><td>0.000149 ***</td><td></td></tr> </tbody> </table>							Estimate	Std. Error	t value	Pr(> t)		(Intercept)	3.973283	0.059343	66.954	< 2e-16 ***		AGEP	0.011717	0.001352	8.666	< 2e-16 ***		SEXF	-0.093133	0.023405	-3.979	7.80e-05 ***		COWFederal government employee	0.059727	0.060927	0.980	0.327343		COWLocal government employee	-0.034417	0.048030	-0.717	0.473928		COWPrivate not-for-profit employee	-0.133524	0.039223	-3.404	0.000709 ***		COWSelf-employed incorporated	-0.072427	0.068093	-1.064	0.287928		COWSelf-employed not incorporated	-0.060609	0.069244	-0.875	0.381779		COWState government employee	-0.081284	0.057796	-1.406	0.160146		SCHLAssociate's degree	0.221432	0.052094	4.251	2.49e-05 ***		SCHLBachelor's degree	0.393834	0.043249	9.106	< 2e-16 ***		SCHLDoctorate degree	0.306554	0.160127	1.914	0.056058 .		SCHLGED or alternative credential	0.137768	0.078192	1.762	0.078612 .		SCHLMaster's degree	0.477284	0.050895	9.378	< 2e-16 ***		SCHLProfessional degree	0.678774	0.087321	7.773	3.52e-14 ***		SCHLRegular high school diploma	0.101746	0.042628	2.387	0.017316 *		SCHLSome college credit, no degree	0.163152	0.042729	3.818	0.000149 ***	
	Estimate	Std. Error	t value	Pr(> t)																																																																																																													
(Intercept)	3.973283	0.059343	66.954	< 2e-16 ***																																																																																																													
AGEP	0.011717	0.001352	8.666	< 2e-16 ***																																																																																																													
SEXF	-0.093133	0.023405	-3.979	7.80e-05 ***																																																																																																													
COWFederal government employee	0.059727	0.060927	0.980	0.327343																																																																																																													
COWLocal government employee	-0.034417	0.048030	-0.717	0.473928																																																																																																													
COWPrivate not-for-profit employee	-0.133524	0.039223	-3.404	0.000709 ***																																																																																																													
COWSelf-employed incorporated	-0.072427	0.068093	-1.064	0.287928																																																																																																													
COWSelf-employed not incorporated	-0.060609	0.069244	-0.875	0.381779																																																																																																													
COWState government employee	-0.081284	0.057796	-1.406	0.160146																																																																																																													
SCHLAssociate's degree	0.221432	0.052094	4.251	2.49e-05 ***																																																																																																													
SCHLBachelor's degree	0.393834	0.043249	9.106	< 2e-16 ***																																																																																																													
SCHLDoctorate degree	0.306554	0.160127	1.914	0.056058 .																																																																																																													
SCHLGED or alternative credential	0.137768	0.078192	1.762	0.078612 .																																																																																																													
SCHLMaster's degree	0.477284	0.050895	9.378	< 2e-16 ***																																																																																																													
SCHLProfessional degree	0.678774	0.087321	7.773	3.52e-14 ***																																																																																																													
SCHLRegular high school diploma	0.101746	0.042628	2.387	0.017316 *																																																																																																													
SCHLSome college credit, no degree	0.163152	0.042729	3.818	0.000149 ***																																																																																																													
Signif. codes: 0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1																																																																																																																	
Residual standard error: 0.2691 on 578 degrees of freedom Multiple R-squared: 0.3383, Adjusted R-squared: 0.3199 F-statistic: 18.47 on 16 and 578 DF, p-value: < 2.2e-16																																																																																																																	
Model quality summary																																																																																																																	

Figure 7.7 Model result summary

Figure 7.7 looks intimidating. But it contains a lot of useful information and diagnostics. You are very likely to be asked about elements of figure 7.7 when presenting results. So we will demonstrate how all of these fields are derived and their meanings.

We will first break the `summary()` down into pieces.

THE MODEL CALL SUMMARY

The first part of the `summary()` is how the `lm()` model was constructed:

```
Call:  
lm(formula = log(PINCP, base = 10) ~ AGEP + SEX + COW + SCHL,  
    data = dtrain)
```

This is a good place to double check you used the correct data frame, performed your intended transformations and used the right variables. For example we can double check we used the data frame `dtrain` and not the data frame `dtest`.

THE RESIDUALS SUMMARY

The next part of the `summary()` is the residuals summary:

```
Residuals:  
    Min      1Q  Median      3Q     Max  
-1.29220 -0.14153  0.02458  0.17632  0.62532
```

In linear regression the residuals are everything. Most of what you want to know about the quality of your model fit is in the residuals. The residuals are our errors in prediction: `log(dtrain$PINCP,base=10) - predict(model,newdata=dtrain)`. We can find useful summaries of the residuals for both the training and test sets as shown in the listing below:

```
> summary(log(dtrain$PINCP,base=10) - predict(model,newdata=dtrain))  
    Min.  1st Qu.  Median  Mean  3rd Qu.  Max.  
-1.29200 -0.14150  0.02458  0.00000  0.17630  0.62530  
> summary(log(dtest$PINCP,base=10) - predict(model,newdata=dtest))  
    Min.  1st Qu.  Median  Mean  3rd Qu.  Max.  
-1.494000 -0.165300  0.018920 -0.004637  0.175500  0.868100
```

In linear regression the coefficients are picked to minimize the sum of squares of the residuals. This is the why the method is also often called "the least squares method." So we expect the residuals to be small.

In the residual summary you are given the min and max, which are smallest and largest residual seen. You are also given 3 quantiles of the residuals: 1Q, median and 3Q. An r-quantile is a number x such that an r-fraction of the residuals are less

than x and a $(1-r)$ fraction of residuals are greater than x . The 1Q, median and 3Q quantiles values are the values of the 0.25, 0.5 and 0.75 quantiles.

What you hope to see in the residual summary is the median near zero, and symmetry in that 1Q is near -3Q (with both not too large). The 1Q and 3Q quantiles are very interesting because exactly half of the training data has a residual in this range. If you drew a random training example its residual would be in this range exactly half the time. So you really expect to commonly see prediction errors of these magnitudes. If these errors are too big for your application, you do not have a usable model.

THE SUMMARY COEFFICIENTS TABLE

The next part of the `summary(model)` is the coefficients table as shown in figure 7.8. A matrix form of this table can be gotten as `summary(model)$coefficients`.

Name of coefficient Coefficients:	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	3.973283	0.059343	66.954	< 2e-16 ***
AGEP	0.011717	0.001352	8.666	< 2e-16 ***
SEXF	-0.093133	0.023405	-3.979	7.80e-05 ***
COWFederal government employee	0.059727	0.060927	0.980	0.327343
COWLocal government employee	-0.034417	0.048030	-0.717	0.473928
COWPrivate not-for-profit employee	-0.133524	0.039223	-3.404	0.000709 ***
COWSelf-employed incorporated	-0.072427	0.068093	-1.064	0.287928
COWSelf-employed not incorporated	-0.060609	0.069244	-0.875	0.381779
COWState government employee	-0.081284	0.057796	-1.406	0.160146
SCHLAssociate's degree	0.221432	0.052094	4.251	2.49e-05 ***
SCHLBachelor's degree	0.393834	0.043249	9.106	< 2e-16 ***
SCHLDoctorate degree	0.306554	0.160127	1.914	0.056058 .
SCHLGED or alternative credential	0.137768	0.078192	1.762	0.078612 .
SCHLMaster's degree	0.477284	0.050895	9.378	< 2e-16 ***
SCHLProfessional degree	0.678774	0.087321	7.773	3.52e-14 ***
SCHLRegular high school diploma	0.101746	0.042628	2.387	0.017316 *
SCHLSome college credit, no degree	0.163152	0.042729	3.818	0.000149 ***

Figure 7.8 Model summary coefficient columns

Each model coefficient forms a row of the coefficients summary table. The columns report the estimated coefficient, the uncertainty of the estimate, how large

the coefficient is relative to the uncertainty and how likely such a ratio would be due to mere chance. Figure 7.8 give the names and interpretations of the columns.

We set out to study income and the impact on income of getting a bachelor's degree. But we must look at all of the coefficients to check for interfering effects.

For example: the coefficient of -0.083 for SEXF means that our model learned a penalty of -0.083 to $\log(PINCP, \text{base}=10)$ for being female. Females are modeled as earning $1 - 10^{-0.083}$ relative to males or 17% less, all other model parameters being equal. Notice we said "all other model parameters being equal" not "all other things being equal." That is because we are not modeling the number years in the workforce (which age may not be a reliable proxy for) or occupation/industry type (which has a big impact on income). This model is not, with the features it was given, capable of testing if on average a female in the same job with the same number of years of experience is paid less.

TIP**Statistics as an attempt to correct bad experimental design**

The absolute best experiment to test if there is a sex driven difference in income distribution would be to compare incomes of individuals who were identical in all possible variables (age, education, years in industry, performance reviews, race, region and so on) but differ only in sex. We are unlikely to have access to such data so we would settle for instead a good experimental design: a population where there is no correlation between any other feature and sex. Random selection can help in experimental design, but are not a panacea. Barring a good experimental design the usual pragmatic strategy is: introduce extra variables to represent effects that may have been interfering with the effect we were trying to study. Thus a study of the effect of sex on income may include other variables like education and age to try to disentangle the competing effects.

The p-value (also called the significance) is one of the most important diagnostic columns in the coefficient summary. The p-value estimates the probability of seeing a coefficient with a magnitude as large as you observed if the true coefficient is really zero (e.g. the variable has no effect on the outcome). Do not trust the estimate of any coefficient with a large p-value. The general rule of thumb is $p \geq 0.05$ is not to be trusted. The estimate of the coefficient may be good,

but you want to use more data to build a model that reliably shows that the estimate is good. On the flip side lower p-values are not always "better" once they are good enough.

SIDE BAR

Collinearity also lowers significance

Sometimes, a predictive variable will not appear significant because it is collinear (or correlated) with another predictive variable. For example, if we did try to use both age and number of years in the workforce to predict income, neither variable may appear significant. This is because age tends to be correlated with number of years in the workforce. If you remove one of the variables and the other one gains significance, this is a good indicator of correlation.

Another possible indication of collinearity in the inputs is seeing coefficients with an unexpected sign: for example seeing that income is *negatively* correlated with years in the workforce.

The overall model can still predict income quite well, even when the inputs are correlated; it just can't determine which variable "deserves the credit" for the prediction. Using regularization (especially Ridge regression as found in `lm.ridge()` in the package MASS) is helpful in collinear situations (we prefer it to "x alone" variable pre-processing such as principal components analysis). If you want to use the coefficient values as advice as well as make good predictions, try to avoid collinearity in the inputs as much as possible.

OVERALL MODEL QUALITY SUMMARIES

The last part of the `summary(model)` report, is the overall model quality statistics. It is a good idea to check the overall model quality before sharing any predictions or coefficients. The summary are as follows:

```
Residual standard error: 0.2691 on 578 degrees of freedom
Multiple R-squared:  0.3383,    Adjusted R-squared:  0.3199
F-statistic: 18.47 on 16 and 578 DF,  p-value: < 2.2e-16
```

The *degrees of freedom* is the number of data rows minus the number of coefficients fit. In our case it is:

```
df <- dim(dtrain)[1] - dim(summary(model)$coefficients)[1]
```

The degrees of freedom is thought of as the number of training data rows you have after correcting for the number of coefficients you tried to solve for. You want the degrees of freedom to be large compared to the number of coefficients fit to avoid "over fitting." Over fitting is when you find chance relations in your training data that are not present in the general population. Over fitting is very bad: you think you have a good model when you do not.

The *residual standard error* is the sum of the square of the residuals (that is the sum of squared error) divided by the degrees of freedom. So it is similar to the RMSE (root mean squared error) that we discussed above, except with the number of data rows adjusted to be the degrees of freedom. In R this is:

```
modelResidualError <- sqrt(sum(residuals(model)^2)/df)
```

Multiple R² is the R² (discussed in section 7.1.3).

The *adjusted R²* is the multiple R² penalized by the ratio of the degrees of freedom to the number of training examples. This attempts to correct the fact that more complex models tend to look better on training data due to over fitting. Usually it is better to rely on the adjusted R². Better still is to compute the R² between predictions and actuals on a hold-out test data. In section 7.1.3 we showed the R² on test data was 0.26, which is significantly lower than the reported adjusted R² of 0.32. So the adjusted R² discounts for over-fitting, but not always enough. This is one of the reasons we advise preparing both training and test data sets; the test set estimates can be more representative of production model performance than statistical formulas.

The F-statistic is similar to the t-values we saw with the model coefficients. It is used to measure if the linear regression model predicts outcome better than the constant mode (the mean value of y). The F-statistic gets its name from the F-test which is the technique used to check if two variances -- in this case the variance of residuals from the constant model, and the variance of the residuals from the linear model -- are significantly different. The corresponding p-value is the estimate of the probability that we would have observed an F-statistic this large or larger if the two variances in question are in reality the same. So you want the p-value to be small (rule of thumb: less than 0.05).

In our example, the model is doing better than just the constant model, and the improvement is incredibly unlikely to have arisen from sampling error.

7.1.6 Linear regression takeaways

What you should remember about linear regression:

- Linear regression is the "go to" statistical modeling method for quantities.
- You should always try linear regression first and only use more complicated methods if they actually out perform a linear regression model.
- However, linear regression will have trouble with problems that have a very large number of variables, or categorical variables with a very large number of levels.
- You can enhance linear regression by adding new variables or transforming variables (such as we did with the `log()` transform of `y`, but always be wary when transforming `y` as it changes the error model).
- With linear regression you think in terms of residuals. You look for variables that correlate with your errors and add them to try and eliminate systematic modeling errors.
- Linear regression can predict well even in the presence of correlated variables; however correlated variables lower the quality of the advice.
- Overly large coefficient magnitudes, overly large standard errors on the coefficient estimates, and the wrong sign on a coefficient could be indications of correlated inputs.
- Linear regression has some of the best built-in diagnostics available, but re-checking your model on test data is still your most effective diagnostic.

7.2 Using Logistic Regression

Logistic regression is the most important (and probably most used) member of a class of models called *generalized linear models*. Unlike linear regression, logistic regression can directly predict values that are restricted to the (0,1) interval, such as probabilities. It is the "go-to" method for predicting probabilities or rates, and like linear regression, the coefficients of a logistic regression model can be treated as "advice." It is also a good first choice for binary classification problems.

In this section, we will use a medical classification example (predicting whether a newborn will need extra medical attention) to work through all of the steps of producing and using a logistic regression model.⁴

Footnote 4 Logistic regression is usually used to perform classification, but logistic regression and its close cousin beta regression are also very useful in estimating *rates*. In fact R's standard `glm()` call will work with prediction numeric values between zero and one in addition to predicting classifications.

7.2.1 Understanding logistic regression

Logistic regression predicts the probability `y` that an instance belongs to a specific category -- for instance, the probability that a flight will be delayed. When `x[i,]` is a row of inputs (for example, a flight's origin and destination, the time of year, the weather, the air carried) logistic regression finds the function $f(x)$ such that:

```
f(x[i,]) = s(b[1] x[i,1] + b[2] x[i,2] + ... b[n] x[i,n]).
```

Here, $s(z)$ is the so-called *sigmoid function* defined as $s(z) = 1/(1+\exp(z))$. If the $y[i]$ are the probabilities that the $x[i,]$ belong to the class of interest (in our example, that a flight with certain characteristics will be delayed), then the task of fitting is to find the $b[1], \dots, b[n]$ such that $f(x[i,])$ is the best possible estimate of $y[i]$. R supplies a one-line command to find these coefficients: `glm()`⁵. Note that we don't need to supply $y[i]$ that are probability estimates to run `glm()`, the training method only requires $y[i]$ that say if a given training example is in the target class or not.

Footnote 5 Logistic regression can be used for classifying into any number of categories (as long as the categories are disjoint and cover all possibilities: every x has to belong to one of the given categories). However, `glm()` only handles the two-category case, so our discussion will focus on this case.

The sigmoid function maps real numbers to the interval $(0,1)$ -- that is, to probabilities. The inverse of the sigmoid is the *logit*, which is defined as $\log(p/(1-p))$, where p is a probability. The ratio $p/(1-p)$ is known as the *odds*, so in the flight example, the logit is the log of the odds (or "log-odds") that a flight will be delayed. In other words, you can think of logistic regression as a linear regression that finds the log-odds of the probability that you are interested in.

In particular, logistic regression assumes that `logit(y)` is linear in the values of x . Like linear regression, logistic regression will find the best coefficients to predict y , including finding advantageous combinations and cancellations when the inputs are correlated.

For the example task, imagine that you are working at a hospital. The goal is to design a plan that provisions neonatal emergency equipment to delivery rooms. Newborn babies are assessed at one and five minutes after birth using what is called the *Apgar test*, which is designed to determine if a baby needs immediate emergency care or extra medical attention. A baby who scores below seven (on a scale from zero to ten) on the Apgar scale needs extra attention.

Such at-risk babies are rare, so the hospital does not want to provision extra emergency equipment for every delivery. On the other hand, at-risk babies may need attention quickly, so provisioning resources proactively to appropriate deliveries can save lives. The goal of this project is to identify ahead of time situations with a higher probability of risk, so that resources can be allocated appropriately.

We will use a sample data set from the CDC 2010 Natality Public Use data file⁶. This data set records statistics for all births registered in the 50 US States and the District of Columbia, including facts about the mother and father, and about the delivery. We will use a sample of just over 26,000 births in a data frame called `sdata`⁷. The data is split into a training and test set, using the random grouping column that we added, as recommended in Section XRF:sect2_2.2.2:sectLoadFromDB.

Footnote 6 http://www.cdc.gov/nchs/data_access/Vitalstatsonline.htm

Footnote 7 Our pre-prepared file is:

<https://github.com/WinVector/zmPDSwR/tree/master/CDC/NatalRiskData.rData>; we also provide a script file <https://github.com/WinVector/zmPDSwR/blob/master/CDC/PrepNatalRiskData.R>, which prepares the data frame from an extract of the full natality data set. Details found at <https://github.com/WinVector/zmPDSwR/blob/master/CDC/README.md>.

```
load("NatalRiskData.rData")
train <- sdata[sdata$ORIGRANDGROUP<=5,]
test <- sdata[sdata$ORIGRANDGROUP>5,]
```

Table 7.1 lists the columns of the dataset that we will use. Because the goal is to anticipate at-risk infants ahead of time, you should restrict yourself to variables whose values are known before delivery, or can be determined during labor. For example, facts about the mother's weight or health history are valid inputs, but post-birth facts like infant birth weight are not. You can consider in-labor complications like breech birth, by reasoning that the model can be updated in the delivery room (via a protocol or checklist) in time for emergency resources to be allocated before delivery.

Table 7.1 Some Variables in Natality Data set

Variable	Type	Description
atRisk	boolean	TRUE if 5-minute Apgar score > 7; FALSE otherwise
PWGT	numeric	Mother's prepregnancy weight
UPREVIS	numeric (integer)	Number of prenatal medical visits
CIG_REC	boolean	TRUE if smoker; FALSE otherwise
GESTREC3	categorical	Two categories: "<37 weeks" (premature) and ">=37 weeks"
DPLURAL	categorical	birth plurality, three categories: single/twin/triplet+
ULD_MECO	boolean	TRUE if moderate/heavy fecal staining of amniotic fluid
ULD_PRECIP	boolean	TRUE for unusually short labor (< three hours)
ULD_BREECH	boolean	TRUE for Breech (pelvis first) birth position
URF_DIAB	boolean	TRUE if mother is diabetic
URF_CHYPER	boolean	TRUE if mother has chronic hypertension
URF_PHYPER	boolean	TRUE if mother has pregnancy-related hypertension
URF_ECLAM	boolean	TRUE if mother experienced eclampsia: pregnancy-related seizures

Now you are ready to build the model.

7.2.2 Building a logistic regression model

The command to build a logistic regression model in R is `glm()`. In our case, the dependent variable `y` is the boolean `atRisk`; all the other variables in table 7.1 are the dependent variables `x`. The formula for building a model to predict `atRisk` using the above variables is rather long to type in by hand; you can generate the formula with the following commands:

```
complications <- c("ULD_MECO", "ULD_PRECIP", "ULD_BREECH")
riskfactors <- c("URF_DIAB", "URF_CHYPER", "URF_PHYPER",
                 "URF_ECLAM")
y <- "atRisk"
x <- c("PWGT",
       "UPREVIS",
       "CIG_REC",
       "GESTREC3",
       "DPLURAL",
       complications,
       riskfactors)
fmla <- paste(y, paste(x, collapse="+"), sep=~")
```

Now you build the logistic regression model, using the training data set.

```
fmla
[1] "atRisk ~ PWGT+UPREVIS+CIG_REC+GESTREC3+DPLURAL+ULD_MECO+ULD_PRECIP+
     ULD_BREECH+URF_DIAB+URF_CHYPER+URF_PHYPER+URF_ECLAM"

model <- glm(fmla, data=train, family=binomial(link="logit"))
```

This is similar to the linear regression call to `lm()`, with one additional argument: `family=binomial(link="logit")`. The `family` function specifies the assumed distribution of the dependent variable `y`. In our case, we are modeling `y` as a binomial distribution, or as a coin whose probability of heads depends on `x`. The link function "links" the output to a linear model: that is, you pass `y` through the link function, then model the resulting value as a linear function of the `x`. Different combinations of `family` functions and `link` functions lead to

different kinds of generalized linear model (for example, poisson, or probit). In this book we will only discuss logistic models, so we will only need to use the binomial family with the logit link.

WARNING **Don't forget the `family` argument!**

Without an explicit `family` argument, `glm` defaults to standard linear regression (like `lm`).

As before, we've stored the results in the object `model`.

7.2.3 Making Predictions

Making predictions with a logistic model is similar to making predictions with a linear model: you again use the `predict()` function.

```
train$pred <- predict(model, newdata=train, type="response")
test$pred <- predict(model, newdata=test, type="response")
```

We've again stored the predictions for the training and test sets as the column `pred` in the respective data frames. Note the additional parameter `type="response"`. This tells the `predict()` function to return the predicted probabilities `y`. If you don't specify `type="response"`, then by default `predict()` will return the output of the link function, `logit(y)`.

A strength of logistic regression is that it preserves the marginal probabilities of the training data. That means that if you sum up the predicted probability scores for the entire training set, that quantity will be equal to the number of positive outcomes (`atRisk == T`) in the training set. This is also true at finer granularity: for example, that if you sum up the predicted probabilities for all the subjects in the training set for whom `train$GESTREC=="<37 weeks"` (the baby was premature), the summed quantity will be equal to the count of all positive examples in the training set where the baby was premature.

CHARACTERIZING PREDICTION QUALITY

If your goal is use the model to classify new instances into one of two categories (in this case at-risk or not at-risk), then you would like the model to give high scores to positive instances, and low scores otherwise. You can check if this is so by plotting the distribution of scores for both the positive and negative instances. Let's do this on the training set (you should also plot the test set, to make sure that the performance is of similar quality).

```
library(ggplot2)
ggplot(train, aes(x=pred, color=atRisk, linetype=atRisk)) +
  geom_density()
```

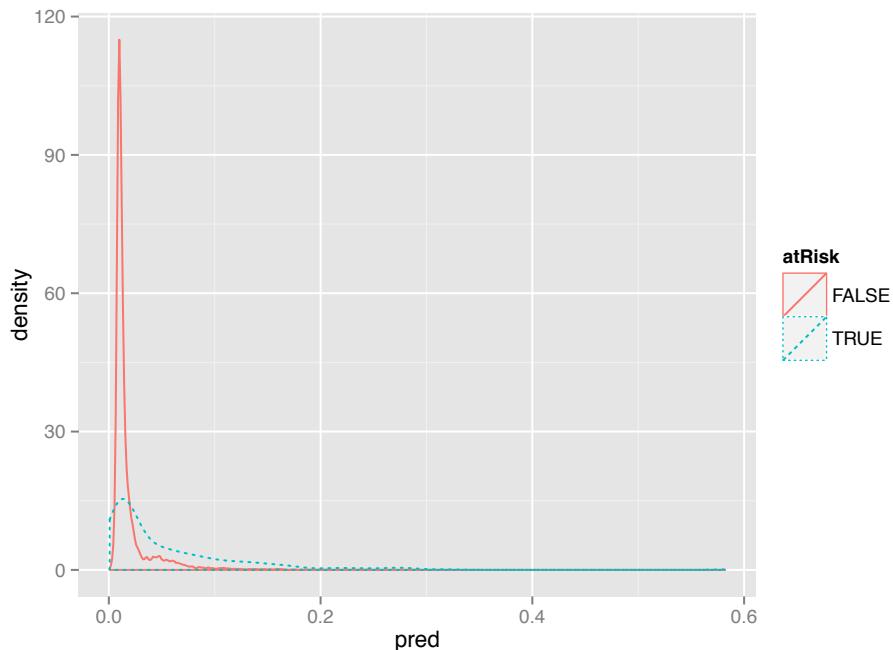


Figure 7.9 Distribution of score broken up by positive examples (TRUE) and negative examples (FALSE)

The result is shown in Figure 7.9. Ideally, you would like the distribution of scores to be separated, with the scores of the negative instances (FALSE) to be concentrated on the left, and the distribution for the positive instances to be concentrated on the right. You will see an example of a classifier that separates the positives and the negatives quite well in Figure XRF:figure_5.2.3:classProbDensity. In the current case, both distributions are concentrated on the left, meaning that both positive and negative instances score

low. This isn't surprising, since the positive instances are quite rare (about 1.8% of all births in the dataset). The distribution of scores for the negative instances dies off sooner than the distribution for positive instances. This means that the model did identify subpopulations in the data where the rate of at-risk newborns is higher than the average.

In order to use the model as a classifier, you must pick a threshold; scores above the threshold will be classified as positive, those below as negative. When you pick a threshold, you are trying to balance the *precision* of the classifier (what fraction of the predicted positives are true positives) and its *recall* (how many of the true positives does the classifier find).

If the score distributions of the positive and negative instances are well separated, as in Figure XRF:figure_5.2.3:classProbDensity, you can pick an appropriate threshold in the "valley" between the two peaks. In the current case, the two distributions are not well separated, which indicates that the model can't build a classifier that simultaneously achieves good recall and good precision. However, you can build a classifier that identifies a subset of situations with a higher than average rate of at-risk births, so pre-provisioning resources to those situations may be advised. We will call the ratio of the classifier precision to the average rate of positives the *enrichment rate*.

The higher you set the threshold, the more precise the classifier will be (that is, you will identify a set of situations with a much higher than average rate of at-risk births); however, you will also miss a higher percentage of at-risk situations, as well. When picking the threshold, you should use the training set, since picking the threshold is part of classifier-building. You can then use the test set to evaluate classifier performance.

To help pick the threshold, you can use a plot like Figure 7.10, which shows both enrichment and recall as a function of the threshold

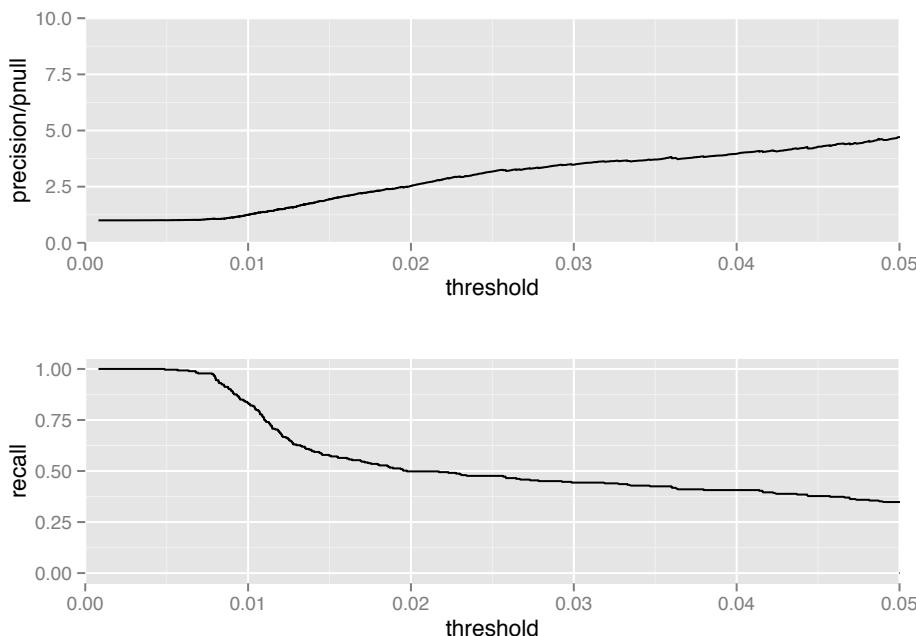


Figure 7.10 Enrichment (top) and recall (bottom) plotted as a function of threshold for the training set

Looking at Figure 7.10, you see that higher thresholds result in more precise classifications, at the cost of missing more cases; a lower threshold will identify more cases, at the cost of many more false positives. The best tradeoff between precision and recall is a function of how many resources the hospital has available to allocate, and how many they can keep in reserve (or re-deploy) for situations that the classifier missed. A threshold of 0.02 (which incidentally is about the overall rate at-risk births in the training data) might be a good tradeoff. The resulting classifier will identify a set of potential at-risk situations that finds about half of all the true at-risk situations, with a true positive rate 2.5 times higher than the overall population.

You can produce Figure 7.10 with the ROCR package, which we discussed in more detail in Chapter XRF:chapter_5:chCritiquingModels.

```

library(ROCR)                                ①
library(grid)                                 ②

predObj <- prediction(train$pred, train$atRisk) ③
precObj <- performance(predObj, measure="prec") ④
recObj <- performance(predObj, measure="rec")   ⑤

```

```

precision <- (precObj@y.values)[[1]]6
prec.x <- (precObj@x.values)[[1]]7
recall <- (recObj@y.values)[[1]]

rocFrame <- data.frame(threshold=prec.x, precision=precision,8
                        recall=recall)

nplot <- function(plist) {
  n <- length(plist)
  grid.newpage()
  pushViewport(viewport(layout=grid.layout(n,1)))
  vplayout=function(x,y) {viewport(layout.pos.row=x, layout.pos.col=y)}
  for(i in 1:n) {
    print(plist[[i]], vp=vplayout(i,1))
  }
}

pnull <- mean(as.numeric(train$atRisk))10

p1 <- ggplot(rocFrame, aes(x=threshold)) +
  geom_line(aes(y=precision/pnull)) +
  coord_cartesian(xlim = c(0,0.05), ylim=c(0,10) )10

p2 <- ggplot(rocFrame, aes(x=threshold)) +12
  geom_line(aes(y=recall)) +
  coord_cartesian(xlim = c(0,0.05) )

nplot(list(p1, p2))13

```

- ➊ Load the ROCR library
- ➋ Load the grid library (you will need this for the nplot function, below)
- ➌ Create a ROCR prediction object.
- ➍ Create a ROCR object to calculate precision as a function of threshold.
- ➎ Create a ROCR object to calculate recall as a function of threshold
- ➏ ROCR objects are what R calls S4 objects; the slots (or fields) of an S4 object are stored as lists within the object. You extract the slots from an S4 object using "@" notation.
- ➐ The x values (thresholds) are the same in both predObj and recObj, so you only need to extract them once.
- ➑ Build a data frame with thresholds, precision, and recall.
- ➒ A function to plot multiple plots on one page (stacked).
- ➓ Calculate the rate of at-risk births in the training set.
- ➔ Plot the enrichment rate as a function of threshold.
- ➕ Plot the recall as a function of threshold.
- ➖ Show both plots simultaneously.

Once you have picked an appropriate threshold, you can evaluate the resulting classifier by looking at the confusion matrix, as we discussed in Section

XRF:sect2_5.2.1:sectScoringClassifiers. Let's use the test set to evaluate the classifier, with a threshold of 0.02.

```
> ctab.test <- table(pred=test$pred>0.02, atRisk=test$atRisk) ①
> ctab.test
      atRisk
pred   FALSE TRUE
  FALSE  9487  93
  TRUE   2405 116
> precision <- ctab.test[2,2]/sum(ctab.test[2,])
> precision
[1] 0.04601349
> recall <- ctab.test[2,2]/sum(ctab.test[,2])
> recall
[1] 0.5550239
> enrich <- precision/mean(as.numeric(test$atRisk))
> enrich
[1] 2.664159
```

- ① Build the confusion matrix
- ② The rows contain the predicted negatives and positives; the columns contain the actual negatives and positives.

The resulting classifier has a precision of 4.6%, and identifies a set of potential at-risk cases that contains 55.5% of the true positive cases in the test set, at a rate 2.66 times higher than the overall average. This is consistent with the results on the training set.

In addition to making predictions, a logistic regression model also helps you extract useful information and advice. We will see this in the next section.

7.2.4 Finding Relations and Extracting Advice

The coefficients of a logistic regression model encode the relationships between the input variables and the output in a similar way the coefficients of a linear regression model do. You can get the model's coefficients with the call `coefficients(model)`.

```
> coefficients(model)
(Intercept)                  PWGT
-4.41218940                 0.00376166
UPREVIS                      CIG_RECTRUE
-0.06328943                  0.31316930
GESTREC3< 37 weeks DPLURALtriplet or higher
1.54518311                   1.39419294
```

DPLURALtwin	ULD_MECACTURE
0.31231871	0.81842627
ULD_PRECIPTRUE	ULD_BREECHTRUE
0.19172008	0.74923672
URF_DIABTRUE	URF_CHYPERTRUE
-0.34646672	0.56002503
URF_PHYPERTRUE	URF_ECLAMTRUE
0.16159872	0.49806435

Negative coefficients that are statistically significant⁸ correspond to variables that are negatively correlated to the odds (and hence to the probability) of a positive outcome. Positive coefficients that are statistically significant are positively correlated to the odds of a positive outcome.

Footnote 8 We will show how to check for statistical significance in the next section.

As with linear regression, every categorical variable is expanded to a set of indicator variables. If the original variable has n levels, there will be $n-1$ indicator variables; the remaining level is the reference level.

For example, the variable DPLURAL has three levels, corresponding to single births, twins, and triplets or higher. The logistic regression model has two corresponding coefficients: DPLURALtwin and DPLURALtriplet or higher. The reference level is single births. Both the DPLURAL coefficients are positive, indicating that multiple births have higher odds of being at-risk than single births do, all other variables being equal.

WARNING **Logistic regression also dislikes a very large variable count**
And as with linear regression, you should avoid categorical variables with too many levels.

INTERPRETING THE COEFFICIENTS

Interpreting coefficient values is a little more complicated with logistic regression than with linear. If the coefficient for the variable $x[, k]$ is $b[k]$, then the odds of a positive outcome are multiplied by a factor of $\exp(b[k])$ for every unit change in $x[, k]$.

The coefficient for GESTREC3< 37 weeks (that is, for a premature baby) is 1.545183. So for a premature baby, the odds of being at-risk are $\exp(1.545183)=4.68883$ times higher compared to a baby that is born full-term, with all other input variables unchanged. To make this concrete, suppose a full-term baby with certain characteristics has a 1% probability of being at-risk

(odds are $p / (1-p)$, or $0.01/0.99 = 0.0101$). Then the odds for a premature baby with the same characteristics are $0.0101 * 4.68883 = 0.047$. This corresponds to a probability of being at risk of $\text{odds} / (1+\text{odds})$, or $0.047/1.047$ -- about 4.5%.

Similarly, the coefficient for UPREVIS (number of prenatal medical visits) is about -0.06. This means every prenatal visit lowers the odds of an at-risk baby by a factor of $\exp(-0.06)$, or about 0.94. Suppose the mother of our premature baby above had made no prenatal visits; a baby in the same situation whose mother had made three prenatal visits would have odds of being at-risk of about $0.047 * 0.94 * 0.94 * 0.94 = 0.039$. This corresponds to a probability of being at risk of 3.75%.

So the general advice in this case might be to keep a special eye on premature births (and multiple births), and encourage expectant mothers to make regular prenatal visits

7.2.5 Reading the Model Summary and Characterizing Coefficients

As we mentioned above, conclusions about the coefficient values are only to be trusted if the coefficient values are statistically significant. You also want to make sure that the model is actually explaining something. The diagnostics in the model summary will help you determine some facts about model quality. The call, as before, is `summary(model)`.

```
> summary(model)

Call:
glm(formula = fmla, family = binomial(link = "logit"), data = train)

Deviance Residuals:
    Min      1Q  Median      3Q      Max 
-0.9732 -0.1818 -0.1511 -0.1358  3.2641 

Coefficients:
              Estimate Std. Error z value Pr(>|z|)    
(Intercept) -4.412189  0.289352 -15.249 < 2e-16 ***
PWGT         0.003762  0.001487   2.530 0.011417 *  
UPREVIS     -0.063289  0.015252  -4.150 3.33e-05 *** 
CIG_RECTRUE  0.313169  0.187230   1.673 0.094398 .  
GESTREC3< 37 weeks 1.545183  0.140795  10.975 < 2e-16 *** 
DPLURALtriplet or higher 1.394193  0.498866   2.795 0.005194 ** 
DPLURALtwin   0.312319  0.241088   1.295 0.195163    
ULD_MECOTRUE  0.818426  0.235798   3.471 0.000519 *** 
ULD_PRECIPTRUE 0.191720  0.357680   0.536 0.591951    
ULD_BREECHTRUE 0.749237  0.178129   4.206 2.60e-05 *** 
URF_DIABTRUE  -0.346467  0.287514  -1.205 0.228187    
URF_CHYPERTTRUE 0.560025  0.389678   1.437 0.150676    
URF_PHYPERTTRUE 0.161599  0.250003   0.646 0.518029
```

```

URF_ECLAMTRUE          0.498064  0.776948  0.641 0.521489
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 2698.7  on 14211  degrees of freedom
Residual deviance: 2463.0  on 14198  degrees of freedom
AIC: 2491

Number of Fisher Scoring iterations: 7

```

THE MODEL CALL SUMMARY

The first line of the summary is the call to `glm()`.

```

Call:
glm(formula = fmla, family = binomial(link = "logit"), data = train)

```

Here is where you check that you have used the correct training set and the correct formula (although in our case, the formula itself is in another variable). You can also verify that you used the correct family and link function to produce a logistic model.

THE DEVIANCERESIDUALS SUMMARY

The deviance residuals are the analog to the residuals of a linear regression model.

```

Deviance Residuals:
    Min      1Q  Median      3Q      Max
-0.9732 -0.1818 -0.1511 -0.1358  3.2641

```

In linear regression, the residuals are the vector of differences between the true outcome values and the predicted output values (the errors). In logistic regression, the deviance residuals are related to the *log-likelihoods* of having observed the true outcome, given the predicted probability of that outcome. The idea behind log-likelihood is that positive instances `y` should have high probability `py` of occurring under the model; negative instances should have low probability of

occurring (or putting it another way, $(1-py)$ should be large). The log-likelihood function rewards "matches" between the outcome y and the predicted probability py , and penalizes mis-matches (high py for negative instances, and vice-versa).

```
pred <- predict(model, newdata=train, type="response")      1
llcomponents <- function(y, py) {
  y*log(py) + (1-y)*log(1-py)
}

edev <- sign(as.numeric(train$atRisk) - pred) *
  sqrt(-2*llcomponents(as.numeric(train$atRisk), pred))    3

> summary(edev)
   Min. 1st Qu. Median Mean 3rd Qu. Max.
-0.9732 -0.1818 -0.1511 -0.1244 -0.1358 3.2640
```

- ① Create the vector of predictions for the training data.
- ② A function to return the log-likelihoods for each data point. The argument y is the true outcome (as a numeric variable, 0/1), the argument py is the predicted probability.
- ③ Calculate the deviance residuals.

Linear regression models are found by minimizing the sum of the squared residuals; logistic regression models are found by minimizing the sum of the squared residual deviances, which is equivalent to maximizing the log-likelihood of the data, given the model.

Logistic models can also be used to explicitly compute rates: given several groups of identical data points (identical except the outcome) predict the rate of positive outcomes in each group. This kind of data is called *grouped data*. In the case of grouped data, the deviance residuals can be used as a diagnostic for model fit. This is why the deviance residuals are included in the summary. We are using *ungrouped data* -- every data point in the training set is potentially unique. In the case of ungrouped data, the model fit diagnostics that use the deviance residuals are no longer valid⁹.

Footnote 9 Powers, Daniel and Yu Xie. *Statistical Methods for Categorical Data Analysis*, 2008.

THE SUMMARY COEFFICIENTS TABLE

The summary coefficients table for logistic regression has the same format as the coefficients table for linear regression.

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	-4.412189	0.289352	-15.249	< 2e-16 ***
PWGT	0.003762	0.001487	2.530	0.011417 *
UPREVIS	-0.063289	0.015252	-4.150	3.33e-05 ***
CIG_RECTRUE	0.313169	0.187230	1.673	0.094398 .
GESTREC3< 37 weeks	1.545183	0.140795	10.975	< 2e-16 ***
DPLURALtriplet or higher	1.394193	0.498866	2.795	0.005194 **
DPLURALtwin	0.312319	0.241088	1.295	0.195163
ULD_MECONTRUE	0.818426	0.235798	3.471	0.000519 ***
ULD_PRECIPTRUE	0.191720	0.357680	0.536	0.591951
ULD_BREECHTRUE	0.749237	0.178129	4.206	2.60e-05 ***
URF_DIABTRUE	-0.346467	0.287514	-1.205	0.228187
URF_CHYPERTURE	0.560025	0.389678	1.437	0.150676
URF_PHYPERTURE	0.161599	0.250003	0.646	0.518029
URF_ECLAMTRUE	0.498064	0.776948	0.641	0.521489

Signif. codes:	0 ****	0.001 ***	0.01 **	0.05 *
	.	.	0.1	' 1

Each column of the table represents

- a coefficient
- its estimated value
- the error around that estimate
- the signed distance of the estimated coefficient value from zero (using the standard error as the unit of distance)
- the probability of seeing a coefficient value at least as large as you observed, under the null hypothesis that the coefficient value is really zero

This last value, called the *p-value* or *significance*, tells you whether or not you should trust the estimated coefficient value. The standard rule of thumb is that coefficients with p-value less than 0.05 are reliable, although some researchers prefer stricter thresholds.

For the birth data, you can see from the coefficient summary that premature birth and triplet birth are strong predictors of the newborn needing extra medical attention: the coefficient magnitudes are large and the p-values indicate significance. Other variables that affect the outcome are the mother's prepregnancy weight (heavier mothers indicate higher risk -- slightly surprising); the number of prenatal medical visits (the more visits, the lower the risk); meconium staining in the amniotic fluid; and breech position at birth. There might be a positive

correlation between mother's smoking and an at-risk birth, but the data doesn't indicate it definitively. None of the other variables show a strong relationship to an at-risk birth.

SIDE BAR**Lack of significance could mean collinear inputs**

As with linear regression, logistic regression can predict well with collinear (or correlated) inputs, but the correlations can mask good advice.

To see this for yourself, we left data about the babies' birth weight in grams in the data set `sdata`. It is present in both the test and training data as the column `DBWT`. Try adding `DBWT` to the logistic regression model in addition to all the other variables; you will see that the coefficient for baby's birth weight will be large and significant, and negatively correlated with risk. The coefficient for `DPLURALtriplet` or higher will appear not significant, and the coefficient for `GESTREC3< 37 weeks` has a much smaller magnitude. This is because low birth weight is correlated to both prematurity and multiple birth. Of the three related variables, birth weight does "most of the explaining" about the outcome: knowing that the baby is a triplet adds no additional useful information, and knowing the baby is premature adds only a little information.

In the context of the modeling goal -- to proactively allocate emergency resources where they are more likely to be needed -- birth weight is not as useful a variable, because you don't know the baby's weight until it is born. You do know ahead of time if it is being born prematurely, or if it is one of multiple babies. So it's better to use `GESTREC3` and `DPLURAL` as input variables instead.

Other signs of possibly collinear inputs: coefficients with the wrong sign, and unusually large coefficient magnitudes.

OVERALL MODEL QUALITY SUMMARIES

The next section of the summary contains the model quality statistics.

```
Null deviance: 2698.7 on 14211 degrees of freedom
Residual deviance: 2463.0 on 14198 degrees of freedom
AIC: 2491
```

Null and Residual Deviances

Deviance is again a measure of how well the model fits the data. It is -2 times the

log-likelihood of the data set, given the model. If you think of deviance as analogous to variance, then the *Null deviance* is similar to the variance of the data around the average rate of positive examples. The *Residual deviance* is similar to the variance of the data around the model. We can calculate the deviances for both the training and test sets.

```

loglikelihood <- function(y, py) { ①
  sum(y * log(py) + (1-y)*log(1 - py))
}

pnull <- mean(as.numeric(train$atRisk)) ②
null.dev <- -2*loglikelihood(as.numeric(train$atRisk), pnull) ③

> pnull
[1] 0.01920912
> null.dev
[1] 2698.716
> model$null.deviance ④
[1] 2698.716

pred <- predict(model, newdata=train, type="response") ⑤
resid.dev <- -2*loglikelihood(as.numeric(train$atRisk), pred) ⑥

> resid.dev
[1] 2462.992
> model$deviance ⑦
[1] 2462.992

testy <- as.numeric(test$atRisk)
testpred <- predict(model, newdata=test,
                     type="response")
pnull.test <- mean(testy)
null.dev.test <- -2*loglikelihood(testy, pnull.test)
resid.dev.test <- -2*loglikelihood(testy, testpred)

> pnull.test
[1] 0.0172713
> null.dev.test
[1] 2110.91
> resid.dev.test
[1] 1947.094

```

- ➊ A function to calculate the log-likelihood of a dataset. The variable `y` is the outcome in numeric form (1 for positive examples, 0 for negative). The variable `py` is the predicted probability that `y==1`.
- ➋ Calculate the rate of positive examples in the data set.
- ➌ Calculate the null deviance.

- 4 For the training data, the null deviance is stored in the slot `model$null.deviance`.
- 5 Predict the probabilities for the training data.
- 6 Calculate the deviance of the model and the training data.
- 7 For the training data, the model deviance is stored in the slot `model$deviance`.
- 8 Calculate null deviance and residual deviance for the test data.

The first thing you can do with the null and residual deviances is check whether or not the model's probability predictions are actually better than just guessing the average rate of positives, statistically speaking. In other words, is the reduction in deviance from the model meaningful, or just something that was observed by chance? This is similar to calculating the F-test statistics that are reported for linear regression. In the case of logistic regression, the test you will run is the chi-squared test. To do that, you need to know the degrees of freedom for the null model and the actual model (which are reported in the summary). The degrees of freedom of the null model is the number of data points minus one: `df.null = ddim(train)[[1]] - 1`. The degrees of freedom of the model that you fit is the number of data points minus the number of coefficients in the model: `df.model = dim(train)[[1]] - length(model$coefficients)`.

If the number of data points in the training set is large, and `df.null - df.model` is small, then the probability of the difference in deviances `null.dev - resid.dev` being as large as we observed is approximately distributed as a chi-squared distribution with `df.null - df.model` degrees of freedom.

```

df.null <- dim(train)[[1]] - 1          ①
df.model <- dim(train)[[1]] - length(model$coefficients) ②

> df.null
[1] 14211
> df.model
[1] 14198

delDev <- null.dev - resid.dev        ③
delfdf <- df.null - df.model
p <- pchisq(delDev, delfdf, lower.tail=F)

> delDev
[1] 235.724
> delfdf
[1] 13

```

```
> p  
[1] 5.84896e-43
```

- 1 The null model has (number of data points - 1) degrees of freedom
- 2 The fitted model has (number of data points - number of coefficients) degrees of freedom
- 3 Compute the difference in deviances and the difference in degrees of freedom
- 4 Estimate the probability of seeing the observed difference in deviances under the null model (the p-value) using a chi-squared distribution.

The p-value is very small; it is extremely unlikely that we could have seen this much reduction in deviance by chance.

The pseudo-R²

A useful goodness of fit measure based on the deviances is the pseudo-R²: $1 - (\text{dev.model}/\text{dev.null})$. The pseudo-R² is the analog to the R² measure for linear regression. It is a measure of how much of the deviance is "explained" by the model. Ideally, you want the pseudo-R² to be close to 1. Let's calculate the pseudo-R² for both the test and training data.

```
pr2 <- 1-(resid.dev/null.dev)  
  
> pr2  
[1] 0.08734674  
  
pr2.test <- 1-(resid.dev.test/null.dev.test)  
  
> pr2.test  
[1] 0.07760427
```

The model only explains about 7.7-8.7% of the deviance; it's not a highly predictive model (you should have suspected that already, from Figure 7.9). This tells you that you haven't yet identified all the factors that actually predict at-risk births.

SIDE BAR**Goodness-of-fit versus significance**

It's worth noting that the model we found is a legitimate model, just not a complete one. The good p-value tells us that the model is legitimate: it gives us more information than the average rate of at-risk births does alone. The poor pseudo-R² means that the model is not giving us enough information to predict at-risk births with high reliability.

It is also possible to have good pseudo-R² (on the training data) with a bad p-value. This is an indication of overfit. That is why it is a good idea to check both, or better yet, check the pseudo-R² of the model on both training and test data.

The AIC

The last metric given in the section of the summary is the AIC, or the *Akaike information criterion*. The AIC is the log-likelihood adjusted for the number of coefficients. Just as the R² of a linear regression is generally higher when the number of variables is higher, the log-likelihood also increases with the number of variables.

```
aic <- 2*(length(model$coefficients) -  
          loglikelihood(as.numeric(train$atRisk), pred))  
> aic  
[1] 2490.992
```

The AIC is usually used to decide which and how many input variables to use in the model. If you train many different models with different sets of variables on the same training set, you can consider the model with the lowest AIC to be the best fit.

FISHER SCORING ITERATIONS

The last line of the model summary is the number of Fisher scoring iterations.

```
Number of Fisher Scoring iterations: 7
```

The Fisher scoring method is an iterative optimization method similar to Newton's method that `glm()` uses to find the best coefficients for the logistic regression model. You should expect that it will converge in about six to eight

iterations. If there are more iterations than that, then the algorithm may not have converged, and the model may not be valid.

Separation and Quasi-Separation

The probable reason for non-convergence is separation or quasi-separation: one of the model variables or some combination of the model variables predicts the outcome perfectly for at least a subset of the training data. You would think this would be a good thing, but ironically logistic regression fails when the variables are too powerful. Ideally, `glm()` will issue a warning when it detects separation or quasi-separation:

```
Warning message:  
glm.fit: fitted probabilities numerically 0 or 1 occurred
```

Unfortunately there seem to be situations when no warning is issued, but there are other warning signs:

- An unusually high number of Fisher iterations
- Very large coefficients, usually with extremely large standard errors
- Residual deviances larger than the null deviances

If you see any of the above signs, the model is suspect. To try to fix the problem, remove any variables with unusually large coefficients; they are probably causing the separation. You can try using a decision tree on the variables that you remove to detect regions of perfect prediction. The data that the decision tree doesn't predict perfectly on can still be used for building a logistic regression model. The overall model would then be a hybrid: the decision tree to predict the "too good" data, and a logistic regression model for the rest.

SIDE BAR**The right way to deal with separation**

We admit, it doesn't feel right to remove variables or data that are "too good" from the modeling process. The correct way to handle separation is to regularize: regularization techniques prevent the model coefficients from becoming too large by using some form of smoothing. Unfortunately, `glm()` does not regularize. The package `glmnet` can. However, its calling interface is not the standard interface that `lm()`, `glm()`, and other modeling functions in this book use. It also does not have the nice diagnostic output of the other packages. For these reasons, we consider a discussion of `glmnet` beyond the scope of this book.

To regularize `glm()` in an ad-hoc way, you can use the `weights` argument. The `weights` argument lets you pass a vector of weights (one per datum) to the `glm()` call. Add another copy of the data, but with *opposite* outcomes, and use this faux data in `glm()` with a small weight. An example of this trick can be found in the blog article "A Pathological GLM Problem that Doesn't Issue a Warning¹⁰."

Footnote 10

<http://www.win-vector.com/blog/2013/05/a-pathological-glm-problem-that-doesnt-issue-a-warning/>

7.2.6 Logistic Regression Takeaways

What you should remember about logistic regression:

- Logistic regression is the "go to" statistical modeling method for binary classification. Try logistic regression first, then more complicated methods if logistic regression doesn't perform well.
- Logistic regression will have trouble with problems with a very large number of variables, or categorical variables with a very large number of levels
- Logistic regression is well-calibrated: it reproduces the marginal probabilities of the data.
- Logistic regression can predict well even in the presence of correlated variables; however correlated variables lower the quality of the advice.
- Overly large coefficient magnitudes, overly large standard errors on the coefficient estimates, and the wrong sign on a coefficient could be indications of correlated inputs.
- Too many Fisher iterations, or overly large coefficients with very large standard errors could be signs that an input or combination of inputs is perfectly correlated with a subset of your responses. You may have to segment the data to deal with this issue.
- `glm()` provides good diagnostics, but re-checking your model on test data is still your most effective diagnostic.
- Pseudo-R² is a useful goodness-of-fit heuristic.

7.3 Summary

In this chapter, you have learned how to predict numerical quantities with linear regression models, and to predict probabilities or classify using logistic regression models. You have also learned how to interpret the models that you've produced.

Both linear and logistic regression assume that the outcome is a linear function of the inputs. This seems quite restrictive, but in practice linear and logistic regression models can perform well even when the theoretical assumptions aren't exactly met -- which is most of the time.

Linear and logistic regression can also provide "advice" by quantifying the relationships between the outcomes and the model's inputs. Since the models are expressed completely by their coefficients, they are small and portable, and make predictions quite quickly -- all important issues when putting a model into production. If the model's errors are homoscedastic (uncorrelated with y), the model might be trusted to extrapolate predictions outside the training range. Extrapolation is never completely safe, but it is sometimes necessary.

The methods that we discussed in this chapter and in the previous chapter use data about known outcomes to build models that predict future outcomes. But what if you don't yet know what to predict? The next chapter looks at *unsupervised methods*: algorithms that discover previously unknown relationships in data.

7.4 External links section

Chapter 8: Using Unsupervised Methods



This chapter covers

- Using R's clustering functions to explore data and look for similarities
- Choosing the right number of clusters
- Evaluating a clustering
- Using R's association rules functions to find patterns of co-occurrence in data
- Evaluating a set of association rules

The methods that we've discussed in previous chapters build models to predict outcomes. In this chapter we look at methods to discover unknown relationships in data. These methods are called *unsupervised methods*. With unsupervised method, there is no outcome that you are trying to predict; instead, you want to discover patterns in the data that perhaps you had not previously suspected. For example, you may want to find groups of customers with similar purchase patterns, or correlations between population movement and socio-economic factors. Unsupervised analyses are often not ends in themselves; rather, they are ways of finding relationships and patterns that then can be used to build predictive models. In fact, we encourage you to think of unsupervised methods as exploratory -- procedures that help you get your hands in the data -- rather than as black-box approaches that mysteriously and automatically give you "the right answer."

In this chapter we will look at two classes of unsupervised methods. *Cluster analysis* finds groups in your data with similar characteristics. *Association rule mining* finds elements or properties in the data that tend to occur together.

8.1 Cluster Analysis

In cluster analysis, the goal is to group the observations in your data into *clusters* such that every datum in a cluster is more similar to other datums in the same cluster than it is to datums in other clusters. For example, a company that offers guided tours might want to cluster its clients by behavior and tastes: which countries they like to visit, whether they prefer adventure tours, luxury tours, or educational tours, what kinds of activities they participate in and what sorts of sites they like to visit. Such information can help the company design attractive travel packages, and target them to the appropriate segments of their client base.

Cluster analysis is a topic worth a book in itself; in this chapter we will discuss two approaches. *Hierarchical clustering* finds nested groups of clusters. An example of hierarchical clustering might be the standard plant taxonomy, which classifies plants by family, then genus, then species, and so on. The second approach we will cover is *K-means*, which is a quick and popular way of finding clusters in quantitative data.

SIDE BAR Clustering and Density Estimation

Historically, cluster analysis is related to the problem of *density estimation*: if you think of your data as living in a large dimensional space, then you want to find the regions of the space where the data is densest. If those regions are distinct, or nearly so, then you have clusters.

8.1.1 Distances

In order to cluster, you need a notion of "similarity" and "dissimilarity." Dissimilarity can be thought of as distance, so that the points in a cluster are closer to each other than they are to the points in other clusters. This is shown in Figure 8.1.

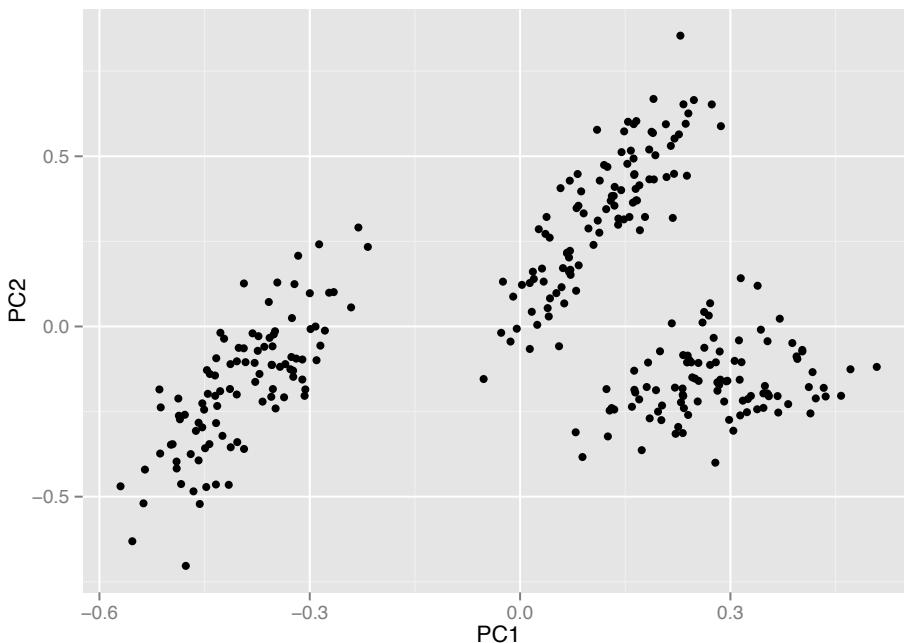


Figure 8.1 An example of data in three clusters

Different application areas will have different notions of distance and dissimilarity. In this section, we will cover a few of the most common ones:

- Euclidean Distance
- Hamming Distance
- Metropolis (City Block) Distance
- Cosine Similarity

Euclidean Distance

The most common distance is *Euclidean distance*. The euclidean distance between two vectors \mathbf{x} and \mathbf{y} is defined as:

```
edist(x, y) <- sqrt((x[1]-y[1])^2 + (x[2]-y[2])^2 + ...)
```

This is the measure people tend to think of when they think of "distance." Optimizing squared euclidean distance is the basis of K-means. Of course, euclidean distance only makes sense when all the data is real-valued (quantitative). If the data is categorical (in particular, binary), then other distances can be used.

Hamming Distance

For categorical variables (male/female, or small/medium/large), you can define the distance as zero if two points are in the same category, and one otherwise. If all the variables are categorical, then you can use *Hamming distance*, which counts the number of mismatches.

```
hdist(x, y) <- sum((x[1] != y[1]) + (x[2] != y[2]) + ...)
```

Here, $a \neq b$ is defined to have a value of one if the expression is true, and a value of zero if the expression is false.

You can also expand categorical variables to indicator variables (as we discussed in Section XRF:sect2_7.1.4:sectFindingRelns), one for each level of the variable.

If the categories are ordered (like small/medium/large) so that some categories are "closer" to each other than others, then you can convert them to a numerical sequence. For example, (small/medium/large) might map to (1/2/3). Then you can use euclidean distance, or other distances for quantitative data.

Metropolis (City Block) Distance

Metropolis (or Manhattan) distance measures distance in the number of horizontal and vertical units it takes to get from one (real-valued) point to the other (no diagonal moves).

```
mdist(x, y) <- sum(abs(x[1]-y[1]) + abs(x[2]-y[2]) + ...)
```

This is also known as "L1 distance" (and squared euclidean distance is "L2 distance").

Cosine similarity

Cosine similarity is a common similarity metric in text analysis. It measures the smallest angle between two vectors (that is, the angle `theta` between two vectors is assumed to be between 0 and 90 degrees). Two perpendicular vectors (`theta = 90` degrees) are the most dissimilar; the cosine of 90 degrees is zero. Two parallel vectors are the most similar (identical, if you assume they are both based at the origin); the cosine of 0 degrees is 1. From elementary geometry, you can derive

that the cosine of the angle between two vectors is given by the normalized dot product between the two vectors.

```
dot(x, y) <- sum( x[1]*y[1] + x[2]*y[2] + ... )
cossim(x, y) <- dot(x, y)/(sqrt(dot(x,x)*dot(y,y)))
```

You can turn the cosine similarity into a pseudo-distance by subtracting it from one (though to get an actual metric you should use $1 - 2 * \text{acos}(\text{cossim}(x, y)) / \pi$).

Different distance metrics will give you different clusters, as will different clustering algorithms. The application domain may give you a hint as to the most appropriate distance, or you can try several. In this chapter, we will use (squared) euclidean distance, as it is the most natural distance for quantitative data.

8.1.2 Preparing the Data

To demonstrate clustering, we will use a small dataset from 1973 on protein consumption from nine different food groups in 25 countries in Europe¹. The goal is to group the countries based on patterns in their protein consumption. The dataset is loaded into R as a dataframe called `protein`.

Footnote 1 The original data set can be found at <http://lib.stat.cmu.edu/DASL/Datafiles/Protein.html>. A tab-separated text file with the data can be found at

<https://github.com/WinVector/zmPDSwR/tree/master/Protein/>. The data file is called `protein.txt`; additional information can be found in the file `protein_README.txt`.

```
> summary(protein)
   Country      RedMeat       WhiteMeat        Eggs
Albania      : 1  Min.   : 4.400  Min.   : 1.400  Min.   :0.500
Austria      : 1  1st Qu.: 7.800  1st Qu.: 4.900  1st Qu.:2.700
Belgium      : 1  Median  : 9.500  Median  : 7.800  Median  :2.900
Bulgaria     : 1  Mean    : 9.828  Mean    : 7.896  Mean    :2.936
Czechoslovakia: 1  3rd Qu.:10.600 3rd Qu.:10.800 3rd Qu.:3.700
Denmark      : 1  Max.    :18.000  Max.    :14.000  Max.    :4.700
(Other)       :19
   Milk          Fish         Cereals        Starch
Min.   : 4.90  Min.   : 0.200  Min.   :18.60  Min.   : 0.600
1st Qu.:11.10 1st Qu.: 2.100  1st Qu.:24.30  1st Qu.:3.100
Median  :17.60  Median  : 3.400  Median  :28.00  Median  :4.700
Mean    :17.11  Mean    : 4.284  Mean    :32.25  Mean    :4.276
3rd Qu.:23.30  3rd Qu.: 5.800  3rd Qu.:40.10  3rd Qu.:5.700
Max.    :33.70  Max.    :14.200  Max.    :56.70  Max.    :6.500
```

Nuts	Fr.Veg
Min. :0.700	Min. :1.400
1st Qu.:1.500	1st Qu.:2.900
Median :2.400	Median :3.800
Mean :3.072	Mean :4.136
3rd Qu.:4.700	3rd Qu.:4.900
Max. :7.800	Max. :7.900

UNITS AND SCALING

The documentation for this dataset doesn't mention what the units of measurement are, though we can assume all the columns are measured in the same units. This is important: units (or more precisely, disparity in units) affect what clusterings an algorithm will discover. If you measure vital statistics of your subjects as age in years, height in feet, and weight in pounds, you will get different distances -- and possibly different clusters -- than if you measure age in years, height in meters, and weight in kilograms.

Ideally, you want a unit of change in each coordinate to represent the same degree of difference. In the `protein` data set, we assume that the measurements are all in the same units, so it might seem that we are okay. This may well be a correct assumption; however, different food groups provide different amounts of protein. Animal-based food sources in general have more grams of protein per serving than plant-based food sources, so one could argue that a change in consumption of 5 grams is a bigger difference in terms of vegetable consumption than it is in terms of red meat consumption.

One way to try to make the clustering more "coordinate-free" is to transform all the columns to have a mean value of zero and a standard deviation of one. This makes the standard deviation the unit of measurement in each coordinate. Assuming that your training data has a distribution that accurately represents the population at large, then a standard deviation represents approximately the same degree of difference in every coordinate. You can scale the data in R using the function `scale()`

```
vars.to.use <- colnames(protein)[-1] ①
pmatrix <- scale(protein[,vars.to.use]) ②
pcenter <- attr(pmatrix, "scaled:center")
pscale <- attr(pmatrix, "scaled:scale") ③
```

- ① Use all the columns except the first (Country)
- ② The output of scale() is a matrix. For the purposes of this chapter, you can think of a matrix as a data frame with all numeric columns (this isn't strictly true, but it's close enough).
- ③ The scale() function annotates its output with two attributes: scaled:center returns the mean values of all the columns; scaled:scale returns the standard deviations. We will store these away so we can "unscale" the data later.

Now on to clustering. We'll start with hierarchical.

8.1.3 Hierarchical clustering with `hclust`

The `hclust()` function takes as input a distance matrix (as an object of class `dist`), which records the distances between all pairs of points in the data (using any one of a variety of metrics). It returns a *dendrogram*: a tree that represents the nested clusters. `hclust()` uses one of a variety of clustering methods to produce a tree that records the nested cluster structure. You can compute the distance matrix using the function `dist()`.

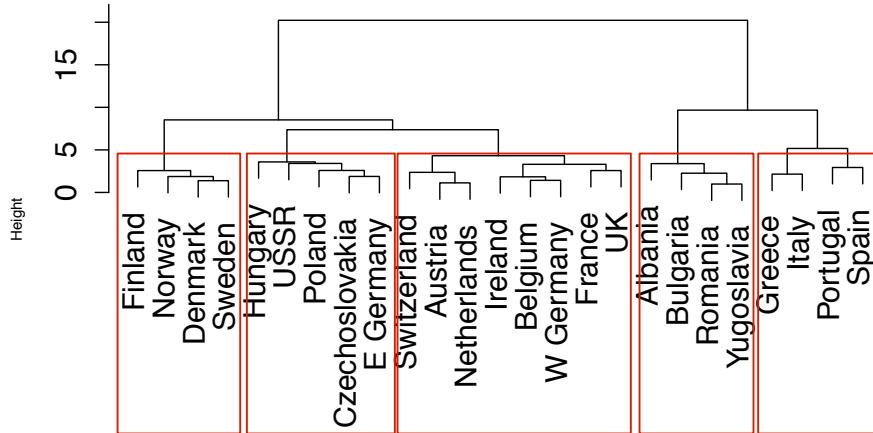
`dist()` will calculate distance functions using the (squared) euclidean distance (`method="euclidean"`), the Metropolis distance (`method="manhattan"`), and something like the Hamming distance, when categorical variables are expanded to indicators (`method="binary"`). If you want to use another distance metric, you will have to compute the appropriate distance matrix and convert it to a `dist` object using the `as.dist()` call. See the help pages for `dist` for further details.

Let's cluster the protein data. We will use Ward's method, which starts out with each data point as an individual cluster, and merges clusters iteratively so as to minimize the total within-sum-of-squares (WSS) of the clustering (we will explain more about WSS later in the chapter).

```
d <- dist(pmatrix, method="euclidean")      ①
pfit <- hclust(d, method="ward")            ②
plot(pfit, labels=protein$Country)          ③
```

- ① Create the distance matrix
- ② Do the clustering
- ③ Plot the dendrogram

Cluster Dendrogram



`hclust (*, "ward")`

Figure 8.2 Dendrogram of countries clustered by protein consumption

To my eye, the dendrogram suggests five clusters (as shown in Figure 8.2). You can draw the rectangles on the dendrogram using the function `rect.hclust()`.

```
rect.hclust(pfit, k=5)
```

To extract the members of each cluster from the `hclust` object, use `cutree()`

```
groups <- cutree(pfit, k=5)

print_clusters <- function(labels, k) {
  for(i in 1:k) {
    print(paste("cluster", i))
    print(protein[labels==i,c("Country", "RedMeat", "Fish", "Fr.Veg")])
  }
}

> print_clusters(groups, 5)
[1] "cluster 1"
   Country RedMeat Fish Fr.Veg
1   Albania     10.1  0.2    1.7
4   Bulgaria      7.8  1.2    4.2
18  Romania      6.2  1.0    2.8
```

```

25 Yugoslavia      4.4  0.6   3.2
[1] "cluster 2"
  Country RedMeat Fish Fr.Veg
2    Austria     8.9  2.1   4.3
3    Belgium    13.5  4.5   4.0
9    France     18.0  5.7   6.5
12   Ireland    13.9  2.2   2.9
14 Netherlands   9.5  2.5   3.7
21 Switzerland  13.1  2.3   4.9
22      UK       17.4  4.3   3.3
24   W Germany   11.4  3.4   3.8
[1] "cluster 3"
  Country RedMeat Fish Fr.Veg
5  Czechoslovakia  9.7  2.0   4.0
7    E Germany    8.4  5.4   3.6
11   Hungary     5.3  0.3   4.2
16    Poland     6.9  3.0   6.6
23    USSR       9.3  3.0   2.9
[1] "cluster 4"
  Country RedMeat Fish Fr.Veg
6  Denmark     10.6  9.9   2.4
8  Finland      9.5  5.8   1.4
15  Norway      9.4  9.7   2.7
20  Sweden      9.9  7.5   2.0
[1] "cluster 5"
  Country RedMeat Fish Fr.Veg
10 Greece      10.2  5.9   6.5
13 Italy        9.0  3.4   6.7
17 Portugal     6.2 14.2   7.9
19 Spain        7.1  7.0   7.2

```

- ➊ A convenience function for printing out the countries in each cluster, along with the values for red meat, fish, and fruit/vegetable consumption. We will use this function throughout this section. Note that the function is hard-coded for the protein dataset.

There's a certain logic to these clusters; the countries in each cluster tend to be in the same geographical region. It makes sense that countries in the same region would have similar dietary habits. We can also see that:

- Cluster 2 is made of countries with higher than average red meat consumption.
- Cluster 4 contains countries with higher than average fish consumption but low produce consumption.
- Cluster 5 contains countries with high fish and produce consumption.

This data set has only 25 points; it's harder to "eyeball" the clusters and the

cluster members when there are very many data points. In the next few sections we will look at some ways to examine clusters more holistically.

VISUALIZING CLUSTERS

As we mentioned in Chapter XRF:chapter_3:chExploringData, visualization is an effective way to get an overall view of the data, or in this case, the clusterings. We can try to visualize the clustering by projecting the data onto the first two *principal components* of the data². If N is the number of variables that describe the data, then the principal components describe the hyperellipsoid in N -space that bounds the data. If you order the principal components by the length of the hyperellipsoid's corresponding axes (longest first), then the first two principal components describe a plane in N -space that captures as much of the variation of the data as can be captured in two dimensions. We'll use the `prcomp()` call to do the principal components decomposition.

Footnote 2 We can project the data onto any two of the principal components, but the first two are the most likely to show useful information.

```
library(ggplot2)
princ <- prcomp(pmatrix)      ①
nComp <- 2
project <- predict(princ, newdata=pmatrix)[,1:nComp]      ②
project.plus <- cbind(as.data.frame(project),
                      cluster=as.factor(groups),
                      country=protein$Country)      ③
ggplot(project.plus, aes(x=PC1, y=PC2)) +
  geom_point(aes(shape=cluster)) +
  geom_text(aes(label=country),
            hjust=0, vjust=1)      ④
```

- ① Calculate the principal components of the data
- ② The `predict()` function will rotate the data into the space described by the principal components. We only want the projection on the the first two axes.
- ③ Create a data frame with the transformed data, along with the cluster label and country label of each point.
- ④ Plot it.

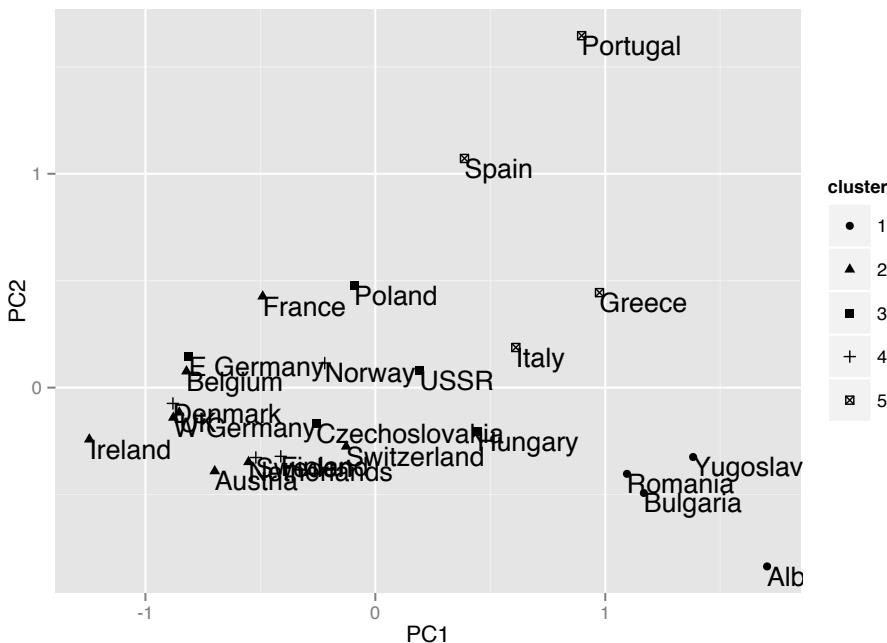


Figure 8.3 Plot of countries clustered by protein consumption, projected onto first two principal components.

You can see in Figure 8.3 that the "Romania/Yugoslavia/Bulgaria/Albania" cluster, and the "Mediterranean" cluster (Spain, etc.) are separated from the others. The other three clusters co-mingle in this projection, though they are probably more separated in other projections.

BOOTSTRAP EVALUATION OF CLUSTERS

An important question when evaluating clusters is whether or not a given cluster is "real:" does the cluster represent actual structure in the data, or is it an artifact of the clustering algorithm? As we will see, this is especially important with clustering algorithms like K-means, where the user has to specify the number of clusters *a priori*. It's been our experience that clustering algorithms will often produce several clusters that represent actual structure or relationships in the data, and then one or two clusters that are buckets that represent "Other," or "Miscellaneous." Clusters of "other" tend to be made up of data points that have no real relationship to each other; they just don't fit anywhere else.

One way to assess if a cluster represents true structure or not is to see if the cluster holds up under plausible variations in the dataset. The `fpc` package has a function called `clusterboot()` that uses bootstrap resampling to evaluate how

stable a given cluster is³. `clusterboot()` is an integrated function that both performs the clustering, and evaluates the final produced clusters. It has interfaces to a number of R clustering algorithms, including both `hclust` and `kmeans`.

Footnote 3 For a full description of the algorithm, see Henning, Christian, "Cluster-wise assessment of cluster stability," Research Report 271, Dept. of Statistical Science, University College London, December 2006. The report can be found online at <http://www.ucl.ac.uk/statistics/research/pdfs/rr271.pdf>.

`clusterboot`'s algorithm uses the *Jaccard coefficient*, a similarity measure between sets. The Jaccard similarity between two sets A and B is the ratio of the number of elements in the intersection of A and B over the number of elements in the union of A and B. The basic general strategy is as follows:

1. Cluster the data as usual.
2. Draw a new data set (of the same size as the original) by resampling the original data set with replacement ("with replacement" means that some of the data points may show up more than once, and others not at all). Cluster the new data set
3. For every cluster in the original clustering, find the most similar cluster in the new clustering (the one which gives the maximum Jaccard coefficient) and record that value. If this maximum Jaccard coefficient is less than 0.5, the original cluster is considered to be "dissolved" -- that is, it didn't show up in the new clustering. A cluster that is dissolved too often is probably not a "real" cluster.
4. Repeat steps 2-3 several times.

The cluster stability of each cluster in the original clustering is the mean value of its Jaccard coefficient over all the bootstrap iterations. As a rule of thumb, clusters with a stability value less than 0.6 should be considered unstable. Values between 0.6 and 0.75 indicate that the cluster is measuring a pattern in the data, but there is not high certainty about which points should be clustered together. Clusters with stability values above about 0.85 can be considered highly stable (that is, they are likely to be "real" clusters).

Different clustering algorithms can give different stability values, even when the two algorithms produce highly similar clusterings, so `clusterboot()` is also measuring how stable the clustering algorithm is, as well.

Let's run `clusterboot()` on the protein data, using hierarchical clustering with five clusters.

```
library(fpc)
①
kbest.p<-5
```

②

```

cboot.hclust <- clusterboot(pmatrix,clustermethod=hclustCBI,
                           method="ward", k=kbest.p) ③

> summary(cboot.hclust$result)
      Length Class Mode
result       7   hclust list
noise        1   -none- logical
nc          1   -none- numeric
clusterlist  5   -none- list
partition    25  -none- numeric
clustermethod 1   -none- character
nccl         1   -none- numeric

④

> groups<-cboot.hclust$result$partition
> print_clusters(groups, kbest.p) ⑤
[1] "cluster 1"
      Country RedMeat Fish Fr.Veg
1     Albania    10.1  0.2   1.7
4     Bulgaria    7.8   1.2   4.2
18    Romania     6.2   1.0   2.8
25 Yugoslavia    4.4   0.6   3.2

[1] "cluster 2"
      Country RedMeat Fish Fr.Veg
2     Austria     8.9   2.1   4.3
3     Belgium    13.5   4.5   4.0
9     France     18.0   5.7   6.5
12    Ireland    13.9   2.2   2.9
14 Netherlands    9.5   2.5   3.7
21 Switzerland   13.1   2.3   4.9
22        UK     17.4   4.3   3.3
24 W Germany    11.4   3.4   3.8

[1] "cluster 3"
      Country RedMeat Fish Fr.Veg
5 Czechoslovakia  9.7   2.0   4.0
7 E Germany      8.4   5.4   3.6
11 Hungary        5.3   0.3   4.2
16 Poland         6.9   3.0   6.6
23 USSR           9.3   3.0   2.9

[1] "cluster 4"
      Country RedMeat Fish Fr.Veg
6 Denmark        10.6  9.9   2.4
8 Finland         9.5  5.8   1.4
15 Norway         9.4  9.7   2.7
20 Sweden         9.9  7.5   2.0

[1] "cluster 5"
      Country RedMeat Fish Fr.Veg
10 Greece         10.2  5.9   6.5
13 Italy          9.0  3.4   6.7
17 Portugal       6.2 14.2   7.9
19 Spain          7.1  7.0   7.2

⑥

> cboot.hclust$bootmean ⑦
[1] 0.7905000 0.7990913 0.6173056 0.9312857 0.7560000

> cboot.hclust$bootbrd ⑧
[1] 25 11 47  8 35

```

- ① Load the fpc package. You may have to install it first. We discuss installing R packages in Appendix .
- ② Set the desired number of clusters.
- ③ Run clusterboot() with hclust ('clustermethod=hclustCBI') using Ward's method ('method="ward"') and kbest.p clusters ('k=kbest.p'). Return the results in an object called cboot.hclust
- ④ The results of the clustering are in cboot.hclust\$result. The output of the hclust() function is in cboot.hclust\$result\$result.
- ⑤ cboot.hclust\$result\$partition returns a vector of clusterlabels.
- ⑥ The clusters are the same as those produced by a direct call to hclust().
- ⑦ The vector of cluster stabilities.
- ⑧ The count of how many times each cluster was dissolved. By default clusterboot() runs 100 bootstrap iterations.

The `clusterboot()` results show that the cluster of countries with high fish consumption (cluster 4) is highly stable. Clusters 1 and 2 are also quite stable, cluster 5 less so (we can see in Figure 8.4 that the members of cluster 5 are separated from the other countries, but also fairly separated from each other). Cluster 3 has the characteristics of what we've been calling the "Other" cluster.

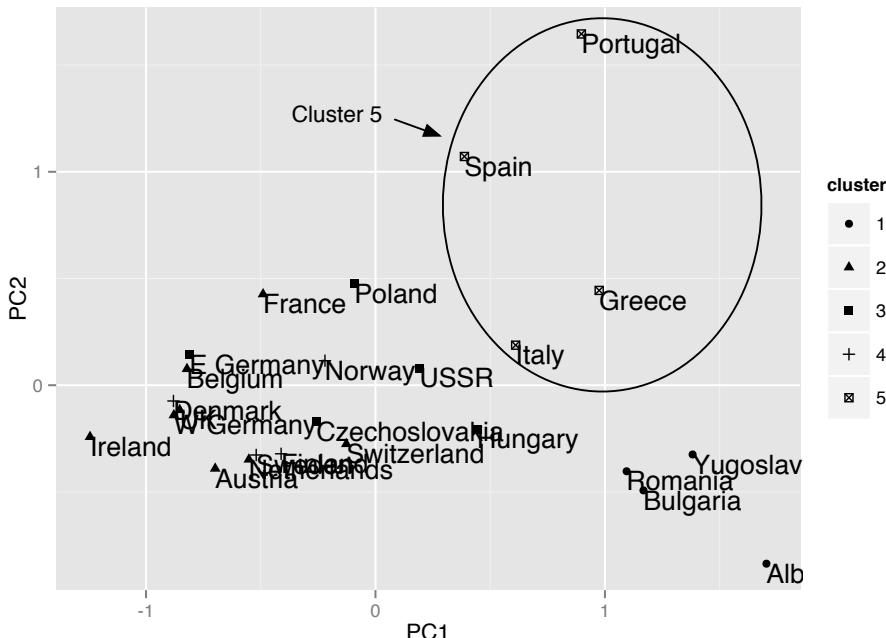


Figure 8.4 Cluster 5: The "Mediterranean" cluster. Its members are separated from the other clusters, but also from each other.

`clusterboot()` assumes that you know the number of clusters, k . We

eyeballed the appropriate k from the dendrogram, but this is not always feasible with a large data set. Can we pick a plausible k in a more automated fashion? We'll look at this question in the next section.

PICKING THE NUMBER OF CLUSTERS

There are a number of heuristics and rules-of-thumb for picking clusters; a given heuristic will work better on some data sets than others. It's best to take advantage of domain knowledge to help set the number of clusters, if that is possible. Otherwise, try a variety of heuristics, and perhaps a few different values of k .

Within Sum of Squares

One simple heuristic is to compute the *total within sum of squares* for different values of k and look for an "elbow" in the curve. Define the cluster's *centroid* as the point that is the mean value of all the points in the cluster. The *within sum of squares* (WSS) for a single cluster is the average squared distance of each point in the cluster from the cluster's centroid. The total within sum of squares is the sum of the within sum of squares of all the clusters. We show the calculation in the listing below.

```
sqr_edist <- function(x, y) {  
  sum((x-y)^2)  
}  
  
wss.cluster <- function(clustermat) {  
  c0 <- apply(clustermat, 2, FUN=mean) ②  
  sum(apply(clustermat, 1, FUN=function(row){sqr_edist(row,c0)})) ④  
}  
  
wss.total <- function(dmatrix, labels) {  
  wsstot <- 0  
  k <- length(unique(labels))  
  for(i in 1:k)  
    wsstot <- wsstot + wss.cluster(subset(dmatrix, labels==i)) ⑥  
  wsstot  
}
```

- ① Function to calculate squared distance between two vectors.
- ② A function to calculate the WSS for a single cluster, which is represented as a matrix (one row for every point)
- ③ Calculate the centroid of the cluster (the mean of all the points).
- ④ Calculate the squared difference of every point in the cluster from the centroid, and

sum all the distances.

- ⑤ Function to compute the total WSS from a set of data points and cluster labels.
- ⑥ Extract each cluster, calculate the cluster's WSS, and sum all the values.

The total WSS will decrease as the number of clusters increases, because each cluster will be smaller and tighter. The hope is that the rate at which the WSS decreases will slow down for k beyond the optimal number of clusters. In other words, the graph of WSS versus k should flatten out beyond the optimal k , so the optimal k will be at the "elbow" of the graph. Unfortunately, this elbow can be difficult to see.

Calinski-Harabasz Index

The *Calinski-Harabasz index* of a clustering is the ratio of the between-cluster variance (which is essentially the variance of all the cluster centroids from the dataset's grand centroid) to the total within-cluster variance (basically the average WSS of the clusters in the clustering). For a given data set, the total sum of squares (TSS) is the squared distance of all the data points from the data set's centroid. The TSS is independent of the clustering. If $\text{WSS}(k)$ is the total WSS of a clustering with k clusters, then the *between sum of squares* $\text{BSS}(k)$ of the clustering is given by $\text{BSS}(k) = \text{TSS} - \text{WSS}(k)$. $\text{WSS}(k)$ measures how close the points in a cluster are to each other. $\text{BSS}(k)$ measures how far apart the clusters are from each other. A good clustering has small $\text{WSS}(k)$ and large $\text{BSS}(k)$.

The within-cluster variance W is given by $\text{WSS}(k) / (n-k)$, where n is the number of points in the data set. The between-cluster variance B is given by $\text{BSS}(k) / (k-1)$. The within-cluster variance will decrease as k increases; the rate of decrease should slow down past the optimal k . The between-cluster variance will increase as k , but the rate of increase should slow down past the optimal k . So in theory, the ratio of B to W should be maximized at the optimal k .

Let's write a function to calculate the Calinski-Harabasz index. The function will accommodate both a `kmeans` clustering and an `hclust` clustering.

```
totss <- function(dmatrix) {  
  grandmean <- apply(dmatrix, 2, FUN=mean)  
  sum(apply(dmatrix, 1, FUN=function(row){sqr_edist(row, grandmean)}))  
}  
  
ch_criterion <- function(dmatrix, kmax, method="kmeans") {  
  if(!(method %in% c("kmeans", "hclust"))) {  
    stop("method must be 'kmeans' or 'hclust'")  
  }  
  totss(dmatrix)  
  ch_index <- numeric(kmax)  
  for(k in 1:kmax) {  
    if(k == 1) {  
      ch_index[k] <- totss(dmatrix)  
    } else {  
      # Compute the between-cluster variance B  
      # Compute the within-cluster variance W  
      # Compute the Calinski-Harabasz index CH  
      # Add CH to ch_index  
    }  
  }  
  ch_index  
}
```

```

stop("method must be one of c('kmeans', 'hclust')")
}
npts <- dim(dmatrix)[1] # number of rows.

totss <- totss(dmatrix) 3

wss <- numeric(kmax)
crit <- numeric(kmax)

wss[1] <- (npts-1)*sum(apply(dmatrix, 2, var)) 4

for(k in 2:kmax) {
  if(method=="kmeans") {
    clustering<-kmeans(dmatrix, k, nstart=10, iter.max=100)
    wss[k] <- clustering$tot.withinss
  } else { # hclust
    d <- dist(dmatrix, method="euclidean")
    pfit <- hclust(d, method="ward")
    labels <- cutree(pfit, k=k)
    wss[k] <- wss.total(dmatrix, labels)
  }
}

bss <- totss - wss 7

crit.num <- bss/(0:(kmax-1)) 8

crit.denom <- wss/(npts - 1:kmax) 9

list(crit = crit.num/crit.denom, wss = wss, totss = totss) 10
}

```

- 1 Convenience function to calculate the total sum of squares.
- 2 A function to calculate the Calinski-Harabasz index for a number of clusters from 1 to kmax.
- 3 The total sum of squares is independent of the clustering.
- 4 Calculate WSS for k=1 (which is really just total sum of squares).
- 5 Calculate WSS for k from 2 to kmax. kmeans() returns the total WSS as one of its outputs.
- 6 For hclust(), calculate total WSS by hand.
- 7 Calculate BSS for k from 1 to kmax.
- 8 Normalize BSS by k-1
- 9 Normalize WSS by npts - k
- 10 Return a vector of Calinski-Harabasz indices and of WSS for k from 1 to kmax.
Also return total sum of squares.

We can calculate both indices for the `protein` data set and plot them.

```

library(reshape2) 1
clustcrit <- ch_criterion(pmatrix, 10, method="hclust") 2
critframe <- data.frame(k=1:10, ch=scale(clustcrit$crit), 3

```

```

wss=scale(clustcrit$wss))
critframe <- melt(critframe, id.vars=c("k"),
                   variable.name="measure",
                   value.name="score")
ggplot(critframe, aes(x=k, y=score, color=measure)) +
  geom_point(aes(shape=measure)) + geom_line(aes(linetype=measure)) +
  scale_x_continuous(breaks=1:10, labels=1:10)

```

- ➊ Load the reshape2 package (for the melt() function).
- ➋ Calculate both criteria for one to ten clusters.
- ➌ Create a data frame with the number of clusters, the CH criterion, and the WSS criterion. We'll scale both the CH and WSS criteria to similar ranges so that we can plot them both on the same graph.
- ➍ Use the melt() function to put the data frame in a shape suitable for ggplot
- ➎ Plot it.

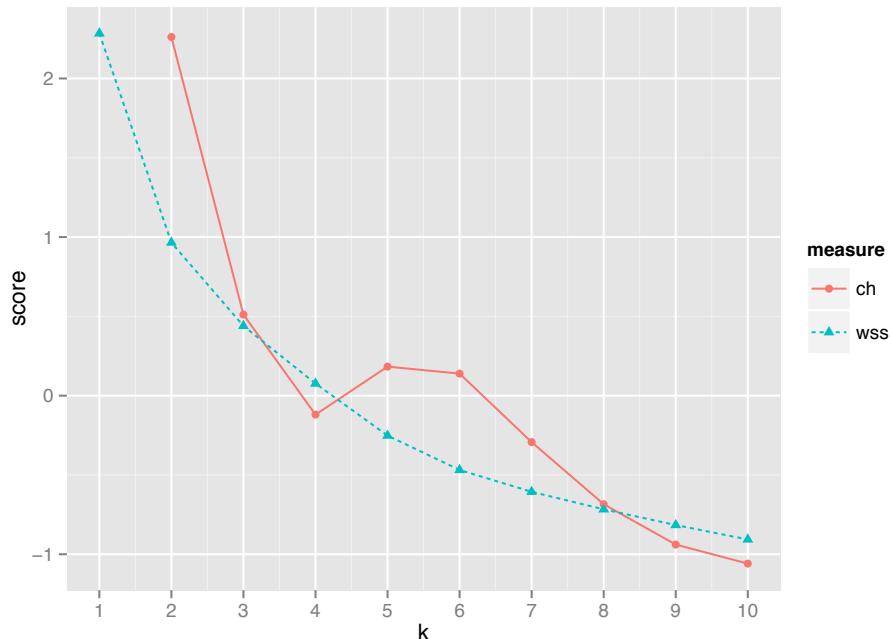


Figure 8.5 Plot of the Calinski-Harabasz and WSS indices for one to ten clusters, on protein data.

Looking at Figure 8.5, you see that the CH criterion is maximized at $k=2$, with another local maximum at $k=5$. If you squint your eyes, you can convince yourself that the WSS plot has an elbow at $k=2$. The $k=2$ clustering corresponds to the first split of the dendrogram in Figure 8.2; if you use `clusterboot()` to do the clustering, you will see that the clusters are highly stable, though perhaps not very informative.

There are several other indices that you can try when picking k . The *gap statistic*⁴ is an attempt to automate the "elbow finding" on the WSS curve. It works best when the data comes from a mix of populations that all have approximately gaussian distributions (a "mixture of gaussians"). We will see one more measure, the *average silhouette width*, when we discuss `kmeans()`.

Footnote 4 Tibshirani, Robert, Guenther Walther and Trevor Hastie. "Estimating the number of clusters in a data set via the gap statistic," *Jou. of the Royal Statistical Society B*, 63(2), pp. 411-423, 2001. Online: www.stanford.edu/~hastie/Papers/gap.pdf

8.1.4 The K-means algorithm

K-means is a popular clustering algorithm when the data is all numeric and the distance metric is squared euclidean (though you could in theory run it with other distance metrics). It's fairly ad-hoc, and has the major disadvantage that you must pick k in advance. On the plus side, it's easy to implement (one reason it's so popular), and can be faster than hierarchical clustering on large data sets. It works best on data that looks like a mixture of gaussians (the `protein` data does not).

THE `KMEANS()` FUNCTION

The function to run k-means in R is called `kmeans()`. The output of `kmeans()` includes the cluster labels, the centers (also called *centroids*) of the clusters, the total sum of squares, total WSS, total BSS, and the WSS of each cluster.

The k-means algorithm is illustrated in Figure 8.6, with $k = 2$.

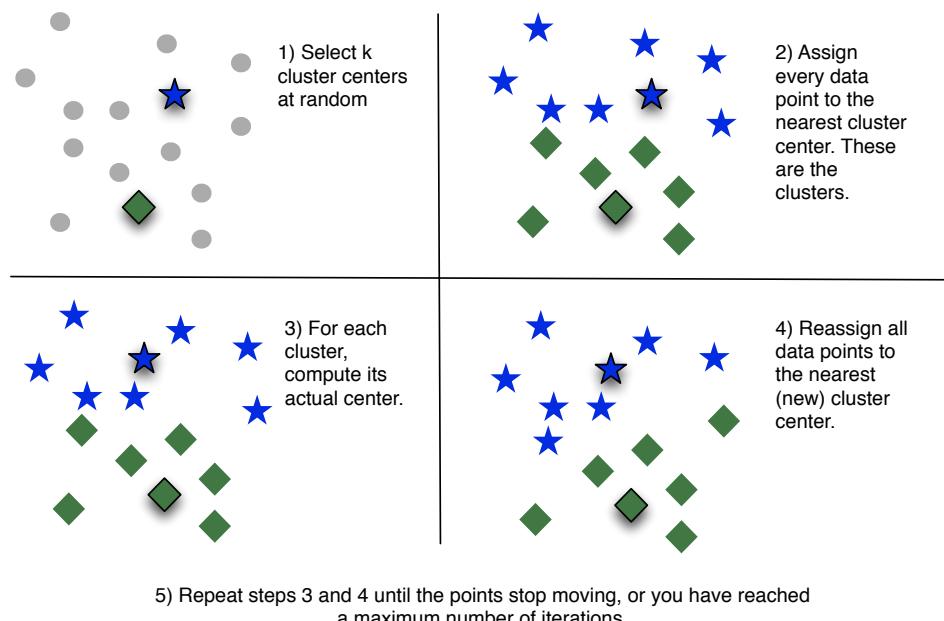


Figure 8.6 The k-means procedure. The two cluster centers are represented by the shadowed star and diamond.

This algorithm is not guaranteed to have a unique stopping point. K-means can be fairly unstable, in that the final clusters depend on the initial cluster centers. It's good practice to run K-means several times with different random starts, then select the clustering with the lowest total WSS. The `kmeans()` function can do this automatically, though it defaults to only using one random start.

Let's run `kmeans()` on the `protein` data (scaled to zero mean and unit standard deviation, as before). We'll use `k=5`.

```
> pclusters <- kmeans(pmatrix, kbest.p, nstart=100, iter.max=100) ①
> summary(pclusters)
```

	Length	Class	Mode
cluster	25	-none-	numeric
centers	45	-none-	numeric
totss	1	-none-	numeric
withinss	5	-none-	numeric
tot.withinss	1	-none-	numeric
betweenss	1	-none-	numeric
size	5	-none-	numeric

```
> pclusters$centers ③
   RedMeat WhiteMeat      Eggs      Milk      Fish
1 -0.807569986 -0.8719354 -1.55330561 -1.0783324 -1.0386379
2  0.006572897 -0.2290150  0.19147892  1.3458748  1.1582546
3 -0.570049402  0.5803879 -0.08589708 -0.4604938 -0.4537795
4  1.011180399  0.7421332  0.94084150  0.5700581 -0.2671539
5 -0.508801956 -1.1088009 -0.41248496 -0.8320414  0.9819154
   Cereals    Starch     Nuts   Fr.Veg
1  1.7200335 -1.4234267  0.9961313 -0.64360439
2 -0.8722721  0.1676780 -0.9553392 -1.11480485
3  0.3181839  0.7857609 -0.2679180  0.06873983
4 -0.6877583  0.2288743 -0.5083895  0.02161979
5  0.1300253 -0.1842010  1.3108846  1.62924487
```

```
> pclusters$size ④
[1] 4 4 5 8 4
```

```
> groups <- pclusters$cluster
> print_clusters(groups, kbest.p)
```

	Country	RedMeat	Fish	Fr.Veg
1	Albania	10.1	0.2	1.7
4	Bulgaria	7.8	1.2	4.2
18	Romania	6.2	1.0	2.8
25	Yugoslavia	4.4	0.6	3.2

	Country	RedMeat	Fish	Fr.Veg
6	Denmark	10.6	9.9	2.4
8	Finland	9.5	5.8	1.4

```

15 Norway      9.4 9.7   2.7
20 Sweden      9.9 7.5   2.0
[1] "cluster 3"
    Country RedMeat Fish Fr.Veg
5  Czechoslovakia     9.7  2.0   4.0
7   E Germany       8.4  5.4   3.6
11    Hungary        5.3  0.3   4.2
16    Poland         6.9  3.0   6.6
23    USSR          9.3  3.0   2.9
[1] "cluster 4"
    Country RedMeat Fish Fr.Veg
2    Austria        8.9  2.1   4.3
3    Belgium       13.5  4.5   4.0
9    France        18.0  5.7   6.5
12   Ireland       13.9  2.2   2.9
14 Netherlands     9.5  2.5   3.7
21 Switzerland    13.1  2.3   4.9
22    UK           17.4  4.3   3.3
24  W Germany      11.4  3.4   3.8
[1] "cluster 5"
    Country RedMeat Fish Fr.Veg
10   Greece        10.2  5.9   6.5
13   Italy          9.0  3.4   6.7
17 Portugal       6.2 14.2   7.9
19   Spain          7.1  7.0   7.2

```

- ➊ Run kmeans() with five clusters, 100 random starts and 100 maximum iterations per run.
- ➋ kmeans() returns all the sum of squares measures
- ➌ pclusters\$centers is a matrix whose rows are the centroids of the clusters. Note that centers is in the scaled coordinates, not the original protein coordinates.
- ➍ pclusters\$size returns the number of points in each cluster. Generally (though not always) a good clustering will be fairly well balanced: no extremely small clusters and no extremely large ones.
- ➎ pclusters\$cluster is a vector of cluster labels.
- ➏ In this case, kmeans() and hclust() returned the same clustering. This will not always be true.

THE KMEANSRUNS() FUNCTION FOR PICKING K

To run `kmeans()`, you must know `k`. The `fpc` package (the same package that has `clusterboot()`) has a function called `kmeansruns()` that calls `kmeans()` over a range of `k` and estimates the best `k`. It then returns its pick for the best value of `k`, the output of `kmeans()` for that value, and a vector of criterion values as a function of `k`. Currently, `kmeansruns()` has two criteria: the Calinski-Harabasz Index ("ch"), and the *average silhouette width* ("asw")⁵. It's a good idea to plot the criterion values over the entire range of `k`, since you may see evidence for a `k` that the algorithm didn't automatically pick (as we did in Figure 8.5).

Footnote 5 [http://en.wikipedia.org/wiki/Silhouette_\(clustering\)](http://en.wikipedia.org/wiki/Silhouette_(clustering))

```
> clustering.ch <- kmeansruns(pmatrix, krange=1:10, criterion="ch") ①
> clustering.ch$bestk
[1] 2 ②
> clustering.asw <- kmeansruns(pmatrix, krange=1:10, criterion="asw") ③
> clustering.asw$bestk
[1] 3

> clustering.ch$crit
[1] 0.000000 14.094814 11.417985 10.418801 10.011797 9.964967 ④
[7] 9.861682 9.412089 9.166676 9.075569

> clustcrit$crit
[1]       Nan 12.215107 10.359587 9.690891 10.011797 9.964967 ⑤
[7] 9.506978 9.092065 8.822406 8.695065

> critframe <- data.frame(k=1:10, ch=scale(clustering.ch$crit), ⑥
  asw=scale(clustering.asw$crit))
> critframe <- melt(critframe, id.vars=c("k"),
  variable.name="measure",
  value.name="score")
> ggplot(critframe, aes(x=k, y=score, color=measure)) +
  geom_point(aes(shape=measure)) + geom_line(aes(linetype=measure)) +
  scale_x_continuous(breaks=1:10, labels=1:10)
> summary(clustering.ch) ⑦
      Length Class Mode
cluster     25   -none- numeric
centers      18   -none- numeric
totss        1   -none- numeric
withinss      2   -none- numeric
tot.withinss  1   -none- numeric
betweenss     1   -none- numeric
size          2   -none- numeric
crit         10   -none- numeric
```

```
bestk      1      -none- numeric
```

- 1 Run kmeansruns() from one to ten clusters, and the CH criterion. By default, kmeansruns() uses 100 random starts and 100 maximum iterations per run.
- 2 The CH criterion picks two clusters.
- 3 Run kmeansruns() from one to ten clusters, and the average silhouette width criterion. Average silhouette width picks 3 clusters.
- 4 The vector of criterion values is called crit.
- 5 Compare the CH values for kmeans() and hclust(). They are not quite the same, because the two algorithms did not pick the same clusters.
- 6 Plot the values for the two criteria.
- 7 kmeansruns() also returns the output of kmeans for k=bestk.

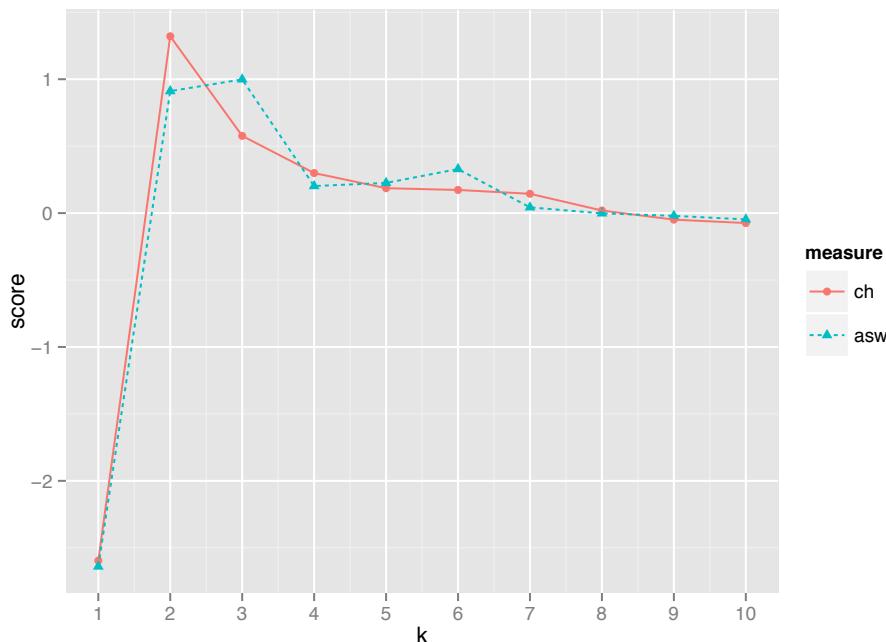


Figure 8.7 Plot of the Calinski-Harabasz and Average Silhouette Width indices for one to ten clusters, on protein data.

Figure 8.7 shows the results of the two clustering criteria provided by `kmeansruns`. They suggest two to three clusters as the best choice. If you compare the values of `clustering.ch$crit` and `clustcrit$crit` in the listing above, you will see that the CH criterion produces different curves for `kmeans()` and `hclust()` clusterings; however, it did pick the same value (which probably means it picked the same clusters) for `k=5`, and `k=6`, which might be taken as evidence that either five or six is the optimal choice for `k`.

CLUSTERBOOT() REVISITED

We can run `clusterboot()` using the k-means algorithm, as well.

```
kbest.p<-5
cboot<-clusterboot(pmatrix, clustermethod=kmeansCBI,
                     runs=100,iter.max=100,
                     krange=kbest.p, seed=15555) ①

> groups <- cboot$result$partition
> print_clusters(cboot$result$partition, kbest.p)
[1] "cluster 1"
  Country RedMeat Fish Fr.Veg
1  Albania    10.1  0.2   1.7
4  Bulgaria     7.8  1.2   4.2
18 Romania      6.2  1.0   2.8
25 Yugoslavia    4.4  0.6   3.2

[1] "cluster 2"
  Country RedMeat Fish Fr.Veg
6  Denmark     10.6  9.9   2.4
8  Finland      9.5  5.8   1.4
15 Norway       9.4  9.7   2.7
20 Sweden       9.9  7.5   2.0

[1] "cluster 3"
  Country RedMeat Fish Fr.Veg
5  Czechoslovakia  9.7  2.0   4.0
7  E Germany      8.4  5.4   3.6
11 Hungary        5.3  0.3   4.2
16 Poland         6.9  3.0   6.6
23 USSR           9.3  3.0   2.9

[1] "cluster 4"
  Country RedMeat Fish Fr.Veg
2  Austria       8.9  2.1   4.3
3  Belgium        13.5  4.5   4.0
9  France         18.0  5.7   6.5
12 Ireland        13.9  2.2   2.9
14 Netherlands    9.5  2.5   3.7
21 Switzerland    13.1  2.3   4.9
22 UK             17.4  4.3   3.3
24 W Germany      11.4  3.4   3.8

[1] "cluster 5"
  Country RedMeat Fish Fr.Veg
10 Greece        10.2  5.9   6.5
13 Italy          9.0  3.4   6.7
17 Portugal       6.2  14.2   7.9
19 Spain          7.1  7.0   7.2

> cboot$bootmean
[1] 0.8670000 0.8420714 0.6147024 0.7647341 0.7508333
> cboot$bootbrd
[1] 15 20 49 17 32
```

 We've set the seed for the random generator so the results are reproducible.

Note that the stability numbers (and the number of times that the clusters were "dissolved") are different for the hierarchical clustering and K-means, even though the discovered clusters are the same. This shows that the stability of a clustering is partly a function of the clustering algorithm, not just the data. Again, the fact that both clustering algorithms discovered the same clusters might be taken as an indication that five is the optimal number of clusters.

8.1.5 Assigning new points to clusters

Clustering is often used as part of data exploration, or as a precursor to other supervised learning methods. However, you may want to use the clusters that you discovered to categorize new data, as well. One common way to do so is to treat the centroid of each cluster as the representative of the cluster as a whole, and then assign new points to the cluster with the nearest centroid. Note that if you scaled the original data before clustering, then you should also scale the new data point the same way, before assigning it to a cluster.

```
assign_cluster <- function(newpt, centers, xcenter=0, xscale=1) { ①
  xpt <- (newpt - xcenter)/xscale ②
  dists <- apply(centers, 1, FUN=function(c0){sqr_edist(c0, xpt)}) ③
  which.min(dists) ④
}
```

- ① A function to assign a new data point `newpt` to a clustering described by `centers`, a matrix where each row is a cluster centroid. If the data was scaled (using `scale()`) before clustering, then `xcenter` and `xscale` are the `scaled:center` and `scaled:scale` attributes, respectively.
- ② Center and scale the new data point.
- ③ Calculate how far the new data point is from each of the cluster centers.
- ④ Return the cluster number of the closest centroid.

Note that the function `sqr_edist` (the squared euclidean distance) was defined previously, in Section 8.1.1.

Let's show an example of assigning points to clusters, using synthetic data.

```
rnorm.multidim <- function(n, mean, sd, colstr="x") { ①
```

```

ndim <- length(mean)
data <- NULL
for(i in 1:ndim) {
  col <- rnorm(n, mean=mean[[i]], sd=sd[[i]])
  data<-cbind(data, col)
}
cnames <- paste(colstr, 1:ndim, sep=' ')
colnames(data) <- cnames
data
}

mean1 <- c(1, 1, 1) 2
sd1 <- c(1, 2, 1)

mean2 <- c(10, -3, 5)
sd2 <- c(2, 1, 2)

mean3 <- c(-5, -5, -5)
sd3 <- c(1.5, 2, 1)

clust1 <- rnorm.multidim(100, mean1, sd1) 3
clust2 <- rnorm.multidim(100, mean2, sd2)
clust3 <- rnorm.multidim(100, mean3, sd3)
toydata <- rbind(clust3, rbind(clust1, clust2))

tmatrix <- scale(toydata) 4
tcenter <- attr(tmatrix, "scaled:center") 5
tscale<-attr(tmatrix, "scaled:scale")
kbest.t <- 3
tclusters <- kmeans(tmatrix, kbest.t, nstart=100, iter.max=100) 6

tclusters$size 7
[1] 100 101 99

unscale <- function(scaledpt, centervec, scalevec) { 8
  scaledpt*scalevec + centervec
}

> unscale(tclusters$centers[1,], tcenter, tscale) 9
  x1          x2          x3
  9.978961 -3.097584  4.864689
> mean2
[1] 10 -3  5

> unscale(tclusters$centers[2,], tcenter, tscale) 10
  x1          x2          x3
 -4.979523 -4.927404 -4.908949
> mean3
[1] -5 -5 -5

> unscale(tclusters$centers[3,], tcenter, tscale) 11
  x1          x2          x3

```

```

1.0003356 1.3037825 0.9571058
> mean1
[1] 1 1 1

> assign_cluster(rnorm.multidim(1, mean1, sd1),      ⑫
                  tclusters$centers,
                  tcenter, tscale)
3
3

> assign_cluster(rnorm.multidim(1, mean2, sd1),      ⑬
                  tclusters$centers,
                  tcenter, tscale)
1
1

> assign_cluster(rnorm.multidim(1, mean3, sd1),      ⑭
                  tclusters$centers,
                  tcenter, tscale)
2
2

```

- ① A function to generate n points drawn from a multidimensional gaussian distribution with centroid mean and standard deviation sd. The dimension of the distribution is given by the length of the vector mean.
- ② The parameters for three gaussian distributions.
- ③ Create a dataset with one hundred points each drawn from the above distributions.
- ④ Scale the dataset.
- ⑤ Store the centering and scaling parameters for future use.
- ⑥ Cluster the dataset, using K-means with three clusters.
- ⑦ The resulting clusters are about the right size.
- ⑧ A function to "unscale" datapoints (put them back in the coordinates of the original data set).
- ⑨ Unscale the first centroid. It corresponds to our original distribution 2.
- ⑩ The second centroid corresponds to the original distribution 3.
- ⑪ The third centroid corresponds to the original distribution 1.
- ⑫ Generate a random point from the original distribution 1 and assign it to one of the discovered clusters.
- ⑬ It is assigned to cluster 3 as we would expect.
- ⑭ Generate a random point from the original distribution 2 and assign it.
- ⑮ It is assigned to cluster 1.
- ⑯ Generate a random point from the original distribution 3 and assign it
- ⑰ It is assigned to cluster 2.

8.1.6 Clustering Takeaways

What you should remember about clustering:

- The goal of clustering is to discover or draw out similarities among subsets of your data.
- In a good clustering, points in the same cluster should be more similar (nearer) to each other than they are to points in other clusters.
- When clustering, the units that each variable is measured in matter. Different units cause different distances and potentially different clusterings.
- Ideally, you want a unit change in each coordinate to represent the same degree of change. One way to approximate this is to transform all the columns to have a mean value of zero and a standard deviation of one, for example by using the function `scale()`.
- Clustering is often used for data exploration, or as a precursor to supervised learning methods.
- Like visualization, it is more iterative and interactive, less automated than supervised methods.
- Different clustering algorithms will give different results. You should consider different approaches, with different numbers of clusters.
- There are many heuristics for estimating the best number of clusters. Again, you should consider the results from different heuristics, and explore various numbers of clusters.

Sometimes, rather than looking for subsets of data points that are highly similar to each other, you would like to know what kind of data (or which data attributes) tend to occur together. In the next section, we will look at one approach to this problem.

8.2 Association Rules

Association rule mining is used to find objects or attributes that frequently occur together: for example, products that are often bought together during a shopping session, or queries that tend to occur together during a session on a web site's search engine. Such information can be used to recommend products to shoppers, to place frequently bundled items together on store shelves, or redesign web sites for easier navigation.

8.2.1 Overview of Association Rules

The unit of "togetherness" when mining association rules is called a *transaction*. Depending on the problem, a transaction could be a single shopping basket, a single user session on a website, or even a single customer. The objects that comprise a transaction are referred to as *items* in an *itemset*: the products in the shopping basket, the pages visited during a website session, the actions of a customer. Sometimes transactions are referred to as "baskets," from the shopping basket analogy.

Mining for association rules occurs in two steps.

1. Look for all the itemsets (subsets of transactions) that occur more than in a minimum fraction of the transactions

2. Turn those itemsets into rules

Let's consider the example of books that are checked out from a library. When a library patron checks out a set of books, that is a transaction; the books that the patron checked out are the itemset that comprise the transaction. Table 8.1 represents a database of transactions.

Table 8.1 A Database of Library Transactions

Transaction ID	Books Checked Out
1	The Hobbit, The Princess Bride
2	The Princess Bride, The Last Unicorn
3	The Hobbit
4	The Neverending Story
5	The Last Unicorn
6	The Hobbit, The Princess Bride, The Fellowship of the Ring
7	The Hobbit, The Fellowship of the Ring, The Two Towers, The Return of the King
8	The Fellowship of the Ring, The Two Towers, The Return of the King
9	The Hobbit, The Princess Bride, The Last Unicorn
10	The Last Unicorn, The Neverending Story

Looking over all the transactions in Table 8.1, you find that *The Hobbit* is in 50% of all transactions, and *The Princess Bride* is in 40% of them (you run a library where fantasy is quite popular). Both books are checked out together in 30% of all transaction. We would say the *support* of the itemset $\{\text{The Hobbit}, \text{The Princess Bride}\}$ is 30%. Of the five transactions that include *The Hobbit*, three (60%) also include *The Princess Bride*. So you can make a rule "People who check out *The Hobbit* also check out *The Princess Bride*." This rule should be correct (according to your data) 60% of the time. We would say that the *confidence* of the rule is 60%. Conversely, of the four times *The Princess Bride* was checked out, *The Hobbit* appeared three times, or 75%. So the rule "People who check out *The Princess Bride* also check out *The Hobbit*" has 75% confidence.

Let's define support and confidence formally. A rule "if X, then Y" means that every time you see the itemset X in a transaction, you expect to also see Y (with a given confidence). For the apriori algorithm (which we will look at in this section), Y is always an itemset with one item. Suppose that your database of transactions is called T. Then $\text{support}(X)$ is the number of transactions that contain X divided by the total number of transactions in T. The confidence of the rule "if X, then Y" is given by $\text{conf}(X \Rightarrow Y) = \text{support}(\text{union}(X, Y)) / \text{support}(X)$, where $\text{union}(X, Y)$ means that we are referring to itemsets that contain both the items in X and the items in Y.

The goal in association rule mining is to find all the "interesting" rules in the database with at least a given minimum support (say 10%) and a minimum given confidence (say 60%).

8.2.2 The Example Problem

For our example problem, let's imagine that we are working for a bookstore, and we are interested in identifying books that our customers are interested in, based on (all of) their previous purchases and book interests. We can get information about their book interests two ways: either they have purchased a book from us, or they have rated the book on our website (even if they bought the book somewhere else). In this case, a transaction is a customer, and an itemset is all the books that they have expressed an interest in, either by purchase or by rating.

The data that we will use is based on data collected in 2004, from the book community Book-Crossing⁶, for research conducted at the Institut für Informatik, University of Freiburg. We have condensed the information into a single tab-separated text file called `bookdata.tsv`. Each row of the file consists of a

user id, a book title (which we've designed as a unique id for each book) and the rating (which we won't actually use in this example).

Footnote 6 The original data repository can be found at <http://www.informatik.uni-freiburg.de/~cziegler/BX/>. Since some artifacts in the original files caused errors when reading into R, we are providing copies of the raw data as an Rdata object:<https://github.com/WinVector/zmPDSwR/blob/master/Bookdata/bxBooks.RData>. The prepared version of the data that we will use in this section is <https://github.com/WinVector/zmPDSwR/blob/master/Bookdata/bookdata.tsv>. Further information and scripts for preparing the data are can be found at <https://github.com/WinVector/zmPDSwR/tree/master/Bookdata>. The researchers' original paper is "Improving Recommendation Lists Through Topic Diversification", Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, Georg Lausen; *Proceedings of the 14th International World Wide Web Conference (WWW '05)*, May 10-14, 2005, Chiba, Japan. It can be found online at <http://www.informatik.uni-freiburg.de/~cziegler/BX/WWW-2005-Preprint.pdf>.

```
"token" "userid"      "rating"      "title"  
" a light in the storm" 55927    0      " A Light in the Storm"
```

The `token` column contains lower-cased column strings; we used the tokens to identify books with different ISBNs (the original book ids) that had the same title except for casing. The `title` column holds properly capitalized title strings; these are unique per book, so we will use them as book ids.

In this format, the transaction (customer) information, is diffused through the data, rather than being all in one row; this reflects the way the data would naturally be stored in a database, since the customer's activity would be diffused throughout time. Books generally come in different editions, or from different publishers; we've condensed all different versions into a single item: hence different copies or printings of *Little Women* will all map to the same item id in our data (namely, the title "Little Women").

The original data includes approximately a million ratings from 278,858 readers about 271,379 books. Our data will have fewer books, due to the mapping that we discussed above.

Now we are ready to mine.

8.2.3 Mining Association Rules with the `arules` package.

We will use the package `arules` for association rule mining. `arules` includes an implementation of the association rule algorithm *apriori*, as well as implementations to read in and examine transaction data⁷. The package uses special datatypes to hold and manipulate the data; we will explore these data types as we work the example.

Footnote 7 For a more comprehensive introduction to `arules` than we can give in this chapter, please see Hahsler, Grin, Hornik and Buchta, "Introduction to arules - A computational environment for mining association rules and frequent item sets," online at cran.r-project.org/web/packages/arules/vignettes/arules.pdf

READING IN THE DATA

We can read the data directly from the `bookdata.tsv` file, into the object `bookbaskets` using the function `read.transaction()`.

```
library(arules)    ①
bookbaskets <- read.transactions("bookdata.tsv", format="single",
                                 sep="\t",          ②
                                 cols=c("userid", "title"), ③
                                 rm.duplicates=T)        ④⑤
```

- ① Load the `arules` package.
- ② Specify the file and the file format.
- ③ Specify the column separator (a tab).
- ④ Specify the column of transaction ids and of item ids, respectively.
- ⑤ Tell the function to look for and remove duplicate entries (for example multiple entries for "The Hobbit" by the same user).

The `read.transactions()` function reads data in two formats: the format where every row corresponds to a single item (like `bookdata.tsv`), and a format where each row corresponds to a single transaction, possibly with transaction id, like Table 8.1. To read data in the first format, use the argument `format="single"`; to read data in the second format, use the argument `format="basket"`.

It sometimes happens that a reader will buy one edition of a book, and then later add a rating for that book under a different edition. Because of the way we are representing books for this example, these two actions will result in duplicate entries. The `rm.duplicates=T` argument will eliminate them.

Once you've read in the data, you can inspect the resulting object.

EXAMINING THE DATA

Transactions are represented as a special object called `transactions`. You can think of a `transactions` object as a 0/1 matrix, with one row for every transaction and one column for every possible item. The matrix entry (i,j) is 1 if the i the transaction contains item j . There are a number of calls you can use to examine the transaction data.

```
> class(bookbaskets) ①
[1] "transactions"
attr(,"package")
[1] "arules"
> bookbaskets
transactions in sparse format with ②
92108 transactions (rows) and
220447 items (columns)
> dim(bookbaskets) ③
[1] 92108 220447
> colnames(bookbaskets)[1:5] ④
[1] " A Light in the Storm:[...]"
[2] " Always Have Popsicles"
[3] " Apple Magic"
[4] " Ask Lily"
[5] " Beyond IBM: Leadership Marketing and Finance for the 1990s"
> rownames(bookbaskets)[1:5] ⑤
[1] "10"      "1000"    "100001"  "100002"  "100004"
```

- ① The object is of class transactions.
- ② Printing the object tells you its dimensions.
- ③ You can also use `dim()` to see the dimensions of the "matrix."
- ④ The columns are labeled by book title.
- ⑤ The rows are labeled by customer.

You can examine the distribution of transaction sizes (or basket sizes) with the function `size()`.

```
> basketSizes <- size(bookbaskets)
> summary(basketSizes)
   Min. 1st Qu. Median     Mean 3rd Qu.    Max.
   1.0    1.0    1.0    11.1    4.0 10250.0
```

Most customers (at least half of them, in fact) only express interest in one book. But someone has expressed interest in over ten thousand! You probably want to look more closely at the size distribution to see what's going on.

```
> quantile(basketSizes, probs=seq(0,1,0.1)) ①
  0%   10%   20%   30%   40%   50%   60%   70%   80%   90%  100%
  1     1     1     1     1     1     2     3     5    13  10253
> library(ggplot2)
> ggplot(data.frame(count=basketSizes)) +
  geom_density(aes(x=count), binwidth=1) +
  scale_x_log10() ②
```

- ① Look at the basket size distribution, in 10% increments.
- ② Plot the distribution to get a better look.

Figure 8.8 shows the distribution of basket sizes. 90% of customers express interest in fewer than 15 books; most of the remaining customers express interest in up to about 100 books or so (the call `quantile(bookbaskets, probs=c(0.99, 1))` will show you that 99% of customers expressed interest in 179 books or fewer). Still, there are a few people who have expressed interest in several hundred, or even several thousand books....

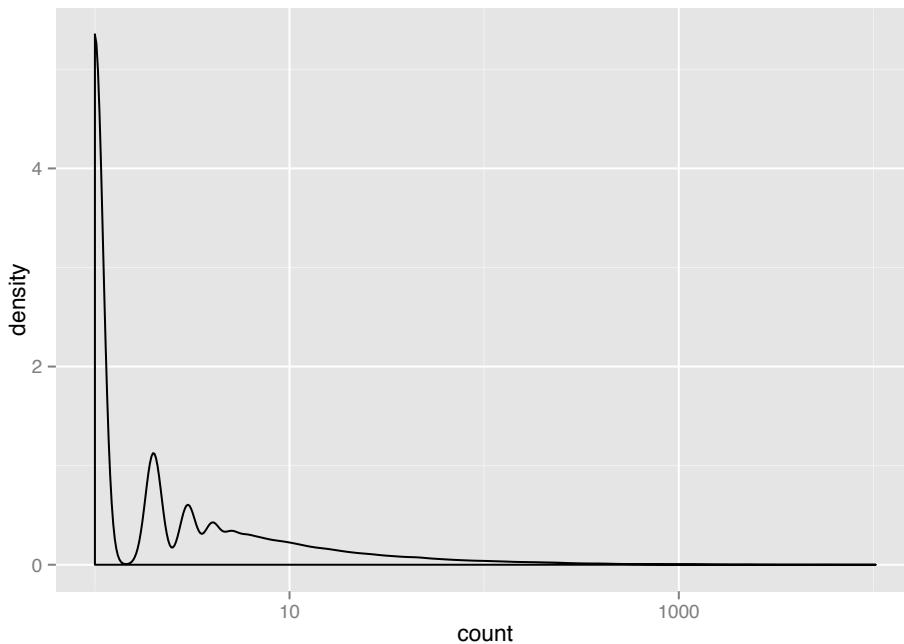


Figure 8.8 A densityplot of basket sizes

Which books are they reading? The function `itemFrequency()` will give you the relative frequency of each book in the transaction data.

```
> bookFreq <- itemFrequency(bookbaskets)
> summary(bookFreq)
   Min. 1st Qu. Median      Mean 3rd Qu.      Max.
1.086e-05 1.086e-05 1.086e-05 5.035e-05 3.257e-05 2.716e-02

> sum(bookFreq)
[1] 11.09909
```

Note that the frequencies don't sum to one. You can recover the number of times that each book occurred in the data by normalizing the item frequencies and multiplying by the total number of items.

```
> bookCount <- (bookFreq/sum(bookFreq))*sum(basketSizes) ①
> summary(bookCount)
   Min. 1st Qu. Median      Mean 3rd Qu.      Max.
1.000 1.000 1.000 4.637 3.000 2502.000
> orderedBooks <- sort(bookCount, decreasing=T) ②
> orderedBooks[1:10]
          Wild Animus
          2502
          The Lovely Bones: A Novel
          1295
          She's Come Undone
          934
          The Da Vinci Code
          905
          Harry Potter and the Sorcerer's Stone
          832
          The Nanny Diaries: A Novel
          821
          A Painted House
          819
          Bridget Jones's Diary
          772
          The Secret Life of Bees
          762
          Divine Secrets of the Ya-Ya Sisterhood: A Novel
          737
> orderedBooks[1]/dim(bookbaskets)[1] ③
Wild Animus
0.02716376
```

- ① Get the absolute count of book occurrences.
- ② Sort the count and list the ten most popular books.
- ③ The most popular book in the data set occurred in fewer than 3% of the baskets.

The last observation in the listing above highlights one of the issues with mining high-dimensional data: when you have thousands of variables, or thousands of items, almost every event is rare. Keep this point in mind when deciding on support thresholds for rule mining; your thresholds will often need to be quite low.

Before we get to the rule mining, let's refine the data a bit more. As we observed above, half of the customers in the data only expressed interest in a single book. Since you want to find books that occur together in people's interest lists, you can't make any direct use of people who have not yet shown interest in multiple books. You can restrict the data set to customers who have expressed interest in at least two books.

```
> bookbaskets_use <- bookbaskets[basketSizes > 1]
> dim(bookbaskets_use)
[1] 40822 220447
```

Now you are ready to look for association rules.

THE APRIORI() FUNCTION

In order to mine rules, you need to decide on a minimum support level and a minimum threshold level. For this example, let's try restricting the itemsets that you will consider to those that are supported by at least one hundred people. This leads to a minimum support of $100/\dim(\text{bookbaskets_use})[1] = 100/40822$. This is about 0.002, or two tenths of a percent. We will use a confidence threshold of 75%.

```
> rules <- apriori(bookbaskets_use,
                     parameter = list(support = 0.002, confidence=0.75)) ①
> summary(rules)
set of 191 rules ②

rule length distribution (lhs + rhs):sizes ③
  2   3   4   5
 11 100  66  14
```

```
Min. 1st Qu. Median     Mean 3rd Qu.     Max.  
2.000 3.000 3.000    3.435 4.000    5.000
```

summary of quality measures:
support confidence lift
Min. :0.002009 Min. :0.7500 Min. : 40.89
1st Qu.:0.002131 1st Qu.:0.8113 1st Qu.: 86.44
Median :0.002278 Median :0.8468 Median :131.36
Mean :0.002593 Mean :0.8569 Mean :129.68
3rd Qu.:0.002695 3rd Qu.:0.9065 3rd Qu.:158.77
Max. :0.005830 Max. :0.9882 Max. :321.89

mining info:
data ntransactions support confidence
bookbaskets_use 40822 0.002 0.75

4

5

- 1 Call `apriori()` with a minimum support of 0.002 and a minimum confidence of 0.75.
- 2 The summary of the `apriori()` output reports the number of rules found...
- 3 ... the distribution of rule lengths. In this example, most rules contain 3 items (2 on the left hand side X (lhs) and one on the right hand side Y (rhs))...
- 4 ...A summary of rule quality measures, including support and confidence...
- 5 ...and some information on how `apriori()` was called.

The quality measures on the rules include not only the rules' support and confidence, but also a quantity called *lift*. Lift compares the frequency of an observed pattern with how often you would expect to see that pattern just by chance. The lift of a rule "If X, then Y" is given by `support(union(X, Y)) / (support(X)*support(Y))`. If the lift is near one, then there's a good chance that the pattern you observed is occurring just by chance. The larger the lift, the more likely that the pattern is "real." In this case, all the discovered rules have a lift of at least 40, so they are likely to be real patterns in customer behavior.

INSPECTING AND EVALUATING RULES

There are also other metrics and interest measures you can use to evaluate the rules by using the function `interestMeasure()`. We will look at two of these measures: *coverage* and *fishersExactTest*. Coverage is the support of the left hand side of the rule (X); it tells you how often the rule would be applied in the data set. Fishers Exact Test is a significance test for whether or not an observed pattern is real, or chance (the same thing lift measures; Fisher's test is more formal). Fisher's exact test returns the p-value, or the probability that you would see the observed pattern by chance; you want the p-value to be small.

```

> measures <- interestMeasure(rules,
+                               method=c("coverage", "fishersExactTest"),
+                               transactions=bookbaskets_use)
> summary(measures)
      coverage      fishersExactTest
Min.   :0.002082   Min.   : 0.000e+00
1st Qu.:0.002511   1st Qu.: 0.000e+00
Median :0.002719   Median : 0.000e+00
Mean   :0.003039   Mean   :5.080e-138
3rd Qu.:0.003160   3rd Qu.: 0.000e+00
Max.   :0.006982   Max.   :9.702e-136

```

1
2
3

- 1 The call to `interestMeasure()` takes as arguments the discovered rules...
- 2 ... a list of interest measures to apply...
- 3 ... and a data set to evaluate the interest measures over. This is usually the same set used to mine the rules, but it needn't be. For instance, we can evaluate the rules over the full data set, `bookbaskets` to get coverage estimates that reflect all the customers, not just the ones who showed interest in more than one book.

The coverage of the discovered rules ranges from 0.002 to 0.007, equivalent to a range of about 100 to 250 people. All the p-values from Fisher's test are small, so it's likely that the rules reflect actual customer behavior patterns.

You can also call `interestMeasure()` with methods "support," "confidence," and "lift," among others. This would be useful in our example if you wanted to get support, confidence and lift estimates for the full data set `bookbaskets`, rather than the filtered data set `bookbaskets_use` -- or for a subset of the data, for instance only customers from the United States.

The function `inspect()` pretty-prints the rules. The function `sort()` allows you to sort the rules by a quality or interest measure, like confidence. To print the five most confident rules in the data set, you could use the command:

```
inspect(head((sort(rules, by="confidence"))), n=5))
```

For legibility, we show the output of the above command in Table 8.2.

Table 8.2 The Five most Confident Rules in the data

Left Hand Side	Right Hand Side	Support	Confidence	Lift
<ul style="list-style-type: none"> • Four to Score • High Five • Seven Up • Two for the Dough 	Three To Get Deadly	0.002	0.988	165
<ul style="list-style-type: none"> • Harry Potter and the Order of the Phoenix • Harry Potter and the Prisoner of Azkaban • Harry Potter and the Sorcerer's Stone 	Harry Potter and the Chamber of Secrets	0.003	0.966	73
<ul style="list-style-type: none"> • Four to Score • High Five • One for the Money • Two for the Dough 	Three To Get Deadly	0.002	0.966	162
<ul style="list-style-type: none"> • Four to Score • Seven Up • Three to Get Deadly • Two for the Dough 	High Five	0.002	0.966	181
<ul style="list-style-type: none"> • High Five • Seven Up • Three to Get Deadly • Two for the Dough 	Four to Score	0.002	0.966	168

There are two things to notice from Table 8.2. First, the rules concern books that come in series: the numbered series of novels about bounty hunter Stephanie

Plum, and the Harry Potter series. So these rules essentially say that if a reader has read four Stephanie Plum or Harry Potter books, they are almost sure to buy another one.

The second thing to notice is that rules 1, 4, and 5 are permutations of the same itemset. This is quite likely to happen when the rules get long.

RESTRICTING WHICH ITEMS TO MINE

You can restrict which items appear in the left hand side or right hand side of a rule. Suppose you are interested specifically in books that tend to co-occur with the novel *The Lovely Bones*. You can do this by restricting which books appear on the right hand side of the rule, using the `appearance` parameter.

```
brules <- apriori(bookbaskets_use,
                    parameter =list(support = 0.001,
                                    confidence=0.6),
                    appearance=list(rhs=c("The Lovely Bones: A Novel"),
                                   default="lhs"))
①
②
③
> summary(brules)
set of 46 rules

rule length distribution (lhs + rhs):sizes
 3   4
44   2

      Min. 1st Qu. Median     Mean 3rd Qu.     Max.
 3.000  3.000  3.000  3.043  3.000  4.000

summary of quality measures:
      support      confidence      lift
Min. :0.001004  Min. :0.6000  Min. :21.81
1st Qu.:0.001029 1st Qu.:0.6118 1st Qu.:22.24
Median :0.001102  Median :0.6258  Median :22.75
Mean   :0.001132  Mean   :0.6365  Mean   :23.14
3rd Qu.:0.001219 3rd Qu.:0.6457 3rd Qu.:23.47
Max.   :0.001396  Max.   :0.7455  Max.   :27.10

mining info:
      data ntransactions support confidence
bookbaskets_use        40822      0.001       0.6
```

- ① Relax the minimum support to 0.001 and the minimum confidence to 0.6
- ② Only *The Lovely Bones* is allowed to appear on the right hand side of the rules
- ③ By default, all the books can go into the left hand side of the rules

The supports, confidences, and lifts are lower than they were in our previous example, but the lifts are still much greater than one, so it's likely that the rules reflect real customer behavior patterns.

Let's inspect the rules, sorted by confidence. Since they will all have the same right hand side, we can use the `lhs()` function to only look at the left hand sides.

```
brulesConf <- sort(brules, by="confidence")      ①

> inspect(head(lhs(brulesConf), n=5))            ②
  items
1 {Divine Secrets of the Ya-Ya Sisterhood: A Novel,
  Lucky : A Memoir}
2 {Lucky : A Memoir,
  The Notebook}
3 {Lucky : A Memoir,
  Wild Animus}
4 {Midwives: A Novel,
  Wicked: The Life and Times of the Wicked Witch of the West}
5 {Lucky : A Memoir,
  Summer Sisters}
```

- ① Sort the rules by confidence
- ② Use the `lhs()` function to get the left hand itemsets of each rule, then inspect the top five.

Notice that four of the five most confident rules include *Lucky : A Memoir* in the left hand side, which perhaps isn't surprising, since *Lucky* was written by the author of *The Lovely Bones*. Suppose you want to find out about works by other authors that are interesting to people who showed interest in *The Lovely Bones*; you can use `subset()` to filter down to only rules that do not include *Lucky*.

```
brulesSub <- subset(brules, subset=!lhs %in% "Lucky : A Memoir")      ①
brulesConf <- sort(brulesSub, by="confidence")

> inspect(head(lhs(brulesConf), n=5))
  items
1 {Midwives: A Novel,
  Wicked: The Life and Times of the Wicked Witch of the West}
2 {She's Come Undone,
  The Secret Life of Bees,
  Wild Animus}
3 {A Walk to Remember,
  The Nanny Diaries: A Novel}
```

```
4 {Beloved,  
  The Red Tent}  
5 {The Da Vinci Code,  
  The Reader}
```

- ➊ Restrict to the subset of rules where Lucky is not in the left hand side.

These examples show that association rule mining is often highly interactive. To get interesting rules, you must often set the support and confidence levels fairly low; as a result you can get many, many rules. Some rules will be more interesting or surprising to you than others; to find them requires sorting the rules by different interest measures, or perhaps restricting yourself to specific subsets of rules.

8.2.4 Association Rule Takeaways

What you should remember about association rules

- The goal of association rule mining is to find relationships in the data: items or attributes that tend to occur together.
- A good rule "if X then Y" should occur more often than we would expect to observe just by chance. You can use lift or Fisher's exact test to check if this is true.
- When there are a large number of different possible items that can be in a basket (in our example, thousands of different books), most events will be rare (have low support).
- Association rule mining is often interactive, as there can be many many rules to sort and sift through.

8.3 Summary

In this chapter, you have learned how to find similarities in data using two different clustering methods in R, and how to find items that tend to occur together in data using association rules. You have also learned how to evaluate your discovered clusters, and your discovered rules.

Unsupervised methods like the ones we have covered in this chapter are really more exploratory in nature. Unlike with supervised methods, there is no "ground truth" to evaluate your findings against. However, the findings from unsupervised methods can be the starting point for more focused experiments and modeling.

In the last few chapters, we have covered the most basic modeling and data analysis techniques; they are all good first approaches to consider when you are starting a new project. In the next chapter, we will touch on a few more advanced methods.

8.4 External links section

Chapter 9: Exploring Advanced Methods

This chapter covers using:

- Bagging and random forests.
- Generalized additive models.
- Kernel Methods.
- Support Vector Machines.

In the last few chapters we have covered basic predictive modeling algorithms that you should have in your toolkit. These machine learning methods are usually a good place to start. In this chapter, we will look at more advanced methods that resolve specific weaknesses of the basic approaches. The main weaknesses we will address are: training variance, non-monotone effects and linearly inseparable data.

To illustrate the issues let's consider a silly "health" prediction model. Suppose we have for a number of patients (of widely varying, but unrecorded, ages) recorded height (as h in meters) and weight (as w in kilograms) and an appraisal of "healthy" or "unhealthy." The modeling question is: can height and weight accurately predict health appraisal? Models built off such limited features provide quick examples of the following common weaknesses:

Training variance

Training variance is when small changes in the make-up of the training set yield models that make substantially different predictions. Decision trees can exhibit this effect. Both *bagging* and *random forests* can both reduce training variance and sensitivity to over-fitting.

Non-monotone effects

Linear regression and logistic regression (see Chapter

XRF:chapter_7:chFunctionalModels) both treat numeric variables in a monotone matter. If more of a quantity is good then much more of the quantity is better. This is often not the case in the real world. For example: ideal healthy weight is in a bounded range, not arbitrarily heavy or arbitrarily light. *Generalized additive models* add the ability to model interesting variable effects and ranges to linear models and generalized linear models (such as logistic regression).

Linearly inseparable data

Often the concept we are trying to learn is not a linear combination of the original variables. Take BMI or body mass index for example: BMI purports to relate m/h^2 to health (rightly or wrongly). The term m/h^2 is not a linear combination of m and h , so neither linear regression or logistic regression would directly discover such a relation. It is reasonable to expect that a model that has a term of m/h^2 could produce better predictions of health appraisal than a model that only has linear combinations of h and m . This is because the data is more "separable" with respect to a m/h^2 shaped decision surface than to a h shaped decision surface. *Kernel methods* allow the data scientist to introduce new non-linear combination terms to models (like m/h^2) and *Support Vector Machines* use both kernels and training data to build useful decision surfaces.

The above issues don't always cause modeling efforts to *visibly* fail. Instead they often leave you with a model that is less powerful than possible. In this chapter we will use a few advanced methods to fix such modeling weaknesses lurking in earlier examples. We start with a demonstration of bagging and random forests.

9.1 Using Bagging and Random Forests to reduce training variance

In Section XRF:sect2_6.3.2:sectUsingDecisionTrees we looked at using decision trees for classification and regression. As we mentioned there, decision trees are an attractive method for a number of reasons:

- They take any type of data, numerical or categorical, without any distributional assumptions and without preprocessing.
- Most implementations (in particular, R's) handle missing data; the method is also robust to redundant and non-linear data.
- The algorithm is easy to use, and the output (the tree) is relatively easy to understand.
- Once the model is fit, scoring is fast.

On the other hand, decision trees do have some drawbacks:

- They have a tendency to overfit, especially without pruning.
- They have high training variance: samples drawn from the same population can produce trees with different structure, and different prediction accuracy.
- Prediction accuracy can be low, compared to other methods¹.

Footnote 1 Lim, Loh, and Shih, "A Comparison of Prediction Accuracy, Complexity, and Training Time of Thirty-three Old and New Classification", *Machine Learning* 40, 203–229 (2000). Online at <http://rbras.org.br/lib/exe/fetch.php/projetos:machlearn:comparisonofclassifiers.pdf>

For these reasons a technique called *bagging* is often used to improve decision trees and a more specialized technique called *random forests* directly combines decision tree and bagging methods. We will work examples of both techniques.

9.1.1 Using Bagging to improve prediction

One way to mitigate the shortcomings of decision tree models is by bootstrap aggregation, also called "bagging." In bagging, you draw bootstrap samples (random samples with replacement) from your data. From each sample, you build a decision tree model. The final model is the average of all the individual decision trees². To make this concrete, suppose that x is an input datum, $y_i(x)$ is the output of the i th tree, $c(y_1(x), y_2(x), \dots, y_n(x))$ is the vector of individual outputs, and y is the output of the final model.

Footnote 2 Bagging and Random Forests (which we will describe in the next section) are two variations of a general technique called *ensemble learning*. An ensemble model is composed of the combination of several smaller simple models (often small decision trees). Seni and Elder's text *Ensemble Methods in Data Mining* is an excellent introduction to the general theory of ensemble learning.

- For regression, or for estimating class probabilities, $y(x)$ is the average of the scores returned by the individual trees: $y(i) = \text{mean}(c(y_1(x), \dots, y_n(x)))$.
- For classification, the final model assigns the class that got the most votes from the individual trees.

Bagging decision trees stabilizes the final model by lowering the variance; this improves the accuracy. A bagged ensemble of trees is also less likely to overfit the data.

SIDE BAR**Bagging Classifiers**

The proofs that bagging reduces variance are only valid for regression and for estimating class probabilities, not for classifiers (a model that only returns class membership, not class probabilities). Bagging a bad classifier can make it worse. So you definitely want to work over estimated class probabilities, if they are at all available. However, it can be shown that for CART trees (which is the decision tree implementation in R) under mild assumptions, bagging will tend to increase classifier accuracy. See Sutton, Clifton D. "Classification and Regression Trees, Bagging, and Boosting," *Handbook of Statistics*, Vol. 24 for more details.

The Spambase dataset (also used in Chapter XRF:chapter_6:chMemorizationMethods) provides a good example for the bagging technique. The dataset consists of about 4600 documents and 57 features that describe the frequency of certain key words and characters. First we'll train a decision tree to estimate the probability that a given document is spam, then we will evaluate the tree's deviance (which we recall from discussions in Chapters XRF:chapter_6:chMemorizationMethods and XRF:chapter_7:chFunctionalModels is similar to variance) and its prediction accuracy.

First, let's load the data. Download a copy of `spamD.tsv`³. Then we write a few convenience functions, and train a decision tree as in listing 9.1.

Footnote 3 <https://github.com/WinVector/zmPDSwR/raw/master/Spambase/spamD.tsv>

Listing 9.1 Preparing SpamBase data, and evaluating the performance of decision trees

```
spamD <- read.table('spamD.tsv', header=T, sep='\t')      ①
spamTrain <- subset(spamD, spamD$rgroup>=10)
spamTest <- subset(spamD, spamD$rgroup<10)

spamVars <- setdiff(colnames(spamD), list('rgroup', 'spam'))
spamFormula <- as.formula(paste('spam=="spam" ',           ②
                                paste(spamVars, collapse=' + '), sep=' ~ '))

loglikelihood <- function(y, py) {                      ③
  pysmooth <- ifelse(py==0, 1e-12,
                     ifelse(py==1, 1-1e-12, py))

  sum(y * log(pysmooth) + (1-y)*log(1 - pysmooth))
}

accuracyMeasures <- function(pred, truth, name="model") { ④
  dev.norm <- -2*loglikelihood(as.numeric(truth), pred)/length(pred)    ⑤
  ctable <- table(truth=truth,
                  pred=(pred>0.5))                                     ⑥
  accuracy <- sum(diag(ctable))/sum(ctable)
  precision <- ctable[2,2]/sum(ctable[,2])
  recall <- ctable[2,2]/sum(ctable[2,])
  f1 <- precision*recall
  data.frame(model=name, accuracy=accuracy, f1=f1, dev.norm)
}

library(rpart)                                         ⑦
treemodel <- rpart(spamFormula, spamTrain)

accuracyMeasures(predict(treemodel, newdata=spamTrain),
                  spamTrain$spam=="spam",
                  name="tree, training")                                ⑧

accuracyMeasures(predict(treemodel, newdata=spamTest),
                  spamTest$spam=="spam",
                  name="tree, test")
```

- ① Load the data and split into training (90% of data) and test (10% of data) sets.
- ② Use all the features and do binary classification, where TRUE corresponds to spam documents.
- ③ A function to calculate log-likelihood (for calculating deviance).
- ④ A function to calculate and return various measures on the model: normalized deviance, prediction accuracy, and f1, which is the product of precision and recall.

- 5 We normalize the deviance by the number of data points so that we can compare the deviance across training and test sets.
- 6 We convert the class probability estimator into a classifier by labeling documents that score greater than 0.5 as spam.
- 7 Load the rpart library and fit a decision tree model.
- 8 Evaluate the decision tree model against the training and test sets.

The output of the last two calls to `accuracyMeasures()` produces the output below. As expected, the accuracy and f1 scores both degrade on the test set, and the deviance increases (we want the deviance to be small).

```
model accuracy      f1  dev.norm
tree, training 0.9104514 0.7809002 0.5618654

model accuracy      f1  dev.norm
tree, test    0.8799127 0.7091151 0.6702857
```

Now let's try bagging the decision trees in listing 9.2.

Listing 9.2 Bagging decision trees

```
ntrain <- dim(spamTrain)[1]          ①
n <- ntrain
ntree <- 100

samples <- sapply(1:ntree,           ②
                  FUN = function(iter)
                  {sample(1:ntrain, size=n, replace=T)})

treelist <- lapply(1:ntree,          ③
                  FUN=function(iter)
                  {samp <- samples[,iter];
                   rpart(spamFormula, spamTrain[samp,])})

predict.bag <- function(treelist, newdata) {          ④
  preds <- sapply(1:length(treelist),
                  FUN=function(iter) {
                    predict(treelist[[iter]], newdata=newdata)})
  predsums <- rowSums(preds)
  predsums/length(treelist)
}

accuracyMeasures(predict.bag(treelist, newdata=spamTrain),      ⑤
                  spamTrain$spam=="spam",
                  name="bagging, training")

accuracyMeasures(predict.bag(treelist, newdata=spamTest),
                  spamTest$spam=="spam",
                  name="bagging, test")
```

- ① We'll use bootstrap samples the same size as the training set, with one hundred trees.
- ② Build the bootstrap samples by sampling the row indices of spamTrain with replacement. Each column of the matrix samples represents the row indices into spamTrain that comprise the bootstrap sample.
- ③ Train the individual decision trees and return them in a list.
- ④ A function that scores a data set against each tree, then averages all the prediction probabilities. predict.bag is assuming the underlying classifier is return decision probabilities, not decisions.
- ⑤ Evaluate the bagged decision trees against the training and test sets.

This results in:

```
model  accuracy      f1  dev.norm
bagging, training 0.9220372 0.8072953 0.4702707
```

```
model accuracy      f1 dev.norm  
bagging, test 0.9061135 0.7646497 0.528229
```

As you see, bagging improves accuracy and f1, and reduces deviance over both the training and test sets when compared to the single decision tree (we will see a direct comparison of the scores a little later on). The improvement is more dramatic on the test set: the bagged model has less generalization error⁴ than the single decision tree. We can further improve model performance by going from bagging to *random forests*.

Footnote 4 Generalization error is the difference in accuracy of the model on data it's never seen before, as compared to its error on the training set.

9.1.2 Using Random Forests to further improve prediction

In bagging, the individual trees are built using randomized datasets, but each tree is built by considering the exact same set of features. This means that all the individual trees are likely to use very similar sets of features (perhaps in a different order or with different split values). Hence, the individual trees will tend to be overly correlated with each other. If there are regions in feature space where one tree tends to make mistakes, then all the trees are likely to make mistakes there, too, losing our chance at correction. The random forest approach tries to de-correlate the trees by randomizing the set of variables that each tree is allowed to use. For each individual tree in the ensemble the random forest method:

1. Draws a bootstrapped sample from the training data.
2. For each sample, grows a decision tree. At each node of the tree:
 1. Randomly draws a subset of `mtry` variables from the `p` total features that are available .
 2. Picks the best variable and the best split from that set of `mtry` variables.
 3. Continues until the tree is fully grown.

The final ensemble of trees is then bagged to make the random forest predictions. The above is quite involved, but fortunately all done by a the single-line random forest call.

By default, the `randomForest()` function in R draws `mtry = p/3` variables at each node for regression trees and `m = sqrt(p)` variables for classification tree. In theory, random forests are not terribly sensitive to the value of `mtry`. Smaller values will grow the trees faster; however if you have a very large number of variables to choose from, of which only a small fraction are

actually useful, then using a larger `mtry` is better, since with a larger `mtry` you are more likely to draw some useful variables at every step of the tree-growing procedure.

Continuing from the data in section 9.1 let's build a spam model using random forests in listing 9.3

Listing 9.3 Using random forests

```
library(randomForest)          ①  
  
fmodel <- randomForest(spamFormula,  
                        data=spamTrain,  
                        ntree=100,           ②  
                        nodesize=7,          ③  
                        importance=T)        ④  
                                         ⑤
```

- ① Load the `randomForest` package.
- ② Call the `randomForest()` function to build the model.
- ③ Use 100 trees to be compatible with our bagging example. The default is 500 trees.
- ④ Specify that each node of a tree must have a minimum of 7 elements, to be compatible with the default minimum node size that `rpart()` uses on this training set.
- ⑤ Tell the algorithm to save information to be used for calculating variable importance (we will see this later).

You can use the `accuracyMeasures()` function to evaluate `fmodel` on the training and test sets. Let's summarize the results for all three of the models we've looked at.

```
# Performance on the training set  
    model accuracy      f1 dev.norm  
      Tree 0.9104514 0.7809002 0.5618654  
      Bagging 0.9220372 0.8072953 0.4702707  
Random Forest 0.9925175 0.9810408 0.1159385  
  
# Performance on the test set  
    model accuracy      f1 dev.norm  
      Tree 0.8799127 0.7091151 0.6702857  
      Bagging 0.9061135 0.7646497 0.5282290  
Random Forest 0.9628821 0.9063584 0.3020674  
  
# Performance change between training and test:  
# The decrease in accuracy and f1 in the test set  
# from training, and the increase in dev.norm  
# in the test set from training.
```

```
# (So in every case, smaller is better)
  model accuracy      f1 dev.norm
    Tree 0.03053870 0.07178505 0.10842030
  Bagging 0.01592363 0.04264557 0.05795832
Random Forest 0.02963540 0.07468237 0.18612886
```

The random forest model performed dramatically better than the other two models in both training and test. However, the random forest's generalization error was comparable to that of a single decision tree (and almost twice that of the bagged model).⁵

Footnote 5 When a machine learning algorithm shows an implausibly good fit (like 0.99+ accuracy) it can be a symptom that you do not have enough training data to falsify bad modeling alternatives. Limiting the complexity of the model can cut down on generalization error and over-fitting and can be worth-while, even if it decreases training performance.

WARNING Random Forests can Overfit!

It's lore among random forest proponents that "random forests don't overfit." In fact, they can. Hastie, et.al. back up this observation in their chapter on random forests in *The Elements of Statistical Learning*. Look for unreasonably good fits on the training data as evidence of useless overfit and memorization. Also, it is important to evaluate your model's performance on a holdout set.

You can also mitigate the overfitting problem by limiting how deep the trees can be grown (using the `maxnodes` parameter to `randomForest()`). When you do this you are deliberately degrading model performance on training data, so that you can more usefully distinguish between models and falsify bad training decisions.

EXAMINING VARIABLE IMPORTANCE

A useful feature of the `randomForest()` function is its variable importance calculation. Since the algorithm uses a large number of bootstrap samples, each data point `x` has a corresponding set of *out-of-bag samples*: those samples that do not contain the point `x`. The out-of-bag samples can be used as a way similar to N-fold cross validation, to estimate the accuracy of each tree in the ensemble.

To estimate the "importance" of a variable `v`, the variable's values are randomly permuted in the out-of-bag samples, and the corresponding decrease in each tree's accuracy is estimated. If the average decrease over all the trees is large, then the variable is considered "important": that is, its value makes a big difference in predicting the outcome. If the average decrease is small, then the variable does not

make very much difference to the outcome. The algorithm also measures the decrease in node purity that occurs from splitting on a permuted variable (that is, how this variable affects the quality of the tree).

You can calculate the variable importance by setting `importance=T` in the `randomForest()` call, and then calling the functions `importance()` and `varImpPlot()`.

```
> varImp <- importance(fmodel) 1  
  
> varImp[1:10, ] 2  
   %IncMSE IncNodePurity  
word.freq.make     2.296096    2.4266521  
word.freq.address 6.019336    2.7132819  
word.freq.all      3.252322    3.2194184  
word.freq.3d       2.407552    0.9644499  
word.freq.our      13.028400   18.6669055  
word.freq.over     6.187187    4.5312607  
word.freq.remove   33.615974   122.9724112  
word.freq.internet 8.615904    8.2494350  
word.freq.order    5.905917    3.0257696  
word.freq.mail     5.893832    5.7076050  
  
varImpPlot(fmodel, type=1) 3
```

- ① Call `importance()` on the spam model.
- ② The `importance()` function returns a matrix of importance measures (larger values = more important).
- ③ Plot the variable importance as measured by MSE (`type=1`).

The result of the `varImpPlot()` call is shown in Figure 9.1.

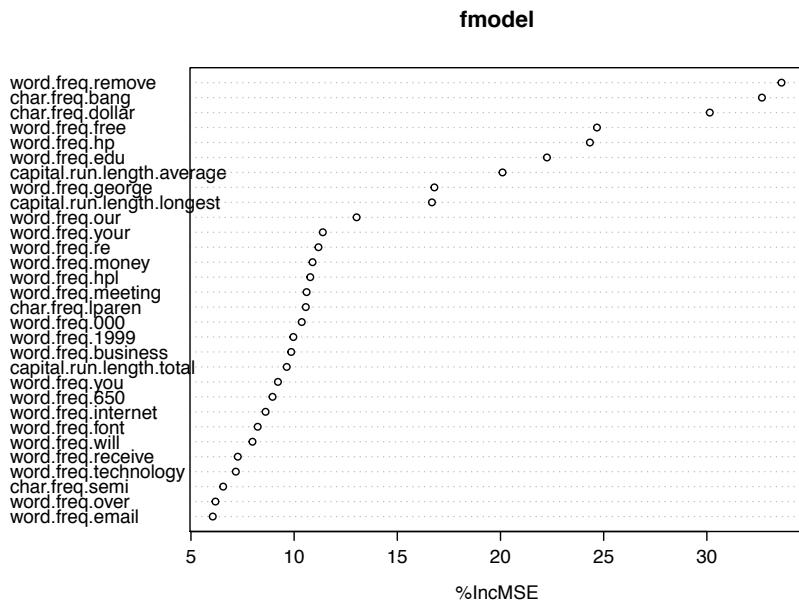


Figure 9.1 Plot of the most important variables in the spam model, as measured by MSE

Knowing which variables are most important (or at least, which variables contribute the most to the structure of the underlying decision trees) can help you with variable reduction. This is useful not only for building smaller, faster trees, but for choosing variables to be used by another modeling algorithm, if that is desired. We can reduce the number of variables in this spam example from 57 to 25 without affecting the quality of the final model.

```

ordered <- sort(varImp[,1], decreasing=T) ①

spamFormula2 <- as.formula(paste('spam=="spam"', ②
                                paste(names(ordered[1:25]),
                                      collapse=' + '),
                                sep=' ~ '))

```

```

fmodel_small <- randomForest(spamFormula2,data=spamTrain,
                               ntree=100,
                               nodesize=7)

```

- ① Sort the variables by their importance, as measured by MSE.
- ② Build a random forest model using only the 25 most important variables.

The smaller model performs just as well as the random forest model built using all 57 variables.

```

model accuracy      f1 dev.norm
Random Forest, train 0.9925175 0.9810408 0.1159385
    RF small, train 0.9934830 0.9834694 0.1198645

Random Forest, test 0.9628821 0.9063584 0.3020674
    RF small, test 0.9606987 0.9015467 0.2932659

```

9.1.3 Bagging and Random Forest Takeaways

What you should remember about bagging and random forests:

- Bagging stabilizes decision trees and improves accuracy by reducing variance.
- Bagging reduces generalization error.
- Random forests further improve decision tree performance by de-correlating the individual trees in the bagging ensemble.
- Random forest's variable importance measures can help you determine which variables are contributing the most strongly to your model.
- Because the trees in a random forest ensemble are un-pruned and potentially quite deep, there is still a danger of overfitting. Be sure to evaluate the model on holdout data to get a better estimate of model performance.

Bagging and Random Forests are "after the fact" improvement we can try on to improve model outputs. In our next section we will work with Generalized Additive Models which work to improve how model inputs are used.

9.2 Using Generalized Additive Models (GAMs) to learn non-monotone relationships

In Chapter XRF:chapter_7:chFunctionalModels we looked at how to use linear regression to model and predict quantitative output, and how to use logistic regression to predict class probabilities. Linear and logistic regression models are powerful tools, especially when you want to understand the relationship between the input variables and the output. They are robust to correlated variables (when regularized), and logistic regression preserves the marginal probabilities of the data. The primary shortcoming of both these models is that they assume that the relationship between the inputs and the output is monotone. That is: if more is good than much more is always better.

But what if the actual relationship is non-monotone? For example: for underweight patients increasing weight can increase health. But there is a limit, at some point more weight is bad. Linear and logistic regression miss this distinction

(but still often perform surprisingly well, hiding the issue). *Generalized Additive Models (GAMs)* are a way to model non-monotone responses within the framework of a linear or logistic model (or any other generalized linear model).

9.2.1 Understanding GAMs

Recall that, if $y[i]$ is the numeric quantity you want to predict, and $x[i,]$ is a row of inputs that corresponds to output $y[i]$, then linear regression finds a function $f(x)$ such that:

$$f(x[i,]) = b_0 + b[1] x[i,1] + b[2] x[i,2] + \dots b[n] x[i,n]$$

and $f(x[i,])$ is as close to $y[i]$ as possible.

In its simplest form, a GAM model relaxes the linearity constraint, and finds a set of functions $s_i()$ (and a constant term a_0) such that

$$f(x[i,]) = a_0 + s_1(x[i,1]) + s_2(x[i,2]) + \dots s_n(x[i,n])$$

and $f(x[i,])$ is as close to $y[i]$ as possible. The functions $s_i()$ are smooth curve fits that are built up from polynomials. The curves are called *splines*, and are designed to pass as closely as possible through the data without being too "wiggly" (that is, without overfitting). An example of a spline fit is shown in Figure 9.2.

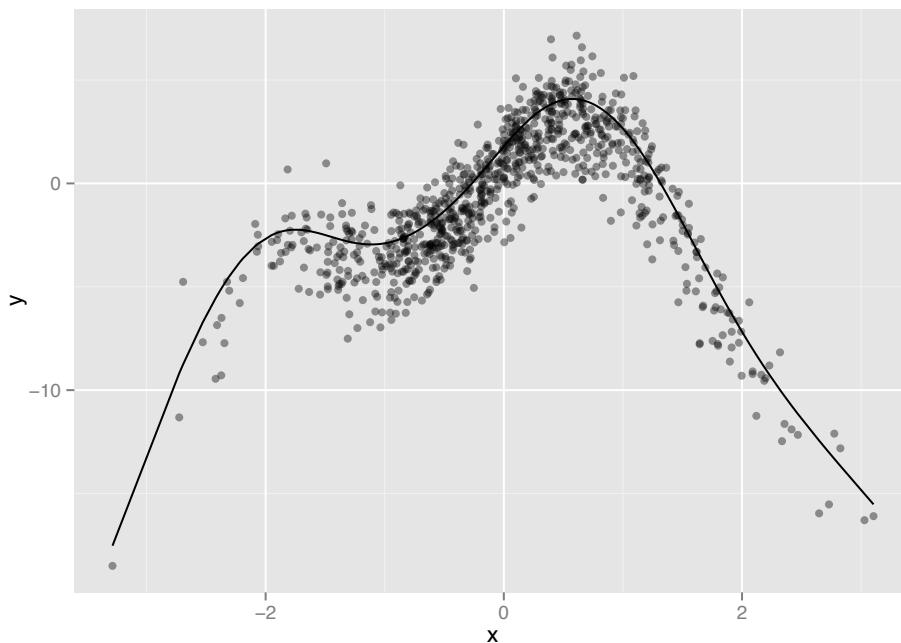


Figure 9.2 A spline that has been fit through a series of points

Let's work on a concrete example.

9.2.2 A one-dimensional regression example

Let's consider a toy example where the response y is a noisy nonlinear function of the input variable x (in fact, it is the function shown in Figure 9.2). As usual, we will split the data into training and test sets as shown in listing 9.4.

Listing 9.4 Preparing an artificial problem

```
set.seed(602957)

x <- rnorm(1000)
noise <- rnorm(1000, sd=1.5)

y <- 3*sin(2*x) + cos(0.75*x) - 1.5*(x^2) + noise

select <- runif(1000)
frame <- data.frame(y=y, x = x)

train <- frame[select > 0.1,]
test <- frame[select <= 0.1,]
```

Given the data is from the non-linear functions `sin()` and `cos()`, there should not be a good linear fit from x to y . We start by building a (poor) linear regression in listing 9.5.

Listing 9.5 Linear regression applied to our artificial example

```
> lin.model <- lm(y ~ x, data=train)
> summary(lin.model)
Call:
lm(formula = y ~ x, data = train)

Residuals:
    Min      1Q  Median      3Q     Max 
-17.698 -1.774   0.193   2.499   7.529 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -0.8330    0.1161 -7.175 1.51e-12 ***
x             0.7395    0.1197  6.180 9.74e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.485 on 899 degrees of freedom
Multiple R-squared:  0.04075, Adjusted R-squared:  0.03968 
F-statistic: 38.19 on 1 and 899 DF,  p-value: 9.737e-10

#
# calculate the root mean squared error (rmse)
#
> resid.lin <- train$y-predict(lin.model)
> sqrt(mean(resid.lin^2))
[1] 3.481091
```

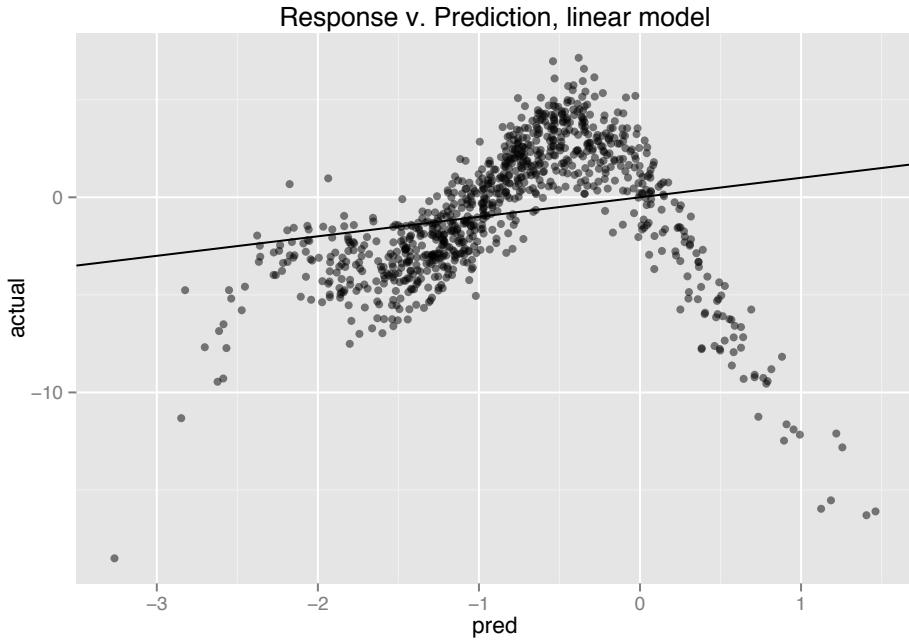


Figure 9.3 Linear model's predictions vs. actual response. The solid line is the line of perfect prediction (prediction=actual).

The resulting model's predictions are plotted versus true response in Figure 9.3. As expected, it's a very poor fit, with an R-squared of about 0.04. In particular, the errors are *heteroscedastic*⁶: there are regions where the model systematically under predicts, and regions where it systematically over predicts. If the relationship between x and y were truly linear (with noise), then the errors would be *homoscedastic*: that is, the errors would be evenly distributed (mean zero) around the predicted value everywhere.

Footnote 6 Heteroscedastic errors are whose magnitude is correlated with the quantity to be predicted. Heteroscedastic errors are bad because they are systematic and violate the assumptions of that errors are uncorrelated with outcomes that are used in many proofs of the good properties of regression methods.

Let's try finding a non-linear model that maps x to y . We will use the function `gam()` in the package `mgcv`⁷. When using `gam()`, you can model variables as either linear or non-linear. You model a variable x as non-linear by wrapping it in the `s()` notation. In this example, rather than using the formula $y \sim x$ to describe the model, you would use the formula $y \sim s(x)$. Then `gam()` will search for the spline `s()` that best describes the relationship between x and y as shown in listing 9.6. Only terms surrounded by `s()` get the GAM/spline treatment.

Footnote 7 There is an older package called `gam`, written by Hastie and Tibshirani, the inventors of GAMs. The `gam` package works fine. However, it is incompatible with the `mgcv` package, which `ggplot` already loads. Since we are using `ggplot` for plotting, we will use `mgcv` for our examples.

Listing 9.6 GAM applied to our artificial example

```
> library(mgcv) ①
> glin.model <- gam(y~s(x), data=train) ②
> glin.model$converged ③
[1] TRUE

> summary(glin.model)

Family: gaussian
Link function: identity ④

Formula:
y ~ s(x)

Parametric coefficients: ⑤
  Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.83467   0.04852  -17.2   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Approximate significance of smooth terms: ⑥
  edf Ref.df      F p-value
s(x) 8.685 8.972 497.8 <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

R-sq.(adj) =  0.832  Deviance explained = 83.4%
GCV score =  2.144  Scale est. = 2.121    n = 901 ⑦

#
# calculate the root mean squared error (rmse)
#
> resid.glin <- train$y-predict(glin.model)
> sqrt(mean(resid.glin^2))
[1] 1.448514
```

- ① Load the `mgcv` package.
- ② Build the model, specifying that `x` should be treated as a non-linear variable.
- ③ The `converged` parameter tells you if the algorithm converged. You should only trust the output if this is `TRUE`.
- ④ `Family=gaussian` and `link=identity` tells you that the model was treated with the same distributions assumptions as a standard linear regression.
- ⑤ The parametric coefficients are the linear terms (in this example, only the constant

term). This section of the summary tells you which linear terms were significantly different from zero.

- 6 The smooth terms are the non-linear terms. This section of the summary tells you which non-linear terms were significantly different from zero. It also tells you the effective degrees of freedom (edf) used up to build each smooth term. An edf near one indicates that the variable has an approximately linear relationship to the output.
- 7 "R-sq (adj)" is the adjusted R-squared. "Deviance explained" is the raw R-squared (0.834).

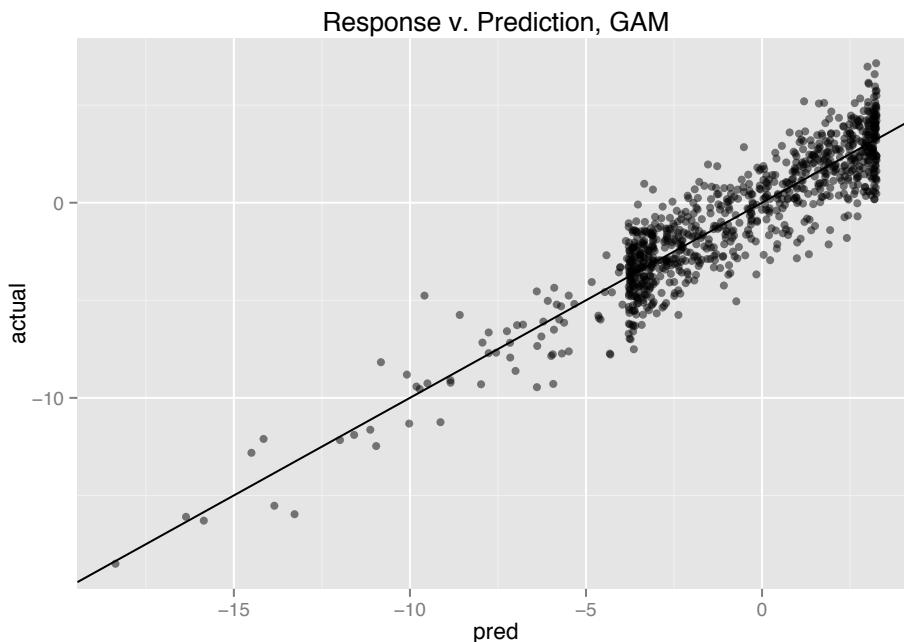


Figure 9.4 GAM's predictions vs. actual response. The solid line is the theoretical line of perfect prediction (prediction=actual).

The resulting model's predictions are plotted versus true response in Figure 9.4. This fit is much better: the model explains over 80% of the variance (R-squared of 0.83), and the root mean squared error (RMSE) over the training data is less than half the RMSE of the linear model. Notice that the points in Figure 9.4 are distributed more or less evenly around the line of perfect prediction. The GAM has been fit to be homoscedastic, and any given prediction is as likely to be an over prediction as an under prediction.

SIDE BAR **Modeling linear relationships using `gam()`**

By default, `gam()` will perform standard linear regression. If you were to call `gam()` with the formula `y ~ x`, you would get the same model that you got using `lm()`, above. More generally, the call `gam(y ~ x1 + s(x2), data=...)` would model the variable `x1` as having a linear relationship with `y`, and try to fit the best possible smooth curve to model the relationship between `x2` and `y`. Of course, the best smooth curve could be a straight line, so if you are not sure whether or not the relationship between `x` and `y` is linear, you can use `s(x)`. If you see that the coefficient has an edf (effective degrees of freedom -- see the model summary in the listing above) of about 1, then you can try refitting the variable as a linear term.

The use of splines gives GAMs a richer model space to choose from; this increased flexibility brings a higher risk of overfitting. Let's check the models' performances on the test data in listing 9.7.

Listing 9.7 Comparing linear regression and GAM performance

```
> actual <- test$y
> pred.lin <- predict(lin.model, newdata=test) ①
> pred.glin <- predict(glin.model, newdata=test)
> resid.lin <- actual-pred.lin
> resid.glin <- actual-pred.glin

> sqrt(mean(resid.lin^2)) ②
[1] 2.792653
> sqrt(mean(resid.glin^2))
[1] 1.401399

> cor(actual, pred.lin)^2 ③
[1] 0.1543172
> cor(actual, pred.glin)^2
[1] 0.7828869
```

- ① Call both models on the test data.
- ② Compare the root mean squared error (RMSE) of the linear model and the GAM on the test data.
- ③ Compare the R-squared of the linear model and the GAM on test data.

The GAM performed similarly on both sets (RMSE of 1.40 on test versus 1.45 on training; R-squared of 0.78 on test versus 0.83 on training). So there is likely no overfit.

9.2.3 Extracting the non-linear relationships

Once you fit a GAM, you will probably be interested in what the `s()` functions look like. Calling `plot()` on a GAM will give you a plot for each `s()` curve, so you can visualize what non-linearities. In our example, `plot(glin.model)` produces the top curve in Figure 9.5.

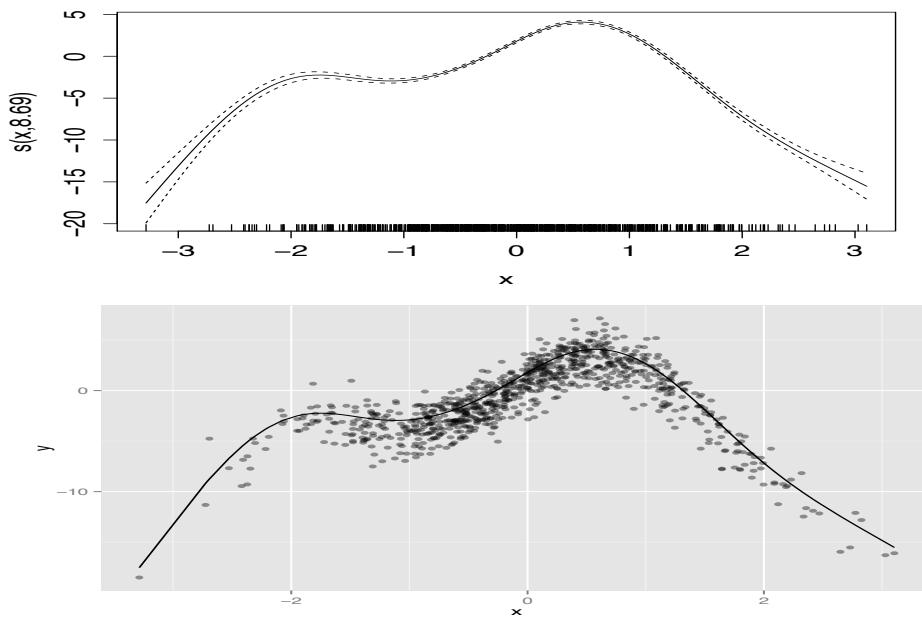


Figure 9.5 Top: The non-linear function `s(PWGT)` discovered by `gam()`, as output by `plot(gam.model)`. **Bottom:** The same spline superimposed over the training data.

The shape of the curve is quite similar to the scatterplot we saw in Figure 9.2 (which is reproduced as the bottom figure of Figure 9.5). In fact, the spline that is superimposed on the scatterplot in Figure 9.2 is the same curve.

You can extract the data points that were used to make this graph, by using the `predict()` function with the argument `type="terms"`. This produces a matrix where the `i`th column represents `s(x[, i])`. Listing 9.8 demonstrates how to reproduce the bottom plot of Figure 9.5.

Listing 9.8 Extracting a learned spline from a GAM

```
> sx <- predict(glin.model, type="terms")
> summary(sx)
  s(x)
Min.   :-17.527035
1st Qu.: -2.378636
Median  :  0.009427
Mean    :  0.000000
3rd Qu.:  2.869166
Max.   :  4.084999

> xframe <- cbind(train, sx=sx[,1])

> ggplot(xframe, aes(x=x)) + geom_point(aes(y=y), alpha=0.4) +
  geom_line(aes(y=sx))
```

Now that we've worked through a simple example, let's try a more realistic example with more variables.

9.2.4 Using GAM on actual data

For this example, we will predict a newborn baby's weight (DBWT) using data from the CDC 2010 Natality data set that we used in Chapter XRF:chapter_7:chFunctionalModels⁸. As input, we will consider mother's weight (PWGT), mother's pregnancy weight gain (WTGAIN), mother's age (MAGER) and the number of pre-natal medical visits (UPREVIS)⁹.

Footnote 8 The data set can be found at

<https://github.com/WinVector/zmPDSwR/blob/master/CDC/NatalBirthData.rData>. A script for preparing the data set from the original CDC extract can be found at

<https://github.com/WinVector/zmPDSwR/blob/master/CDC/prepBirthWeightData.R>.

Footnote 9 We've chosen this example to highlight the mechanisms of `gam()`, not to find the best model for birth weight. Adding other variables beyond the four we've chosen will improve the fit, but obscure the exposition.

In listing 9.9 we fit a linear model and a GAM, and compare.

Listing 9.9 Applying linear regression (with and without GAM) to health data

```
> form.lin <- as.formula("DBWT ~ PWGT + WTGAIN + MAGER + UPREVIS")
> linmodel <- lm(form.lin, data=train) ①
> summary(linmodel)

Call:
lm(formula = form.lin, data = train)

Residuals:
    Min      1Q  Median      3Q     Max 
-3044.35 -267.02   27.13  323.23 2797.13 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 2236.9901    31.7913   70.36 <2e-16 ***
PWGT         2.8869     0.1371   21.06 <2e-16 ***
WTGAIN       8.5438     0.3078   27.76 <2e-16 ***
MAGER        7.8842     0.7402   10.65 <2e-16 ***
UPREVIS     14.1309     1.1445   12.35 <2e-16 *** 
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 519.6 on 13568 degrees of freedom
Multiple R-squared:  0.09793, Adjusted R-squared:  0.09766 ②
F-statistic: 368.2 on 4 and 13568 DF,  p-value: < 2.2e-16

> form.glin <- as.formula("DBWT ~ s(PWGT) + s(WTGAIN) +
+                           s(MAGER) + s(UPREVIS)") 
> glinmodel <- gam(form.glin, data=train) ③
> glinmodel$converged ④
[1] TRUE
> summary(glinmodel)

Family: gaussian
Link function: identity

Formula:
DBWT ~ s(PWGT) + s(WTGAIN) + s(MAGER) + s(UPREVIS)

Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 3307.187     4.417   748.8 <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Approximate significance of smooth terms:
          edf Ref.df      F p-value    
s(PWGT)    4.792  5.844 89.47 <2e-16 ***
s(WTGAIN)  4.669  5.693 134.04 <2e-16 ***
s(MAGER)   5.711  6.710 15.96 <2e-16 ***
s(UPREVIS) 4.875  5.815 44.50 <2e-16 ***
```

```

---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

R-sq.(adj) =  0.115  Deviance explained = 11.6%      5
GCV score = 2.652e+05  Scale est. = 2.6478e+05  n = 13573

```

- 1 Build a linear model with four variables
- 2 The model explains about 10% of the variance; all coefficients are significantly different from zero.
- 3 Build a GAM with the same variables.
- 4 Verify that the model has converged.
- 5 The model explains about 11.5% of the variance; all variables have a non-linear effect significantly different from zero.

The GAM has improved the fit, and all four variables seem to have a non-linear relationship with birth weight, as evidenced by edfs all greater than one. We could use `plot(glinmodel)` to examine the shape of the `s()` functions; instead, we will compare them with a direct smoothing curve of each variable against mother's weight (in listing 9.10).

Listing 9.10 Plotting GAM results

```

> terms <- predict(glinmodel, type="terms")          1
> tframe <- cbind(DBWT = train$DBWT, as.data.frame(terms))    2
> colnames(tframe) <- gsub('[]()', '', colnames(tframe))   3
> pframe <- cbind(tframe, train[,c("PWGT", "WTGAIN",
"MAGER", "UPREVIS")])           4
>
> p1 <- ggplot(pframe, aes(x=PWGT)) +
  geom_point(aes(y=scale(sPWGT, scale=F))) +      5
  geom_smooth(aes(y=scale(DBWT, scale=F)))        6
[...]          7

```

- 1 Get the matrix of `s()` functions.
- 2 Bind in birth weight, convert to data frame.
- 3 Make the column names reference-friendly ("`s(PWGT)`" is converted to "`sPWGT`", etc.).
- 4 Bind in the input variables
- 5 Plot `s(PWGT)` shifted to zero mean versus PWGT (mother's weight) as points
- 6 Plot the smoothing curve of DWBT (birth weight) shifted to zero mean versus PWGT (mother's weight).
- 7 Repeat for remaining variables (omitted, for brevity).

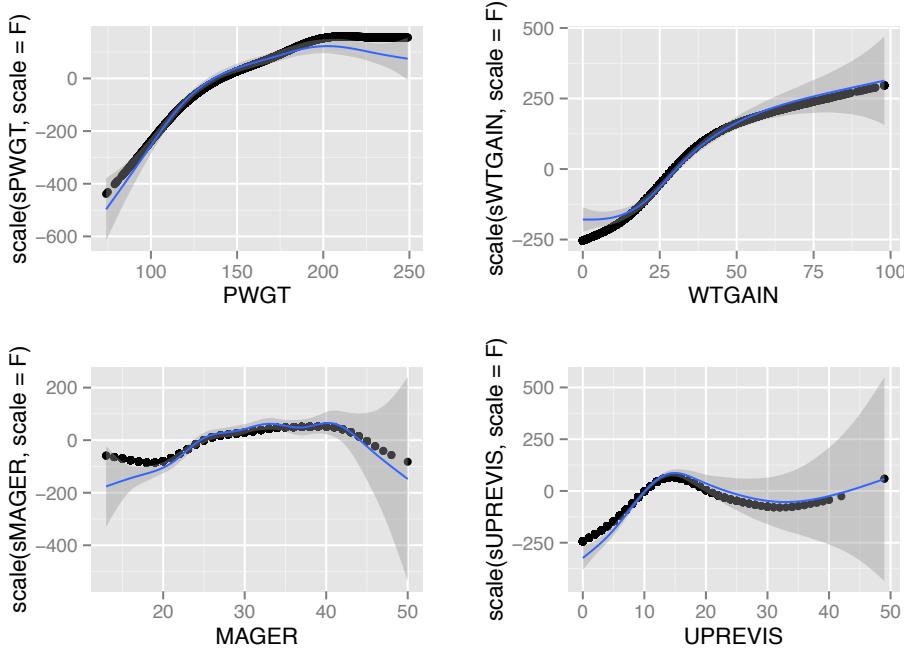


Figure 9.6 Smoothing curves of each of the four input variables plotted against birth weight, compared with the splines discovered by `gam()`. All curves have been shifted to be zero-mean for comparison of shape.

The plots of the `s()` splines compared with the smooth curves directly relating the input variables to birth weight are shown in Figure 9.6. The smooth curves in each case are quite similar to the corresponding `s()` in shape, and non-linear for all of the variables. As usual, you should check for overfit with hold-out data.

```

pred.lin <- predict(linmodel, newdata=test) ①
pred.glin <- predict(glinmodel, newdata=test)

cor(pred.lin, test$DBWT)^2
# [1] 0.09656043
cor(pred.glin, test$DBWT)^2
# [1] 0.1160727

```

- ① Run both the linear model and the GAM on the test data.
- ② Calculate R-squared for both models.

The performance of both the linear model and GAM were similar on the test set as they were on the training set, so in this example there is no overfit.

9.2.5 Using GAM for Logistic Regression

The `gam()` function can be used for logistic regression as well. Suppose that we wanted to predict the birth of underweight babies (defined as `DBWT < 2000`) from the same variables we have been using. The logistic regression call to do that would be:

```
form <- as.formula("DBWT < 2000 ~ PWGT + WTGAIN + MAGER + UPREVIS")
logmod <- glm(form, data=train, family=binomial(link="logit"))
```

The corresponding call to `gam()` also specifies the binomial family with logit link:

```
> form2 <- as.formula("DBWT<2000~s(PWGT)+s(WTGAIN) +
+ s(MAGER)+s(UPREVIS)")
> glogmod <- gam(form2, data=train, family=binomial(link="logit"))

> glogmod$converged
[1] TRUE

> summary(glogmod)
Family: binomial
Link function: logit

Formula:
DBWT < 2000 ~ s(PWGT) + s(WTGAIN) + s(MAGER) + s(UPREVIS)

Parametric coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept) -4.35651   0.08617 -50.55  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Approximate significance of smooth terms: ①
              edf Ref.df Chi.sq p-value
s(PWGT)     4.072  5.033  7.372  0.197
s(WTGAIN)   3.110  3.910 60.471 2.94e-12 ***
s(MAGER)    1.000  1.001  0.021  0.886
s(UPREVIS)  4.816  5.684 159.071 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

R-sq.(adj) =  0.0337  Deviance explained = 10.6% ②
UBRE score = -0.82445  Scale est. = 1           n = 13573
```

- ① Note that in this case, only the mother's weight gain during pregnancy (WTGAIN) and the number of pre-natal visits (UPREVIS) have a significant effect on

outcome.

- ② "Deviance explained" is the pseudo-R-squared: $1 - (\text{deviance}/\text{null.deviance})$.

As with the standard logistic regression call, you recover the class probabilities with the call `predict(glogmodel, newdata=train, type="response")`.

SIDE BAR The `gam()` package requires explicit formulas as input

You may have noticed that when calling `lm()`, `glm()`, or `rpart()`, we can input the formula specification as a string. These three functions quietly convert the string into a formula object. Unfortunately, neither `gam()` nor `randomForest()`, which we saw in a previous section, will do this automatic conversion. You must explicitly call `as.formula()` to convert the string into a formula object.

9.2.6 GAM Takeaways

What you should remember about generalized additive models (GAMs):

- GAMs let you represent non-linear and non-monotone relationships between variables and outcome in a linear or logistic regression framework.
- In the `mgcv` package, you can extract the discovered relationship from the GAM model using the `predict()` function with the `type="terms"` parameter.
- You can evaluate the GAM with the same measures you would use for standard linear or logistic regression: residuals, deviance, R-squared and pseudo-R-squared. The `gam()` summary also gives you an indication of which variables have a significant effect on the model.
- Because GAMs have increased complexity compared to standard linear or logistic regression models, there is more risk of overfit.

GAMs allow us to extend linear methods (and generalized linear methods) to allow variables to have non-linear (or even non-monotone) effects on outcome. But we have only considered each variable's impact individually. Another approach is to form new variables from non-linear combinations of existing variables. The hope is: with access to enough of these new variables, your modeling problem becomes easier.

In the next two sections we will work with two of the most popular ways to add and manage new variables: *kernel methods* and *support vector machines*.

9.3 Using Kernel Methods to increase data separation

Often your available variables are not quite good enough to meet your modeling goals. The most powerful way to get new variables is to get new, better, measurements from the domain expert. Acquiring new measurements may not be practical, so you would also use methods to create new variables from combinations of the measurements you already have at hand. We call these new variables *synthetic* to emphasize that they are synthesized from combinations of existing variables and do not represent actual new measurements. Kernel methods are one way to produce new variables from old and to in fact increase the power of machine learning methods.¹⁰ With enough synthetic variables data where points from different classes are mixed together can often be lifted to a space where the points from each class are grouped together, and separated from out-of-class points.

Footnote 10 The standard method to create synthetic variables is to add *interaction* terms. An interaction between variables occurs when a change in outcome due to two (or more) variables is more than the changes due to each variable alone. For example, too high a sodium intake will increase the risk of hypertension, but this increase is disproportionately higher for people with a genetic susceptibility to hypertension. The probability of becoming hypertensive is a function of the interaction of the two factors (diet and genetics). For use interaction terms in R see `help('formula')` for some details. In models such as `lm()` you can introduce interaction terms by adding `:`'s to pairs of terms in your formula specification.

One misconception of Kernel methods is that they are automatic or self-adjusting. They are not; beyond a few "automatic bandwidth adjustments" it is up to the data scientist to specify a useful kernel instead of the kernel being automatically found from the data. However, many of the standard kernels (inner-product, Gaussian and cosine) are so useful that it is often profitable to try a few kernels to see what improvements they offer.

WARNING**The word kernel is used in many different senses**

The word "kernel" has many different incompatible definitions in mathematics and statistics. The machine learning sense of the word used here is taken from operator theory and the sense used in Mercer's theorem. The kernels we want are two argument functions that behave a lot like an inner product. The other common (incompatible) statistical use of the word "kernel" is in density estimation where kernels are single argument functions that represent probability density functions or distributions.

In next few sections we will work through the definition of kernel function. We

will give a few examples of transformation that can be implemented by kernels and a few examples of transformations that can not be implemented as kernels. We will then work through a few examples.

9.3.1 Understanding kernel functions

THE FORMAL DEFINITION OF A KERNEL FUNCTION

In our application a kernel is a function with a very specific definition. Let u and v be any pair of variables. u and v are typically vectors of input or independent variables (possibly taken from two rows of a data set). A function $k(,)$ that maps pairs (u, v) to numbers is called a kernel function if and only if there is some function $\phi()$ mapping (u, v) 's to a vector space such that

$$k(u, v) = \phi(u) \%*% \phi(v)$$

for all u, v .¹¹ We will informally call the expression $k(u, v) = \phi(u) \%*% \phi(v)$ the "Mercer expansion of the kernel" (in reference to Mercer's theorem¹²) and consider $\phi()$ the certificate that tells us $k(,)$ is a good kernel. This is much easier to understand from a concrete example. In listing 9.11 we develop an example function $k(,)$ and the matching $\phi()$ that demonstrates that $k(,)$ is in fact a kernel over two dimensional vectors.

Footnote 11 $\%*%$ is R's notation for dot product or inner product, see `help('%%')` for details. Note that $\phi()$ is allowed to map to very large (and even infinite) vector spaces.

Footnote 12 http://en.wikipedia.org/wiki/Mercer's_theorem

Listing 9.11 An artificial kernel example

```
> u <- c(1,2)
> v <- c(3,4)
> k <- function(u,v) {    ①
  u[1]*v[1] + u[2]*v[2] +
  u[1]*u[1]*v[1]*v[1] + u[2]*u[2]*v[2]*v[2] +
  u[1]*u[2]*v[1]*v[2]
}
> phi <- function(x) {    ②
  x <- as.numeric(x)
  c(x,x*x,combn(x,2,FUN=prod))
}
> print(k(u,v))    ③
[1] 108
> print(phi(u))
[1] 1 2 1 4 2
> print(phi(v))
[1] 3 4 9 16 12
> print(as.numeric(phi(u) %*% phi(v)))    ④
[1] 108
```

- ① Define a function of two vector variables (both two dimensional) as the sum of various products of terms.
- ② Define a function of a single vector variable that returns a vector containing the original entries plus all products of entries.
- ③ Example evaluation of k().
- ④ Confirm phi() agrees with k(). (phi() is the certificate that shows k() is in fact a kernel).

Figure 9.7 illustrates what we hope for from a good kernel: out data being pushed around so it is easier to sort or classify.

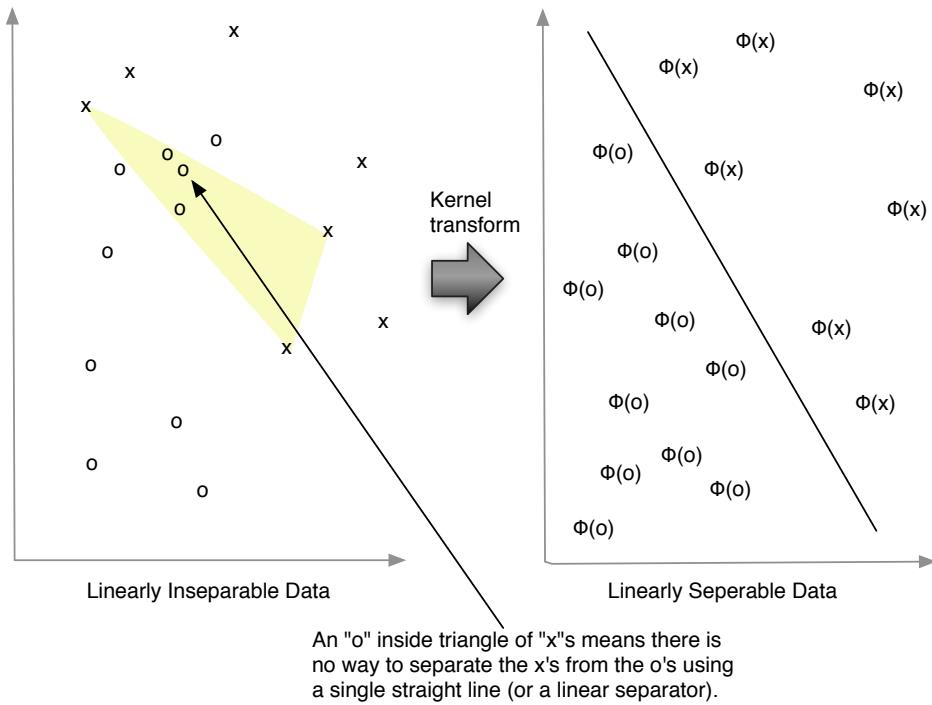


Figure 9.7 Notional illustration of a Kernel transform. Based on Cristianini and Shawe-Taylor (2000).

Most kernel methods use the function $k(\cdot, \cdot)$ directly and only use properties of $k(\cdot, \cdot)$ guaranteed by the matching $p(\cdot)$ to ensure method correctness. The $k(\cdot, \cdot)$ function is usually quicker to compute than the notional function $\phi(\cdot)$. A simple example of this is the what we will call the dot-product similarity of documents. The dot-product document similarity is defined as the dot product of two vectors where each vector is derived from a document by building a huge vector of indicators, one for each possible feature. For instance, if the features you are considering are word pairs, then for every pair of words in a given dictionary the document gets a feature of 1 if the pair occurs as a consecutive utterance in the document and 0 if not. This method is the $\phi(\cdot)$, but in practice we never use the $\phi(\cdot)$ procedure. Instead for one document each consecutive pair of words is generated and a bit of score is added if this pair is both in the dictionary and found consecutively in the other document. For moderate sized documents and large dictionaries this direct $k(\cdot, \cdot)$ implementation is vastly more efficient than the $\phi(\cdot)$ implementation.

WHY ARE KERNEL FUNCTIONS USEFUL?

Kernel functions are very useful for a number of reasons:

- Cover's theorem. Inseparable data sets (data where examples from more than one training

class appear to be inter-mixed when plotted) become separable (and so we can build a good classifier) under common non-linear transforms. Non-linear kernels are a good compliment to many linear machine learning techniques.

- Many $\phi()$ can be directly implemented as a data preparation. Never be too proud to stoop to trying some interaction variables in a model.
- Some very powerful and expensive $\phi()$ that cannot be directly implemented as data preparation have very efficient matching kernel function $k(,)$ that can be used directly in select machine learning algorithms without needing access to the highly complex $\phi()$.
- Mercer's theorem: all symmetric positive semi-definite functions $k(,)$ mapping pairs of variables to the reals can be represented as $k(u,v) = \phi(u) \%*% \phi(v)$ for some function $\phi()$. So by restricting to functions with a Mercer expansion we are not giving up much.

Our next goal is to demonstrate some useful kernels and some machine learning algorithms that use them efficiently to solve problems. The most famous kernelized machine learning algorithm is the support vector machine, which we will demonstrate in section 9.4. But first it helps to demonstrate some useful kernels.

SOME IMPORTANT KERNEL FUNCTIONS

Figure 9.8 give the formal definitions some of the most useful kernels, but first lets look at their practical uses in table 9.1.

Table 9.1 Some important kernels and their uses

Kernel Name	Informal description and use
Definitional (or explicit) kernels	Any method that explicitly adds additional variables (such as interactions) can be expressed as a kernel over the original data. These are kernels where we explicitly implement and use <code>phi()</code> .
Linear transformation kernels	Any positive semi-definite linear operation (like projecting to principal components) can be also be expressed as a kernel.
Gaussian or radial kernel	Many decreasing non-negative functions of distance can be expressed as kernels. This is also an example of a kernel where <code>phi()</code> maps into an infinite dimensional vector space (essentially the Taylor series of <code>exp()</code>) and therefore <code>phi(u)</code> does not have an easy to implement representation (you must instead use <code>k(,)</code>).
Cosine similarity kernel	Many similarity measures (measures that are large on identical items and small for dissimilar items) can be expressed as kernels.
Polynomial kernel	Much is made of the fact that positive integer powers of kernels are also kernels. The derived kernel does have many more terms derived from powers and products of terms from the original kernel, but the modeling technique is not able to independently pick coefficients for all of these terms simultaneously. Polynomial kernels do introduce some extra options, but they are not magic.

$$\kappa(u, v) = \phi(u) \cdot \phi(v) \quad \text{definitional kernel}$$

$$\kappa(u, v) = u \cdot v \quad \text{dot product or identity kernel}$$

$$\kappa(u, v) = u^\top L^\top Lv \quad \text{linear transformation}$$

$$\kappa(u, v) = e^{-c||u-v||^2} \quad \text{Gaussian radial kernel}$$

$$\kappa(u, v) = \frac{u \cdot v}{\sqrt{u \cdot u \ v \cdot v}} \quad \text{cosine similarity kernel}$$

$$\kappa(u, v) = (su \cdot v + c)^d \quad \text{polynomial kernel}$$

Figure 9.8 Some important formal kernel definitions

At this point it is important to point out that not everything is a kernel. For example the common squared distance function (`k = function(u,v) { (u-v) %*% (u-v) }`) is not a kernel. So we see kernels can express similarities, but can not directly express distances.¹³

Footnote 13 Some more examples of kernels (and how to build new kernels from old) can be found at:
<http://www.win-vector.com/blog/2011/10/kernel-methods-and-support-vector-machines-de-mystified/>

9.3.2 Using an explicit kernel on a problem

REVISITING THE PUMS LINEAR REGRESSION MODEL

To demonstrate using a kernel on an actual problem we re-prepare the data used in section XRF:sect2_7.1.3:SectMakingPredictions to again build a model predicting the logarithm of income from a few other factors. We resume this analysis by using `load()` to reload the data from a copy of the file <https://github.com/WinVector/zmPDSwR/raw/master/PUMS/psub.RData>. Recall that the basic model (for purposes of demonstration) used only a few variables, we re-do producing a step-wise improved linear regression model for `log(PINCP)` in listing 9.12. Applying stepwise linear regression to PUMS data

```
dtrain <- subset(psub,ORIGINRANDECODEDGROUP >= 500)
dtest <- subset(psub,ORIGINRANDECODEDGROUP < 500) ①
m1 <- step( ②
  lm(log(PINCP,base=10) ~ AGEP + SEX + COW + SCHL,
    data=dtrain), ③
  direction='both')
rmse <- function(y, f) { sqrt(mean( (y-f)^2 )) } ④
print(rmse(log(dtest$PINCP,base=10),
  predict(m1,newdata=dtest))) ⑤
# [1] 0.2752171
```

- ① Split data into test and training
- ② Ask that the linear regression model we are building be "stepwise" improved, which is a powerful automated procedure for removing variables that don't seem to have significant impacts (can improve generalization performance).
- ③ Build the basic linear regression model.
- ④ Define the root mean square error function.
- ⑤ Calculate the root mean square error between the prediction and the actuals.

The quality of prediction was middling (the RMSE is not that small), but the model exposed some of the important relationships. In a real project you would do your utmost to find new explanatory variables. But you would also be interested to

see if any combination of variables you are already using would help with prediction. We will work through finding some of these combinations using an explicit `phi()`.

INTRODUCING AN EXPLICIT TRANSFORM

Explicit kernel transforms are a formal way to unify ideas like re-shaping variables and adding interaction terms.¹⁴

Footnote 14 See `help('formula')` for how to add interactions using the `:` and `*` operators.

In listing 9.13 we set up a `phi()` function and use it to build a new larger data frame with new modeling variables.

Listing 9.13 Applying an example explicit kernel transform

```
phi <- function(x) {    1
  x <- as.numeric(x)
  c(x,x*x,combn(x,2,FUN=prod))
}

phiNames <- function(n) {    2
  c(n,paste(n,n,sep=':'),
    combn(n,2,FUN=function(x) {paste(x,collapse=':')}))}

modelMatrix <- model.matrix(~ 0 + AGEP + SEX + COW + SCHL,psub)    3
colnames(modelMatrix) <- gsub('[^a-zA-Z0-9]+','_',
  colnames(modelMatrix))    4

pM <- t(apply(modelMatrix,1,phi))    5
vars <- phiNames(colnames(modelMatrix))
vars <- gsub('[^a-zA-Z0-9]+','_',vars)

colnames(pM) <- vars    6
pM <- as.data.frame(pM)
pM$PINCP <- psub$PINCP
pM$ORIGINRANDECODE <- psub$ORIGINRANDECODE
pMtrain <- subset(pM,ORIGINRANDECODE >= 500)
pMtest <- subset(pM,ORIGINRANDECODE < 500)    7
```

- ➊ Define our primal kernel function: map a vector to a copy of itself plus all square terms and cross multiplied terms.
- ➋ Define a function similar to our primal kernel, but working on variable names instead of values.
- ➌ Convert data to a matrix where all categorical variables are encoded as multiple numeric indicators.
- ➍ Remove problematic characters from matrix column names.
- ➎ Apply the primal kernel function to every row of the matrix and transpose results so they are written as rows (not as a list as returned by `apply()`).
- ➏ Extend names from original matrix to names for compound variables in new

matrix.

- ⑦ Add in outcomes, test/train split columns and prepare new data for modeling.

The steps to use this new expanded data frame to build a model are given in example 9.14

Listing 9.14 Modeling using the explicit kernel transform

```
formulaStr2 <- paste('log(PINCP,base=10)',  
                     paste(vars,collapse=' + '),  
                     sep=' ~ ')  
m2 <- lm(as.formula(formulaStr2),data=pMtrain)  
coef2 <- summary(m2)$coefficients  
interestingVars <- setdiff(rownames(coef2)[coef2[, 'Pr(>|t|)']<0.01],  
                           '(Intercept)')  
interestingVars <- union(colnames(modelMatrix),interestingVars) ①  
formulaStr3 <- paste('log(PINCP,base=10)',  
                     paste(interestingVars,collapse=' + '),  
                     sep=' ~ ')  
m3 <- step(lm(as.formula(formulaStr3),data=pMtrain),direction='both') ②  
print(rmse(log(pMtest$PINCP,base=10),predict(m3,newdata=pMtest))) ③  
# [1] 0.2735955
```

- ① Select a set of interesting variables by building an initial model using all of the new variables and retaining an interesting subset. This is an ad-hoc move to speed up the stepwise regression by trying to quickly dispose of many useless derived variables. The primal kernel method by introducing many new variables also introduces many new degrees of freedom which can invite over-fitting.
- ② Stepwise regress on subset of variables to get new model.
- ③ Calculate the root mean square error between the prediction and the actuals.

We see RMSE is improved by a very small amount on the test data. With such a small improvement we have extra reason to want to confirm its statistical significance using a cross validation procedure as demonstrated in section XRF:sect2_6.2.3:sectCrossVal. Leaving these issues aside, let's look at the summary of the new model to see what new variables the `phi()` procedure introduced. Listing 9.15 shows the structure of the new model.

Listing 9.15 Inspecting the results of the explicit kernel model

```
> print(summary(m3))

Call:
lm(formula = log(PINCP, base = 10) ~ AGEP + SEXM +
    COWPrivate_not_for_profit_employee +
    SCHLAssociate_s_degree + SCHLBachelor_s_degree +
    SCHLDoctorate_degree +
    SCHLGED_or_alternative_credential + SCHLMaster_s_degree +
    SCHLProfessional_degree + SCHLRegular_high_school_diploma +
    SCHLsome_college_credit_no_degree + AGEP_AGEPE, data = pmtrain)

Residuals:
    Min      1Q  Median      3Q     Max 
-1.29264 -0.14925  0.01343  0.17021  0.61968 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 2.9400460  0.2219310 13.248 < 2e-16 ***
AGEP        0.0663537  0.0124905  5.312 1.54e-07 ***
SEXM        0.0934876  0.0224236  4.169 3.52e-05 ***
COWPrivate_not_for_profit_em -0.1187914  0.0379944 -3.127 0.00186 ** 
SCHLAssociate_s_degree       0.2317211  0.0509509  4.548 6.60e-06 ***
SCHLBachelor_s_degree        0.3844459  0.0417445  9.210 < 2e-16 ***
SCHLDoctorate_degree         0.3190572  0.1569356  2.033 0.04250 *  
SCHLGED_or_alternative_creden 0.1405157  0.0766743  1.833 0.06737 .  
SCHLMaster_s_degree          0.4553550  0.0485609  9.377 < 2e-16 ***
SCHLProfessional_degree      0.6525921  0.0845052  7.723 5.01e-14 ***
SCHLRegular_high_school_diplo 0.1016590  0.0415834  2.445 0.01479 *  
SCHLsome_college_credit_no_de 0.1655906  0.0416345  3.977 7.85e-05 *** 
AGEP_AGEPE      -0.0007547  0.0001704 -4.428 1.14e-05 *** 
---
Signif. codes:  0 '****' 0.001 '***' 0.01 '**' 0.05 '*' 0.1 '.' 1 

Residual standard error: 0.2649 on 582 degrees of freedom
Multiple R-squared:  0.3541,   Adjusted R-squared:  0.3408 
F-statistic: 26.59 on 12 and 582 DF,  p-value: < 2.2e-16
```

In this case the only new variable is: AGEP_AGEPE. The model is using AGEP*AGEP to build a non-monotone relation between age and log income.¹⁵

Footnote 15 Of course, this sort of relation could be handled quickly by introducing a AGEP*AGEP term directly in the model or by using a generalized additive model to discover the optimal (possibly non-linear) shape of the relation between AGEP and log income (see section 9.2).

The `phi()` method is automatic and can therefore be applied in many modeling situations. In our example we can think of the crude function that multiplies all pairs of variables as our `phi()` or think of the implied function that

took the original set of variables to the new set called `interestingVars` as the actual training data dependent `phi()`. Explicit `phi()` kernel notation adds some capabilities, but algorithms that are designed to work directly with implicit kernel definitions in `k(,)` notation can be much more powerful. The most famous such method is the support vector machine, which we will use in the next section.

9.3.3 Kernel Takeaways

What you should remember about kernel methods:

- Kernels are systematic way of creating interaction and other synthetic variables that are combinations of individual variables.
- The goal of kernelizing is to lift the data into a space where the data is separable, or where linear methods can be used directly.

Now we are ready to work with the most famous user of kernel methods, support vector machines.

9.4 Using Support Vector Machines to model complicated decision boundaries

The idea of SVMs is to use entire training examples as classification landmarks (called support vectors). We will describe the bits of the theory that affect use and move on to applications.

9.4.1 Understanding support vector machines

A support vector machine with a given function `phi()` builds a model where for a given example \mathbf{x} the machine decides \mathbf{x} is in the class if

```
w %*% phi(x) + b >= 0
```

for some w and b , and not in the class otherwise. The model is completely determined by the vector w and the scalar offset b . The general idea is sketched out in Figure 9.9. In "real space" (left), the data are separated by a non-linear boundary. When the data are lifted into the higher-dimensional kernel space (right), the lifted points are separated by a hyperplane whose normal is w , and that is offset from the origin by b (not shown). Essentially, all the data that makes a positive dot product with w is on one side of the hyperplane (and all belong to one class); data that makes a negative dot product with the w belong to the other class.

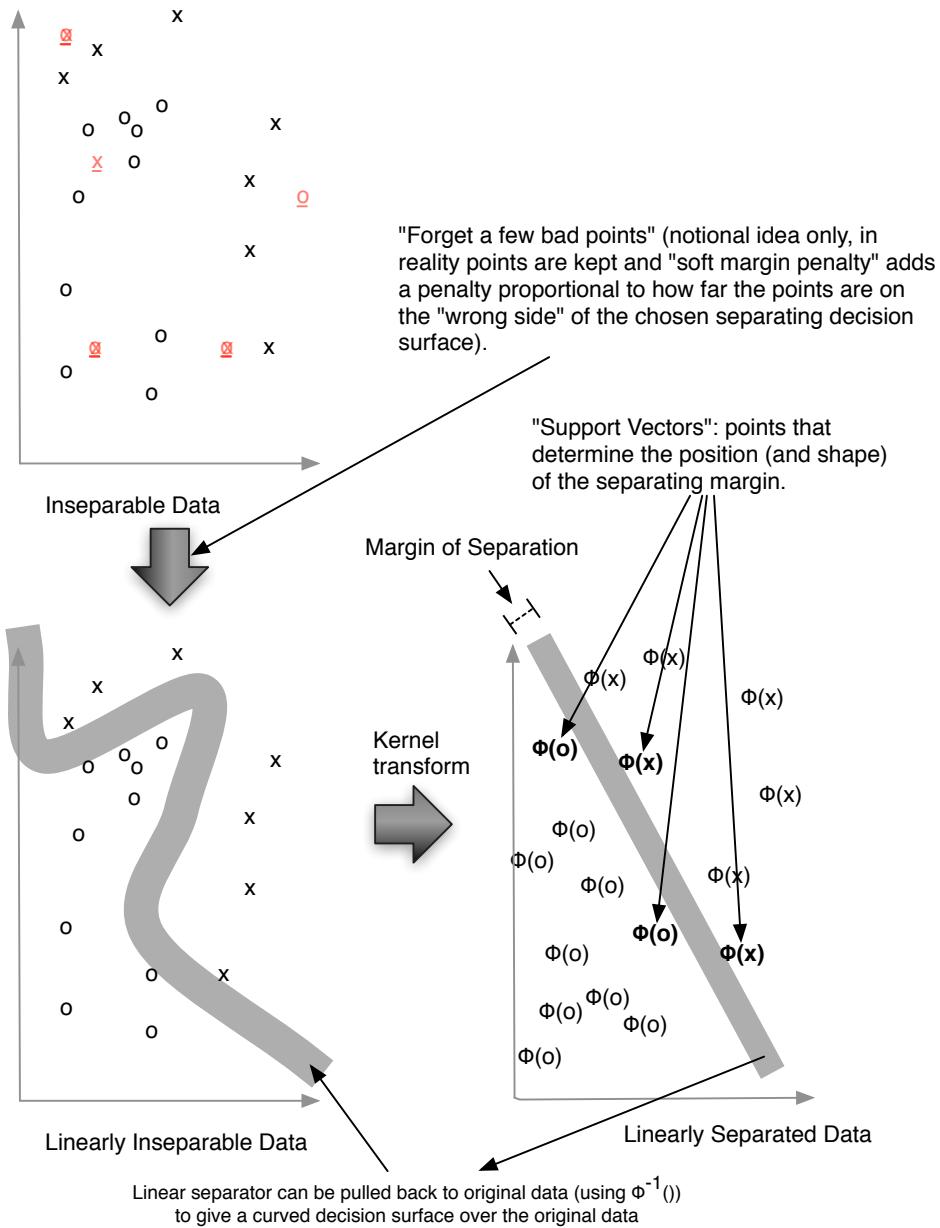


Figure 9.9 Notional illustration of SVM.

Finding w and b is performed by the support vector training operation. There are variations on the support vector machine that make decisions between more than two classes, perform scoring/regression, and detect novelty. But we will discuss only the support vector machines for simple classification.

As a user of support vector machines you don't immediately need to know how the training procedure works, that is what the software is doing for you. But you do need to have some notion of what it is trying to do. The model w, b is ideally picked so that

```
w %*% phi(x) + b >= u
```

for all training x that were in the class and

```
w %*% phi(x) + b <= v
```

for all training examples not in the class. The data is called *separable* if $u>v$ and the size of the separation $(u-v)/\sqrt{w^T w}$ is called the *margin*. The goal of the SVM optimizer is to maximize the margin. Large margin can actually ensure good behavior on future data (good generalization performance). In practice real data is not always separable even in the presence of a kernel. To work around this most SVM implementations implement the so-called *soft margin* optimization goal.

A soft margin optimizer adds additional error terms that are used to allow a limited fraction of the training examples to be on the wrong side of the decision surface.¹⁶ The model does not actually perform well on the altered training examples, but trades the error on these examples against increased margin on the remaining training examples. For most implementations there is a control that determines the trade-off between margin width for the remaining data and how much data is pushed around to achieve the margin. Typically the control is named C and setting it to values higher than 1 increases the penalty for moving data (so builds a thinner margin that may not generalize as well at the expense of adjusting less of the training data).¹⁷

Footnote 16 A common type of data set that is inseparable under any kernel is any data set where there are at least two examples belonging to different outcome classes with the exact same values for all input or x variables. The original "hard margin" SVM could not deal with this sort of data and was for that reason not considered to be practical.

Footnote 17 For more details on support vector machines we recommend Christianini and Shawe-Taylor's "An Introduction to Support Vector Machines."

THE SUPPORT VECTORS

The support vector machine gets its name from how the vector w is usually represented: as a linear combination of training examples called the support vectors. Recall we said in section 9.3.1 that the function `phi()` is allowed, in principle, to map into a very large or even infinite vector space. Support vector machines can get away with this because they never explicitly compute $\phi(x)$ ever. What is done instead is that any time the algorithm wants to compute $\phi(u) \cdot \phi(v)$ for a pair of data points it instead computes $k(u, v)$ which is, by definition, equal. But then how do we evaluate the final model $w \cdot \phi(x) + b$? It would be nice if there were a s such that $w = \phi(s)$, as we could then again use `k(,)` to do the work. In general there is usually no s such that $w = \phi(s)$. However there are always a set of vectors s_1, \dots, s_m and numbers a_1, \dots, a_m such that

```
w = sum(a1*phi(s1), ..., am*phi(sm))
```

With some math we can show this means:

```
w %*% phi(x) + b = sum(a1*k(s1,x), ..., am*k(sm,x)) + b
```

The right hand side is a quantity we can compute.

The vectors s_1, \dots, s_m are actually the features from m training examples and are called the support vectors. The work of the support vector training algorithm is to find the vectors s_1, \dots, s_m , the scalars a_1, \dots, a_m and the offset b .¹⁸

Footnote 18 Because SVMs work in terms of support vectors, not directly in terms of original variables or features, a feature that is predictive can be lost if it doesn't show up strongly in kernel specified similarities between support vectors.

The reason the user must know about the support vectors is that they are stored in the support vector model and there can be a very large number of them (causing the model to be large and expensive to evaluate). In the worst case the number of support vectors in the model can be almost as large as the number of training examples (making support vector model evaluation potentially as expensive as nearest neighbor evaluation). There are some tricks to work around this: lowering C , training models on random subsets of the training data and "primalizing."

The easy case of primalizing is when you have a kernel $\phi()$ that has a simple representation (such as the identity kernel or a low degree polynomial kernel) and you can explicitly compute $w = \sum(a_1\phi(s_1), \dots, a_m\phi(s_m))$. In this case you can discard the a_i and s_i and save only w and b as your model representation.

For kernels that don't map into a finite vector space (such as the very popular radial or Gaussian kernel) you can also hope to find a vector function $p()$ such that $p(u) \approx p(v)$ is very near $k(u, v)$ for all of your training data and then use

```
w ~ sum(a1*p(s1), ..., am*p(sm))
```

along with b as an approximation of your support vector model. However, many support vector packages are not able to convert to a primal form model (it is mostly seen in Hadoop implementations) and often converting to primal form takes as long as the original model training.

9.4.2 Trying a SVM on artificial example data

Support vector machines excel at learning concepts of the form "examples that are near each other should be given the same classification." This is because they can use support vectors and margin to erect a moat that groups training examples into classes. In this section we will quickly work some examples. One thing to notice is how little knowledge of the internal working details of the support vector machine are needed. The user mostly has to choose the kernel to control what is similar/dissimilar, adjust C to try and control model complexity, and pick `class.weights` to try and value different types of errors.

THE SPIRAL EXAMPLE

Let's start with an example adapted from R's kernlab library documentation. Listing 9.16 shows the recovery of the famous spiral machine learning counter-example using kernlab's spectral clustering method.

Listing 9.16 Setting up the spirals data as an example classification problem

```
> library('kernlab')  
> data('spirals')    ①  
> sc <- specc(spirals, centers = 2)    ②  
> s <- data.frame(x=spirals[,1],y=spirals[,2],class=as.numeric(sc))    ③  
> library('ggplot2')  
> ggplot(data=s) +  
  geom_point(aes(x=x,y=y,shape=as.factor(class))) +  
  coord_fixed()    ④
```

- ① Load the kernlab kernel and support vector machine package and then ask that the included example "spirals" be made available.
- ② Use kernlab's spectral clustering routine to identify the two different spirals in the example data set.
- ③ Combine the spiral coordinates and the spiral label into a data frame.
- ④ Plot the spirals with class labels.

Figure 9.10 shows the labeled spiral data set. Two classes (represented by circles and triangles) of data are arranged in two interwoven spirals. This dataset is hard to learn for learners that don't have a rich enough concept space (perceptrons, shallow neural nets) and easy for more sophisticated learners that can introduce the right new features. Support vector machines, with the right kernel, are a technique which finds the spiral easily.

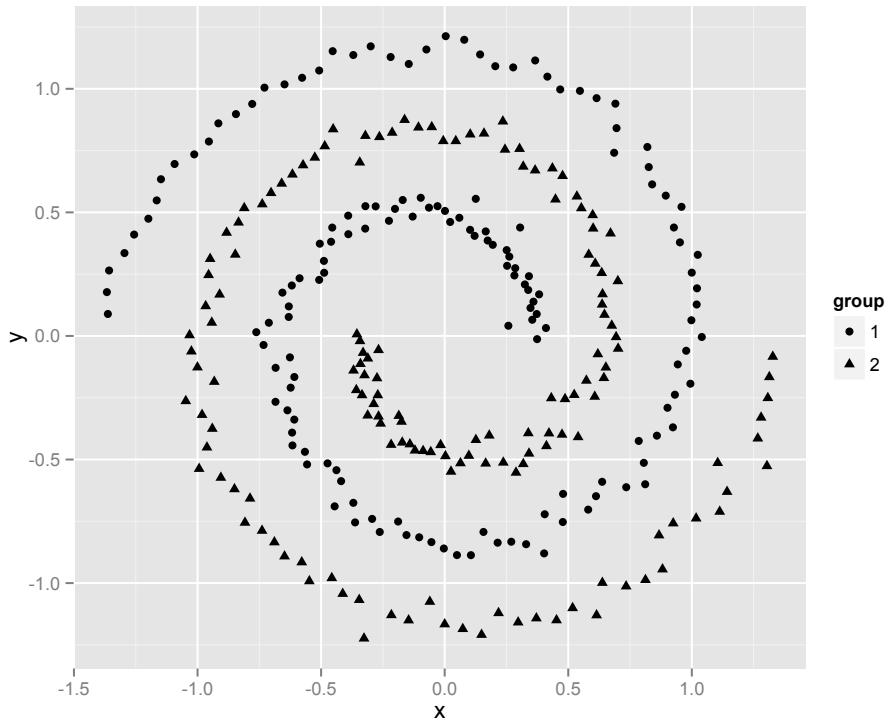


Figure 9.10 The spiral counter-example

SUPPORT VECTOR MACHINE WITH THE WRONG KERNEL

Support vector machines are very powerful, but without the correct kernel they have difficulty with some concepts (such as the spiral example). Listing 9.17 shows a failed attempt to learn the spiral concept with a support vector machine using the identity or dot-product kernel.

Listing 9.17 SVM with a poor choice of kernel

```
set.seed(2335246L)
s$group <- sample.int(100, size=dim(s)[[1]], replace=T)
sTrain <- subset(s, group>10)
sTest <- subset(s, group<=10) ①
mSVMV <- ksvm(class~x+y, data=sTrain, kernel='vanilladot') ②
sTest$predSVMV <- predict(mSVMV, newdata=sTest, type='response') ③
ggplot() +
  geom_point(data=sTest, aes(x=x, y=y, shape=predSVMV=='1'),
             show_guide=T) +
  geom_point(data=s, aes(x=x, y=y, shape=class=='1'), alpha=0.2,
             show_guide=F) +
  coord_fixed() ④
```

- ① Prepare to try to learn spiral class label from coordinates using a support vector machine.
- ② Build the support vector model using a vanilladot kernel (not a very good kernel).
- ③ Use the model to predict class on held-out data.
- ④ Plot the predictions on top of a grey copy of all the data so we can see if predictions agree with the original markings or not.

This attempt results in figure 9.11. In figure 9.11 we plot the total data set in light grey and the SVM classifications of the test data set in solid black. Notice that the plotted predictions look a lot more like the concept $y < 0$ than the spirals. The SVM did not produce a good model with the identity kernel. In the next section we repeat the process with the Gaussian radial kernel and get a very good result.

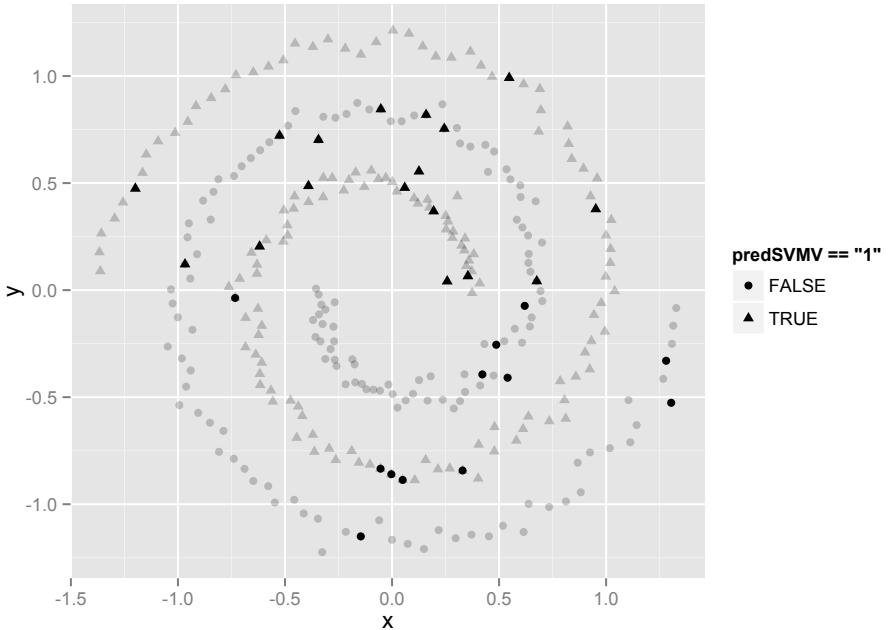


Figure 9.11 Identity kernel failing to learn the spiral concept

SUPPORT VECTOR MACHINE WITH A GOOD KERNEL

In listing 9.18 we repeat the SVM fitting process, but this time specifying the Gaussian or radial kernel. We again plot the SVM test classifications in black (with the entire data set in light grey) in figure 9.12. Notice that this time the actual spiral has been learned and predicted.

Listing 9.18 SVM with a good choice of kernel

```
mSVMG <- ksvm(class~x+y,data=sTrain,kernel='rbfdot')
sTest$predSVMG <- predict(mSVMG,newdata=sTest,type='response')
ggplot() +
  geom_point(data=sTest,aes(x=x,y=y,shape=predSVMG=='1'),
             show_guide=T) +
  geom_point(data=s,aes(x=x,y=y,shape=class=='1'),alpha=0.2,
             show_guide=F) +
  coord_fixed()
```

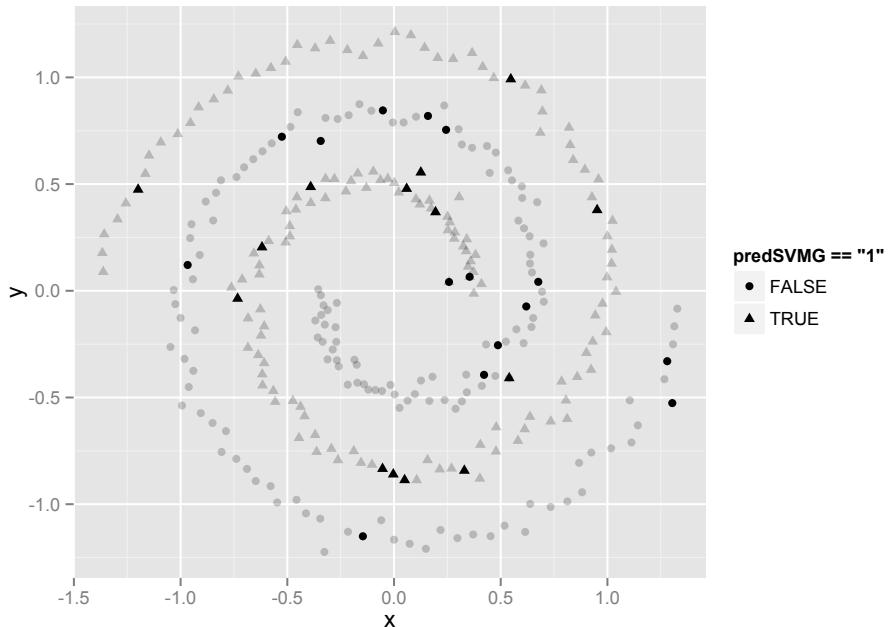


Figure 9.12 Radial kernel successfully learning the spiral concept

9.4.3 Using SVMs on real data

To demonstrate the use of SVMs on real data we will quickly re-do the analysis of the Spambase data from section XRF:sect2_5.2.1:sectScoringClassifiers.

REPEATING THE SPAMBASE LOGISTIC REGRESSION ANALYSIS

In section XRF:sect2_5.2.1:sectScoringClassifiers we originally built a logistic regression model and confusion matrix. We continue working on this example in listing 9.19 (after downloading a copy of <https://github.com/WinVector/zmPDSwR/raw/master/Spambase/spamD.tsv>).

Listing 9.19 Re-visiting the SpamBase example with GLM

```
spamD <- read.table('spamD.tsv', header=T, sep='\t')
> spamFormula <- as.formula(paste('spam=="spam" ',
  paste(spamVars, collapse=' + '), sep=' ~ '))
> spamModel <- glm(spamFormula, family=binomial(link='logit'),
  data=spamTrain)
> spamTest$pred <- predict(spamModel, newdata=spamTest,
  type='response')
> print(with(spamTest, table(y=spam, glPred=pred>=0.5)))
      glPred
y      FALSE TRUE
non-spam    264   14
spam        22 158
```

APPLYING A SUPPORT VECTOR MACHINE TO THE SPAMBASE EXAMPLE

The SVM modeling steps are about as simple and are shown in listing 9.20.

Listing 9.20 Applying SVM to the SpamBase example

```
> library('kernlab')
> spamFormulaV <- as.formula(paste('spam',
  paste(spamVars, collapse=' + '), sep=' ~ '))
> svmM <- ksvm(spamFormulaV, data=spamTrain, ①
  kernel='rbfdot', ②
  C=10, ③
  prob.model=T, cross=5, ④
  class.weights=c('spam'=1, 'non-spam'=10) ⑤
  )
> spamTest$svmPred <- predict(svmM, newdata=spamTest, type='response')
> print(with(spamTest, table(y=spam, svmPred=svmPred)))
      svmPred
y      non-spam spam
non-spam      271    7
spam        27 153
```

- ① Build a support vector model for the Spambase problem.
- ② Ask for the radial dot or Gaussian kernel (in fact the default kernel).
- ③ Set the "soft margin penalty" high, prefer not moving training examples to getting a wider margin. Essentially preferring a complex model that applies weakly to all the data to a simpler model that applies strongly on a subset of the data.
- ④ Ask that in addition to a predictive model that an estimate of model estimating class probabilities also be built. Not all SVM libraries support this operation and the probabilities are essentially built after the model (through a cross-validation procedure) and may not be of as high a quality as the model itself.
- ⑤ Explicitly control the trade-off between false positive and false negative errors. In

this case we are say non-spam classified as spam (a false positive) should be considered an expensive mistake.

Listing 9.21 shows the standard summary and print display for the support vector model. Very few model diagnostics are included (other than training error, which is a simple accuracy measure), so we definitely recommend using the model critique techniques from Chapter XRF:chapter_5:chCritiquingModels to validate model quality and utility. A few things to look for are: what kernel was used, the "SV type" (classification as we wanted)¹⁹, and the number of support vectors retained (this is the degree of memorization going on, in this case 1117 training examples were retained).

Footnote 19 The ksvm call only performs classification on factors, if a boolean or numeric quantity is used as the quantity to be predicted the ksvm call may return a regression model (instead of the desired classification model).

Listing 9.21 Printing the SVM results summary

```
> print(summary(svmM))
Length Class Mode
      1   ksvm   S4
> print(svmM)
Support Vector Machine object of class "ksvm"

SV type: C-svc (classification)
parameter : cost C = 10

Gaussian Radial Basis kernel function.
Hyperparameter : sigma = 0.0300518888320615

Number of Support Vectors : 1117

Objective Function Value : -4635.266
Training error : 0.02824
Cross validation error : 0.076275
Probability model included.
```

COMPARING RESULTS

Notice the two confusion matrices are very similar. The GLM model has a recall of 158/180 and the SVM model has a recall of 153/180. However the SVM model has a much lower false positive count of 7 than the GLM's 14. Some of this is due to setting `C=10` (which tells the SVM to prefer training accuracy and margin to model simplicity) and setting `class.weights` (telling the SVM to prefer precision to recall). For a more apples to apples comparison we can look at the GLM model's top 160 spam candidates (the same number the SVM model proposed) as shown in listing 9.22.

Listing 9.22 Shifting decision point to perform an apples to apples comparison

```
> sameCut <- sort(spamTest$pred)[length(spamTest$pred)-160] ①
> print(with(spamTest,table(y=spam,glPred=pred>sameCut))) ②
      glPred
y      FALSE TRUE
non-spam    268   10
spam        30  150
```

- ① Find out what GLM score threshold has 160 examples above it.
- ② Ask the GLM model for its predictions that are above the threshold. We are essentially asking the model for its 160 best candidate spam prediction results.

Notice the new confusion matrix in listing 9.22 has slightly higher false positive and false negative counts than the support vector model. The SVM model represents a small improvement on the GLM/logistic model, and it required no new data so may be worth using. Where SVMs excel is in cases where unknown combination of variables are important effects and also when similarity of examples is strong evidence of examples being in the same class (not a property of the email spam example). Problems of this nature tend to benefit from use of either SVM or nearest neighbor techniques.²⁰

Footnote 20 For some examples of the connections between support vector machines and kernelized nearest neighbor methods please see

<http://www.win-vector.com/blog/2011/10/kernel-methods-and-support-vector-machines-de-mystified/>.

9.4.4 Support Vector Machine Takeaways

What you should remember about SVMs:

- Support Vector Machines are a kernel-based classification approach where the kernels are represented in terms of a (possibly a very large) subset of the training examples.

- SVMs try to lift the problem into a space where the data is linearly separable (or as near to separable as possible).
- SVMs are useful in cases where the useful interactions or other combinations of input variables aren't known in advance. They are also useful when similarity is strong evidence of belonging to the same class.

9.5 Summary

In this chapter, we demonstrated some advanced methods to fix specific issues with basic modeling approaches. We used:

- *Bagging and random forests*: to reduce the sensitivity of models to early modeling choices and reduce modeling variance.
- *Generalized additive models*: to remove the (false) assumption that each model feature contributes to the model in a monotone fashion.
- *Kernel Methods*: to introduce new features that are non-linear combinations of existing features, increasing the power of our model.
- *Support Vector Machines*: to use training examples as landmarks (called support vectors), again increasing the power of our model.

You should understand that you bring in advanced methods and techniques to fix specific modeling problems, not because they have exotic names or exciting histories. We also feel you should at least try to find an existing technique to fix a problem you suspect is hiding in your data *before* building your own custom technique (often the existing technique incorporates a lot of tuning and wisdom). Finally: the goal of learning the theory of advanced techniques is not to be able to recite the steps of the common implementation but to know when the technique applies and what trade-offs it represents. The data scientist needs to supply thought and judgement and realize that the platform can supply implementations.

The actual point of a modeling project is to deliver results for deployment and to present useful documentation and evaluations to your partners. The next part of this book will address best practices for delivering your results.

9.6 External links section

Part 3

In Part 3 we conclude with the important steps of deploying work into production, documenting work and building effective presentations.

10

Documentation and Deployment

In this chapter we will work through producing effective milestone documentation, code comments, version control records, and demonstration deployments. The idea is that these can all be thought of as important documentation of the work you have done. We want to be able to do the following:

Table 10.1 Chapter goals

Goal	Description
Produce effective milestone documentation	A readable summary of project goals, data provenance, steps taken and technical results (numbers and graphs). Milestone documentation is usually read by collaborators and peers, so it can be concise and often include actual code. We will demonstrate a very good tool for producing excellent milestone documentation is the R <code>knitr</code> package. <code>knitr</code> is a product of the "reproducible research" movement and is a excellent way to produce a reliable snapshot that not only shows the state of a project, but can allow others to confirm the project works.
Manage a complete project history	It makes little sense to have exquisite milestone or checkpoint documentation of how your project worked last February if you can't get a copy of February's code and data. This is why you need a good version control discipline.
Deploy demonstrations	True production deployments are best done by experienced engineers. These engineers know the tools and environment they will be deploying to. A good way to jump start production deployment is to have a reference deployment. This allows engineers to experiment with your work, test corner cases and build acceptance tests.

Concretely, this chapter will cover:

- Using `knitr` to create substantial project milestone documentation and to automate reproduction of graphs and other results.
- Effective use of comments in code.
- Using `git` for version management, and for collaboration.
- Deploying models as http services and exporting model results.

10.1 The buzz dataset

Our example dataset for this and the following chapter is the "buzz dataset" from <http://ama.liglab.fr/datasets/buzz/>. We are going to work with the data found in `TomsHardware-Relative-Sigma-500.data.txt`.¹ The original supplied documentation (`TomsHardware-Relative-Sigma-500.names.txt` and `BuzzDataSetDoc.pdf`) tells us the buzz data is structured as follows.

Footnote 1 All files mentioned in this chapter are available from
<https://github.com/WinVector/zmPDSwR/tree/master/Buzz>.

Table 10.2 Buzz data description

Attribute	Description
Rows	Each row represents many different measurements of the popularity of a technical personal computer discussion topic.
Topics	Topics include technical issues about personal computers such as: brand names, "memory", "overclocking" and so on.
Measurement types	For each topic measurement types are quantities such as the number of discussions started, number of posts, number of authors, number of readers, and so on. Each measurement is taken at eight different times.
Times	The eight relative times are named 0 through 7 and are likely days (the original variable documentation is not completely clear and the matching paper has not yet been released). For each measurement type all eight relative times are stored in different columns in the same data row.
Buzz	The quantity to be predicted is called "buzz" and is defined as being true or "1" if the ongoing rate of additional discussion activity is at least 500 events per day averaged over a number of days after the observed days. Likely buzz is a future average of the seven variables labeled "NAC" (the original documentation is unclear on this).

In our initial buzz documentation we list what we know (and importantly, admit what we are not sure about). We do not intend any disrespect in calling out issues in the supplied buzz documentation. That documentation is about as good as you see at the beginning of a project. In an actual project you would clarify and improve unclear points through discussions and work cycles. This is one reason having access to active project sponsors and partners is critical in real world projects.

The buzz problem demonstrates some features that are common in actual data science projects:

- This is a project where you are trying to predict the future from past features. These sort of projects are particularly desirable as you can expect to produce a lot of training data by saving past measurements.
- The quantity to be predicted is a function of future values of variables you are measuring. So part of the problem is re-learning the business rules that make the determination. In such cases it may be better to steer the project to predict estimates of the future values in question and leave the decision rules to the business.
- A domain specific re-shaping of the supplied variables would be appropriate. We are given daily popularities of articles over 8 days, we would prefer variables that represent popularity summed over the measured days, variables that measure topic age, variables that measure shape (indicating topics that are falling off fast or slow), and other time-series specific features.

In this chapter we are going to use the buzz data set as is and concentrate on demonstrating the tools and techniques used in producing documentation, deployments and presentations. In actual projects we advise you to start by producing notes like in table 10.2. You would also incorporate meeting notes to document your actual project goals. As this is only a demonstration we will emphasize technical documentation: data provenance and an initial trivial analysis to demonstrate we have control of the data. Our example initial buzz analysis is found here: <https://github.com/WinVector/zmPDSwR/blob/master/Buzz/buzzm.md>.² We suggest you skim it before we work through the tools and steps used to produce these documents in our next section.

Footnote 2 Also available in PDF form: <https://github.com/WinVector/zmPDSwR/raw/master/Buzz/buzz.pdf>

10.2 Using knitr to produce milestone documentation

The first audience you will have to prepare documentation for is: peers and yourself. You may need to return to previous work months later, and it may be in an urgent situation like an important bug fix, presentation or feature improvement. For self/peer documentation you want to concentrate on facts: what the stated goals were, where the data came from and what techniques were tried. You assume as long as you use standard terminology or references that the reader can figure out anything else they need to know. You want to emphasize any surprises or exceptional issues, as they are exactly what is very expensive to re-learn. You can't expect to share this sort of documentation with clients, but you can later use it as a basis for building wider documentation and presentations.

The first sort of documentation we recommend is project milestone or checkpoint documentation. At major steps of the project you should take some time out to repeat your work in a clean environment (proving you know what is in

intermediate files and you can in fact recreate them). An important, and often neglected, milestone is the start of a project. In this section we will use the `knitr` R package to document starting work with the buzz data.

10.2.1 What is knitr?

knitr is a R package allows the inclusion of R code and results inside documents. knitr's operation is similar in concept to Knuth's literate programming and to the R Sweave package. In practice you maintain a master file that contains both user readable documentation and chunks of program source code. The document types supported by knitr include Latex, Markdown and HTML. During a "knit" knitr extracts and executes all of the R code and then builds a new result document that assembles the contents of the original document plus pretty-printed code and results (see figure 10.1).

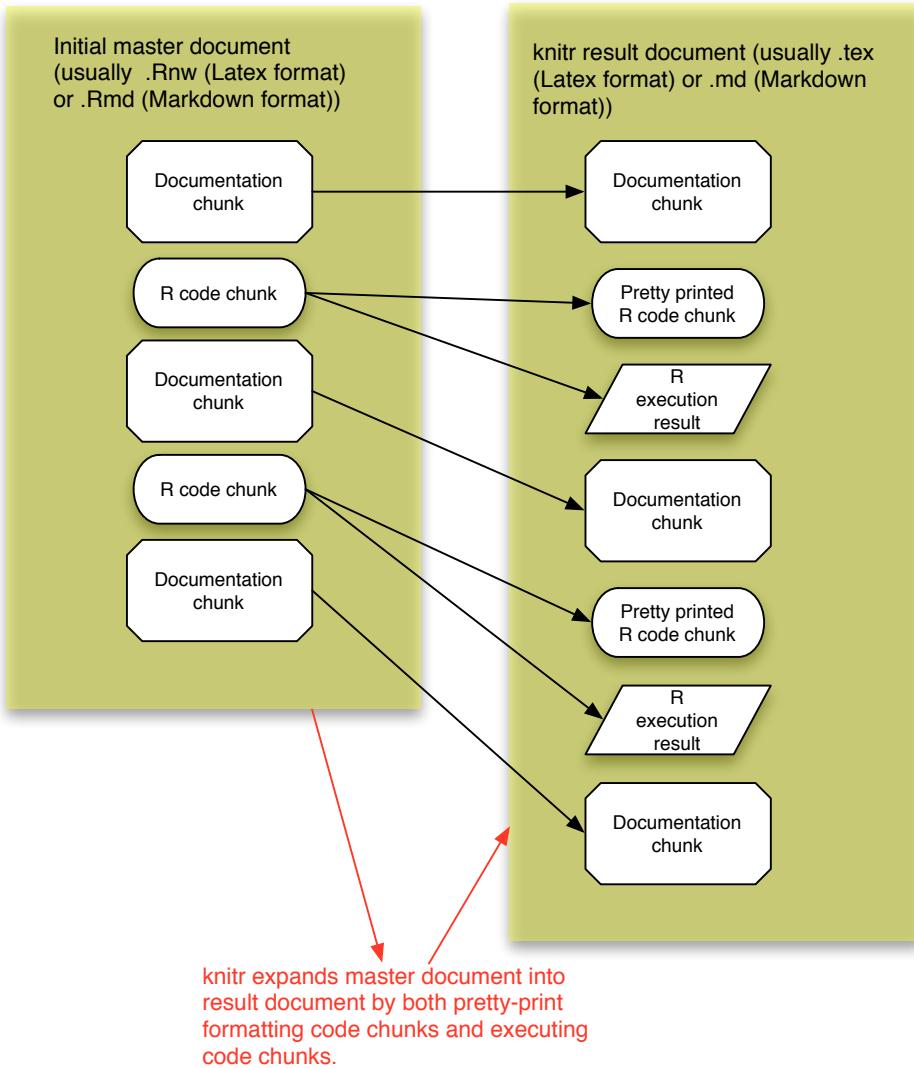


Figure 10.1 knitr process schematic

The process is best demonstrated with a few examples.

A SIMPLE KNITR MARKDOWN EXAMPLE

Markdown³ is a simple web-ready format that is used in many wikis. Listing 10.1 shows a simple Markdown document with knitr annotation blocks denoted with `````.

Footnote 3 <http://daringfireball.net/projects/markdown/>

Listing 10.1 knitr annotated Markdown

```
# Simple knitr Markdown example ①

Two examples:

* plotting
* calculating

Plot example:
```{r plotexample, fig.width=2, fig.height=2, fig.align='center'} ②
library(ggplot2) ③
ggplot(data=data.frame(x=c(1:100),y=sin(0.1*c(1:100)))) +
 geom_line(aes(x=x,y=y))
``` ④

Calculation example: ⑤
```{r calceexample} ⑥
pi*pi
```
```

- ① Markdown text and formatting
- ② Knitr chunk open with option assignments
- ③ R code
- ④ Knitr chunk close
- ⑤ More Markdown text
- ⑥ Another R code chunk

We save listing 10.1 in a file named "simple.Rmd." In R we would process this as follows:

```
library(knitr)
knit('simple.Rmd')
```

This produces the new file "simple.md" which is in Markdown format and appears (with the proper viewer) as in figure 10.2⁴.

Footnote 4 We used `pandoc -o simple.html simple.md` to convert the file to easily viewable html.

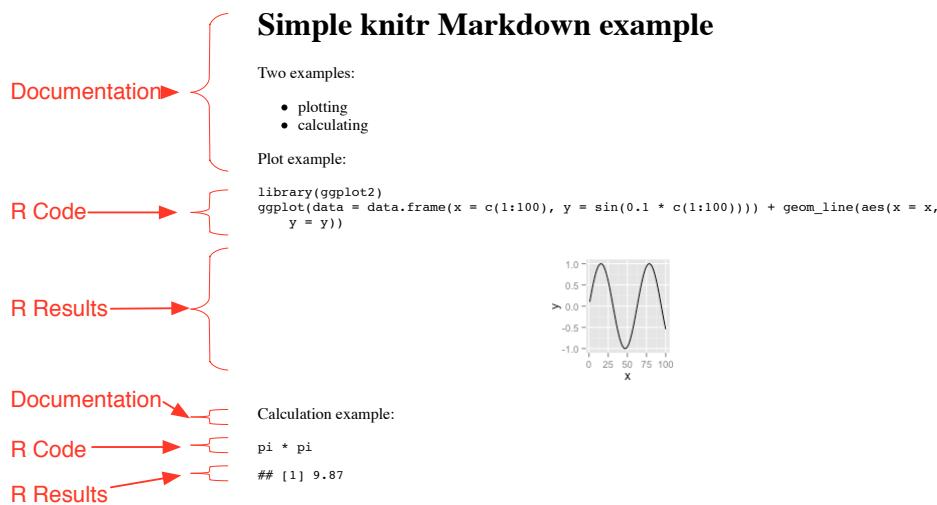


Figure 10.2 Simple knitr Markdown result

A SIMPLE KNITR LATEX EXAMPLE

Latex is a powerful document preparation system suitable for publication quality typesetting both for articles and entire books. To show how to use knitr with Latex we work through a simple example. The main new feature is in Latex code-blocks are marked with << and @ instead of `~~~. A simple Latex document with knitr chunks looks like the following:

```
\documentclass{article} ①
\begin{document} ②
<<nameofblock>>= ③
1+2 ④
@ ⑤
\end{document} ⑥
```

- ① Latex declarations (not knitr).
- ② Latex declarations (not knitr).
- ③ knit start chunk marker
- ④ R code
- ⑤ knit end chunk marker
- ⑥ Latex declarations (not knitr).

We save this content into a file named "add.Rnw" and then (using the bash shell) run R in batch to produce the file "add.tex". At a shell we then run Latex to create the final "add.pdf" file as we show below.

```
echo "library(knitr); knit('add.Rnw')" | R --vanilla ①  
pdflatex add.tex ②
```

- ① Use R in batch mode to create add.tex from add.Rnw.
- ② Use Latex to create add.pdf from add.tex.

This produces the PDF as shown in figure 10.3.



Figure 10.3 Simple knitr Latex result

THE PURPOSE OF KNITR

The purpose of knitr is to produce reproducible work.⁵ When you distribute your work in knitr format (as we do in section 10.2.3) anyone can download your work and without great effort: re-run it to confirm they get the same results you did. This is the ideal standard of scientific research, but is rarely met as scientists usually are deficient in sharing all of their code, data and actual procedures. knitr collects and automates all the steps, so it becomes obvious if something is missing or does not actually work as claimed. knitr automation may seem like a mere convenience: but it makes the following essential work much easier (and therefore more likely to actually be done):

Footnote 5 The knitr community calls this "reproducible research," but that is because scientific work is often called "research."

Table 10.3 Maintenance tasks made easier by knitr

| Task | Discussion |
|---|--|
| Keeping code in sync with documentation | With only one copy of the code (already in the document) you have a harder time getting out of sync. |
| Keeping results in sync with data | Eliminating all by-hand steps (such as cutting and pasting results, picking file names and including figures) makes it much more likely you will correctly re-run and re-check your work. |
| Handing off of correct work to others | If the steps are sequenced so a machine can run them, then it is much easier to re-run and confirming them. Also having a container (the master document) to hold all your work makes managing dependencies much easier. |

10.2.2 knitr technical details

To use knitr on a substantial project we need to know more about how knitr code chunks work. In particular we need to be clear how chunks are marked and what common chunk options you will need to manipulate.

KNITR BLOCK DECLARATION FORMAT

In general a knitr code block starts with the block declaration (` `` in Markdown and << in Latex). The first string is the name of the block (must be unique across the entire project). After that a number of comma separated `option=value` chunk option assignments are allowed.

KNITR CHUNK OPTIONS

A sampling of useful option assignments is given below.

Table 10.4 Some useful knitr options

| Option name | purpose |
|-------------|---|
| cache | Controls if results are cached. With <code>cache=F</code> (the default) the code chunk is always executed. With <code>cache=T</code> the code chunk is not executed if valid cached results are available from previous runs. Cached chunks are essential when you are revising knitr documents, but you should always delete the cache directory (found as a sub-directory of where you are using knitr) and do clean re-run to make sure your calculations are in fact using current versions of the data and settings you have specified in your document. |
| echo | Controls if source code is copied into the document. With <code>echo=T</code> (the default) pretty formatted code is added to the document. With <code>echo=F</code> code is not echoed (useful when you only want to display results) |
| eval | Controls if code is evaluated. With <code>eval=T</code> (the default) code is executed. With <code>eval=F</code> it is not (useful for displaying instructions). |
| message | Set <code>message=F</code> to direct R <code>message()</code> commands to the console running R instead of to the document. This is useful for issuing progress messages to the user that you don't want in the final document. |
| results | Controls what is to be done with R output. Usually you do not set this option and output is intermingled (with <code>##</code> comments) with the code. A useful option is <code>results='hide'</code> which suppresses output. |
| tidy | Controls if source code is re-formatted before being printed. You almost always want to set <code>tidy=F</code> as the current version of knitr often breaks code due to mishandling of R comments when re-formatting. |

Most of these options are demonstrated in our buzz example, which we work through in the next section.

10.2.3 Using knitr to document the buzz data

For a more substantial example we use knitr to document the initial data treatment and initial trivial model for the buzz data (recall: buzz is records of computer discussion topic popularity introduced in section 10.1). We will produce a document that outlines the initial steps of working with the buzz data (the sorts of steps we had, up until now, been including in this book whenever we introduce a new data set). This example works through advanced knitr topics such as: caching (to speed up re-runs), messages (to alert the user) and advanced formatting. We supply two examples of knitr for the buzz data at <https://github.com/WinVector/zmPDSwR/tree/master/Buzz>. The first example is in Markdown format and found in the knitr file `buzzm.Rmd` which knits to the Markdown file `buzzm.md`. The second example is in Latex format and found in the knitr file `buzz.Rnw` which kits to the Latex file `buzz.tex` (which in turn is used to produce the viewable file `buzz.pdf`). All steps we mention in this section are completely demonstrated in both of these files. We will show excerpts from `buzz.Rmd` (using the ````` delimiter) and excerpts from `buzz.Rnw` (using the `<<` delimiter).

NOTE**Buzz data notes**

For the buzz data the notes we usually include in this book are found in the files `buzz.md` and `buzz.pdf`. We strongly suggest reading one of these files and the feature table in section 10.1. The original description files from the buzz project (`TomsHardware-Relative-Sigma-500.names.txt` and `BuzzDataSetDoc.pdf`) are also available in our GitHub repository.

SETTING UP CHUNK CACHE DEPENDENCIES

For a substantial knitr project you are going to want to enable caching. Otherwise re-running knitr to correct typos becomes is prohibitively expensive. The standard way to enable knitr caching is to add the `cache=T` option to all knitr chunks. You also probably want to set up the chunk cache dependency calculator by inserting the following invisible chunk towards the top of your file.

```
% set up caching and knitr chunk dependency calculation  
% note: you will want to do clean re-runs once in a while to make sure
```

```
% you are not seeing stale cache results.  
<<setup,tidy=F,cache=F,eval=T,echo=F,results='hide'>>=  
opts_chunk$set(autodep=T)  
dep_auto()  
@
```

CONFIRMING DATA PROVENANCE

Because knitr is automating steps you can afford to take a couple of extra steps to confirm the data you are analyzing is in fact the data you thought you had. For example we start our buzz data analysis by confirming the sha cryptographic hash of the data we are starting from matches what we thought we had downloaded. This is done (assuming your system has the `sha` cryptographic hash installed) as follows (note: always look to the first line of chunks for chunk options such as `cache=T`).

```
<<dataprep,tidy=F,cache=T>>=  
infile <- "TomsHardware-Relative-Sigma-500.data.txt"  
paste('checked at',date())  
system(paste('shasum',infile),intern=T) # write down file hash  
buzzdata <- read.table(infile, header=F, sep=",")  
...
```

This code sequence produces the output shown in figure 10.4. In particular we have documented that the data we loaded has the same cryptographic hash we recorded when we first downloaded the data. Having confidence in this can speed up debugging when things go wrong.

```
infile <- "TomsHardware-Relative-Sigma-500.data.txt"  
paste('checked at',date())  
## [1] "checked at Fri Nov  8 15:01:39 2013"  
system(paste('shasum',infile),intern=T) # write down file hash  
## [1] "c239182c786baf678b55f559b3d0223da91e869c TomsHardware-Relative-Sigma-500.data.txt"
```

Figure 10.4 knitr documentation of buzz data load

RECORDING THE PERFORMANCE OF THE NAIVE ANALYSIS

The initial milestone is a good place to try and record the results of a naive "just apply a standard model to whatever variables are present" analysis. For the buzz this analysis (and its results) a random forest applied to test data performs as follows:

```
rtest <- data.frame(truth=buzztest$buzz,
  pred=predict(fmodel, newdata=buzztest))
print(accuracyMeasures(rtest$pred, rtest$truth))
## [1] "precision= 0.809782608695652 ; recall= 0.84180790960452"
##      pred
## truth    0   1
##       0 579  35
##       1  28 149
##   model accuracy      f1 dev.norm
## 1 model    0.9204 0.6817     4.401
```

USING MILESTONES TO SAVE TIME

Now that we have gone to all the trouble to implement, write-up and run the buzz data preparation steps we end our knitr analysis by saving the R workspace. We can then start additional analyses (such as introducing better shape features for the time varying data) from the saved workspace. In the following code snippet we show a conditional saving of the data (to prevent needless file churn) and again produce a cryptographic hash of the file (so we can confirm work that starts from a file with the same name is in fact starting from the same data).

```
Save prepared R environment.
% Another way to conditionally save, check for file.
% message=F is letting message() calls get routed to console instead
% of the document.
<<save,tidy=F,cache=F,message=F,eval=T>>=
fname <- 'thRS500.Rdata'
if(!file.exists(fname)) {
  save(list=ls(),file=fname)
  message(paste('saved',fname)) # message to running R console
  print(paste('saved',fname)) # print to document
} else {
  message(paste('skipped saving',fname)) # message to running R console
  print(paste('skipped saving',fname)) # print to document
}
paste('checked at',date())
system(paste('shasum',fname),intern=T) # write down file hash
```

@

Figure 10.5 shows the result. The data scientist can safely start their analysis on the saved workspace and has documentation that allows them to confirm that a workspace file they are using is in fact one produced by this version of the preparation steps.

Save prepared R environment.

```
fname <- 'thRS500.Rdata'
if(!file.exists(fname)) {
  save(list=ls(),file=fname)
  message(paste('saved',fname)) # message to running R console
  print(paste('saved',fname)) # print to document
} else {
  message(paste('skipped saving',fname)) # message to running R console
  print(paste('skipped saving',fname)) # print to document
}

## [1] "skipped saving thRS500.Rdata"

paste('checked at',date())

## [1] "checked at Fri Nov  8 15:32:54 2013"

system(paste('shasum',fname),intern=T) # write down file hash

## [1] "304895b8b5860ac5c995e10bd3b8c995820d60a0 thRS500.Rdata"
```

Figure 10.5 knitr documentation of prepared buzz workspace

KNITR TAKEAWAY

In our knitr example we worked through the steps we have done for every dataset in this book: load data, manage columns/variables, perform an initial analysis, present results, and save a workspace. The key point is: because this time we took the extra effort to do this work in knitr we have the following:

- Nicely formatted documentation (`buzz.md` and `buzz.pdf`).
- Shared executable code (`buzz.Rmd` and `buzz.Rnw`).

This makes debugging (which usually involves repeating and investigating earlier work), sharing and documentation much easier and more reliable.

10.3 Using comments and version control for running documentation

Another essential record of your work is what we call running documentation. Running documentation is more informal than milestone/checkpoint documentation and is easiest maintained in the form of code comments and version control records. Undocumented and untracked code runs up large *technical debt*⁶ that can cause problems down the road.

Footnote 6 See http://en.wikipedia.org/wiki/Technical_debt

In this section we will work through producing effective code comments and using git for version control record keeping.

10.3.1 Writing effective comments

R's comment style is very simple: everything following a # (that is not quoted) until the end of a line is a comment and ignored by the R interpreter. Listing 10.2 is an example of a good commented block of R code.

Listing 10.2 Example code comment

```
#     Return the pseudo logarithm of x, which is close to
# sign(x)*log10(abs(x)) for x such that abs(x) is large
# and doesn't "blow up" near zero. Useful
# for transforming wide-range variables that may be negative
# (like profit/loss).
# See: http://www.win-vector.com/blog
# /2012/03/modeling-trick-the-signed-pseudo-logarithm/
#     NB: This transform has the undesirable property of making most
# signed distributions appear bimodal around the origin, no matter
# what the underlying distribution really looks like.
# The argument x is assumed be numeric and can be a vector.
pseudoLog10 <- function(x) { asinh(x/2)/log(10) }
```

Good comments include: what the function does, what the types arguments are expected to be, limits of domain, why you should care about the function, and where it is from. Of critical importance are any "NB" (*nota bene* or "note well") or "TODO" notes. It is vastly more important to document any unexpected features or limitations in your code than to try and explain the obvious. Because R variables do

not have types (only objects they are pointing to have types) you many want to document what types of arguments you are expecting. It is critical to know if a function works correctly on lists, data frame rows, vectors, and so on.

Notice we did not bother with any of the following:

Table 10.5 Things not to worry about in comments

| Item | Why not to bother |
|--|---|
| Pretty "ASCII art" formatting. | It is enough that the comment be there and be readable. Formatting into a beautiful block just makes the comment harder to maintain and decreases the chance of the comment being up to date. |
| Anything we see in the code itself. | There is no point repeating the name of the function, saying it takes only one argument and so on. |
| Anything we can get from version control. | We don't bother recording the author or date the function was written. These facts, while important are easily recovered from your version control system with commands like <code>git blame</code> . |
| Any sort of Javadoc/Doxxygen style annotations | The standard way to formally document R functions is in separate <code>.Rd</code> (R documentation) files in a package structure (see http://cran.r-project.org/doc/manuals/R-exts.html). In our opinion the R package system is too specialized and tiresome to use in regular practice (though it is good for final delivery). For formal code documentation we recommend knitr. |

Also, avoid comments that add no actual content, such as in listing 10.3.

Listing 10.3 Useless comment

```
#####
# Function: addone
# Author: John Mount
# Version: 1.3.11
# Location: RSource/helperFns/addone.R
# Date: 10/31/13
# Arguments: x
# Purpose: Adds one
#####
addone <- function(x) { x + 1 }
```

The only thing worse than no documentation is documentation that is wrong. At all costs avoid comments that are incorrect (notice the comment says "adds one" when the code clearly adds two) as in listing 10.4 (and *do* delete such comments if you find them).

Listing 10.4 Worse than useless comment

```
# adds one
addtwo <- function(x) { x + 2 }
```

10.3.2 Using version control to record history

Version control can both maintain critical snapshots of your work in earlier states and produce running documentation of what was done by whom and when in your project. Figure 10.6 shows a cartoon "version control saves the day scenario" that is in fact very common.

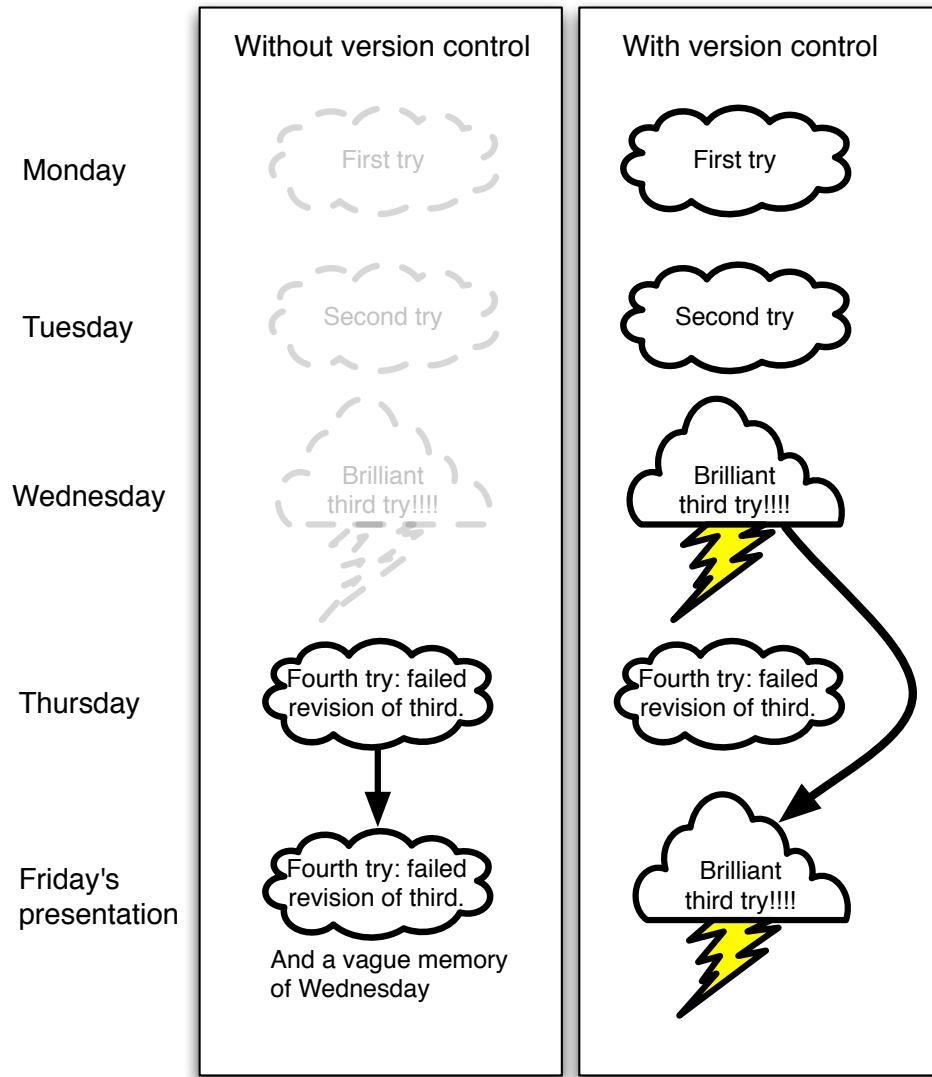


Figure 10.6 Version control saving the day

In this section we explain the basics of using git (<http://git-scm.com/>) as your version control system. To really get familiar with git I recommend a good book such as Jon Loeliger and Matthew McCullough's "Version Control with Git" 2nd Edition, O'Reilly 2012. Or, better yet, work with people who know git. In this chapter we assume you know how to run an interactive shell in on your computer (on Linux and OSX you tend to use "bash" as your shell, on Windows you can install Cygwin⁷).

Footnote 7 <http://www.cygwin.com/>

NOTE**Working in bright light**

Sharing your git repository means you are sharing a lot of information about your work habits and also sharing your mistakes. You are much more exposed than when you just share final work or status reports. Make this a virtue, know you are working in bright light. One of the most critical features in a good data scientist (perhaps even before analytic skill) is: scientific honesty.

For most of the benefit of single-user git you need to become familiar with a few commands:

- git init .
- git add -A .
- git commit
- git status
- git log
- git diff
- git checkout

We, unfortunately, will not have space to explain all of these commands. We will however, demonstrate how to think about git and the main path of commands you need to maintain your work history.

STARTING A GIT PROJECT USING THE COMMAND LINE

When you start a project do the following:

1. Start the project in a new directory. Place any work in either this directory or sub-directories.
2. Move your interactive shell into this directory and type "git init .". It is okay if you have already started working and there are already files present.

You can check if you have already performed the init step by typing "git status". If the init has not been done you will get a message similar to: "fatal: Not a git repository (or any of the parent directories): .git." If the init has been done you will get a status message telling you something like "on branch master" and listing facts about many files.

The init step sets up in your directory a single hidden file tree called ".git" and prepares you to keep extra copies of every file in your directory (including sub-directories). Keeping all of these extra copies is called "versioning" and what is meant by "version control." You can now start working on your project, save

everything related to your work in this directory or some sub-directory of this directory.

Again, you only need to init a project once. Do not worry about accidentally running "git init ." a second time, that is harmless.

USING ADD/COMMIT PAIRS TO CHECKPOINT WORK

TIP

Get nervous about uncommitted state

A good rule of thumb for git: you should be as nervous having uncommitted changes as you should be about not having pressed save. You don't need to push/pull often, but you do need to make local commits often (even if you later squash them with a git technique call "rebasing").

As often as practical enter the following two commands into an interactive shell in your project directory:

```
git add -A .    ①  
git commit      ②
```

- ① Stage results to commit (specify what files should be committed).
- ② Actually perform the commit.

Checking in file is split into two stages: add and commit. This has some advantages (such as allowing you to inspect before committing), but for now just consider the two commands as always going together. The commit command should bring up an editor where you enter a comment as to what you are up to. Until you are a "git expert" allow yourself easy comments like: "update", "going to lunch", "just added a paragraph" or "corrected spelling." Run the add/commit pair of commands after every minor accomplishment on your project. Run these commands every time you leave your project (to go to lunch, to go home or to work on another project). Do not fret if you forget to do this, just run the commands next time you remember.

NOTE**A "wimpy commit" is better than no commit**

We have been a little loose in our instructions of commit often and don't worry too much about having a long commit message. Two things to keep in mind are: usually you want commits to be meaningful with the code working (so you tend not to commit in the middle of an edit with syntax errors), and good commit notes are to be preferred (just don't forgo a commit because you don't feel like writing a good commit note).

USING GIT LOG AND GIT STATUS TO VIEW PROGRESS

Any time you want to know about your work progress type either "git status" to see if there are any edits you can put through the add/commit cycle or "git log" to see the history of your work (from the viewpoint of the add/commit cycles).

For example here is the "git status" from my copy of this book's examples repository⁸:

Footnote 8 <https://github.com/WinVector/zmPDSwR>

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

And a "git log" from the same project:

```
commit c02839e0b34172f54fd68201f64895295b9d7609
Author: John Mount <jmount@win-vector.com>
Date:   Sat Nov 9 13:28:30 2013 -0800

    add export of random forest model

commit 974a8d5b95bdf25b95d23ef75d08d8aa6c0d74fe
Author: John Mount <jmount@win-vector.com>
Date:   Sat Nov 9 12:01:14 2013 -0800

    Add rook examples
```

The indented lines are what text I entered at the `git commit` step, the dates are tracked automatically.

USING GIT THROUGH RSTUDIO

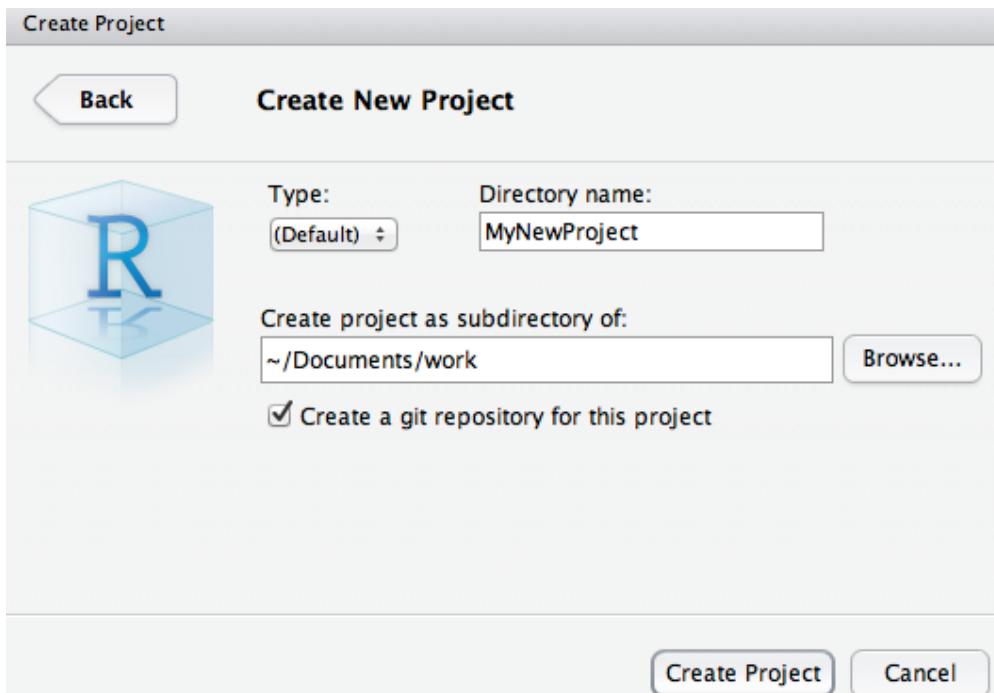


Figure 10.7 RStudio new project pane

The RStudio IDE supplies a graphical user interface to git that you should try. The add/commit cycle can be performed as follows in RStudio.

- Start a new project. From the RStudio command menu select `Project -> Create Project`. Next click on `New Project`. Then select the name of the project, what directory to create the new project directory in, leave type as "(Default)" and make sure "Create a git repository for this project" is checked. When the new project pane looks something like figure 10.7 you press "Create Project" and you have a new project.
- Do some work in your project. Create new files by selecting `File -> New -> R Script`. Type some R code (like 1/5 into the editor pane and then press the save icon to save the file. When saving the file be sure to choose your project directory or a sub-directory of your project.
- Commit your changes to version control. How to do this is shown in figure 10.8. You select the Git control pane in the top-right of RStudio. This pane shows all changed files as line items. Check the "Staged" check box for any files you want to stage for this commit. Then press the "Commit" button and you are done.

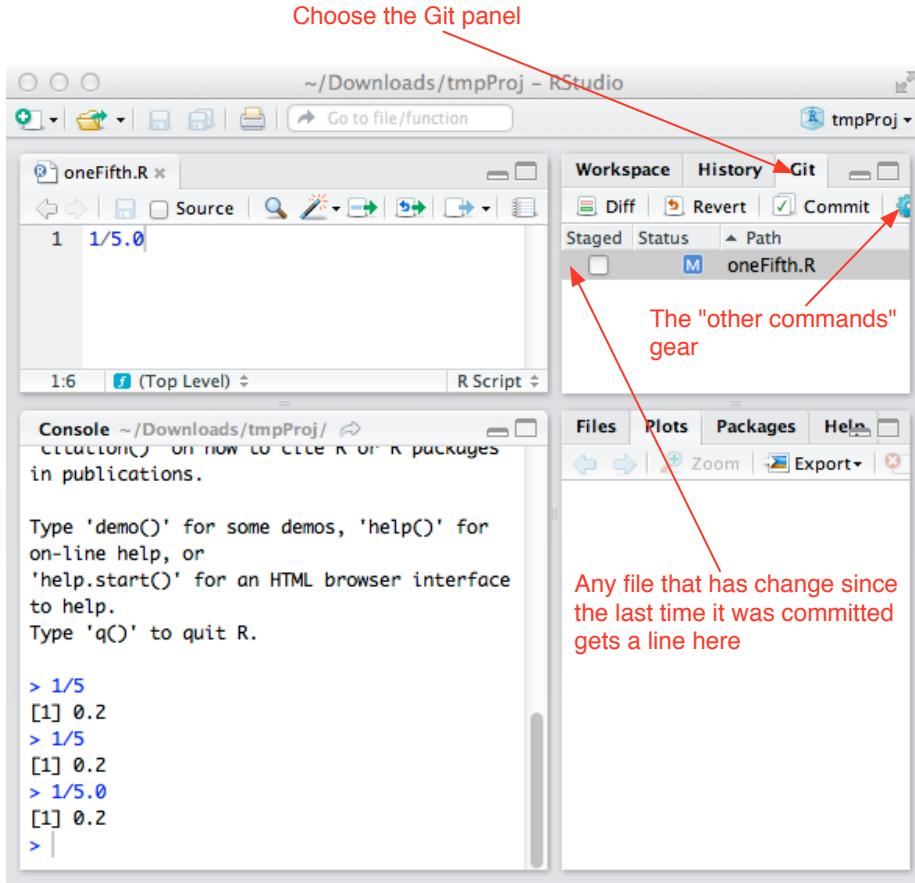


Figure 10.8 RStudio Git controls

You may not yet deeply understand or like Git, but you are able to safely check in all of your changes every time you remember to stage and commit. This means all of your history is there, you can't clobber your committed in work just by deleting your working file. Consider all of your working directory as "scratch work" only work checked in is safe from loss.

Your git history can be seen by pulling down on the "other commands" gear shown in the git pane in figure 10.8 and selecting "history" (don't confuse this with the nearby history pane, which is command history, not git history). In an emergency you can find git help and find your earlier files. If you have been checking in, then your older versions are there, it is just a matter of getting some help in accessing them. Also, if you are working with others you can use the push/pull menu items to publish and receive updates. All we want to say about version control at this point: *commit often and if you are committing often all problems can be solved with some further research*. Also be aware since your primary version control is on your own machine you need to separately make sure

your machine is backed up, as if your machine fails you lose both your work and your version repository.

10.3.3 Using version control to explore your project

Up until now our model of version control is: git keeps a complete copy of all of your files at each time you successfully entered the pair of add/commit lines. We will now use these commits. If you add/commit often enough git is ready to help you with any of the following tasks:

- Tracking your work over time.
- Recovering a deleted file.
- Comparing two past versions of a file.
- Finding when you added a specific bit of text.
- Recovering a whole file or a bit of text from the past (undo an edit).
- Sharing files with collaborators.
- Publicly sharing your project (ala GitHub <https://github.com/>).
- Maintaining different versions (branches) of your work.

And that is why you want to add and commit very often.

TIP

Getting help on git

For any git command you can try `git help command` to get usage information. For example: to learn about `git log` type `git help log`.

FINDING OUT WHO WROTE WHAT AND WHEN

In Section 10.3.1 we implied that a good version control system can produce a lot of documentation on its own. One powerful example is the command `git blame`. Look what happens if we download the git repository <https://github.com/WinVector/zmPDSwR> (with the command `git clone git@github.com:WinVector/zmPDSwR.git`) and run the command `git blame README.md`:

```
git blame README.md
376f9bce (John Mount 2013-05-15 07:58:14 -0700 1) ## Support ...
376f9bce (John Mount 2013-05-15 07:58:14 -0700 2) # by Nina ...
2541bb0b (Marius Butuc 2013-04-24 23:52:09 -0400 3)
2541bb0b (Marius Butuc 2013-04-24 23:52:09 -0400 4) Works deri ...
2541bb0b (Marius Butuc 2013-04-24 23:52:09 -0400 5)
```

We have truncated lines for readability. But the `git blame` information takes each line of the file and prints the following:

- The prefix of the line's git commit hash. This is used to identify which commit the line we are viewing came from.
- Who committed the line.
- When they committed the line.
- The line number.
- And finally the contents of the line.

VIEWING A DETAILED HISTORY OF CHANGES

The two main ways to view the detailed history of your project are commands like `git log --graph --name-status` and the GUI tool `gitk`. Continuing our <https://github.com/WinVector/zmPDSwR> example we see the recent history of the repository by executing the `git log` command as in the example below:

```
git log --graph --name-status
* commit c49c853cbcble5a923d6e1127aa54ec7335d1b3
| Author: John Mount <jmount@win-vector.com>
| Date:   Sat Oct 26 09:22:02 2013 -0700
|
|       Add knitr and rendered result
|
| A     Buzz/.gitignore
| A     Buzz/buzz.Rnw
| A     Buzz/buzz.pdf
|
* commit 6ce20dd33c5705b6de7e7f9390f2150d8d212b42
| Author: John Mount <jmount@win-vector.com>
| Date:   Sat Oct 26 07:40:59 2013 -0700
|
|       update
|
| M     CodeExamples.zip
```

This variation of the `git log` command draws a graph of the history (mostly a straight line, which is the simplest possible history) and what files were added (the `A` lines), modified (the `M` lines), and so on. Commit comments are shown. Note that commit comments can be short. We can say things like "update" instead of

"update CodeExamples.zip" because git records what files were altered in each commit. The `gitk` GUI allows similar views and browsing through the detailed project history as shown in figure 10.9.

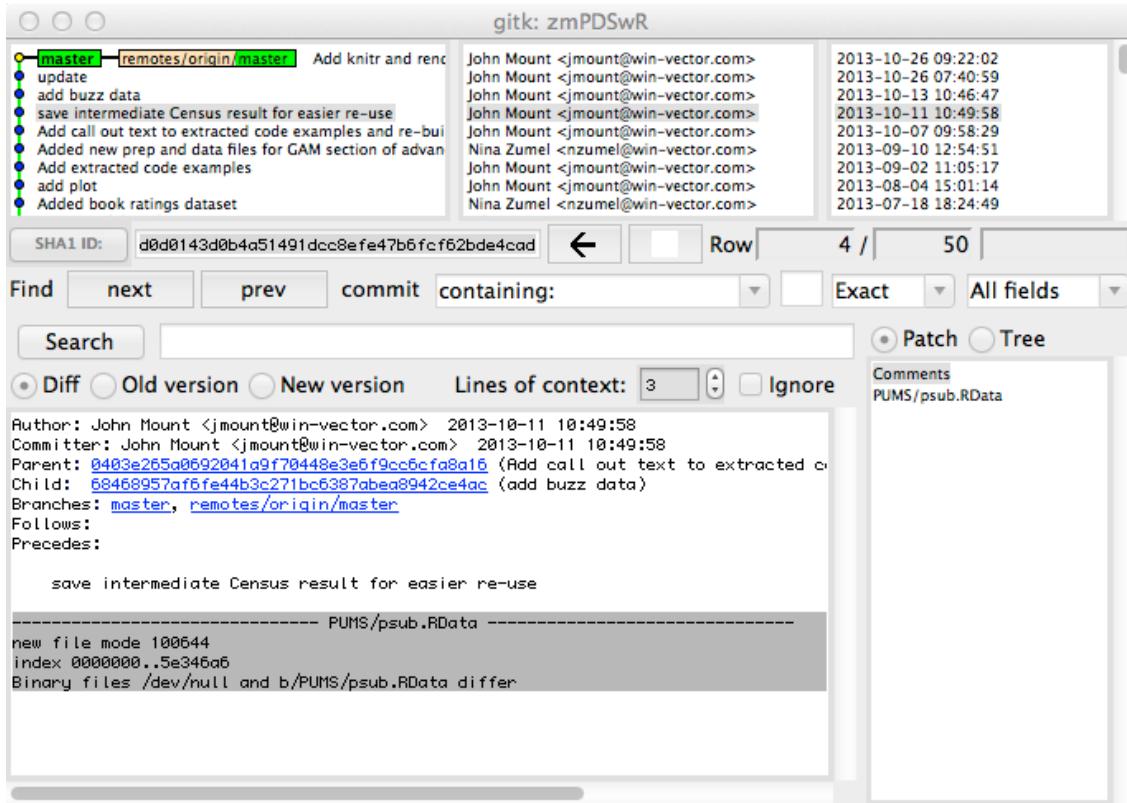


Figure 10.9 gitk browsing <https://github.com/WinVector/zmPDSwR>

USING GIT DIFF TO COMPARE FILES FROM DIFFERENT COMMITS

The `git diff` command allows you to compare any two committed versions of your project or even to compare your current work to any earlier version. In git commits are named by very large hash keys, but you are allowed to use prefixes of the hashes as names of commits.⁹. For example listing 10.5 demonstrates finding the differences in two versions of <https://github.com/WinVector/zmPDSwR> in a diff or patch format.

Footnote 9 You can also create meaningful names for commits with the `git tag` command.

Listing 10.5 Finding line based differences between two different committed versions.

```
diff --git a/CDC/NatalBirthData.rData b/CDC/NatalBirthData.rData
...
+++ b/CDC/prepBirthWeightData.R
@@ -0,0 +1,83 @@
+data <- read.table("natal2010Sample.tsv.gz",
+                    sep="\t", header=T, stringsAsFactors=F)
+
+## make a boolean from Y/N data
+makevarYN = function(col) {
+  ifelse(col %in% c("", "U"), NA, ifelse(col=="Y", T, F))
+}
...
...
```

WARNING Try not to confuse git commits and git branches.

A git commit represents the complete state of a directory tree at a given time. A git branch represents a sequence of commits and changes as you move through time. Commits are immutable branches record progress.

USING GIT LOG TO FIND THE LAST TIME A FILE WAS AROUND

A common question when working with on a project for a while is: when did I delete a certain file and what had been in it at the time. Git makes answering this question very easy. We will demonstrate this in the repository <https://github.com/WinVector/zmPDSwR>. This repository has a README.md (Markdown) file, but I remember starting with a simple text file. When and how did that file get deleted? To file out I run the following (command after the \$ prompt and the rest of the text the result).

```
$ git log --name-status -- README.txt

commit 2541bb0b9a2173eb1d471e11d4aca3b690a011ef
Author: Marius Butuc <marius.butuc@gmail.com>
Date:   Wed Apr 24 23:52:09 2013 -0400

        Translate readme to Markdown

D      README.txt
```

```
commit 9534cff7579607316397ccb40f120d286b7e4b58
Author: John Mount <jmount@win-vector.com>
Date:   Thu Mar 21 17:58:48 2013 -0700

    update licenses

M       README.txt
```

Ah: the file was deleted by Marius Butuc, an early book reader who generously composed a pull request to change my text file to Markdown (I reviewed and accepted the request at the time). I can view the contents of this older file with `git show 9534cf -- README.txt` (the `9534cff` is the prefix of the commit number before the deletion, manipulating these commit numbers is not hard if you use copy and paste). And can recover that copy of the file with `git checkout 9534cf -- README.txt`.

10.3.4 Using version control to share work

In addition to producing work you must often share it with peers. The common (and bad) way to do this is "emailing zip-files." Most of the bad sharing practices take excessive effort, are error prone and rapidly cause confusion. We advise using version control to share work with peers. To do that effectively with git you need to start using additional commands such as "git pull," "git rebase," and "git push." Things seem more confusing at this point (though you still do not yet need to worry about branching in its full generality), but are in fact far less confusing and far less error prone than ad-hoc solutions. I almost always advise sharing work in "star workflow" where each worker has their own repository and a single common "naked" repository (that is a repository with only git data structures and no ready to use files) is used to coordinate (thought of as a server or gold standard, often named "origin").

The usual shared workflow is:

- Continuously: work, work, work.
- Very often: commit results to the local repository using a "git add" "git commit" pair.
- Every once in a while: pull a copy of the remote repository into our view with some variation of "git pull" and then use "git push" to push our work upstream.

The main rule of git is: don't try anything clever (push/pull so on) unless you are in a "clean" state (everything committed, confirm with `git status`).

SETTING UP REMOTE REPOSITORY RELATIONS

For two our more git repositories to share work the repositories need to know about each other through a relation called "remote." A git repository is willing to its work to a remote repository by the push command and pick up work from a remote repository by the pull command. Snippet 10.6 shows the declared remotes for the author's local copy of the <https://github.com/WinVector/zmPDSwR> repository.

Listing 10.6 git remote

```
$ git remote --verbose
origin  git@github.com:WinVector/zmPDSwR.git (fetch)
origin  git@github.com:WinVector/zmPDSwR.git (push)
```

The remote relation is set when you create a copy of a repository using the `git clone` command or can be set using the `git remote add` command. In listing 10.6 we see the remote repository is called "origin", this is the traditional name for a remote repository that you are using as your master or gold standard (git tends not to use the name "master" for repositories because "master" is the name of the branch you are usually working on).

USING PUSH AND PULL TO SYNCHRONIZE WORK WITH REMOTE REPOSITORIES

Once your local repository has declared some other repository as remote you can push and pull between the repositories. When pushing or pulling always make sure you are "clean" (have no uncommitted changes) and you usually want to pull before you push (as that is the quickest way to spot and fix any potential conflicts). For a description of what version control conflicts are and how to deal with them see <http://www.win-vector.com/blog/2013/10/resolving-git-pseudo-conflicts/>.

Usually for simple tasks we don't use branches (a technical version control term) and we use the "rebase" option on pull so that it appears that every piece of work is recorded into a simple linear order, even though collaborators are actually working in parallel. This is what I call an *essential* difficulty of working with others: time and order become separate ideas and become hard to track (and this is *not* a needless complexity added by using git, there are such needlessnesses; but this is not one of them).

The new git commands you need to learn are:

- `git push` (usually used in the `git push -u origin master` variation).
- `git pull` (usually used in the `git pull --rebase origin master` variation).

Typically two authors may be working on different files in the same project at the same time. As we see in figure 10.10 below the second author to try to push their results to the shared repository must decide how to specify the parallel work was performed. Either they can say the work was truly in parallel (represented by two branches being formed and then a merge record joining the work) or they can rebase their own work to claim their work was done "after" the other's work (preserving a linear edit history and avoiding the need for any merge records). Note: "before" and "after" are tracked in terms of arrows, not time.

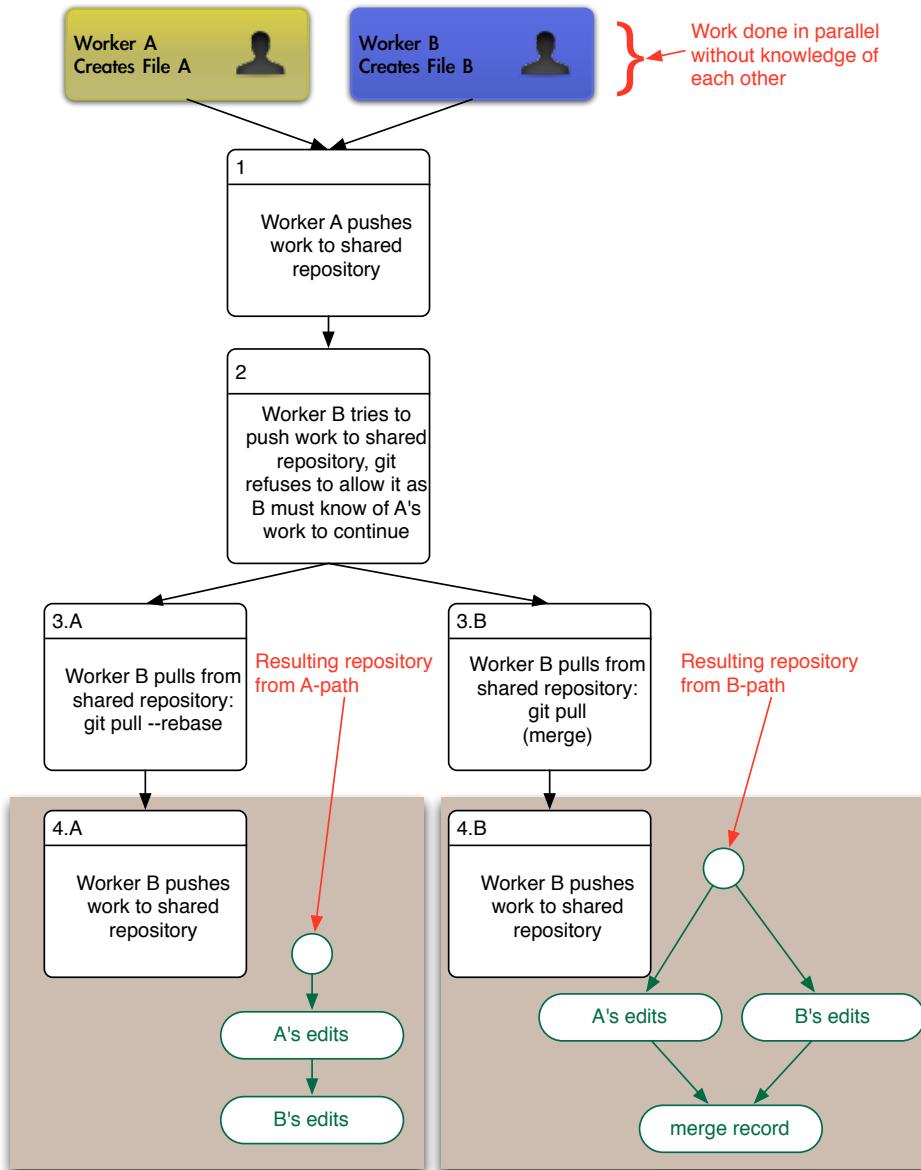


Figure 10.10 git pull: rebase versus merge

The idea: merging is what is really happening, but rebase is a much simpler to read. The general rule is you should only rebase work you have not yet shared (in our above example B should feel free to rebase their edits to appear to be after A's edits as they have not yet successfully pushed their work anywhere). You should avoid rebasing things records people have seen as you are essentially hiding the edit steps they may be basing their work on (forcing them to merge or rebase in the future to catch up with your changed record keeping).

For most projects I try to use a rebase-only strategy. For example: this book itself is maintained in a git repository. We have only two authors who are in close

proximity (so able to easily coordinate) and we are only trying to create one final copy of the book (we are not trying to maintain many branches for other uses). If we always rebase the edit history will appear totally ordered (for each pair of edits one is always recorded as having come before the other) and this makes talking about versions of the book much easier (again: before is determined by in arrows in the edit history, not by time stamp).

TIP**Don't confuse version control with backup**

git keeps multiple copies and records of all of your work. But until you push to a remote destination all of these copies are on your machine in the .git directory. So do not confuse basic version control with remote backups, they are complementary.

A BIT ON THE GIT PHILOSOPHY

Git is rather interesting in that it automatically detects and manages so much of what you would have to specify with other version control systems (example: git finds which files have changed instead of you specifying it, git also decides which files are related on its own). Because of the large degree of automation the beginning user usually a severe under-estimate of how much git in fact tracks for them. This makes git fairly quick except when git insists you help decide how a possible global inconsistency should be recorded in history (either as a rebase or a branch followed by a merge record). The point is: git suspects possible inconsistency based on global state (even when the user may not think there is such) and then forces the committer to decide how to annotate the issue *at the time of commit* (a great service to any possible readers in the future). Git automates so much of the record keeping it is always a shock when you have a conflict and have to express opinions on nuances you did not know were being tracked. Git is also a "anything is possible, but nothing is obvious or convenient" system. This is very hard on the user at first, but in the limit is much better than a "everything is smooth, but very little is possible" version control system (which can leave you stranded).

TIP**Keep notes**

Git commands are confusing, you will want to keep notes. One idea is to write a three by five card for each command you are regularly using. Ideally you can be at the top of your git game with about seven cards.

10.4 Deploying models

A good rule of writing is "show not tell." So while good documentation and presentation are vital for a successful data science collaboration, deploying at least a demonstration of your predictive model is vital. We strongly encourage partnering with a development group to produce the actual production hardened version of your model, but a good demonstration helps recruit these collaborators.

We outline some deployment method below:

Table 10.6 Methods to demonstrate predictive model operation

| Method | Description |
|------------------------|--|
| Batch | Data is brought into R, scored and then written back out. This is essentially an extension of what you are already doing with test data |
| Cross language linkage | R supplies answers to queries from another language (C, C++, Python, Java and so on). R is in fact designed with efficient cross language calling in mind (in particular the Rcpp package), but this is a specialized topic we will not cover here. |
| Services | R can be set up as an HTTP service to take new data as an HTTP query and respond with results. |
| Export | Often model evaluation is fairly simple compared to model construction. In this case the data scientist can export the model and a specification for the code to evaluate the model and the production engineers implement (with tests) model evaluation in the language of their choice (SQL, Java, C++ and so on). |

We have already demonstrated batch operation of models each time we applied a model to a test set. We will not work through a R cross language linkage example as it is very specialized, and requires knowledge of the system you are trying to link to. We will however demonstrate service and export strategies.

10.4.1 Deploying models as R http services

One easy way to demonstrate an R model in operation is to expose it as an http service. In listing 10.7 we show how to do this for our buzz model (predicting discussion topic popularity).

Listing 10.7 Buzz model as a R-based http service

```
library(Rook)          ①
load('thRS500.Rdata') ②
library(randomForest)   ③
numericPositions <- sapply(buzztrain[,varslist],is.numeric)    ④

modelFn <- function(env) { ⑤
  errors <- c()
  warnings <- c()
  val <- c()
  row <- c()
  tryCatch(
    {
      arg <- Multipart$parse(env) ⑥
      row <- as.list(arg[varslist])
      names(row) <- varslist
      row[numericPositions] <- as.numeric(row[numericPositions])
      frame <- data.frame(row)
      val <- predict(fmodel,newdata=frame)
    },
    warning = function(w) { message(w)
      warnings <- c(warnings,as.character(w)) },
    error = function(e) { message(e)
      errors <- c(errors,as.character(e)) }
  )
  body <- paste( ⑦
    'val=' , val , '\n',
    'nerrors=' , length(errors) , '\n',
    'nwarnings=' , length(warnings) , '\n',
    'query=' , env$QUERY_STRING , '\n',
    'errors=' , paste(errors,collapse=' ') , '\n',
    'warnings=' , paste(warnings,collapse=' ') , '\n',
    'data row' , '\n',
    paste(capture.output(print(row)),collapse='\n') , '\n',
    sep=' ')
  list(
    status=ifelse(length(errors)<=0,200L,400L),
    headers=list('Content-Type' = 'text/text'),
    body=body )
}

s <- Rhttpd$new()          ⑧
s$add(name="modelFn",app=modelFn) ⑨
s$start()                  ⑩
##starting httpd help server ... done
##
##Server started on host 127.0.0.1 and port 11953 . App urls are:
```

```
##  
##      http://127.0.0.1:11953/custom/modelFn    ⑪
```

- ① Load the rook http server library.
- ② Load the saved buzz workspace (includes the random forests model).
- ③ Load the random forests library (loading the workspace does not load the library).
- ④ Determine which variables are numeric (in the rook server everything defaults to character).
- ⑤ Declare the modeling service.
- ⑥ This block does the actual work: parse data and apply the model.
- ⑦ Format results, place in a list and return.
- ⑧ Start a new rook http service.
- ⑨ Register our model function as a http service.
- ⑩ Start the http server.
- ⑪ This is the URL the service is running at.

We show how to call the http service in listing 10.8.

Listing 10.8 Calling the buzz http service

```
rowAsForm <- function(url,row) { ①  
  s <- paste('<HTML><HEAD></HEAD><BODY><FORM action=""',url,  
             '" enctype="multipart/form-data" method="POST">\n',sep='')  
  s <- paste(s,'<input type="submit" value="Send"/>',sep='\n')  
  qpaste <- function(a,b) {  
    paste('<p> ',a,' <input type="text" name=""',a,  
          '" value=""',b,'" /> </p>',sep='') }  
  assignments <- mapply('qpaste',varslist,as.list(row)[varslist])  
  s <- paste(s,paste(assignments,collapse='\n'),sep='\n')  
  s <- paste(s,'</FORM></BODY></HTML>',sep='\n')  
  s  
}  
  
url <- 'http://127.0.0.1:11953/custom/modelFn' ②  
cat(rowAsForm(url,buzztest[7,]),file='buzztest7.html') ③
```

- ① Function to convert a row of dataset into a huge HTML form that transmits all of the variable values to HTTP server on submit (when the SEND button is pressed).
- ② The url we started the rook http server on.
- ③ Write the form representing the variables for the 7th test example to a file.

This produces the HTML form buzztest7.html shown in figure 10.11. (also saved in our example GitHub repository).

Send

num.new.disc0

num.new.disc1

num.new.disc2

num.new.disc3

num.new.disc4

Figure 10.11 Top of HTML form that asks server for buzz classification on submit

When the "SEND" button is pressed on buzztest7.html is submitted (and assuming the form's action is pointing to a valid server and port) we get a result (saved in GitHub as buzztest7res.txt). which we excerpt below:

```
val=1
nerrors=0
nwarnings=0
...
```

Notice the result is a prediction of "val=1" which was what we would expect for the 7th row of the test data. If you were pushing this further you could move to more machine friendly formats such as JSON, but this is far enough for an initial demonstration.

10.4.2 Deploying models by export

Because training is often the hard part of building a model it often makes sense to export a finished model for use by other systems. For example: a lot of theory goes into how a random forest pick variables and builds its trees. The structure of our random forest model is large but simple: a big collection of decision trees. But the construction is quite time consuming and technical. The idea is: it can be easier to fax a friend a solved Sudoku puzzle then to teach them your entire solution strategy.

So it often makes sense to export a copy of the finished model from R (instead of attempting to reproduce all of the details of model construction). When exporting a model you are depending on development partners to handle the hard parts of hardening a model for production (versioning, dealing with exceptional conditions, and so on). Software engineers tend to be very good at project management and risk control, so export projects are also a very good opportunity to learn.

The steps required depend a lot on the model and data treatment. For many models you only need to save a few coefficients. For random forests you need to export the trees. In all cases you need to write code to in your target system (be it SQL, Java, C, C++, Python, Ruby, and so on) to evaluate the model.¹⁰

Footnote 10 A fun example is the Salford Systems Random Forests package that exports models as source code instead of data. The package creates a compilable file in your target language (often Java or C++) that implements the decision trees essentially as a series of if statements over class variables.

One of the issues of exporting models is you must repeat any data treatment. So part of exporting a model is producing a specification of the data treatment (so it can be re-implemented outside of R).

In listing 10.9 we show how to export the buzz random forest model. Some investigation of the random forest model and documentation showed that the underlying trees are through a method called `getTree()`. In listing 10.9 we combine the description of all of these trees into a single table.

Listing 10.9 Exporting the random forest model

```
load('thRS500.Rdata')    ①
library(randomForest)     ②

extractTrees <- function(rfModel) { ③
  ei <- function(i) {
    ti <- getTree(rfModel,k=i,labelVar=T)
    ti$nodeid <- 1:dim(ti)[[1]]
    ti$treeid <- i
    ti
  }
  nTrees <- rfModel$ntree
  do.call('rbind',sapply(1:nTrees,ei,simplify=F))
}

write.table(extractTrees(fmodel),      ④
            file='rfmodel.tsv',row.names=F,sep='\t',quote=F)
```

- ① Load the saved buzz workspace (includes the random forests model).
- ② Load the random forests library (loading the workspace does not load the library).
- ③ Define a function that joins the tree-tables from the random forest `getTree()` method into one large table of trees.
- ④ Write the table of trees as a tab separated values table (easy to other software to read).

Figure 10.12 shows an extract of a single tree from the buzz random forest model. The tree is exported as the table of directions shown in the top of the figure and the interpretation is illustrated in the bottom. We have also saved the exported table in our example GitHub repository.¹¹

Footnote 11 at: <https://github.com/WinVector/zmPDSwR/blob/master/Buzz/rfmodel.tsv>

Table representation of decision tree

| left daughter | right daughter | split var | split point | status | prediction | nodeid | treeid |
|---------------|----------------|--------------------------|-------------|--------|------------|--------|--------|
| 2 | 3 | num.displays6 | 2,517.5 | 1 | NA | 1 | 1 |
| 4 | 5 | attention.level.author0 | 0.0004335 | 1 | NA | 2 | 1 |
| 6 | 7 | number.total.disc5 | 3.5 | 1 | NA | 3 | 1 |
| 8 | 9 | avg.disc.length4 | 127 | 1 | NA | 4 | 1 |
| 10 | 11 | contribution.sparseness0 | 0.0037735 | 1 | NA | 5 | 1 |
| 12 | 13 | num.displays7 | 3,948.5 | 1 | NA | 6 | 1 |
| 14 | 15 | num.displays1 | 3,326 | 1 | NA | 7 | 1 |
| 16 | 17 | num.authors.topic7 | 24.5 | 1 | NA | 8 | 1 |
| 0 | 0 | NA | 0 | -1 | 0 | 9 | 1 |
| 18 | 19 | avg.disc.length5 | 9 | 1 | NA | 10 | 1 |
| 20 | 21 | num.displays2 | 959.5 | 1 | NA | 11 | 1 |
| ... | | | | | | | |

Graphical interpretation of decision tree table

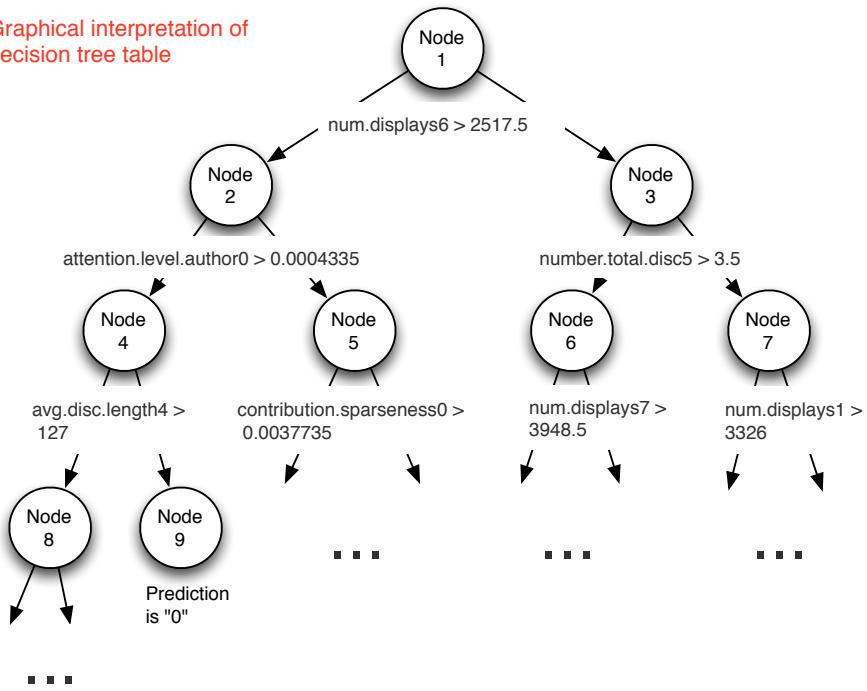


Figure 10.12 One tree from the buzz random forest model

Your developer partners would then build tools to read the model trees and evaluate the trees on new data. Previous test results and demonstration servers become the basis of important acceptance tests.

10.4.3 What to take away

You should now be comfortable demonstrating R models to others. Of particular power is setting up a model as an http service that can be experimented upon by others and also exporting models so model evaluation can be re-implemented in a production environment.

10.5 Summary

This chapter has been about how to mange, share and demonstrate your results. By now you should be comfortable:

- Using knitr to produce significant milestone/checkpoint documentation.
- Writing effective comments.
- Using git to save your work history.
- Using git to collaborate with others
- Setting up http demonstration services.
- Exporting model results for use in production.

10.6 External links section