

Business Intelligence with

Dwight Barry

Business Intelligence with R

From Acquiring Data to Pattern Exploration

Dwight Barry

This book is for sale at <http://leanpub.com/businessintelligencewithr>

This version was published on 2016-09-27



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.



This work is licensed under a [Creative Commons Attribution-ShareAlike 3.0 Unported License](#)

Contents

About	1
About the Author	1
Session Info	1
Version Info	1
Code and Data	1
Cover Image	1
Proceeds	2
Contact	2
install.packages	3
Introduction	4
Overview	4
Conventions	6
What you need	6
Acknowledgments	7
Website/Code	8
Happy coding!	8
Chapter 1: An entire project in a few lines of code	9
The analytics problem	9
Set up	10
Acquire data	11
Wrangling data	12
Analytics	16
Explore the data	16
Run a forecasting model	17
Reporting	19
Create an interactive HTML plot	19
Documenting the project	21
Summary	22
Chapter 2: Getting Data	23
Working with files	24

CONTENTS

Reading flat files from disk or the web	24
Reading big files with <code>data.table</code>	24
Unzipping files within R	25
Reading Excel files	26
Creating a data frame from the clipboard or direct entry	27
Reading XML files	28
Reading JSON files	29
Working with databases	31
Connecting to a database	31
Creating data frames from a database	32
Disconnecting from a database	33
Creating a SQLite database inside R	34
Creating a data frame from a SQLite database	37
Getting data from the web	39
Working through a proxy	39
Scraping data from a web table	39
Working with APIs	41
Creating fake data to test code	42
Writing files to disk	44
Chapter 3: Cleaning and Preparing Data	46
Understanding your data	46
Identifying data types and overall structure	46
Identifying (and ordering) factor levels	47
Identifying unique values or duplicates	47
Sorting	49
Cleaning up	50
Converting between data types	50
Cleaning up a web-scraped table: Regular expressions and more	50
Missing data: Converting dummy NA values	52
Rounding	52
Concatenation	53
Merging data frames	54
Joins	55
Unions and bindings	58
Subsetting: filter and select from a data frame	61
Creating a derived column	62
Peeking at the outcome with <code>dplyr</code>	62
Reshaping a data frame between wide and long	64
Reshaping: melt and cast	64
Reshaping: Crossing variables with <code>~</code> and <code>+</code>	66
Summarizing while reshaping	67
Piping with <code>%>%</code> : Stringing it together in <code>dplyr</code>	68

CONTENTS

Chapter 4: Know Thy Data—Exploratory Data Analysis	70
Creating summary plots	71
Everything at once: ggpairs	71
Create histograms of all numeric variables in one plot	73
A better “pairs” plot	74
Mosaic plot matrix: “Scatterplot” matrix for categorical data	75
Plotting univariate distributions	76
Histograms and density plots	76
Bar and dot plots	77
Plotting multiple univariate distributions with faceting	78
Plotting bivariate and comparative distributions	80
Double density plots	81
Boxplots	81
Beanplots	82
Scatterplots and marginal distributions	84
Mosaic plots	87
Multiple bivariate comparisons with faceting	88
Pareto charts	89
Plotting survey data	90
Obtaining summary and conditional statistics	92
Finding the mode or local maxima/minima	96
Inference on summary statistics	97
Confidence intervals	97
Tolerance intervals	101
Dealing with missing data	103
Visualizing missing data	103
Imputation for missing values	107
Chapter 5: Effect Sizes	112
Overview	112
Effect sizes: Measuring <i>differences</i> between groups	115
Basic differences	116
Standardized differences	120
Determining the probability of a difference	122
Effect sizes: Measuring <i>similarities</i> between groups	127
Correlation	127
Bootstrapping BCa CIs for non-parametric correlation	130
Determining the probability of a correlation	131
Partial correlations	132
Polychoric and polyserial correlation for ordinal data	133
Associations between categorical variables	134
Cohen’s kappa for comparisons of agreement	135
Regression coefficient	135

CONTENTS

R^2 : Proportion of variance explained	136
Chapter 6: Trends and Time	139
Describing trends in non-temporal data	139
Smoothed trends	140
Quantile trends	141
Simple linear trends	142
Segmented linear trends	143
The many flavors of regression	145
Plotting regression coefficients	146
Working with temporal data	147
Calculate a mean or correlation in circular time (clock time)	149
Plotting time-series data	150
Detecting autocorrelation	152
Plotting monthly and seasonal patterns	153
Plotting seasonal adjustment on the fly	154
Decomposing time series into components	155
Using spectral analysis to identify periodicity	159
Plotting survival curves	159
Evaluating quality with control charts	161
Identifying possible breakpoints in a time series	163
Exploring relationships between time series: cross-correlation	165
Basic forecasting	168
Chapter 7: A Dog's Breakfast of Dataviz	170
Plotting multivariate distributions	170
Heatmaps	170
Creating calendar heatmaps	172
Parallel coordinates plots	173
Peeking at multivariate data with dplyr and a bubblechart	175
Plotting a table	176
Interactive dataviz	177
Basic interactive plots	177
Scatterplot matrix	181
Motionchart: a moving bubblechart	182
Interactive parallel coordinates	183
Interactive tables with DT	183
Making maps in R	184
Basic point maps	185
Chloropleth maps	186
Chloropleth mapping with the <i>American Community Survey</i>	188
Using shapefiles and raw data	189
Using ggmap for point data and heatmaps	191

CONTENTS

Interactive maps with leaflet	194
Why map with R?	196
Chapter 8: Pattern Discovery and Dimension Reduction	197
Mapping multivariate relationships	197
Non-metric multidimensional scaling (nMDS)	197
Diagnostics for nMDS results	201
Vector mapping influential variables over the nMDS plot	203
Contour mapping influential variables over the nMDS plot	204
Principal Components Analysis (PCA)	205
nMDS for Categories: Correspondence Analysis	207
Cluster analysis	209
Grouping observations with hierarchical clustering	209
Plotting a cluster dendrogram with ggplot	210
Exploring hierarchical clustering of nMDS results	213
How to partition the results of a hierarchical cluster analysis	213
Identifying and describing group membership with kMeans and PAM	215
How to choose an optimal number of clusters with bootstrapping	218
Determining optimal numbers of clusters with model-based clustering	219
Identifying group membership with irregular clusters	228
Variable selection in cluster analysis	231
Error-checking cluster results with known outcomes	233
Exploring outliers	235
Identifying outliers with distance functions	235
Identifying outliers with the local outlier factor	237
Anomaly detection	238
Extreme value analysis	239
Finding associations in shopping carts	246
Chapter 9: Reporting and Dashboarding	252
Output formats for .Rmd documents	253
Dashboards	253
Simple dashboarding with R Markdown	254
Dashboarding and reporting with flexdashboard	256
Reports and technical memos	258
Slide decks	259
purl: scraping the raw code from your .Rmd files	260
Shiny apps	261
Tweaking the YAML headers	263
Appendix 1: Setting up projects and using “Make” files	266
Setting up a project with RStudio and Git	266
Committing your changes	268

CONTENTS

Rolling back to a previous state	268
Packaging a project	268
“Make” files in R	268
Appendix 2: .Rmd File for the <i>Chapter 1</i> final report example	270
Appendix 3: .R file for <i>Chapter 9</i> Shiny app example	278
Appendix 4: .Rmd file for the <i>Chapter 9</i> flexdashboard example	285
Appendix 5: R Markdown Quick Reference	290

About

About the Author

Dwight Barry is a Lead Data Scientist at Seattle Children's Hospital, in Seattle, Washington, USA.

Session Info

The code in this book was tested on:

```
R version 3.3.1 (2016-06-21)
Platform: x86_64-apple-darwin13.4.0 (64-bit)
Running under: OS X 10.10.5 (Yosemite)
```

```
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 14.04.5 LTS
```

```
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows Server 2008 R2 x64 (build 7601) Service Pack 1
```

Version Info

Version 0.9.9 for final review, September 2016

Code and Data



The code and data are available from the book's [GitHub site](#).

Due to page width constraints, some lines of code *as printed in the book* have unnecessary line breaks, primarily in long URLs to acquire data or code from the web. The book's code on Github does not contain these line breaks, so it should work as-is. So, if you cut & paste from the book, note that long URLs may require that you remove a line break for the code to function correctly.

Cover Image

The [cover photo](#) was taken by [Gerd Altmann](#), and was released as [CC0](#).

Proceeds

All proceeds will benefit the *Agape Girls Junior Guild*, supporting mitochondrial disorder research at *Seattle Children's Research Institute* and *Seattle Children's Hospital*.



Contact



Rmadillo



Dwight Barry



Dwight Barry



@healthstatsdude

install.packages

```
install.packages(c("abd", "acs", "arules", "arulesViz", "asympTest", "beanplot",
"boot", "bootES", "ca", "choroplethr", "choroplethrMaps", "classInt", "cluster",
"clustvarsel", "data.table", "dbSCAN", "devtools", "DMwR", "dplyr", "DT",
"dygraphs", "epitools", "eurostat", "extremeStat", "flexdashboard", "forecast",
"gamlss.tr", "gdata", "GGally", "ggExtra", "ggplot2", "ggseas", "googleVis",
"gridExtra", "htmlTable", "htmlwidgets", "httr", "jsonlite", "kulife",
"leaflet", "likert", "lubridate", "maps", "maptools", "mc2d", "mclust", "mgcv",
"mvoutlier", "NeatMap", "orddom", "pairsD3", "polycor", "plotly", "pscl",
"psych", "qcc", "quantreg", "RColorBrewer", "RCurl", "readxl", "reshape2",
"rjags", "rmarkdown", "RODBC", "scales", "segmented", "shiny", "simpleboot",
"sqldf", "strucchange", "survival", "survminer", "tidyverse", "tolerance",
"useful", "vcd", "vegan", "VIM", "XLConnect", "XML", "xtable", "zipcode",
"zoo"), dependencies = TRUE)

source("https://bioconductor.org/biocLite.R")
biocLite(c("graph", "Rgraphviz", "GenomicRanges", "BiocInstaller"))

devtools::install_github("vqv/ggbiplot")
devtools::install_github("twitter/AnomalyDetection")
devtools::install_github("hrbrmstr/taucharts")
devtools::install_github("timelyportfolio/parcoords")
devtools::install_github("rasmusab/bayesian_first_aid")
```

Introduction

The growth of R has been phenomenal over the past several years, reaching out past academia and research labs into the daily activities of business and industry. More and more line-level analysts are supplementing their SQL or Excel skills with R, and more and more businesses are hiring analysts with R skills to bridge the increasing gap between business needs and on-hand analytic resources.

While there are many books that explore the use of R in applied statistics and data science, nearly all of them are set as a series of case studies or are written predominantly for academic audiences as textbooks. There are also many R cookbooks, but they tend to cover the breadth of R's capabilities and options. This book aims to provide a cookbook of R recipes that are specifically of use in the daily workflow of data scientists and analysts in business and industry.

Business Intelligence with R is a practical, hands-on overview of many of the major BI/analytics tasks that can be accomplished with R. It is not meant to be exhaustive—there is always more than one way to accomplish a given task in R, so this book aims to provide the simplest and/or most robust approaches to meet daily workflow needs. It can serve as the go-to desk reference for the professional analyst who needs to get things done in R.

From setting up a project under version control to creating an interactive dashboard of a data product, this book will provide you with the pieces to put it all together.

Overview

Chapter 1 takes only about 50 lines of R code to cover the spectrum of an analytics workflow, from data acquisition to business forecasting to the creation of an interactive report. Because “learning R” is difficult, this example points out that “learning R to accomplish *x*” is really fairly simple for anyone comfortable with coding. And if all you’ve ever developed are formulas in Excel, you *have* coded—using R is certainly within any analyst’s abilities, and you’ll find that it simplifies your workflow tremendously.

Chapter 2 covers bringing data into R, from text files to databases.

Since the data we get is rarely in the form we need, **Chapter 3** provides some tools and examples for cleaning and wrangling data.

Chapter 4 is the heart of any analytic project, no matter how broad the scope of the business question or the complexity of subsequent analyses. If you aren’t intimately familiar with your data from the start, your final products will be, at best, limited, and at worst, wrong.

Chapter 5 covers effect sizes; many non-analytic business customers have been trained to ask whether a difference or similarity is “statistically significant.” While common to this day, that

question provides a trivial answer to the wrong question. Using *p*-values today is much like continuing to use a rotary telephone. Sure, if properly wired, it still works fine. But if you've got a smart phone, why would you even want to use the old technology? Modern analytics focuses instead on questions of *how much different*, not whether there is a statistically significant difference. Effect size statistics provide what our business customers really need to know: how much different (or similar) are these groups?

Chapter 6 covers the timeless business need to understand trends. It used to be that the analysis of time series and/or autocorrelated data was not at all simple. But R has tools that make this type analysis easy, considerably faster than what it would take to accomplish the same in Excel.

If the glut of dashboarding and graphing in data products is any indication, modern business runs on dataviz. That's a good thing—I've seen far too many examples of decision-makers using tables to make decisions that ended poorly because tables just can't capture the big picture. **Chapter 7** explores data visualization tools not covered in earlier chapters, including multivariate visualization, maps, and interactivity.

Chapter 8 explores pattern discovery, via dimension reduction, clustering, outlier exploration, and association rules. “What things are like/not like other things?” is a common part of analytics daily workflows, so the tools in this chapter provide the means to accomplish those ends.

Finally, **Chapter 9** provides some brief examples of how to use R to create data products for your customers, from static reporting to interactive dashboards. A few tips and tricks to customize those are included as well.

Perhaps the largest advance in scientific research in recent years is the idea of reproducible research. While ensuring accurate and detailed documentation is a key feature of business intelligence and has been for some time, many analysts and even entire companies do not give this the attention it deserves; as a result, analysts are constantly reinventing the wheel by having to pore through thousands of lines of code to try to reproduce another analyst's work. **Chapter 9** emphasizes the idea that creating data products in R provides you both final products and a ready-made way to document the product as you build it.

Although the title should make it clear who the intended audience is for this book, I should point out that this *not* a data science book. There are already some great books out there on using R for data science work. I particularly like Williams' [Data Mining with Rattle and R](#)¹ and Zumel and Mount's [Practical Data Science with R](#)² for those interested in getting into predictive analytics via R. Kuhn and Johnson's [Applied Predictive Modeling](#)³ and James et al.'s [An Introduction to Statistical Learning with Applications in R](#)⁴ are both great for those needing a more thorough text. Provost and Fawcett's [Data Science for Business](#)⁵ is a great way to learn the conceptual side of data science tools *as applied in business settings*, and it's wonderfully well written.

¹<https://www.amazon.com/Data-Mining-Rattle-Excavating-Knowledge/dp/1441998896>

²<https://www.amazon.com/Practical-Data-Science-Nina-Zumel/dp/1617291560/>

³<https://www.amazon.com/Applied-Predictive-Modeling-Max-Kuhn/dp/1461468485/>

⁴<http://www-bcf.usc.edu/~gareth/ISL/>

⁵<https://www.amazon.com/Data-Science-Business-Data-Analytic-Thinking/dp/1449361323/>

Conventions

This book contains one large and one small break from conventional R usage.

First, along with more and more data scientists, I use = for assignment, rather than the traditional <- . Fans of nitpicking flame wars can find many arguments in support of either the traditional or the new convention; I use = because I frequently use other programming languages and it's just easier on my poor brain. If you are a solo analyst or the only person on your team who knows R, you are free to make your own choice. If you are working as part of a team of R programmers, it is important that everyone choose a convention and stick to it. You may wish to adopt one of the style guides promulgated by groups like the [Bioconductor project](#)⁶ or [Google](#)⁷.

The one place your choice of assignment operator is important is if you want to make an assignment in the middle of an existing function call. This is not something you should ever actually want.

Second, I also tend to use require instead of library. In terms of the underlying code, library is often considered the best practice, and some R purists get outwardly annoyed by the use of require. However, require is a verb, and helps me remember that this package is *required* for this data product.

With respect to variable naming, I tend to use pothole_case for variable, dataframe, and database names, and ALL_CAPS for variables or options where you need to supply your own values. The R community hasn't developed much consensus around naming formats, and so once again I have fallen back on habits learned elsewhere (cough Python cough) or to avoid problems in other languages (dot.case causes problems with several other languages but can also cause trouble even within R in certain highly used packages [e.g., knitr]). But again, you should use whatever you want in your own work. The important part is that generally speaking, variable names should probably be nouns, function names should probably be verbs, and all names should be clear and descriptive—the most common variable name in CRAN may be 'x' but that does not mean you should follow suit.

What you need

This is not the best book for a never-used-R-and-need-to-teach-myself analyst. Perhaps you've had an intro course or workshop already—you don't need to be an expert to get value from this book but you will probably struggle without a more detailed overview of how to use R. Rob Kabacoff's stellar [R in Action](#)⁸ is a good book to introduce new users to R, and it works well as a self-teaching or classroom text. There are also a wide variety of websites, videos, and online training programs that can give beginners the initial understanding of R that will make this book more useful.

You don't need to be an R programmer, however; **this book is aimed at people who use R to accomplish day-to-day analytic ends for business purposes, not for those who want to improve their code efficiency, create custom algorithms, or develop new packages.**

⁶<http://master.bioconductor.org/developers/how-to/coding-style/>

⁷<https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml>

⁸<http://www.manning.com/kabacoff2/>

To get the most from this book, you should already be aware of basic R use for analysis (e.g., knowing how to make base plots, use `lm`, etc.) and R's data construct types, such as knowing the differences between a list, a dataframe, and a matrix. You should also know that any result from the functions shown in this book can be placed in an object to be called or modified later, e.g., you should know that `bob = prop.table(table(bike_share_daily[c(3,9)]))` saves the table of proportions as a two-way table in an object named `bob`. This is important because many functions (especially with `ggplot2`) are presented in this book as stand-alone functions that create the output immediately. Obviously, you'll need to assign those functions to R objects if you want to keep them in memory for subsequent use.

Of course, you can always use `?` in front of any function to learn more about its use, syntax, and outputs.

You should also be familiar with the ideas behind many of the tools used in analytics and data science; for the most part, this book does not explain the intricacies of how the methods work or what the results of the examples mean. There are many fine resources out there covering theory and tools, but are necessarily limited in methodological scope as a result. In addition, many data science R books often use a case-study approach, focusing on a few major problems or themes in order to explain the methods and results in greater depth. Again, this book is aimed at providing comprehensive cookbook for common business intelligence and data science tasks, and given how many tools there are, there just isn't the space here to provide recipes, interpretation, *and* theory.

This book uses R, RStudio, and Git software. I used R 3.3 and RStudio 0.99 to create this book. If you are on a Windows system, you'll need to ensure either Perl and/or Java are installed to use some of the Excel import functions (if you are using Mac or Unix/Linux, Perl was probably installed with the OS). Make sure that the architecture for Perl/Java you have or install matches that of your R install, i.e., if you are using 64-bit R make sure your Java is also x64.

You can find the installation list for the packages used at the start this book or on [GitHub](#)⁹.

Acknowledgments

I originally started this book as a way to organize the code snippets I had cluttering up a variety of computers, servers, and repos... and as I started to organize those snippets, I also started getting more and more requests from colleagues who wanted to learn R. So this book is the culmination of both efforts.

Several people generously donated their time to review this book and test its code, including John Tilelli, Matt Sandgren, Howard Rosenfeld, Seth Terrell, Maria Brumm, Sherif Bates, Robert Samohyl, Deb Curley, and Alexis Gaevsky. Remaining errors are, of course, my own—so if you find any, please submit a pull request or otherwise let me know so that I can correct and update the book.

I've been using R for about a decade, and along the way, I have taken ideas from across the web wherever I could find them. If you see some of *your* code here and I have not given you or the source proper attribution, please let me know! I will fix that immediately.

⁹https://github.com/Rmadillo/business_intelligence_with_r/blob/master/manuscript/code/BIWR_install_package_list.R

The cover photo¹⁰ was taken by Gerd Altmann¹¹, and was released as CC0¹².

Website/Code



The code is available at [this book's GitHub site¹³](#).

Due to page width constraints, some lines of code *as printed in the book* have unnecessary line breaks, primarily in long URLs to acquire data or code from the web. The book's code on GitHub does not contain these line breaks, so it should work as-is. So, if you cut & paste from the book, note that long URLs may require that you remove a line break for the code to function correctly.

This book will be updated with additional functions and ideas on an ad hoc basis—check the *About* page to see the version number and differences from previous versions.

Feedback, criticism, and improvements are always welcome; send me an email to dbarryanalytics@gmail.com, or submit pull requests or issues on [GitHub¹⁴](#).

Happy coding!

I hope that this book serves well in its intended purpose as a desktop reference for busy professionals—giving you fingertip access to a variety of BI analytic methods done in R as simply as possible.

¹⁰<https://pixabay.com/en/toronto-ontario-canada-road-1439133/>

¹¹<https://pixabay.com/en/users/geralt-9301/>

¹²<https://pixabay.com/en/service/terms/#usage>

¹³https://github.com/Rmadillo/business_intelligence_with_r/tree/master/manuscript/code

¹⁴https://github.com/Rmadillo/business_intelligence_with_r

Chapter 1: An entire project in a few lines of code

Learning R is an infamously difficult thing to do. However, if you need to accomplish a particular analytics goal, a solution in R can be found relatively easily if you focus on that single task. Over time, completing a variety of single tasks leads you to a practical working knowledge of R in general. So don't think about "learning R"—think instead about "learning how to solve the problem at hand with R," and things get a lot easier.

Essentially, the technical part of the typical BI workflow is to:

1. Set up and develop project space
2. Acquire data
3. Wrangle data
4. Perform analytics
5. Develop report/data product
6. Document the process

This chapter shows how simple it can be to do some advanced analytics in R, working through each of these six steps.

The analytics problem

The data we'll use in this chapter contains 2+ million records of minute-scale power consumption in a single household for about 47 months, collected by Electricité de France and archived at the UCI Machine Learning Repository. The data set's page can be accessed [here¹⁵](#) for more details, but for convenience, the attributes of this dataset are as follows:

1. **date**: date in format dd/mm/yyyy
2. **time**: time in format hh:mm:ss
3. **global_active_power**: household global minute-averaged active power (in kilowatts)
4. **global_reactive_power**: household global minute-averaged reactive power (in kilowatts)
5. **voltage**: minute-averaged voltage (in volts)
6. **global_intensity**: household global minute-averaged current intensity (in amperes)

¹⁵<https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption>

7. **sub_metering_1**: energy sub-metering No. 1 (in watt-hours of active energy). It corresponds to the kitchen, containing mainly a dishwasher, an oven, and a microwave (hot plates are not electric but gas powered).
8. **sub_metering_2**: energy sub-metering No. 2 (in watt-hours of active energy). It corresponds to the laundry room, containing a washing-machine, a tumble-drier, a refrigerator, and a light.
9. **sub_metering_3**: energy sub-metering No. 3 (in watt-hours of active energy). It corresponds to an electric water-heater and an air-conditioner.

Our purpose is to describe past monthly power usage and forecast potential usage for six months following the last full month of data in the dataset.

Set up

First, we need to set up the working environment—this is generally easiest to do using the file system (see *Appendix 1*), but for sake of explanation and thoroughness, we'll do the entire process in this chapter within R. To ensure clarity and reproducibility, we need to set up a single directory for each project, with subfolders that contain and separate code, data, results, and so on.

Open RStudio.

R starts in the default directory, symbolized at the top of the console window as `~/`.

Create the directory space with `dir.create`, the R equivalent to `mkdir`:

```
# Creates directory/ies, use recursive=TRUE in the first
# command to create the entire path at once
dir.create("~/BIWR/Chapter1/Code", recursive=TRUE)
dir.create("~/BIWR/Chapter1/Data")
dir.create("~/BIWR/Chapter1/Results")
```

Normally, at this point we'd use the script window to write out the script, often using R Markdown to create the report and documentation as we go. For now, you either cut and paste each line into the console as we go, open a new R Script window and put the code into that, or just load [this chapter's code from Github](#)¹⁶. At the end of this chapter, we'll go through the process using an R Markdown file to document the project and create a final data product.

Remember, if you are cutting and pasting code from the book, you will need to manually edit the URLs, which are broken up here because of page width constraints.

Move into your new directory and load the R packages we'll use in this project:

¹⁶https://github.com/Rmadillo/business_intelligence_with_r/blob/master/manuscript/code/BIWR_Chapter_1_Code.R

```

# Set the working directory
setwd("~/BIWR/Chapter1")

# This package will allow us to interpolate between missing time series values
require(zoo)

# These packages provide functions for easy data wrangling
require(dplyr)
require(reshape2)

# This package provides automated forecasting of time series data
require(forecast)

# This package allows us to create publication-quality plots
require(ggplot2)

# This package allows creation of javascript widgets for use in webpages
require(htmlwidgets)

# This packages uses htmlwidgets to make time series widgets
require(dygraphs)

```

Acquire data

Download the data from a zipped flat file (semi-colon delimited .txt format) on the UCI Machine Learning Repository:

```

# Download the zip file into the data folder (note the line break here!)
download.file("http://archive.ics.uci.edu/ml/machine-learning-databases/00235/
  household_power_consumption.zip", destfile =
  "Data/household_power_consumption.zip")

# Unzip the data from the zip file into the Data folder
unzip("Data/household_power_consumption.zip", exdir="Data")

# Read the data into R
# NAs are represented by blanks and ? in this data, so need to change
power = read.table("Data/household_power_consumption.txt", sep=";", header=T,
  na.strings=c("?", ""), stringsAsFactors=FALSE)

```

Wrangling data

We'll look at the structure of the data to see what's been read in:

```
# str gives you the structure of the dataset
str(power)
```

The screenshot shows the RStudio console window. The title bar says "Console ~/BIWR/Chapter1/". The console area displays the output of the `str(power)` command. It shows a data frame with 2075259 observations and 9 variables. The variables are: Date (character), Time (character), Global_active_power (numeric), Global_reactive_power (numeric), Voltage (numeric), Global_intensity (numeric), Sub_metering_1 (numeric), Sub_metering_2 (numeric), and Sub_metering_3 (numeric). The output ends with a vertical ellipsis.

```
Console ~/BIWR/Chapter1/
> str(power)
'data.frame': 2075259 obs. of 9 variables:
 $ Date : chr "16/12/2006" "16/12/2006" "16/12/2006" "16/12/2006" ...
 $ Time : chr "17:24:00" "17:25:00" "17:26:00" "17:27:00" ...
 $ Global_active_power : num 4.22 5.36 5.37 5.39 3.67 ...
 $ Global_reactive_power: num 0.418 0.436 0.498 0.502 0.528 0.522 0.52 0.52 0.51 0.51 ...
 $ Voltage : num 235 234 233 234 236 ...
 $ Global_intensity : num 18.4 23 23 23 15.8 15 15.8 15.8 15.8 15.8 ...
 $ Sub_metering_1 : num 0 0 0 0 0 0 0 0 0 ...
 $ Sub_metering_2 : num 1 1 2 1 1 2 1 1 1 2 ...
 $ Sub_metering_3 : num 17 16 17 17 17 17 17 17 17 16 ...
> |
```

The Date and Time variables were read in as characters, so we'll convert them to date and time classes, respectively, as well as create a new DateTime column:

```
# Convert character date to an ISO date
power$Date = as.Date(power$Date, format="%d/%m/%Y")

# Create a DateTime object
power$DateTime = as.POSIXct(paste(power$Date, power$Time))

# Obtain the Month and Year for each data point
power$Month = format(power$Date, "%Y-%m")

# Add the first to each Y-m combo and convert back to ISO Date
power$Month = as.Date(paste0(power$Month, "-01"))

# Verify the changes
str(power)
```

```
Console ~/BIWR/Chapter1/
> str(power)
'data.frame': 2075259 obs. of 11 variables:
 $ Date           : Date, format: "2006-12-16" "2006-12-16" "2006-12-16" ...
 $ Time           : chr  "17:24:00" "17:25:00" "17:26:00" "17:27:00" ...
 $ Global_active_power : num  4.22 5.36 5.37 5.39 3.67 ...
 $ Global_reactive_power: num  0.418 0.436 0.498 0.502 0.528 0.522 0.52 0.51 0.51 ...
 $ Voltage         : num  235 234 233 234 236 ...
 $ Global_intensity : num  18.4 23 23 23 15.8 15 15.8 15.8 15.8 15.8 ...
 $ Sub_metering_1   : num  0 0 0 0 0 0 0 0 ...
 $ Sub_metering_2   : num  1 1 2 1 1 2 1 1 1 2 ...
 $ Sub_metering_3   : num  17 16 17 17 17 17 17 17 17 16 ...
 $ DateTime        : POSIXct, format: "2006-12-16 17:24:00" "2006-12-16 17:25:00" "2006-12-16 17:26:00" ...
 $ Month          : Date, format: "2006-12-01" "2006-12-01" "2006-12-01" ...
> |
```

```
# Get an overview of the variables
summary(power)
```

```
Console ~/BIWR/Chapter1/
> summary(power)
   Date           Time      Global_active_power Global_reactive_power    Voltage
Min.   :2006-12-16 Length:2075259   Min.   : 0.076   Min.   :0.000   Min.   :223.2
1st Qu.:2007-12-12 Class :character  1st Qu.: 0.308   1st Qu.:0.048   1st Qu.:239.0
Median :2008-12-06 Mode  :character  Median : 0.602   Median :0.100   Median :241.0
Mean   :2008-12-05             Mean   : 1.092   Mean   :0.124   Mean   :240.8
3rd Qu.:2009-12-01             3rd Qu.: 1.528   3rd Qu.:0.194   3rd Qu.:242.9
Max.   :2010-11-26             Max.   :11.122   Max.   :1.390   Max.   :254.2
NA's   :25979                 NA's   :25979   NA's   :25979   NA's   :25979
Global_intensity Sub_metering_1 Sub_metering_2 Sub_metering_3      DateTime
Min.   : 0.200   Min.   : 0.000   Min.   : 0.000   Min.   : 0.000   Min.   :2006-12-16 17:24:00
1st Qu.: 1.400   1st Qu.: 0.000   1st Qu.: 0.000   1st Qu.: 0.000   1st Qu.:2007-12-12 00:18:30
Median : 2.600   Median : 0.000   Median : 0.000   Median : 1.000   Median :2008-12-06 07:13:00
Mean   : 4.628   Mean   : 1.122   Mean   : 1.299   Mean   : 6.458   Mean   :2008-12-06 06:33:21
3rd Qu.: 6.400   3rd Qu.: 0.000   3rd Qu.: 1.000   3rd Qu.:17.000   3rd Qu.:2009-12-01 14:07:30
Max.   :48.400   Max.   :88.000   Max.   :80.000   Max.   :31.000   Max.   :2010-11-26 21:02:00
NA's   :25979   NA's   :25979   NA's   :25979   NA's   :25979
   Month
Min.   :2006-12-01
1st Qu.:2007-12-01
Median :2008-12-01
Mean   :2008-11-21
3rd Qu.:2009-12-01
Max.   :2010-11-01
> |
```

We can see from the summary that there are about 26k missing values (NAs) in our primary variable, `Global_active_power`—about 1.25% of the ~2 million records. We can quickly get a table and graph of missing data over time by setting a counting marker for missing values with `ifelse`, then using the `dplyr` package to group and summarize the data, and finally pull a ready-made R function from the web to create a calendar graph that shows the daily distribution of the missing values.

```

# Use ifelse to count each minute that is NA
power$Missing = ifelse(is.na(power$Global_active_power), 1, 0)

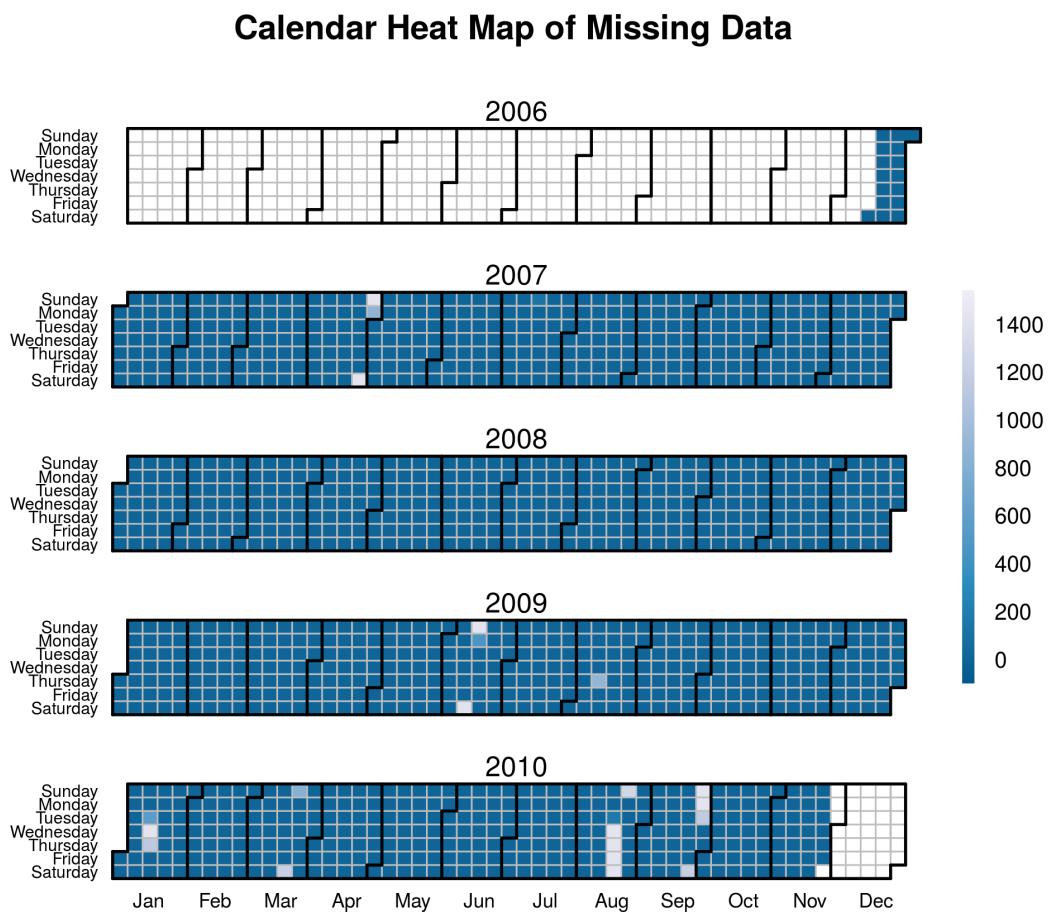
# Use dplyr's group_by function to group the data by Date
power_group_day = group_by(power, Date)

# Use dplyr's summarize function to summarize by our NA indicator
# (where 1 = 1 minute with NA)
power_day_missing = summarize(power_group_day, Count_Missing = sum(Missing))

# Download the 'calendarHeat' function from revolutionanalytics.com
source("http://blog.revolutionanalytics.com/downloads/calendarHeat.R")

# Plot the calendar graph to view the missing data pattern
calendarHeat(power_day_missing$date, power_day_missing$count_missing,
  varname="Missing Data", color="w2b")

```



You can view the actual values of missing data on each day by exploring the `power_day_missing` data frame.

The 4-5 day spans of missing data near the end of the time series may be a little concerning since we want to perform automated forecasting. However, since we're aggregating to months and forecasting into months with very few missing values, we should be ok. But to make automatic forecasting work, we need to fill in those missing values. If we convert each missing value to 0, we'll definitely underestimate usage for those times.

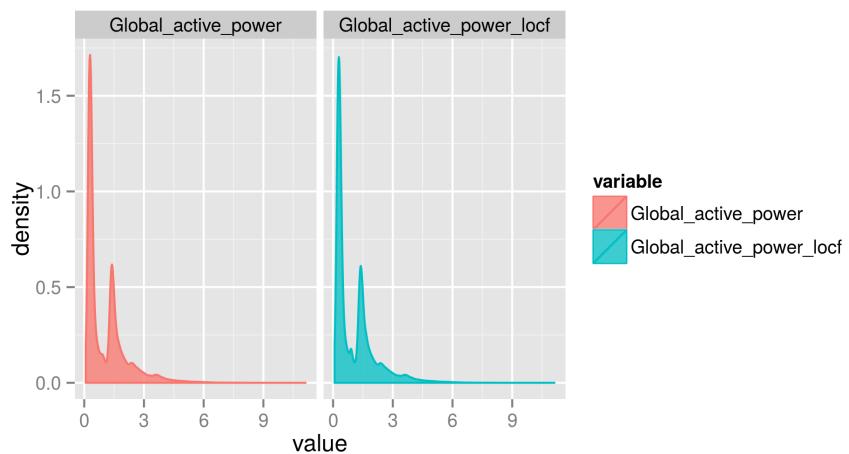
A reasonable approach here is to carry the last value forward, which we can do with the `na.locf` function in the `zoo` package. While other approaches are possible (and perhaps better, e.g., using some sort of mean or median value instead of the last value, or even a seasonal Kalman filter), we'll proceed with this option to keep the example simple.

```
# Use zoo to perform interpolation for missing time series values
power$Global_active_power_locf = na.locf(power$Global_active_power)

# Compare the original and interpolated distributions by
# reshaping the two variables into long form for ggplot
power_long = melt(power, id.vars= "DateTime", measure.vars=
  c("Global_active_power", "Global_active_power_locf"))

# Create density plot
ggplot(power_long, aes(value, fill=variable, color=variable)) +
  geom_density(alpha=0.75) +
  facet_wrap(~variable)

# Save density plot to Results folder
# Note that ggave uses "dpi" to set image resolution
ggsave("Results/density_plot.png", width=6, height=4, dpi=600, units="in")
```



The overall shape hasn't changed, though we can see a small spike at about 1 kW in the last-observation-carried-forward data. This should be fine for our purposes, as the overall pattern is essentially the same.

Now that we have a complete time series, we can determine total monthly use (kWh). While we're at it, we can calculate maximum demand for a given month (kW) over the period of record as an example of how to calculate multiple summaries at once. kWh measures *use*, i.e., how much energy is used, while kW measures *demand*. We're interested in usage, because that's how power companies charge us.

```
# Use dplyr to group by month
power_group = group_by(power, Month)

# Use dplyr to get monthly max demand and total use results
power_monthly = summarize(power_group,
  Max_Demand_kW = max(Global_active_power_locf),
  Total_Use_kWh = sum(Global_active_power_locf)/60)

# Remove partial months from data frame
power_monthly = power_monthly[2:47,]

# Convert Month to Date
power_monthly$Month = as.Date(paste0(power_monthly$Month, "-01"))

# Look at structure of the result
str(power_monthly)
```

The screenshot shows the RStudio Console window with the title 'Console ~/BIWR/Chapter1/'. The console output displays the structure of the 'power_monthly' data frame. It shows 46 observations and 3 variables: 'Month' (Date type), 'Max_Demand_kW' (numeric type), and 'Total_Use_kWh' (numeric type). The 'Month' variable is formatted as '2007-01-01' through '2007-03-01'.

```
Console ~/BIWR/Chapter1/
> str(power_monthly)
Classes 'tbl_df' and 'data.frame': 46 obs. of  3 variables:
 $ Month      : Date, format: "2007-01-01" "2007-02-01" "2007-03-01" ...
 $ Max_Demand_kW: num  9.27 9.41 10.67 8.16 7.67 ...
 $ Total_Use_kWh: num  1150 942 981 617 733 ...
> |
```

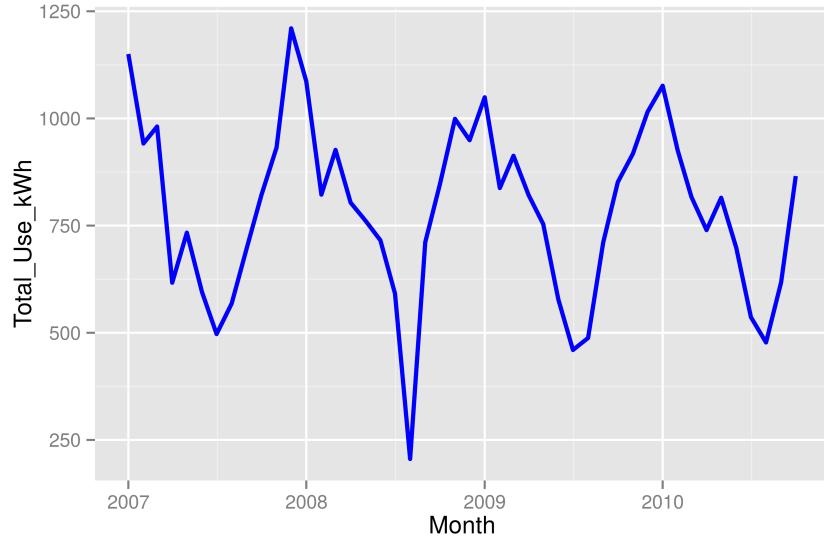
Analytics

Explore the data

Plotting your data is the single most important part of analytics, so this book spends a lot of space on graphical representation. Here, as we're focused on power use over time, we'll plot the monthly summary we've calculated above.

```
# Create plot of total use by month
ggplot(power_monthly, aes(Month, Total_Use_kWh)) +
  geom_line(col="blue", lwd=1)

# Save plot as hi-res png to Results subfolder
ggsave("Results/total_use_plot.png", width=6, height=4, dpi=600, units="in")
```



We can see clear patterns in the data—higher in the winter and lower in the summer.

Run a forecasting model

Now we want to forecast total use for the next six months. We'll create the model with the automated `forecast` function from the `forecast` package, then plot the results and view the model itself.

```
# Create a time series object of monthly total use
total_use_ts = ts(power_monthly$Total_Use_kWh, start=c(2007,1), frequency=12)

# Automatically obtain the forecast for the next 6 months
# using the forecast package's forecast function
# See ?forecast for more details
total_use_fc = forecast(total_use_ts, h=6)

# View the forecast model results
summary(total_use_fc)
```

```

Console ~/BIWR/Chapter1/
> summary(total_use_fc)

Forecast method: ETS(A,N,A)

Model Information:
ETS(A,N,A)

Call:
ets(y = object, lambda = lambda)

Smoothing parameters:
alpha = 0.0613
gamma = 3e-04

Initial states:
l = 829.9356
s=257.3091 165.2708 45.671 -82.8744 -375.8192 -255.0943
-144.7483 -20.4648 -23.0432 97.3446 88.7408 247.708

sigma: 72.9475

      AIC     AICc      BIC
598.7736 612.3220 624.3746

Error measures:
      ME     RMSE      MAE      MPE      MAPE      MASE      ACF1
Training set -12.79661 72.94752 54.94647 -3.944992 9.466253 0.653263 -0.09482945

Forecasts:
      Point Forecast    Lo 80     Hi 80    Lo 95     Hi 95
Nov 2010      959.1288 865.6428 1052.6148 816.1543 1102.1033
Dec 2010      1051.1827 957.5214 1144.8439 907.9402 1194.4252
Jan 2011      1041.6224 947.7862 1135.4586 898.1124 1185.1324
Feb 2011       882.5943 788.5836 976.6051 738.8173 1026.3714
Mar 2011       891.2219 797.0369 985.4069 747.1783 1035.2655
Apr 2011       770.7830 676.4241 865.1420 626.4734 915.0926
> |

```

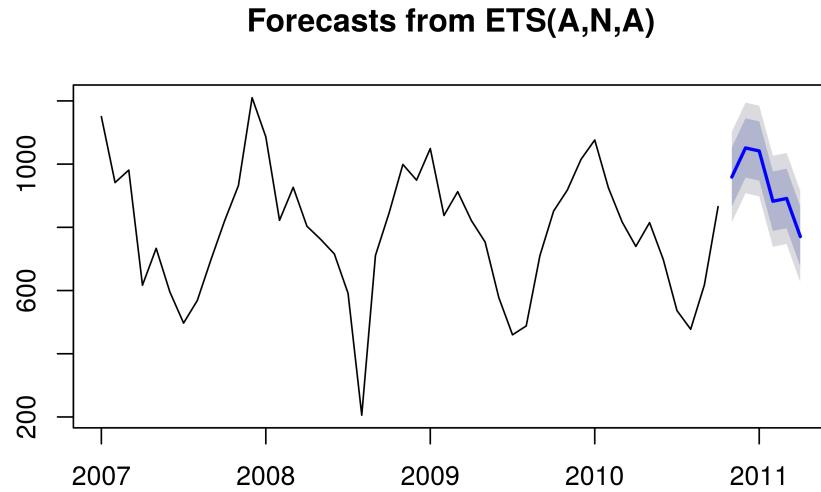
```

# Export a copy of the model results into a text file in the Results folder
sink("Results/Forecast_Model.txt")
summary(total_use_fc)
sink()

# View the forecast plot
plot(total_use_fc)

# Save the base graphics plot to the Results folder
# Note that resolution is called 'res' here
png("Results/total_use_forecast.png", width=6, height=4, res=600, units="in")
plot(total_use_fc)
dev.off()

```



Reporting

With the forecast summary and plot we have the essential pieces for addressing the problem. Now we need to create a report for decision-making. In this case, we'll again keep it simple and just produce an interactive browser app of the monthly trends and forecast results.

Create an interactive HTML plot

```
# Create a data frame with the original data
# and placeholders for the forecast details
use_df = data.frame(Total_Use = power_monthly$Total_Use_kWh,
                    Forecast = NA, Upper_80 = NA,
                    Lower_80 = NA, Upper_95 = NA, Lower_95 = NA)

# Create a data frame for the forecast details
# with a placeholder column for the original data
use_fc = data.frame(Total_Use = NA, Forecast = total_use_fc$mean,
                     Upper_80 = total_use_fc$upper[,1], Lower_80 = total_use_fc$lower[,1],
                     Upper_95 = total_use_fc$upper[,2], Lower_95 = total_use_fc$lower[,2])

# Union the two data frames into one using rbind
use_ts_fc = rbind(use_df, use_fc)

# Create a time series of the data and forecast results
total_use_forecast = ts(use_ts_fc, start=c(2007, 1), freq=12)

# Create the widget
```

```

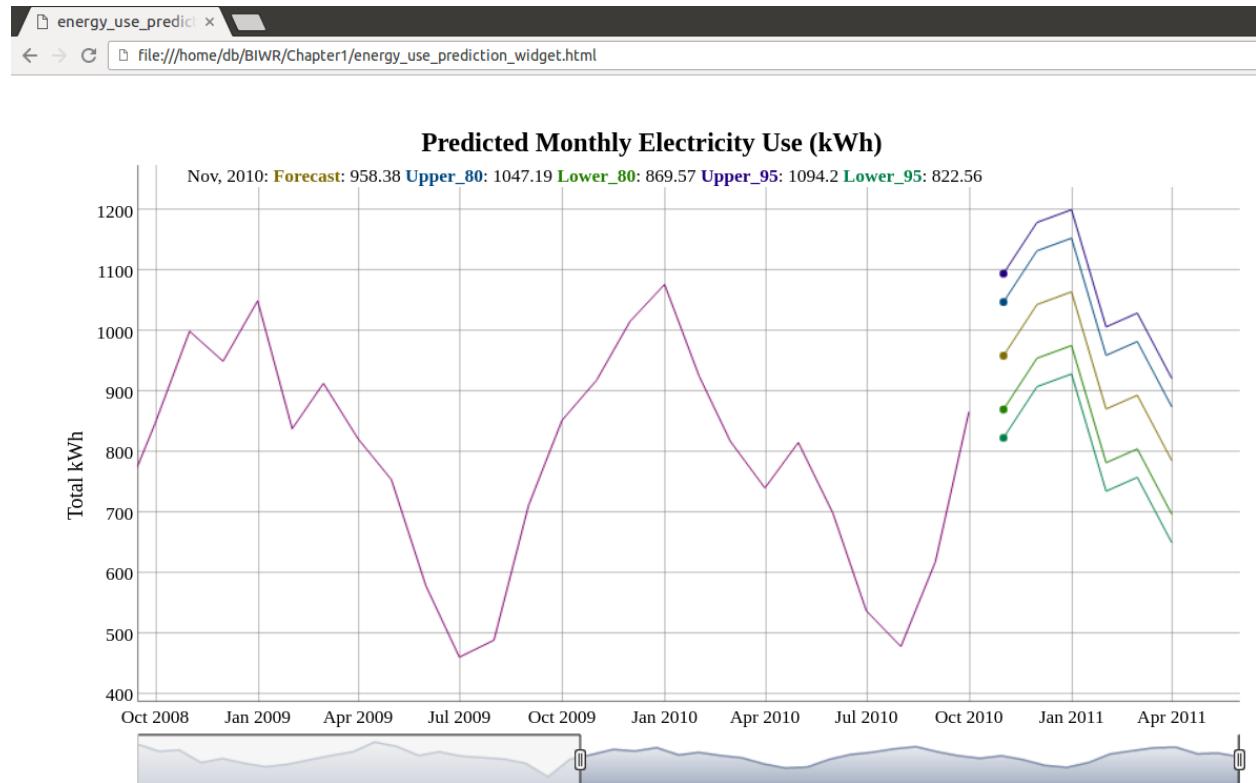
energy_use_prediction_widget = dygraph(total_use_forecast,
  main = "Predicted Monthly Electricity Use (kWh)",
  ylab = "Total kWh", width=900, height=500) %>%
dySeries(c("Total_Use"), label = "Actual kWh Usage") %>%
dyEvent(x = "2008-08-01", "Went on vacation", labelLoc = "top") %>%
dyRangeSelector(dateWindow = c("2008-09-15", "2011-06-01")) %>%
dyLegend(width = 800)

# Display the widget in the Viewer window
# Hit the Zoom button for a pop-out
energy_use_prediction_widget

# Save the widget as a stand-alone HTML file to Results folder
saveWidget(energy_use_prediction_widget, "energy_use_prediction_widget.html")

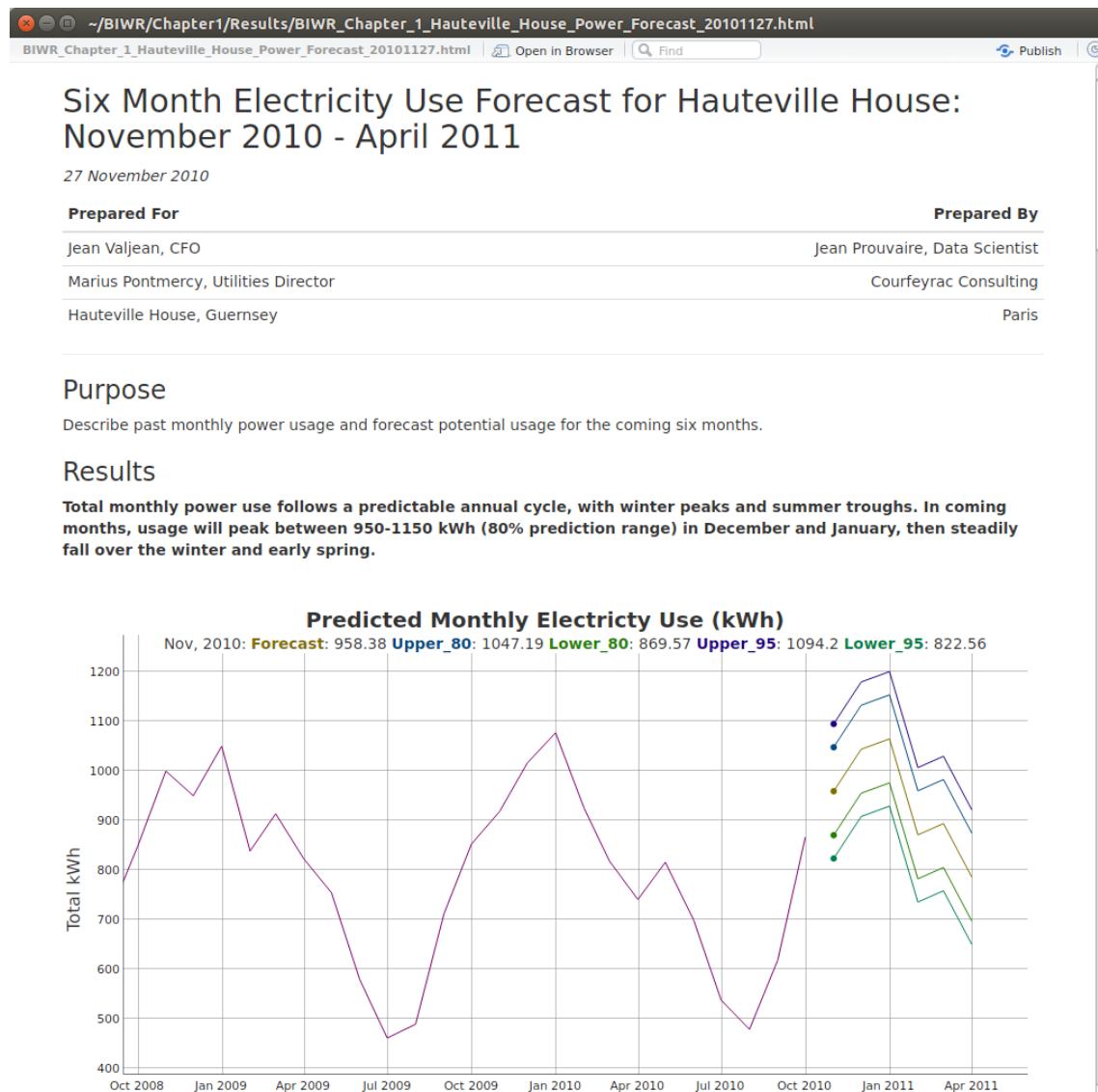
```

Opening the `energy_use_prediction_widget.html` from the Reports folder shows us the result. This stand-alone file can be sent to the decision-maker(s) who can explore the exact trend and forecast values with a mouse hover, while still seeing the overall pattern.



Documenting the project

R Markdown is a brilliant way to create documentation and reports in one shot (see *Chapter 9*). For example, had this forecasting project actually required a report, I'd structure it all inside an .Rmd like I've shown in *Appendix 2*. To run that file manually, just load it into RStudio and click the Knit HTML button in the *Source* window. A preview window will pop up. In some systems, the HTML file is concurrently saved to the working directory, while in others you may need to Open in browser and then Save as from there.



To run an .Rmd file programatically, the rmarkdown package allows you to run it from the console, a

command line, or a “make” file (see *Appendix 1*) via the `render` function:

```
render("Code/Hauteville_House_Power_Forecast_20101127.Rmd", "html_document")
```



Note: as I prefer to have the `.Rmd` file self-contained, I sometimes include the directory creation steps in the code. This will *fail* if you’ve already created the directories—comment those sections out if you’re (re)running it several times locally for testing.

I often have a script window (`.R`) and an R Markdown (`.Rmd`) window open at the same time when working on projects. Once I have the code the way I want it, I transfer that code to the `.Rmd` file if I intend to include it in any data/reporting products. When I have steps or results and want to jot down my rationale for the methods I’m using, interpretations and ideas for future use, and so on, I do that in an `.Rmd` file as well. At the end, I aim for an `.Rmd` file that will allow another user to reproduce the entire analysis and have an understanding of why I chose certain methods or came to the conclusions I report. For longer analyses, this may require a “make” file type of setup (see *Appendix 1*).

Even better is when it’s a small project, like this one—I can contain the code, documentation, and report all in one file. Since decision makers want the answer first, I include all of the code except for results inside a single code chunk near the top. The results I place in separate code chunks, placed where appropriate.

Summary

The entire analytics workflow for this project—from directory set-up through creating a forecast model and developing the final report—took only about 50 lines of code, excluding comments and spacing. We’ve seen in this chapter how much can be accomplished in just a few lines of R code; there are very few languages in which you really can do this much so succinctly.

The rest of this book provides code snippets, examples, and full recipes for using R throughout the main portions of the analytics workflow: data acquisition through exploration and reporting.

Chapter 2: Getting Data

- Working with Files
- Working with Databases
- Getting Data from the Web
- Creating Fake Data to Test Code
- Writing Files to Disk

The tried-and-true method for bringing data into R is via the humble csv file, and in many cases it will be the default approach. But in most professional environments, you'll encounter a wide variety of file formats and production needs, so having a cheat sheet of data import methods is useful.

There are a wide variety of packages and functions to import data from files, but a few provide nearly all the functionality you'll need:

File Type	Function	Package
csv file	<code>read.table("file", sep=",", ...)</code> <code>read.csv("file", ...)</code>	base
Flat file	<code>read.table("file", sep=" ", ...)</code> <code>fread("file", sep=" ", ...)</code>	base data.table
Excel	<code>read_excel("Excelfile", sheet, ...)</code> <code>read.xls("Excelfile", sheet, ...)</code> <code>readWorksheet("Excelfile", sheet, ...)</code>	readxl gdata XLConnect
XML	<code>xmlToDataFrame("xmlfile", ...)</code> <code>readHTMLTable("htmlfile", which= , ...)</code>	XML
JSON	<code>fromJSON()</code>	jsonlite
SPSS	<code>spss.get("SPSSfile.sav")</code>	Hmisc
Stata	<code>read.dta("Statafile.dta")</code>	foreign
SAS	<code>read.sas7bdat("SASfile.sas7bdat")</code>	sas7bdat
Database table(s)	<code>sqlFetch(CONNECTION, "TABLE NAME", ...)</code> <code>sqlQuery(CONNECTION, "SQL QUERY", ...)</code> <code>sqldf("SQL QUERY", dbname="database", ...)</code>	RODBC sqldf

Working with files

Reading flat files from disk or the web

To import flat files, it's probably best to get in the habit of using `read.table` instead of `read.csv`. While comma-delimited files are indeed the most common, tab- and pipe-delimited can also be quite prevalent depending on the context. And, since `read.csv` is just a wrapper for `read.table`, you might save time in the long run by using `read.table` for any flat file, and changing the `sep` value as needed, such as `sep=","` for comma, `sep="\t"` for tab, `sep="|"` for pipe, and so on. I also like to use the `stringsAsFactors=FALSE` option as well, as I often pull in data that has character fields, and it's easier to designate the factor variables later. Finally, `read.table` assumes that the data you're reading in doesn't have a header, so if there is one, you have to specify that with `header=TRUE`.

We saw reading in a semi-colon delimited flat file in the first chapter:

```
power = read.table("~/BIWR/Chapter1/Data/household_power_consumption.txt",
  sep=";", header=T, na.strings=c("?", ""), stringsAsFactors=FALSE)
```

While reading in a flat file is pretty standard fare for any R user, it can sometimes require a little extra work to make sure it comes in correctly. One important option to remember is `na.strings`; while R will read missing data from blank cells as NA automatically, you find a wide variety of ways people code missing values, from -99 to ? and sometimes even '0'. Occasionally you find more than one NA designator in the same file, as you can see in the `power` dataset example, above.

Sometimes file encoding matters as well. This example downloads an online .csv file from the Canadian 2011 Public Service Employee Survey (PSES), using UTF-8 encoding to ensure that special characters are read in correctly:

```
# Read in a csv from the internet with non-ASCII characters
pses2011 = read.table("http://www.tbs-sct.gc.ca/pses-saff/2011/data/
  2011_results-resultats.csv", sep=",", encoding="UTF-8")
```

Note: this file has no header; in a moment, we'll download the Excel file with the documentation for this data and use that to create the header.

Reading big files with `data.table`

The `data.table` package is extremely useful—and much, *much* faster than `read.table`—for larger files. The `data.table` function we used at the start of Chapter 1 on household power consumption took about 24 seconds on my laptop; the `fread` function below only took 2.3 seconds.

```
require(data.table)
# If you want to read in a csv directly, use fread, e.g., if you
# had the raw pses2011 in a local csv, you'd read it in this way:
power = fread("~/BIWR/Chapter1/Data/household_power_consumption.txt",
  sep=";", header=T, na.strings=c("?", ""), stringsAsFactors=FALSE)
```

If you *already* have a data frame and want to speed up reading and accessing it, use the `data.table` function:

```
power = data.table("power")
```

Unzipping files within R

Larger files are often zipped up to save space and bandwidth. Using the `unz` function inside the usual `read.table` takes care of bringing a single file into R. We'll download the zip file for the [Bike Sharing dataset¹⁷](#) from the UCI Machine Learning Archive and unzip the daily use file from it.

```
download.file("http://archive.ics.uci.edu/ml/machine-learning-databases/00275/
  Bike-Sharing-Dataset.zip", "Bike-Data.zip")
bike_share_daily = read.table(unz("Bike-Data.zip", "day.csv"), header=T,
  quote="\"", sep=",")
```

If the zip file has more than one file in it, you can unzip them all from within R with `unzip`.

```
unzip("Bike-Data.zip")
```

Note: although there is an `untar` function, it is no longer necessary—gzipped files can be read directly into R with the standard `read.table` function.

Once you have the data files (or other files) unzipped, you can bring them into R as data frames or other formats using the relevant “reading data” functions, e.g.:

```
bike_share_daily = read.table("day.csv", header=T, sep=",")
bike_share_hourly = read.table("hour.csv", header=T, sep=",")
bike_share_readme = readLines("Readme.txt")
```

¹⁷<https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset>

Reading Excel files

To import Excel files, `readxl`, `gdata`, or `XLConnect` provide pretty much anything you'll need; `gdata` is faster and more powerful but requires Perl, so Windows users must ensure that's installed on their system before they can use it. `XLConnect` has a lot of options for those using Windows and/or are in Excel-heavy work environments, and while it can be slow with large files, it is based on Java so it could be easier to start with on Windows platforms (as long as the Java and R architectures match, i.e., if R is x64, Java should be as well). `readxl` is a new addition to the R ecosystem, and if all you need to do is pull in a worksheet as-is, it's by far the fastest and easiest method.

We'll use an Excel file that comes with the `readxl` package to demonstrate `readxl` and `XLConnect`.

```
# Load the package
require(readxl)

# Load the Excel workbook
datasets = system.file("extdata/datasets.xlsx", package = "readxl")

# Read in the first worksheet
iris_xl = read_excel(datasets, sheet = 1)
```

The `XLConnect` package makes it easy to read and write to and from Excel files in Windows. For sake of reproducibility and data permanency, you should always save data as plain text, e.g., as a csv, and not in proprietary or non-text formats like Excel. So if you must write/save *into* Excel, `XLConnect` will do it... but you'll have to figure that out on your own.

```
# Load the package
require(XLConnect)

# Load the Excel workbook
wb = loadWorkbook(system.file("extdata/datasets.xlsx", package = "readxl"))
```

If you want to read in a single tab/sheet:

```
# Read in the first tab
data_from_sheet = readWorksheet(wb, sheet = 1)
```

The `sheet=` option can use the sheet's name or its position, e.g., using `sheet="NAME OF SHEET"` in either `readxl` or `XLConnect` retrieves the same data.

If you want to read in *every* tab from a single Excel file:

```
# Get a list of worksheets in the workbook
sheets = getSheets(wb)

# Invisibly return each sheet as its own data frame
invisible(
  lapply(sheets, function(sheet)
    assign(sheet, readWorksheet (wb, sheet = sheet ), pos = 1))
)
```

XLConnect doesn't support downloading an Excel file straight from the web. However, gdata can. You can use it to import the documentation file for the 2011 PSES dataset imported in the previous section, and since the sheet has non-data rows in it (at the bottom), we'll exclude those by using the nrows function. It also contains "multi-string" headers, so we'll use the header, skip, and col.name options to ignore their header and create our own:

```
# Load package
require(gdata)

# Download Excel file form web and read in the first sheet to a data frame
pses2011_header = read.xls("http://www.tbs-sct.gc.ca/pses-saff/2011/
  data/PSES2011_Documentation.xls", sheet="Layout-Format", nrows=22,
  header=FALSE, skip=1, col.names=c("Variables", "Size", "Type",
  "Description"), stringsAsFactors=FALSE)
```

Using the colnames function, the metadata we read in from this Excel file can be used to create the header names for the raw data we read in as a csv in the previous section:

```
# Assign the Variable column names from the Excel file to the raw data file
colnames(pses2011) = pses2011_header$Variable
```

Creating a dataframe from the clipboard or direct entry

A really useful short-cut to get spreadsheet or table data with a header row into R quickly is copying it into the clipboard from a spreadsheet and importing it via the "clipboard" option in the read.table function:

```
# Generic code for reading data from the clipboard
my_data_frame = read.table("clipboard", sep="\t", header=TRUE)
```

For a small amount of data, many R users make vectors and create a data frame from them, e.g.,:

```
# Fake data in vectors
Survey_Response = c("Strongly Agree", "Agree", "Neutral", "Disagree",
  "Strongly Disagree")
Send_Email_Ad = c("Yes", "Yes", "No", "No", "No")

# Combine vectors into a data frame
marketing = data.frame(Survey_Response, Send_Email_Ad)
```

Longtime SAS users might remember the datalines (or cards, if you're really an old-timer) method to create tables on the fly. While the clipboard method shown above is probably better in most cases, if you're used to using this old SAS method and you only need a quick, small, use-once table, the textConnection function could be handy to have in the toolbox:

```
# An R take on SAS's datalines
marketing = read.table(textConnection(
  ("Survey_Response, Send_Email_Ad
  Strongly Agree, Yes
  Agree, Yes
  Neutral, No
  Disagree, No
  Strongly Disagree, No"),
  header=TRUE, sep=", "))
```

Of course, whichever works best for you is the method to use.

Reading XML files

We see a way to bring in specific tables from websites below, in the Web section, but this recipe provides an overview of bringing in a complete XML file. For a simple example, we'll use the 2010 Patient Satisfaction results from the Veterans Health Administration's [Hospital Report Card](#)¹⁸.

¹⁸<http://www1.va.gov/health/HospitalReportCard.asp>

```

require(XML)
# Read in the webpage as-is
vha_pt_sat_raw = readLines("http://www1.va.gov/VETDATA/docs/Datagov/
  Data_Gov_VHA_2010_Dataset11_Patient_Satisfaction.xml")

# Convert xml to a data frame
VHA_Patient_Satisfaction = xmlToDataFrame(vha_pt_sat_raw)

```

When there are more complex structures in the file, you may have to use another approach. Here we'll use the Federal Housing Finance Agency's [House Price Index \(HPI\)](#)¹⁹, which measures average price changes in repeat sales or refinancings on single-family homes. If you try the simple approach shown above, it will fail, with the warning '`data`' must be of a vector type, was '`NULL`'. Going through a list first and then into a data frame can often make these types of XML files import successfully:

```

# Read the file from the web
FHFA_HPI_raw = readLines("http://www.fhfa.gov/DataTools/Downloads/Documents/
  HPI/HPI_master.xml")

# Convert the XML to an R list
FHFA_HPI_list = xmlToList(xmlParse(FHFA_HPI_raw))

# Turn the list into a data frame
FHFA_HPI = data.frame(do.call(rbind, FHFA_HPI_list),
  row.names = make.unique(names(FHFA_HPI_list)))

```

Reading JSON files

JSON data can be really useful for nested data, so it is becoming more popular for data visualizations (e.g., in d3-based web apps) and other applications where the usual tidy, *row x column* format would make for a large and unwieldy data structure. Still, sometimes we need to work with these data structures in R, so reading them into a data frame is required. The `jsonlite` package is probably the best way to do this.

```

# Load the jsonlite package
require(jsonlite)

# Get JSON data of UFO sightings
ufos = fromJSON("http://metricsgraphicsjs.org/data/ufo-sightings.json")

```

¹⁹<http://www.fhfa.gov/DataTools/Downloads/Pages/House-Price-Index.aspx>

Usually, the `fromJSON` function will return a list if you’re pulling in anything more complex than raw JSON, so you’ll need to either call the data frame from within the list, or convert that portion of the list into a data frame. We’ll do the latter in this example:

```
# Acquire the JSON data from Yahoo finance
currency_list = fromJSON("http://finance.yahoo.com/webservice/v1/symbols/
  allcurrencies/quote?format=json")

# This takes the piece of the list we need and brings it out as a data frame
currency_data = currency_list$list$resources$resource$fields

# Look at the data structure
str(currency_data)
```

The screenshot shows the RStudio console window with the title 'Console'. The command `> str(currency_data)` is entered, followed by its output. The output shows that 'currency_data' is a data frame with 173 observations and 7 variables. The variables are: name (character), price (character), symbol (character), ts (character), type (character), utctime (character), and volume (character). The values for 'name' include 'USD/KRW', 'SILVER 1 OZ 999 NY', 'USD/VND', 'USD/BOB', etc. The 'price' column contains values like '1102.795044', '0.062096', '21805.000000', '6.890000', etc. The 'symbol' column includes 'KRW=X', 'XAG=X', 'VND=X', 'BOB=X', etc. The 'ts' column lists timestamps such as '1434801300', '1434750474', '1434801300', '1434801300', etc. The 'type' column has entries like 'currency', 'currency', 'currency', etc. The 'utctime' column shows dates and times like '2015-06-20T11:55:00+0000', '2015-06-19T21:47:54+0000', '2015-06-20T11:55:00+0000', '2015-06-20T11:55:00+0000', etc. The 'volume' column has values '0', '4', '0', '0', etc.

```
Console ~/ 
> str(currency_data)
'data.frame': 173 obs. of 7 variables:
 $ name   : chr "USD/KRW" "SILVER 1 OZ 999 NY" "USD/VND" "USD/BOB" ...
 $ price  : chr "1102.795044" "0.062096" "21805.000000" "6.890000" ...
 $ symbol : chr "KRW=X" "XAG=X" "VND=X" "BOB=X" ...
 $ ts     : chr "1434801300" "1434750474" "1434801300" "1434801300" ...
 $ type   : chr "currency" "currency" "currency" ...
 $ utctime: chr "2015-06-20T11:55:00+0000" "2015-06-19T21:47:54+0000" "2015-06-20T11:55:00+0000" "2015-06-20T11:55:00+0000" ...
 $ volume : chr "0" "4" "0" "0" ...
```

You’ll notice that every variable is in character format, so you’ll need to do a little cleaning to get this data into a format we can analyze. We’ll see more on changing formats in the next chapter.

Sometimes you encounter streaming JSON (aka “pseudo-JSON”), which contains JSON formatted-data but throws an error when you try to use the `fromJSON` function. Instead, use the `stream_in` function as follows:

```
# Download Enron emails to the working directory from jsonstudio
download.file("http://jsonstudio.com/wp-content/uploads/2014/02/enron.zip",
  "enron.zip")

# Unzip it
unzip("enron.zip")

# Bring in the streaming JSON as a data frame
enron = stream_in(file("enron.json"))
```

As above, each variable in the data frame is in `chr` format, so you’ll have to convert to other data types where relevant (e.g., the date column).

Working with databases

Connecting to a database

While those who work with server-based databases will typically manipulate the data in those environments before importing into R, there are times when having a direct link to the database can be useful—such as for ongoing and/or automated reporting or modeling. This recipe covers the basics of `RODBC`, one of the packages that allows you to use ODBC to connect R to your database. I've found the `sqldf` package (see the next few recipes) easier to use for working with database tables once you've connected, but as usual there are a variety of options in R—choose the one that works best for your workflow.

As there are dozens of database platforms, all with their own system-specific drivers, we won't cover Data Source Name (DSN) setup in this book. Enterprise systems often have those already set up, but if not, your database admin and/or a search on stackoverflow can provide advice on setting up the proper drivers for your particular context for adding user or system DSN connections. As an example, this is how you might set up a user DSN in a Windows Server environment to access a SQL Server database:

1. Click the Start button. Type in data in the Search programs and files box and select Data Sources (ODBC).
2. Click Add.
3. Select SQL Server Native Client xx.x, or choose the appropriate driver if that's not the one you need. Click Finish.
4. Put the name you want to use to access the database in the Name box, e.g., SQL3. Put the actual name of the server in the Server box, e.g., perhaps this server is called EDWSQLDB3. Click Next.
5. Leave it as Integrated Windows Authentication unless for some reason you like doing things manually. Click Next.
6. Choose the database to match the specific one on the server you wish to access. If you have read/write privs to the data, leave that box as is. Otherwise, change READWRITE to READONLY in the dropdown list. Click Next.
7. Adjust settings as you prefer or just click Finish.
8. Click Test Data Source. Click OK. Click OK again to close the data source administrator.

Once the appropriate DSNs are available for you, you can use the `RODBC` package to access your databases:

```
require(RODBC)
```

Once your system is configured for the ODBC connection, you need to define that connection within R, replacing the CAPS with your specific values:

```
# For example, if you'd followed the user DSN naming, above,  
# the "NAME OF DSN CONNECTION" would be "SQL3"  
connection = odbcConnect("NAME OF DSN CONNECTION", uid="YOUR USER ID",  
                        pwd="YOUR PASSWORD")
```



Using pop-up password entry in RStudio

Instead of hard-coding your password into code, you can add a pop-up prompt for a password in RStudio. In the `odbcConnect` function, use this:

```
pwd = .rs.askForPassword("Please enter your password")
```

We can verify the connection is working:

```
odbcGetInfo(connection)
```

Once the connection is made, data can flow. If you don't already know the names of the tables or you want to verify their spelling, you can get a list by writing the list to an R object and then view that object:

```
# To see all tables:  
sqlTables(connection)  
  
# To see all tables in a given schema, e.g. dbo:  
sqlTables(connection, schema="dbo")
```

If there are a lot of tables or you want to make a record of their names, create an object and then view it within R:

```
connection_tables = sqlTables(connection)  
connection_tables
```

Creating data frames from a database

Once you're connected to database, you can pull in an entire table using the `sqlFetch` function, but if you're using a database, you probably want to query it to get just what you need. To get a whole table, use `sqlFetch`:

```
# Fetch a table
whole_table = sqlFetch(connection, "table_name")
```

To make a SQL query and put the results into a dataframe, set up the query in an R object and then use the `sqlQuery` function. The following query is identical to above as it brings in the whole table:

```
# Within the double quotes you can put a valid SQL query,
# using single quotes to name the table
query_1 = "SELECT * FROM 'table_name'"

# Then use sqlQuery to pull the table into R
query_1_dataframe = sqlQuery(connection, query_1)

# While not as clear for large queries, you can combine them into one step:
query_1_dataframe = sqlQuery(connection, "SELECT * FROM 'table_name'")
```

Any typical SQL query will work; for example, this query will pull in all columns but only those records specified by the where clause, and will sort it by the order by variable(s):

```
# Usual SQL syntax works
query_2 = "SELECT * FROM 'table_name$'
WHERE variable_name = some_condition
ORDER BY ordering_variable_name"

query_2_dataframe = sqlQuery(connection, query_2)
```

Disconnecting from a database

When you have the data you need, or at least by the end of the R session, close the database connection:

```
odbcCloseAll()
```

The `RODBC` package allows R to connect with any database that allows ODBC connections, and then interact with that database using SQL. Since database platforms, local systems, and server environments vary considerably, it can be difficult to specify how to set up the initial DSN connection for any given combination—you'll need to consult with a database administrator or search online for your particular combination of databases, drivers, and system connections.

You can learn more by viewing: `vignette("RODBC", package="RODBC")`.

Creating a SQLite database inside R

You don't even need to understand the intricacies of database planning and management to be able to create your own database on the fly from within R. While there's no need to do so if your data fits into R's memory, it can be really useful if you need to subset and merge several large files to obtain a working dataframe that can be comfortably analyzed within R's memory limits.

This recipe walks through creating a SQLite database and adding tables to it; the following recipe will walk through obtaining data from the database for analysis in R.

The `sqldf` package provides more options for database work without dealing with a formal database—everything can be done from within R.

```
require(sqldf)
```

Creating a new database with `sqldf` is easy:

```
sqldf("attach 'PSES_database.sqlite' as new")
```

It's just a shell at the moment (which is why the command returns `NULL`); next create a connection to that database for use in subsequent table import and manipulation:

```
connect = dbConnect(SQLite(), dbname="PSES_database.sqlite")
```

And now you can read R data frames into the database using `dbWriteTable`:

```
# Write a data frame into a database table
# options are connection name, name of the table to write,
# and name of the data frame
dbWriteTable(connect, "pses2011", pses2011)
```

You can also read csv files directly into the database (i.e., so it is not throttled by passing through R first). We'll use the PSES 2011 data again, this time downloading it directly:

```
# Download the PSES2011 file
download.file("http://www.tbs-sct.gc.ca/pses-saff/2011/data/
  2011_results-resultats.csv", "pses2011.csv")

# Read the csv into the database as a table called pses2011_downloaded
read.csv.sql("pses2011.csv", sql = "CREATE TABLE pses2011_downloaded
  AS SELECT * FROM file", dbname = "PSES_database.sqlite")
```

Because an Excel file must be loaded into R first, and then passed to the database, you need to consider other options. If your Excel file has only a few sheets in it, saving them as csvs might be preferable. If you have many sheets to import, using XLConnect to import the sheets all at once could save a little time. They'll still need to be loaded into the database individually, however, and you should remove the dataframes when you've transferred them to the database.

```
# Download the Excel file
download.file("http://www.tbs-sct.gc.ca/pses-saff/2011/data/
  PSES2011_Documentation.xls", "PSES2011_Documentation.xls")

# Load the Excel file into R
pses2011_xls = loadWorkbook("PSES2011_Documentation.xls")

# Read each worksheet into separate dataframes within a list
pses2011_documentation = readWorksheet(pses2011_xls,
  sheet=getSheets(pses2011_xls))

# The sheet names have spaces and hyphens, which will cause
# trouble for SQLite; run names(pses2011_documentation) to see
# So, this changes the dataframe names inside the list
names(pses2011_documentation) = c("Questions", "Agency", "LEVELID15", "Demcode",
  "PosNegSpecs", "LayoutFormat")

# Add a new row to account for 0 values for LEVEL1ID in
# main pses2011 file
pses2011_documentation$Agency = rbind(pses2011_documentation$Agency,
  c(0,NA,"Other",NA,"OTH"))

# Now each sheet can be loaded into the database as a separate table
with(pses2011_documentation, {
  dbWriteTable(conn=connect, name="Questions", value=Questions,
    row.names=FALSE)
  dbWriteTable(conn=connect, name="Agency", value=Agency,
    row.names=FALSE)
```

```

dbWriteTable(conn=connect, name="LEVELID15", value=LEVELID15,
  row.names=FALSE)
dbWriteTable(conn=connect, name="Demcode", value=Demcode,
  row.names=FALSE)
dbWriteTable(conn=connect, name="PosNegSpecs", value=PosNegSpecs,
  row.names=FALSE)
dbWriteTable(conn=connect, name="LayoutFormat", value=LayoutFormat,
  row.names=FALSE)
} )

# Remove the Excel objects
rm(pses2011_xls, pses2011_documentation)

```

To see names of the tables in the database:

```
sqldf("SELECT * FROM sqlite_master", dbname = "PSES_database.sqlite")$tbl_name
```

To see table names and the SQL statements that generated them:

```
sqldf("SELECT * FROM sqlite_master", dbname = "PSES_database.sqlite")
```

To see names of columns in a particular table:

```

# Note: PRAGMA and TABLE_INFO are SQLite-specific statements
sqldf("PRAGMA TABLE_INFO(Questions)", dbname = "PSES_database.sqlite")$name

```

A shortcut for seeing a list of tables and/or fields (which may not work depending on your system):

```

dbListTables(connect)
dbListFields(connect, "Questions")

```

The equivalent of `head` to look at the start of a table would be:

```
sqldf("SELECT * FROM Questions LIMIT 6", dbname = "PSES_database.sqlite")
```

Finally, whenever you're done interacting with the database, close the connection:

```
dbDisconnect(connect)
```

SQLite was built intentionally to be “lightweight” database, and being able to set it up and access it from entirely within R makes it perfect for occasional to moderate desktop use. As is the case with the many different flavors of SQL, there are nuances in SQLite (as we saw with the PRAGMA statement, above), but most basic uses remain the same across those different types of SQL.

K> You can see a list of the SQL statements available using: K> K> K> .SQL92Keywords K>

More info on the way R interfaces with RDBMSs can be found on the homepage for the [DBI package](#)²⁰, and details on SQLite can be found at its [homepage](#)²¹.

Those with more sophisticated desktop database needs might consider [PostgreSQL](#)²², which also plays well with R via the [RPostgreSQL](#) package.

Creating a dataframe from a SQLite database

Once you have a database connection, you’re able to access exactly what you need and pull just that into R as a dataframe, conserving memory and saving time when working on projects that will need to access a variety of tables.

Load the `sqldf` package if it’s not already loaded, and then reconnect to the database we created in the previous recipe:

```
require(sqldf)
connect = dbConnect(SQLite(), dbname="PSES_database.sqlite")
```

Selecting subsets from a database table using `sqldf` is based on (of course) SQL select statements. For example, this statement brings in three of the columns from the LEVEL1ID table:

```
pses_acr = sqldf("SELECT
  Level1ID
, Acronym_PSES_Acronyme_SAFF AS Acronym
, DeptNameE as Department
FROM Agency",
  dbname="PSES_database.sqlite")
```

You can (and should!) join tables within the database before bringing it into R. For example, this code merges the main data table (`pses2011`) with the agency name/abbreviation table to create a dataframe called `pses2011-agency`:

²⁰<https://github.com/rstats-db/DBI>

²¹<http://www.sqlite.org/>

²²<http://www.postgresql.org/>

```
pses2011_agency = sqldf("SELECT
  maintable.*
, agencytable.DeptNameE as Agency
, agencytable.Acronym_PSES_Acronyme_SAFF as Abbr
FROM pses2011 maintable
LEFT JOIN Agency agencytable
  ON maintable.LEVEL1ID = agencytable.Level1ID",
dbname="PSES_db.sqlite")
```

Don't forget to close the connection:

```
dbDisconnect(connect)
# or
# close(connect)
```

Basically, `sqldf("SQL QUERY", dbname="NAME OF DATABASE")` is all you need to use SQLite within R, and it works on both databases and dataframes—if the latter, it's just `sqldf("SQL QUERY")`. It is important to note that unlike R, SQLite is not case sensitive, so be careful naming tables and columns—a and A are identical to SQLite. (SQL functions are in CAPS in the code examples in this book simply to distinguish them from table and column names, which are left in the same case that R sees them.)

You can do most basic types of SQL work in this fashion; for example, if you want to count the number of rows in your new dataframe where ANSWER3 (“Neutral”) is more than 50% of the total responses, have it grouped by Question and Agency, and order the result by highest to lowest counts, you can use:

```
agency_row_count = sqldf("SELECT
  Question
, Agency
, COUNT(ANSWER3) as Answer3_Count
FROM pses2011_agency
WHERE ANSWER3 > 50
GROUP BY Question, Agency
ORDER BY Answer3_Count desc")
```

The possibilities are fairly limitless by containing large files inside a database and bringing in only what's needed for analysis using SQL, such as via `sqldf` or `RODBC`. A basic knowledge of SQL is required, of course, but the learning curve is very small for R users and well worth taking on given its centrality in data access and manipulation. It also serves as a great bridge for those used to SQL who want to use R more extensively but are not yet accomplished in using the R functions that have SQL equivalents (e.g., R's `merge` vs. SQL joins). For example, the standard SQL `CASE WHEN` statement could be used to generate new columns that sum the number of cases in which the majority either strongly agreed or strongly disagreed:

```
agency_row_count = sqldf("SELECT
    Question
    , SUM(CASE WHEN ANSWER1 > 51 THEN 1 ELSE 0 END) AS Strong_Agreement
    , SUM(CASE WHEN ANSWER5 > 51 THEN 1 ELSE 0 END) AS Strong_Disagreement
  FROM pses2011_agency
  GROUP BY Question, Agency
  ORDER BY Question desc")
```

It's worth pointing out that if your data fits in memory, there usually isn't need for creating a local database; merging and manipulation can occur inside R, and recipes in the next chapter explore ways to do that.

Getting data from the web

Working through a proxy

If you're working behind a company firewall, you may have to use a proxy to pull data into R from the web. The `httr` package makes it simple.

```
require(httr)
```

Enter your proxy address, the port (usually 8080), and your user name/password in place of the CAPS code:

```
set_config(use_proxy(url="YOUR_PROXY_URL", port="YOUR_PORT",
  username="YOUR_USER_NAME", password="YOUR_PASSWORD"))
```

There is a bewildering array of methods to access websites from within R, particularly while having to pass through a proxy, and most of them are obscure to even the most established quants. Thanks to `httr`, all of that complexity has been hidden behind a few simple functions—type `vignette("quickstart", package="httr")` if you want more information.

Sometimes your proxy lets you through for some things but not others; this happens to me most frequently with `https` URLs. In that case, I remove the `s` and use `http`, and the connection is usually fine after that.

Scraping data from a web table

Sometimes, webpages have a treasure trove of data in tables... but they don't have an option to download it as a text or Excel file. And while data scraping add-ins are available for modern web browsers like Chrome (*Scraper*) or Firefox (*Table2Clipboard* and *TableTools2*) that make it as easy

as point-and-click, if the tables aren't set up for easy download or a simple cut-and-paste, you can use the `XML` package to grab what you need.

While there are a few R packages that allow scraping, the `XML` package is the simplest to begin with, and may serve all your needs anyway.

```
require(XML)
```

As a simple example, we can explore the [2010 National Survey on Drug Use and Health²³](#). Like many websites, it has pages that contain multiple tables. Say we just want Table 1.1A. First, read in the raw HTML:

```
drug_use_2010 = readLines("http://archive.samhsa.gov/data/NSDUH/
2k10nsduh/tabs/Sect1peTabs1to46.htm")
```

Then read in the first table (`which=1`) part of the HTML:

```
drug_use_table1_1 = readHTMLTable(drug_use_2010, header=T, which=1,
stringsAsFactors=FALSE)
```

What if you want more than one table? Since we've read in the entire webpage, we can scrape it to extract whatever information we need. For example, let's say we want tables 1.17A and 1.17B. Using the webpage's [table of contents²⁴](#), we find that the tables we want are 31st and 32nd:

```
drug_use_table1_17a = readHTMLTable(drug_use_2010, header=T, which=31,
stringsAsFactors=FALSE)
drug_use_table1_17b = readHTMLTable(drug_use_2010, header=T, which=32,
stringsAsFactors=FALSE)
```

Scraping tables that cover multiple pages

What about a case where you need a table that goes over many pages? For example, ProPublica has a website that lists [deficiencies in nursing home care in the United States²⁵](#). At the time of this writing (mid 2016), the subset that includes Texas contains more than 22 thousand rows that cover 524 different (web)pages.

Obviously, cutting and pasting that would seriously suck.

Here's how you can do it in R with a few lines of code. First, set up an R object that will iteratively list every one of the 524 web URLs:

²³<http://archive.samhsa.gov/data/NSDUH/2k10nsduh/tabs/Sect1peTabs1to46.htm>

²⁴<http://archive.samhsa.gov/data/NSDUH/2k10nsduh/tabs/TOC.htm#TopOfPage>

²⁵<http://projects.propublica.org/nursing-homes/>

```
allpages = paste("http://projects.propublica.org/nursing-homes/findings/
search?page=", 1:524, "&search=&ss=ALL&state=TX", sep="")
```

...then read in each page into an R list format:

```
tablelist = list()
for(i in seq_along(allpages)){
  page = allpages[i]
  page = readLines(page)
  homes = readHTMLTable(page, header=T, which=1, stringsAsFactors = FALSE)
  tablelist[[i]] = homes
}
```

...and finally turn the list into a data frame:

```
nursing_home_deficiencies = do.call(rbind, lapply(tablelist, data.frame,
stringsAsFactors=FALSE))
```

This result still needs some cleaning before we can move to analysis—Chapter 3 provides an example of cleaning this particular scrape in addition to the more general recipes.

Working with APIs

Many data-oriented sites have APIs that make it easy to pull in data to R. Some are wide open, but most true APIs require keys. And of course, each one has its own terms of service and ways to implement.

A good example of a typical API is via the US Census Bureau's American Community Survey (ACS). First, go to their *Terms of Service* page²⁶, and if you agree with those terms, click the Request a KEY button at the bottom of the left side menu bar. A pop-up will ask for your organization and email address (and agreement to the *Terms*, of course), and it will return a key to that email within a few minutes.

The `acs` package provides the link between the ACS and R, and also provides a function to permanently install the key so you don't have to enter it every time you hit the API:

```
require(acs)
api.key.install(key="y0uR K3y H3rE")
```

The ACS is *vast*. Since this is simply a recipe for example's sake, we'll assume you know the table and geographies you want data on... say, population by county in Texas. `acs` keeps the data frame part of the object in the `estimate` slot:

²⁶<http://www.census.gov/data/developers/about/terms-of-service.html>

```
tx_pops_acs = acs.fetch(geography=geo.make(state="TX", county="Harris"),
    table.number="B01003")
tx_county_pops = data.frame(tx_pops_acs@estimate)
```

We'll see more use of the `acs` package in the *Maps* section of Chapter 7.

Creating fake data to test code

Creating fake data is useful when you want to test how code will work before doing it “for real” on larger, more complicated datasets. We’ll create a few fake dataframes in this recipe as both an example as how to do it, as well as for use in other recipes in this book.

While creating and using fake data is useful for lots of cases, it is especially useful for merging/joining (which we’ll explore in the next chapter). Sometimes joins may not behave like you expect that they will due to intricacies in your data, so testing it first on known (fake) data helps you determine whether any problems arose because of the code, or because of something in your data. If the fake data merge as expected but the real data don’t, then you know that there’s something in your data (and not the code!) that is messing with the join process. In large, enterprise-scale databases, it is typically the joins that can cause the most unexpected behavior—problems there can lead to the wrong results, impacting everything downstream of that join for the worse.

Whatever the use case, testing code on fake data can sometimes save considerable time that would have been lost debugging.

Using a variety of sources, we’ll develop four dataframes that mimic a customer sales type of database (of course, these datasets are all small, so a database is unnecessary here).

```
# Table A - customer data
set.seed(1235813)
customer_id = seq(1:10000)
customer_gender = sample(c("M", "F", "Unknown"), 10000, replace=TRUE,
    prob=c(.45, .45, .10))
customer_age = round(runif(10000, 18, 88), 0)
make_NAs = which(customer_age %in% sample(customer_age, 25))
customer_age[make_NAs] = NA
customer_purchases = round(rlnorm(10000)*100, 2)

# Table B - city/state/zip and business density lookup table
download.file("ftp://ftp.census.gov/econ2012/CBP_CSV/zbp12totals.zip",
    "temp.zip", method="curl")
zipcodes = read.table(unz("temp.zip", "zbp12totals.txt"), header=T, quote="\"",
    sep=",", colClasses="character")
zipcodes = zipcodes[,c(1,11:12,10)]
```

```

# End of Table B

# Table A, continued
customer_zip = zipcodes[sample(1:nrow(zipcodes), 10000, replace=TRUE),]$zip
customers = data.frame(customer_id, customer_gender, customer_age,
  customer_purchases, customer_zip, stringsAsFactors=FALSE)

# Table C - results of a product interest survey
ones = seq(1, 1, length.out = 2000)
zeros = seq(0, 0, length.out = 2000)
strongly_agree = c(ones, zeros, zeros, zeros, zeros)
agree = c(zeros, ones, zeros, zeros, zeros)
neutral = c(zeros, zeros, ones, zeros, zeros)
disagree = c(zeros, zeros, zeros, ones, zeros)
strongly_disagree = c(zeros, zeros, zeros, zeros, ones)
survey = data.frame(customer_id, strongly_agree, agree, neutral, disagree,
  strongly_disagree)

# Table D - lookup table to match states to regions
state_regions = data.frame(datasets::state.abb, datasets::state.name,
  datasets::state.region, datasets::state.division, stringsAsFactors=FALSE)
colnames(state_regions) = c("abb", "state", "region", "division")

```

In Table A, we've created a customer demographic and sales dataset. The `sample` function creates a random vector of male and female genders, as well as unknown, proportional to the weights in the `prob` option. The age and purchase values were created using the random number generators based on the uniform and log-normal distributions, respectively. The age value vector was subsequently replaced with 25% NAs. Each customer's zip code was generated from a random selection of the zip codes in Table B. All vectors are then brought together into the `customers` data frame.

Table B, we've downloaded and unzipped some 2012 County Business Patterns data from the US Census to create a “locations” lookup table. (Metadata for this dataset is online [here](#)²⁷; in addition to the city/state/zip fields, we've also kept the `est` field, which is the number of business establishments in this zip code in the first quarter of 2012.)

Table C is just a systematic selection designating a survey response from each customer. This could also be done randomly, but is done this way here simply for illustration.

Finally, Table D takes advantage of R's built-in data from the `datasets` package to generate a lookup table of states and state regions.

Note: if you don't have curl on your system, the one-line ftp call for the zipcode data won't work. You can either `install curl`²⁸ (and restart R), or use the `download.file` and `unzip` approach as shown

²⁷https://www.census.gov/econ/cbp/download/noise_layout/ZIP_Totals_Layout10.txt

²⁸<https://curl.haxx.se/download.html>

in *Chapter 1* within the *Acquire data* section.



wakefield is a new R package that will generate random datasets for you. In addition to the random data frame generation, it also has a nice function to visualize the distribution of NAs and data types in a data frame (`?table_heat`). Check out its GitHub site²⁹ for more info.

Writing files to disk

There is usually no reason to save a dataframe for use outside R in any format other than a flat file: the only way to provide truly reproducible analytics is to ensure everything can be read by any program, language, or platform.

`write.table` using `sep=","` will save your dataframe as a csv, and setting `row.names=FALSE` will save it without the row numbers that R provides by default.

```
write.table(pses2011, "pses2011.csv", sep=",", row.names=FALSE)
```

Sometimes you have a single data frame that you want to export as separate text files based on a factor. Here's how to do that:

```
# Split data frame into a list by a factor
pses2011_split = split(pses2011, pses2011$LEVEL1ID)

# Save each new data frame as an individual .csv file based on its name
lapply(1:length(pses2011_split), function(i)
  write.csv(pses2011_split[[i]],
  file = paste0("~/BIWR/",
  names(pses2011_split[i]), ".csv"),
  row.names = FALSE))
```

Sometimes, you've made a lot of changes to data you've loaded, particularly via changes in column classes, and you want to preserve that for future analysis within R. The `save` function is what you need:

```
save(pses2011, file="pses2011.Rdata")
```

If you want to save a data frame in XML format (at the cost of a larger file size!), use the `write.xml` function in the `kufife` package.

²⁹<https://github.com/trinker/wakefield>

```
kulife::write.xml(iris, file = "iris.xml")
```

If you want to save data in JSON format, use `jsonlite` with the `pretty` option set to true to have it formatted cleanly. Then `sink` it into a file in the working directory.

```
iris_json = jsonlite::toJSON(iris, pretty = TRUE)
sink("iris_json.json")
iris_json
sink()
```

Note you can save memory by calling a function from a package using the `::` connector, which will run that function without having to load its package. This can also be useful if you don't want to mask functions in already-loaded packages that have the same name.

Chapter 3: Cleaning and Preparing Data

- Understanding your data
- Cleaning up
- Merging dataframes: Joins, unions, and bindings
- Subsetting: filter and select from a dataframe
- Creating a derived column
- Reshaping a dataframe between wide and long
- Piping with `%>%` : Stringing it together in dplyr

Understanding your data

Identifying data types and overall structure

Before you do anything else—every dataset should be evaluated with a few basic functions. It's funny how often this is overlooked...but it's not funny how much time you can waste writing code that fails until you realize you don't have the data in the proper format.

Factors, characters, dates, values that are out-of-range for the measurement, inappropriate dummy values, and NAs tend to be the data types or results that will screw up future analyses—so using the the following functions, every time you first pull in a dataset, will save a lot of time and grief in the long run.

We saw the frequent use of `str` in the *Chapter 1* example to understand our dataset and to ensure our changes worked as expected. `head` (and `tail`), `summary`, and `complete.cases` are a few other important functions that help you understand your data set as seen by R.

Using Table A from the fake customer data created in Chapter 2, we might explore our data one or more times a session with the following functions:

```
# View the structure of the dataset, particularly data types
str(customers)

# Look at the first (or last) 6 lines of the data frame
head(customers)
tail(customers)
```

```
# Obtain basic summary stats
summary(customers)

# Look at the rows with missing data
missing_data = customers[!complete.cases(customers),]

missing_data
```

Ironically, a few days before I started this chapter, I was trying to use a function that needed factors on a variable I just *knew* were factors, but R had considered them characters because as is my habit, I had used the `stringsAsFactors=FALSE` option when I imported the data. After banging my head for a while, I pulled up `str` and the problem became clear immediately.

Always check! Even if you're a seasoned professional!

Identifying (and ordering) factor levels

Factors can be ordered or unordered; when created they usually are arranged alphabetically.

```
levels(customers$customer_gender)

"F"      "M"      "Unknown"
```

If you do need to change the order—for example, if you want them to display in plots in a different order—you can use the `ordered` function:

```
customers$customer_gender = ordered(customers$customer_gender,
  c("Unknown", "F", "M"))

"Unknown" "F"      "M"
```

Identifying unique values or duplicates

The `unique` function will provide the distinct values in a variable, in the order that they appear in the data.

```
unique(customers$customer_age)

[1] 72 44 83 74 66 42 18 NA 34 35 32 50 76 23 36 54
[17] 47 49 33 70 71 68 43 61 45 41 86 85 60 87 24 19
[33] 31 78 59 48 40 69 22 21 67 80 65 84 39 81 51 58
[49] 30
```

You can use the `sort` function to see the unique values in order (excluding NAs):

```
sort(unique(customers$customer_age))

[1] 18 19 21 22 23 24 30 31 32 33 34 35 36 39 40 41
[17] 42 43 44 45 47 48 49 50 51 54 58 59 60 61 65 66
[33] 67 68 69 70 71 72 74 76 78 80 81 83 84 85 86 87
```

You can see if any rows are duplicated in the data frame with the `duplicated` function, but by itself just returns TRUE/FALSE. Use this structure instead to find any duplicated rows:

```
customers[duplicated(customers),]

[1] customer_id      customer_gender
[3] customer_age     customer_purchases
[5] customer_zip
<0 rows> (or 0-length row.names)
```

There aren't any, so to illustrate what it shows when it finds a duplicate, we'll create a new row that is the duplicate of row 2718:

```
customers[10001,] = customers[2718,]

customers[duplicated(customers),]

  customer_id customer_gender customer_age customer_purchases customer_zip
10001        2718            M         47          75.43       73140
```

Before moving on, make sure you remove the duplicated row:

```
customers = unique(customers)
```

```
Console ~/ ↵
> tail(customers, 2)
  customer_id customer_gender customer_age customer_purchases customer_zip
10000      10000            M           59          403.44       63431
10001      2718             M           47          75.43        73140
> str(customers)
'data.frame': 10001 obs. of  5 variables:
 $ customer_id    : int  1 2 3 4 5 6 7 8 9 10 ...
 $ customer_gender : Factor w/ 3 levels "F","M","Unknown": 1 2 2 1 1 2 2 2 1 1 ...
 $ customer_age   : num  72 44 83 74 74 66 72 42 18 NA ...
 $ customer_purchases: num  43 176.3 3502.1 30.8 213.5 ...
 $ customer_zip    : Factor w/ 8827 levels "01001","01002",...: 8552 8302 5190 1425 2940 1898 6323 5046 6468 8048
...
> customers = unique(customers)
> tail(customers, 2)
  customer_id customer_gender customer_age customer_purchases customer_zip
9999       9999            F           NA         114.05       46302
10000     10000            M           59          403.44       63431
> str(customers)
'data.frame': 10000 obs. of  5 variables:
 $ customer_id    : int  1 2 3 4 5 6 7 8 9 10 ...
 $ customer_gender : Factor w/ 3 levels "F","M","Unknown": 1 2 2 1 1 2 2 2 1 1 ...
 $ customer_age   : num  72 44 83 74 74 66 72 42 18 NA ...
 $ customer_purchases: num  43 176.3 3502.1 30.8 213.5 ...
 $ customer_zip    : Factor w/ 8827 levels "01001","01002",...: 8552 8302 5190 1425 2940 1898 6323 5046 6468 8048
...
> |
```

Sorting

The `dplyr` package is incredibly useful for data manipulation, as we'll see later in this chapter, but it's also useful for understanding your data. We often want to sort on variables and see the results to start to get a sense of the data and its patterns. The base R installation includes `sort` and `order` functions, but they are clunky when you try to use this this way. `dplyr`'s `arrange` function makes it easy.

```
require(dplyr)

# Look at top 10 customers by purchases (descending)
head(arrange(customers, desc(customer_purchases)), 10)

# Look at top 10 customers by age (ascending) and purchases (descending)
head(arrange(customers, customer_age, desc(customer_purchases)), 10)

# NAs are placed last
tail(arrange(customers, customer_age, desc(customer_purchases)), 10)

# You can store the sorted data frame, of course
customers = arrange(customers, customer_age, desc(customer_purchases))
```

Cleaning up

Converting between data types

In the previous chapter, we saw a JSON import that turned all of the variables into character data types. While this result is relatively rare, there is almost *always* a need to convert a column from one data type to another. After you've used `str` to figure out what type(s) of data you have, you can use conversion functions to change them as you need. The following are the primary conversion functions:

Data type you want	Function
Character / String	<code>as.character(OBJECT, ...)</code>
Factor / Category	<code>as.factor(OBJECT, ...)</code>
Numeric / Double	<code>as.numeric(OBJECT, ...)</code>
Integer	<code>as.integer(OBJECT, ...)</code>
Date	<code>as.Date(OBJECT, format="yyyy-mm-dd", ...)</code>
Datetime	<code>as.POSIXct(OBJECT, tz="CURRENT TIME ZONE", ...)</code>

Sometimes you may need to reduce your data set to only a certain type of variable, say all numeric and none of the other variable types. Use the `sapply` function as follows:

```
customers_numeric = customers[sapply(customers, is.numeric)]
```



Dates and Times

Note that conversion to and between dates and times can be tricky, depending on the structure of the existing date/time variable. If it's already in standard ISO format (e.g., `yyyy-mm-dd`), these basic conversions will work fine. If not, you'll have to mess with the `format` option and/or use other techniques. See *Chapter 6: Trends and Time* for lots of details on working with and converting date and time values.

Cleaning up a web-scraped table: Regular expressions and more

In many cases the data frame will require a little clean up after it's loaded. Clarifying messy or unclear column names, pattern replacement, converting between data types, and concatenating or extracting portions of variables' results are all common cleaning needs before analysis.

`gsub` is probably the most useful general purpose approach to regular expressions in R. You should look at the help files for `?gsub` for regular expressions, `?strsplit` for splitting a character string, and `?substr` for extracting a portion of a string (as well as the links in their *See Also* sections) to get more details on those and similar or otherwise related functions.

There are many possible approaches to cleaning, so as an example, the following code does some cleaning of the ProPublica *Nursing Home Deficiencies* data set scraped from the web in the previous chapter, using a few different functions:

```
# Create cleaner column names
colnames(nursing_home_deficiencies) = c("Date", "Home", "City", "State",
                                         "Deficiency_Count", "Severity")

# Replace abbreviations and extraneous text
nursing_home_deficiencies$State = gsub("Tex.", "TX",
                                         nursing_home_deficiencies$State, fixed = TRUE)

nursing_home_deficiencies$Home = gsub("(REPORT) Home Info", "",
                                         nursing_home_deficiencies$Home, fixed = TRUE)

# Take the first letter (severity score) and remove the
# duplicate and extraneous text
nursing_home_deficiencies$Severity = substr(nursing_home_deficiencies$Severity,
                                              1, 1)

# Clean the date information and convert to date object
nursing_home_deficiencies>Date = gsub(".", "", nursing_home_deficiencies$Date,
                                         fixed = TRUE)

clean = function(col) {
  col = gsub('Jan', 'January', col, fixed = TRUE)
  col = gsub('Feb', 'February', col, fixed = TRUE)
  col = gsub('Aug', 'August', col, fixed = TRUE)
  col = gsub('Sept', 'September', col, fixed = TRUE)
  col = gsub('Oct', 'October', col, fixed = TRUE)
  col = gsub('Nov', 'November', col, fixed = TRUE)
  col = gsub('Dec', 'December', col, fixed = TRUE)
  return(col)
}

nursing_home_deficiencies$Date = clean(nursing_home_deficiencies$Date)

nursing_home_deficiencies$Date = as.Date(nursing_home_deficiencies$Date,
                                         format = "%B %d, %Y")
```

The use of `fixed = TRUE` option above tells `gsub` to use exact matching; the default is `false`, which uses [POSIX 1003.2 extended regular expressions³⁰](#).

³⁰<http://www.cheatography.com/davechild/cheat-sheets/regular-expressions/>

Changing a single column's name

The `colnames` function works in the above example to specify the names for all columns at once. If you just want to change a single column's name, the pattern is slightly different:

```
colnames(my_data)[column_number] = "new_column_name"
```

Missing data: Converting dummy NA values

Sometimes, you'll find values or placeholders where no data should be; for example, -99 and 9999 are common among older data sets to represent missing values (represented by `NA` in R). To fix this, you can use `gsub`, but you could use assignment to ensure the values don't go from numeric to character:

```
# This will work
my_data$my_variable = as.numeric(gsub("9999", NA, my_data$my_variable))

# But this is better, albeit less transparent
my_data$my_variable[my_data$my_variable==9999] = NA

# And this will do it for the entire data frame
my_data[my_data == 9999] = NA
```

It's also useful to remember to use `na.rm=TRUE` when including data with `NA` values in calls to many base installation functions (e.g., `mean`), otherwise R will return `NA` or throw an error.

```
# This won't work
mean(customers$customer_age)
NA

# This will
mean(customers$customer_age, na.rm=TRUE)
53.42279
```

Rounding

Sometimes you'll have data that's already summarized in some form (e.g., `means`) and you need to reduce the number of significant figures or decimal places, either for display purposes or to avoid spurious precision. Use `round` for decimal places, and `signif` for significant figures, using the `digits=` option as necessary:

```
round(mean(customers$customer_age, na.rm=TRUE), 1)
```

53.4

```
signif(mean(customers$customer_age, na.rm=TRUE), 1)
```

50

```
signif(mean(customers$customer_age, na.rm=TRUE), 3)
```

53.4

`ceiling`, `floor`, and `trunc` provide other ways to round numbers to an integer if you need to round up, down, or remove the decimal portion, respectively.

Concatenation

Mashing things together into the same field is done with the `paste` or `paste0` function; the only difference between the two is that `paste` requires a separator (defaults to a space) while `paste0` will concatenate without any separators.

For example, if you want to attach the first of the month to a `yyyy-mm` field, these would be equivalent:

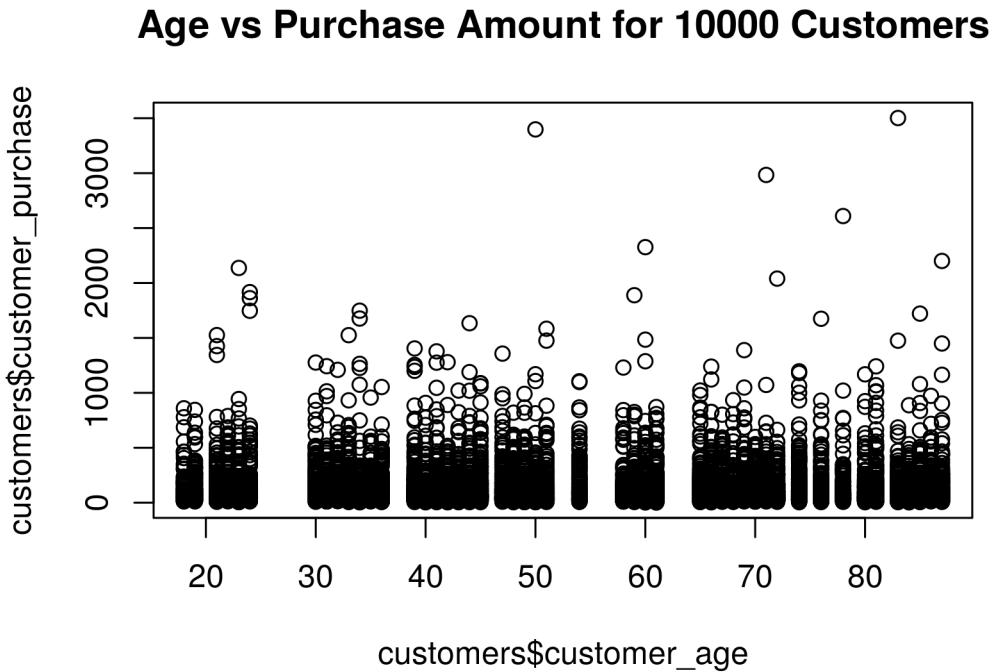
```
my_data$date = as.Date(paste(mydata$yyyy_mm, "01", sep="-"))
```

```
my_data$date = as.Date(paste0(mydata$yyyy_mm, "-01"))
```

`paste` works for anything, but you'll have to pay attention to the data type it creates. The commands above would have created character variables if we hadn't specified that we wanted a date object, for example.

It can also be useful for including soft-coded values in a title or a plot, axis, annotation, etc.:

```
plot(customers$customer_age, customers$customer_purchase,
  main=paste("Age vs Purchase Amount for",
  length(customers$customer_age), "Customers"))
```



Commas in Numbers

Every once in a while you'll get a file that has commas in a numeric field (cough Excel cough). A data type conversion, paste, and gsub can be used together to clean that up in one line of code:

```
my_data$my_variable = as.numeric(paste(gsub(", ", "", my_data$my_variable)))
```

Merging dataframes

We've already touched on merging tables inside a SQLite database to create a dataframe in Chapter 2, but more often you usually need to merge two dataframes inside R.

The best package to do this in R is `dplyr`, and part of its greatness is that it works on dataframes in the same way as it does on databases. However, SQL jockeys may still prefer to use `sqldf`. I tend to use the former in R-heavy projects and the latter in projects where I need to work from a database or want to streamline the join results.

Of course, base R has native joining abilities as well via the `merge` function, but it tends to be somewhat clunkier than `dplyr` or `sqldf`.

Joins

The recipe provides an overview of joining tables via dplyr's *_join functions, shows a little more on doing joins using sqldf, and reviews R's built-in `merge` function for comparison. We'll also cover the concepts of how the types of joins work to make it easy to visualize what we are trying to accomplish.

First, load the libraries:

```
require(dplyr)
require(sqldf)
```

We'll use the fake customer data we created in the previous chapter to illustrate these methods.

Our objective is to merge the four tables so that we have the customer data alongside an expanded view of their location and their answer from the survey. Using dplyr:

```
customer_locations = left_join(customers, zipcodes, by =
  c("customer_zip" = "zip"))

customer_regions = left_join(customer_locations, state_regions, by=
  c("stabbr" = "abb"))

customer_survey = left_join(customer_regions, survey, by =
  c("customer_id" = "customer_id"))
```

The base package's `merge` function is the old standby, but is considerably slower than dplyr, and the type of join is implied by the `all` option, which reduces code clarity.

```
customer_locations = merge(customers, zipcodes, by.x = "customer_zip",
  by.y = "zip", all.x = TRUE)

customer_regions = merge(customer_locations, state_regions, by.x = "stabbr",
  by.y = "abb", all.x = TRUE)

customer_survey = merge(customer_regions, survey, by.x="customer_id",
  by.y="customer_id", all.x=TRUE)
```



R will match on columns with the same names, so while you can usually get away without specifying the `by` variables with `merge` or `dplyr`, you should *always* specify them anyway—remember your downstream users!

With both the `dplyr` joins and the `base` merges, you get everything in both dataframes. You either have to select only the columns you want in a new dataframe before the merge, or remove the excess columns after it. Of course, using SQL you can choose just the columns you want, which is illustrated below.

```
customer_survey = sqldf("
  SELECT
    customers.*
    , zipcodes.city
    , zipcodes.stabbr
    , state_regions.state
    , state_regions.region
    , state_regions.division
    , survey.strongly_agree
    , survey.agree
    , survey.neutral
    , survey.disagree
    , survey.strongly_disagree
  FROM customers
  LEFT JOIN zipcodes
    ON customers.customer_zip = zipcodes.zip
  LEFT JOIN state_regions
    ON zipcodes.stabbr = state_regions.abb
  LEFT JOIN survey
    ON customers.customer_id = survey.customer_id"
  , stringsAsFactors=FALSE)
```

If you're coming to R with some SQL in your background, you might wonder why this wasn't the method to start with—after all, we can do it all in one step, we get exactly what we want, and we can sort (`ORDER BY`) or add any other SQL statement here to get whatever else we need. The reason is that showing each join/merge in a single step allows other R coders to see exactly what was done more clearly. SQL merge code can get ugly fast, so it's useful to consider whether the project (=code) is going to be passed on to others in the future and whether they are familiar with SQL, and code accordingly.

That said, I love using this approach; SQL provides you many more options and ability to customize your output to be exactly what you need. As long as your downstream dependencies are fine with SQL, you might consider this as your *primary* approach to joining.

Obviously, there's more to life than left joins. The following table lays out the different types of joins, and how `dplyr`, `merge`, and `sqldf` perform each, given data frames/tables *A* and *B*.

Type of Join	Visualization	dplyr	merge	sqldf
Left		<pre>left_join(A, B, by=c("A.key" ="B.key"))</pre>	<pre>merge(A, B, by.x="A.key", by.y="B.key", all.x=TRUE)</pre>	<pre>SELECT * FROM A LEFT JOIN B ON A.key = B.key</pre>
Inner		<pre>inner_join(A, B, by=c("A.key" ="B.key"))</pre>	<pre>merge(A, B, by.x="A.key", by.y="B.key", all=FALSE)</pre>	<pre>SELECT * FROM A INNER JOIN B ON A.key = B.key</pre>
Full (Outer)		<i>Not yet implemented</i>	<pre>merge(A, B, by.x="A.key", by.y="B.key", all=TRUE)</pre>	<i>Not yet implemented</i>
Semi		<pre>semi_join(A, B, by=c("A.key" ="B.key"))</pre>	<i>n/a</i>	<pre>SELECT * FROM A LEFT JOIN B ON A.key = B.key WHERE B.key IS NULL</pre>
Anti		<pre>anti_join(A, B, by=c("A.key" ="B.key"))</pre>	<i>n/a</i>	<i>Not yet implemented</i>
Right**		<pre>left_join(B, A, by=c("B.key" ="A.key"))</pre>	<pre>merge(A, B, by.x="A.key", by.y="B.key", all.y=TRUE)</pre>	<pre>SELECT * FROM B LEFT JOIN A ON B.key = A.key</pre>

Type of Join	Visualization	dplyr	merge	sqldf
Right Anti**		semi_join(B, A, n/a by=c("B.key" ="A.key"))		SELECT * FROM B LEFT JOIN A ON B.key = A.key WHERE A.key IS NULL

** Although dplyr doesn't have a formal right join function, simply reversing the table order and using left join serves the same purpose. Though why you'd want to use a right join anyway is beyond me—right joins are just backwards left joins and it makes more sense to focus on your primary dataset as the "A" table.

You can merge large files with `data.tables`, but the way you specify the joins is opaque—e.g., `datatable_AB = datatable_A[datatable_B]` is a left join, given you've specified the keys ahead of time—so for now it's probably best employed for single-user/casual use and not for production code unless speed is more important than clarity.

Unions and bindings

Concatenating two tables can be handled two ways: via `sqldf` or through R's native binding functions. It's often easier just to use the native R functions if you already have the two tables ready to concatenate: `rbind` for stacking and `cbind` for concatenating sideways. "Being ready" means that the columns are identically named, and that there are the same number of columns in each table. The `sqldf` approach is best when you're pulling specific columns from each table to stack into a single data frame.

We'll split the `customer_survey` data frame in half row-wise to illustrate unions:

```
customers_first_half = customer_survey[1:5000,]
customers_second_half = customer_survey[5001:10000,]
```

If you need to stack two tables with the same columns and prefer the SQL approach (and/or already have `sqldf` loaded), just include it in your query, e.g.:

```
customers_whole = sqldf(
  "SELECT * FROM customers_first_half
  UNION
  SELECT * FROM customers_second_half")
```

The `UNION` above would be done with `rbind` as follows:

```
customers_whole = rbind(customers_first_half, customers_second_half)
```

If you have some variables in one table that don't occur in the other table, you can assign them in the other table as a new variable of NA values, and then rbind them together; we saw an example of this in Chapter 1 when combining the actual and forecast values together into a single data frame for plotting:

```
# Create a data frame with the original data
# and space for the forecast details
use_df = data.frame(Total_Use = power_monthly$Total_Use_kWh,
  Forecast = NA, Upper_80 = NA,
  Lower_80 = NA, Upper_95 = NA, Lower_95 = NA)

# Create a data frame for the forecast details
# with a column for the original data
use_fc = data.frame(Total_Use = NA, Forecast = total_use_fc$mean, Upper_80 =
  total_use_fc$upper[,1], Lower_80 = total_use_fc$lower[,1],
  Upper_95 = total_use_fc$upper[,2], Lower_95 = total_use_fc$lower[,2])

# "Union" the two data frames into one
use_ts_fc = rbind(use_df, use_fc)
```

The sqldf version of the above would be considerably larger, as the total_use_fc is actually an R list object, so it would have to be converted to a data frame before you could UNION or rbind.

Binding two tables together side-by-side is done with the cbind or data.frame functions (cbind is just a wrapper for data.frame). This approach is *row-blind*: it will line up the two tables *in the order that they occur*. It is up to you to ensure the observations (rows) match across the two data frames. The two data frames must also have the same numbers of observations (i.e., the same length).

Again, we'll split the customer_survey data column-wide in two for illustration:

```
customers_left_half = customer_survey[,1:7]

customers_right_half = customer_survey[,8:14]
```

We can put them back together again either way:

```
customers_back_together = cbind(customers_left_half, customers_right_half)

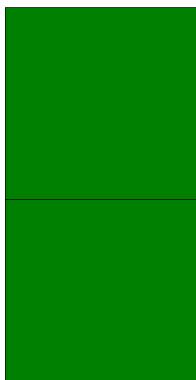
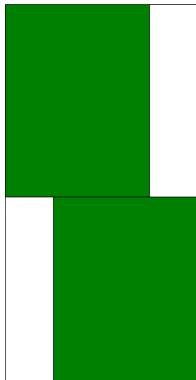
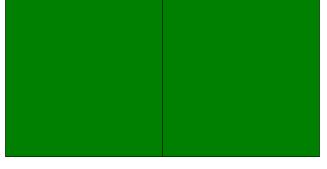
customers_back_together = data.frame(customers_left_half, customers_right_half)
```

An alternative to the cbind is a join with sqldf; a natural or an inner join will work if you have an appropriate key in each table (i.e., customer_id in this example). Then you don't need to ensure the order is correct, either.

```
# Keep the ID in the second table
customers_right_half = customer_survey[,c(1,8:14)]  
  

# Use sqldf to make a natural join
customers_back_together = sqldf("
  SELECT *
  FROM customers_left_half A
  JOIN customers_right_half B
  ON A.customer_id = B.customer_id")
```

As in the last recipe, here's a visualization and function summary for unions and concatenated tables:

Type of Join	Visualization	r/cbind	sqldf
Union		rbind(A, B)	SELECT * FROM A UNION SELECT * FROM B
Semi-union		see above	see below**
Concatenate		data.frame(A, B) cbind(A, B)	SELECT * FROM A JOIN B ON A.key = B.key

** Use UNION to bind the two tables, but you'll need to cast columns as null in the tables where

they don't already occur, and ensure the column names match in both places, e.g., `CAST(NULL as numeric(8)) AS Forecast` could be used as one line in the table with the actual data so it would read NA when unioned with the forecast values.

Subsetting: filter and select from a dataframe

Even after joining, the dataset we have is rarely the dataset we need to complete a project. The `dplyr` package has a few really useful data manipulation functions that work on data frames just like they do on databases, and tend to be simpler, faster, and more intuitive than comparable base installation functions like `merge`, `order`, or `subset`.

We'll use the same fake data generated previously to illustrate filtering (subsetting rows) and selecting (subsetting columns) a data frame.

```
require(dplyr)
```

Need to subset on a couple of conditions? Use `filter`, using `&` for AND and `|` (pipe) for OR:

```
# Subset the data to only those customers who agree or strongly agree
# with the survey question
customers_marketing = filter(customer_survey, strongly_agree == 1 | agree == 1)

# Subset the data to customers who are Californians over age 64 who have spent
# at least $200 OR Oregonians of any age who have spent more than $500
customers_CA_65_200_OR_500 = filter(customer_survey, state == "California" &
customer_age >= 65 & customer_purchases >= 200 | state == "Oregon" &
customer_purchases > 500)
```

You can use regular expressions in filtering with the `grep1` function: `customers_in_lake_cities = filter(customer_survey, grep1('lake', tolower(city)))`

Need to reduce the data to just some of the variables without having to spell them all out? Use `select`, which provides some built-in SQL “like” type of access to make selection of columns from a large dataset more manageable:

```
# Subset the data to only columns with names that start with "customer"
customers_only = select(customer_survey, starts_with("customer"))

# Subset the data to only ID and columns with names that contain "agree"
customers_only_agreement = select(customer_survey, customer_id, matches("agree"))
```

`select` can also remove variables by appending a minus sign (-) to the front of the criteria:

```
# Subset the data to only columns with names that DO NOT start with "customer"
customers_excluded = select(customer_survey, -starts_with("customer"))
```

Definitely explore the vignettes in the `dplyr` package (start with `vignette("introduction", package="dplyr")`), as they provide a wide variety of examples and use cases—there's almost certainly going to be an example of what you need to do there.



RStudio has a great `dplyr` cheat sheet you can download [here³¹](#).

Creating a derived column

You can create a new column based on other columns using either `dplyr`'s `mutate` function or simply by assignment in base R. Both create a new column, but `mutate` can perform slower on large datasets and/or be more awkward to type than simply making a new column in the usual way with `$` and `=`.

Both of these commands create a new column that combine city and the state names.

```
customer_survey$Location = paste(customer_survey$city, customer_survey$stabbr,
  sep=", ")
# or
customer_survey = mutate(customer_survey, Location=paste(city, stabbr, sep=", "))
```

Of course, math operations work with both methods, e.g., to get the proportion an observation is of the whole:

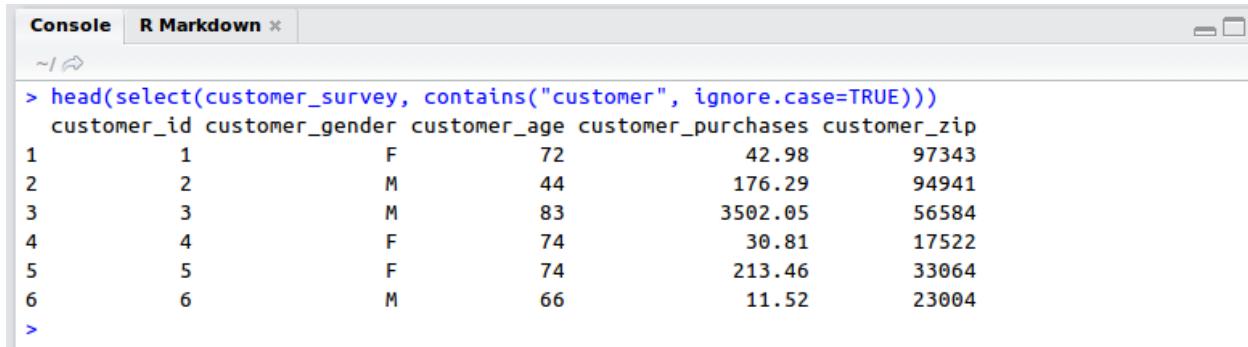
```
customer_survey$proportion = customer_survey$customer_purchases /
  sum(customer_survey$customer_purchases)
# or
customer_survey = mutate(customer_survey, proportion2 = customer_purchases /
  sum(customer_purchases))
```

Peeking at the outcome with `dplyr`

If you want to check on how one or more of the `dplyr` actions will perform in terms of output before updating or creating a dataframe, you can wrap it inside the `head` function. For example:

³¹<http://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>

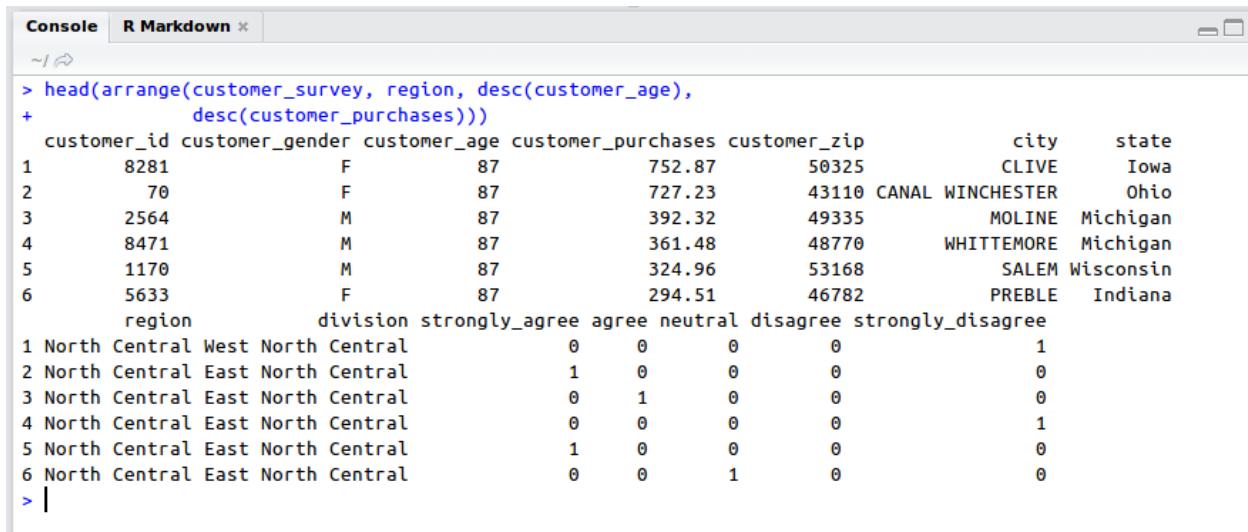
```
head(select(customer_survey, contains("customer", ignore.case=TRUE)))
```



The screenshot shows the RStudio interface with the 'Console' tab selected. The code `head(select(customer_survey, contains("customer", ignore.case=TRUE)))` is run, followed by its output. The output displays the first six rows of the dataset, which include columns: customer_id, customer_gender, customer_age, customer_purchases, and customer_zip.

	customer_id	customer_gender	customer_age	customer_purchases	customer_zip
1	1	F	72	42.98	97343
2	2	M	44	176.29	94941
3	3	M	83	3502.05	56584
4	4	F	74	30.81	17522
5	5	F	74	213.46	33064
6	6	M	66	11.52	23004

```
head(arrange(customer_survey, region, desc(customer_age),
desc(customer_purchases)))
```



The screenshot shows the RStudio interface with the 'Console' tab selected. The code `head(arrange(customer_survey, region, desc(customer_age), desc(customer_purchases)))` is run, followed by its output. The output displays the first six rows of the dataset, which include columns: customer_id, customer_gender, customer_age, customer_purchases, customer_zip, city, state, region, division, strongly_agree, agree, neutral, disagree, and strongly_disagree.

	customer_id	customer_gender	customer_age	customer_purchases	customer_zip	city	state	region	division	strongly_agree	agree	neutral	disagree	strongly_disagree
1	8281	F	87	752.87	50325	CLIVE	Iowa	North	Central	0	0	0	0	1
2	70	F	87	727.23	43110	CANAL WINCHESTER	Ohio	North	Central	1	0	0	0	0
3	2564	M	87	392.32	49335	MOLINE	Michigan	North	Central	0	1	0	0	0
4	8471	M	87	361.48	48770	WHITTEMORE	Michigan	North	Central	0	0	0	0	1
5	1170	M	87	324.96	53168	SALEM	Wisconsin	North	Central	1	0	0	0	0
6	5633	F	87	294.51	46782	PREBLE	Indiana	North	Central	0	0	1	0	0

```
head(mutate(customer_survey, Location = paste(city, state, sep=", "),
purchase_proportion=customer_purchases/sum(customer_purchases)))
```

Reshaping a dataframe between wide and long

A lot of analysis and plotting functions require that data be organized in *long* format, where the variable of interest is in one column and the id/categorical portions of the data are repeated as necessary in other columns. Then again, other tools need the data in *wide* format, so being able to move between the two formats is essential. The `reshape2` package provides just that ability.

Reshaping: melt and cast

We'll use a small dataset to illustrate some of the possibilities `reshape2` offers—this shows the verbal and nonverbal IQ test scores for children before and after neurosurgery (and in which hemisphere, i.e. “Side” of the brain; data from *Table 2* in [this paper³²](#)). Load the package and the data:

```
require(reshape2)

surgery_outcomes = read.table("https://raw.githubusercontent.com/Rmadillo/
  business_intelligence_with_r/master/manuscript/code/surgery_outcomes.tsv",
  header=T, sep="\t")
```

³²<http://thejns.org/doi/10.3171/2015.3.PEDS14359>

```
head(surgery_outcomes, 4)
```

row	ID	Side	Phase	Verbal	Nonverbal
1	1	Right	Post	146	115
2	1	Right	Pre	129	103
3	2	Right	Post	126	115
4	2	Right	Pre	138	115

As is, this data is both wide and long. If you're only interested in looking at either Verbal score or Nonverbal scores, it's long, as the other variables are already long. If you want to look at Verbal and Nonverbal together, it's wide.

Suppose each measurement needs its own row, that is, we need to make this a long data frame so we can compare Verbal and Nonverbal scores. We need to contain those two scores within a single column, such as IQ_score. To do this, you melt a dataframe from wide into long format:

```
surgery_outcomes_melted = melt(surgery_outcomes, id.vars=c(1:4),
                                measure.vars=c(5:6), variable.name="Test_type", value.name="IQ_score")
```

```
head(surgery_outcomes_melted, 4)
```

row	ID	Side	Phase	Test_type	IQ_score
1	1	Right	Post	Verbal	146
2	1	Right	Pre	Verbal	129
3	2	Right	Post	Verbal	126
4	2	Right	Pre	Verbal	138

We now have a completely long form data set, as we have the primary variable of analytic interest contained in a single column.

Any data in long form can be cast into a variety of wide forms, with a d or an a tacked on to the front depending on whether you need a dataframe (d), or a vector, matrix, or array (a). To put this data back into the wide/long shape it started as, use dcast:

```
surgery_outcomes_original = dcast(surgery_outcomes_melted,
                                    ID + Side + Phase ~ Test_type, value.var="IQ_score")
```

```
head(surgery_outcomes_original, 4)
```

row	ID	Side	Phase	Verbal	Nonverbal
1	1	Right	Post	146	115
2	1	Right	Pre	129	103
3	2	Right	Post	126	115
4	2	Right	Pre	138	115

Being able to move back and forth between data shapes makes meeting the needs of the R packages you're working with considerably easier, especially if you're moving between some that work best with long form structures and others that work well with wide.

The important part to remember in melting data is to ensure you've designated the id and measurement variables appropriately. While `melt` can operate with neither or only one of these designated, proper coding etiquette requires that you designate each specifically to avoid ambiguity.



The `tidy` package is a new, simpler way to reshape data. At the time of this writing, it can accomplish any simple `melt/cast` (called `gather/spread`) task. However, more complicated data set transformations can fail in `tidy`; the current consensus is to keep `reshape2` in the toolbox until `tidy` is mature enough to accomplish the full suite of transformation needs. A useful cheat sheet on `tidy` is co-packaged with `dplyr` by RStudio [here](#)³³, and a more complete treatment is available [here](#)³⁴ or by checking out the vignette: `vignette("tidy-data")`.

Reshaping: Crossing variables with ~ and +

There is tremendous variety in reshaping options using the formula structure in `dcast`. The tilde sign (~) separates the variables you want representing the rows (left of the ~) and those you want representing the columns (right of the ~). For example, we can cross phase and the test type in the columns:

```
head(dcast(surgery_outcomes_melted, ID ~ Test_type + Phase,
           value.var="IQ_score"), 4)
```

	ID	Verbal_Post	Verbal_Pre	Nonverbal_Post	Nonverbal_Pre
1	1	146	129	115	103
2	2	126	138	115	115
3	3	110	100	104	100
4	4	104	NA	106	NA

We could also bring back the Side variable into the same wide shape:

³³<http://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>

³⁴<http://vita.had.co.nz/papers/tidy-data.html>

```
head(dcast(surgery_outcomes_melted, ID + Side ~ Test_type + Phase,
           value.var='IQ_score'), 4)
```

	ID	Side	Verbal_Post	Verbal_Pre	Nonverbal_Post	Nonverbal_Pre
1	1	Right	146	129	115	103
2	2	Right	126	138	115	115
3	3	Left	110	100	104	100
4	4	Right	104	NA	106	NA

Perhaps you only want to use Phase to look at Verbal and Nonverbal separately:

```
head(dcast(surgery_outcomes_melted, ID + Phase ~ Test_type,
           value.var='IQ_score'), 4)
```

	ID	Phase	Verbal	Nonverbal
1	1	Post	146	115
2	1	Pre	129	103
3	2	Post	126	115
4	2	Pre	138	115

And so on... the possibilities are as many as your variable set...

Summarizing while reshaping

While dplyr provides you with all you need for summarizing data once your data structure is set, reshape2 offers the option to summarize while reshaping. The summarization works based on which operation you choose as well as which side of the ~ you place the variables (as described above).

Get the mean for each Test_type, by Phase:

```
dcast(surgery_outcomes_melted, Phase ~ Test_type,
      value.var='IQ_score', mean, na.rm=T)
```

	Phase	Verbal	Nonverbal
1	Post	104.6667	102.26667
2	Pre	103.0909	98.90909

Get the mean for each Side, by combination of Test_type and Phase:

```
dcast(surgery_outcomes_melted, Side ~ Test_type + Phase,
      value.var='IQ_score', mean, na.rm=T)

  Side Verbal_Post Verbal_Pre Nonverbal_Post Nonverbal_Pre
1 Left    102.4444   97.71429     100.7778     97.42857
2 Right   108.0000  112.50000     104.5000    101.50000
```

Get the standard deviation for each combination of Test_type, Side, and Phase:

```
dcast(surgery_outcomes_melted, Test_type + Side + Phase ~ . ,
      value.var='IQ_score', sd, na.rm=T)

  Test_type Side Phase .
1   Verbal   Left Post 14.78269
2   Verbal   Left Pre 12.51285
3   Verbal  Right Post 23.80756
4   Verbal  Right Pre 24.93324
5 Nonverbal  Left Post 16.09952
6 Nonverbal  Left Pre 13.96253
7 Nonverbal Right Post 11.46734
8 Nonverbal Right Pre 10.47219
```

Piping with %>%: Stringing it together in dplyr

A useful additional feature of dplyr is that you can string together a bunch of functions with the %>% (“then”) operator, a feature that is growing in popularity across a variety of R packages. So, if you want to filter, group, do some summary stats, create a new column, etc., you can pipe them all together instead of nesting them in parentheses or over multiple lines.

```
# For customers who've purchased more than $500, get count, mean,
# sd, and sum for each state, and calculate the coefficient of
# variation, rounded to 2 decimal places
customer_500_piped = customer_survey %>%
  filter(customer_purchases >= 500) %>%
  group_by(state) %>%
  summarize(count_purchases = n(),
           mean_purchases = mean(customer_purchases, na.rm=T),
           sd_purchases = sd(customer_purchases, na.rm=T),
           sum_purchases = sum(customer_purchases)) %>%
  mutate(cv_purchases = round(sd_purchases / mean_purchases, 2))
```

```
head(customer_500_piped, 4)
```

Source: local **data frame** [4 x 6]

	state	count_purchases	mean_purchases	sd_purchases	sum_purchases	cv_purchases
	(chr)	(int)	(dbl)	(dbl)	(dbl)	(dbl)
1	Alabama	7	879.2857	301.6824	6155.00	0.34
2	Alaska	4	928.5350	234.7453	3714.14	0.25
3	Arizona	8	707.6425	186.9487	5661.14	0.26
4	Arkansas	5	767.0300	124.2218	3835.15	0.16

This is also a good example where `mutate` is more convenient than traditional assignment.

However, not everyone is enamored with this new “piping” trend; when you need to provide verbose code or examples for beginners to understand the processing steps it **may not be the best coding choice**³⁵. As with all “coding standards” advice, do as you think best for you and your context.

³⁵<http://www.fromthebottomoftheheap.net/2015/06/03/my-aversion-to-pipes/>

Chapter 4: Know Thy Data—Exploratory Data Analysis

- Creating summary plots
- Plotting univariate distributions
- Plotting bivariate and comparative distributions
- Plotting survey data
- Obtaining summary and conditional statistics
- Inference on summary statistics
- Dealing with missing data

Exploration and evaluation is the heart of analytics; this chapter covers a wide variety of ways to understand, display, and summarize your data.

`ggplot2` is perhaps the most useful (and logical) graphics package for R, and we'll use it extensively in this book. The `scales` package is often loaded with it, to provide support for more specialized axes, such as for dates and times.

```
require(ggplot2)
require(scales)
```

`ggplot2` can seem daunting at first; the power of its immense flexibility and beauty comes at the cost of a bewildering variety of options. But it's well worth the effort to learn, as you'll be able to customize virtually anything you decide to plot.



Need a `ggplot2` cheat sheet?

RStudio has a great A3/11x17 sized overview of all the primary `ggplot` functions and options³⁶ on their cheatsheet page³⁷.

The `ggplot2` documentation³⁸ is very thorough, but if you need a cheat-sheet to simplify the possibilities down to some common tasks, there's a great introduction on Computer World³⁹. For an overview of common `ggplot2` tasks with worked examples, `Cookbook R`⁴⁰ is quite useful, and Zev Ross has a more thorough but also well done overview on his blog⁴¹.

³⁶<http://www.rstudio.com/wp-content/uploads/2015/12/ggplot2-cheatsheet-2.0.pdf>

³⁷<https://www.rstudio.com/resources/cheatsheets/>

³⁸<http://docs.ggplot2.org/current/>

³⁹<http://www.computerworld.com/article/2935394/business-intelligence/my-ggplot2-cheat-sheet-search-by-task.html>

⁴⁰<http://www.cookbook-r.com/Graphs/>

⁴¹<http://zevross.com/blog/2014/08/04/beautiful-plotting-in-r-a-ggplot2-cheatsheet-3/>

We'll primarily use the bike share data we downloaded from the UCI Machine Learning Repository as one of the examples in Chapter 2. To make it easy, the code below downloads and imports the data we'll use throughout much of this chapter (as well as the rest of the book) and adjusts the column types and levels.

```
download.file("http://archive.ics.uci.edu/ml/machine-learning-databases/
 00275/Bike-Sharing-Dataset.zip", "Bike-Sharing-Dataset.zip")

bike_share_daily = read.table(unz("Bike-Sharing-Dataset.zip", "day.csv"),
  colClasses=c("character", "Date", "factor", "factor", "factor", "factor",
  "factor", "factor", "factor", "numeric", "numeric", "numeric", "numeric",
  "integer", "integer", "integer"), sep=",", header=TRUE)

levels(bike_share_daily$season) = c("Winter", "Spring", "Summer", "Fall")
levels(bike_share_daily$workingday) = c("No", "Yes")
levels(bike_share_daily$holiday) = c("No", "Yes")
bike_share_daily$mnth = ordered(bike_share_daily$mnth, 1:12)
levels(bike_share_daily$mnth) = c(month.abb)
levels(bike_share_daily$yr) = c(2011, 2012)
```

Creating summary plots

If you were to poll data scientists on the “Three Most Important Steps in Any Analysis,” *Plot Thy Data* should account for all three. Far too many business professionals rely on tables and simple summary statistics to make decisions, which are useful but can never provide the richness of information that graphs can. Further, decisions based on single statistic (usually the mean) are made much too often, usually with either good results reached by luck or accident, or—more often—they end up with (predictably) poor results.

The recipes immediately below are aimed at giving you a quick visual summary of the data, not to produce final plots—and so they’re rough and use functions that can be difficult to modify. We’ll see recipes for production plots later in this chapter and throughout the book, but we often just need to see the data before doing anything else.

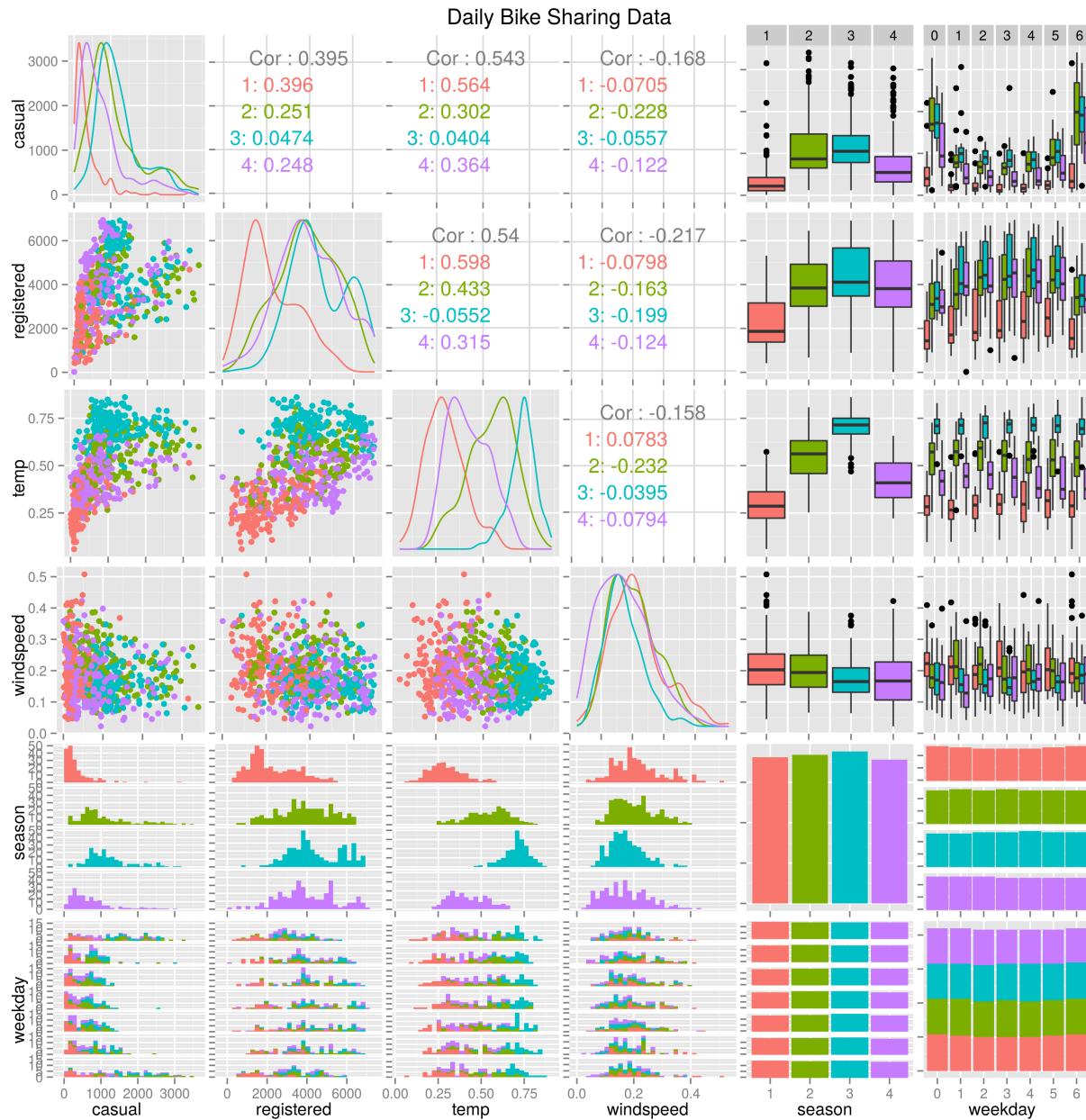
Everything at once: `ggpairs`

The `ggpairs` function in the `GGally` package has a relatively slow but visually wonderful way to plot your dataset, so load that library before continuing.

```
require(GGally)
```

`GGally` is built over `ggplot2`, and its syntax works in more or less the same manner as `qplot`. You can create a summary plot of a subset of the bike share data for an at-a-glance overview like this:

```
ggpairs(data=bike_share_daily, columns=c(14:15, 10, 13, 3, 7),
  title="Daily Bike Sharing Data", axisLabels="show",
  mapping=aes(color=season, alpha=0.3))
```



It's helpful to use a non-parametric correlation method if you don't know *a priori* whether the quantitative variables are linearly related in their raw forms. Unfortunately, `ggpairs` doesn't yet provide this functionality, so know that for continuous pairs of variables that don't show a linear point cloud, the correlation coefficients printed in this graph will be larger than they actually are.

`GGally` is still in active development, so currently it's pretty difficult to customize the `ggpairs` plot,

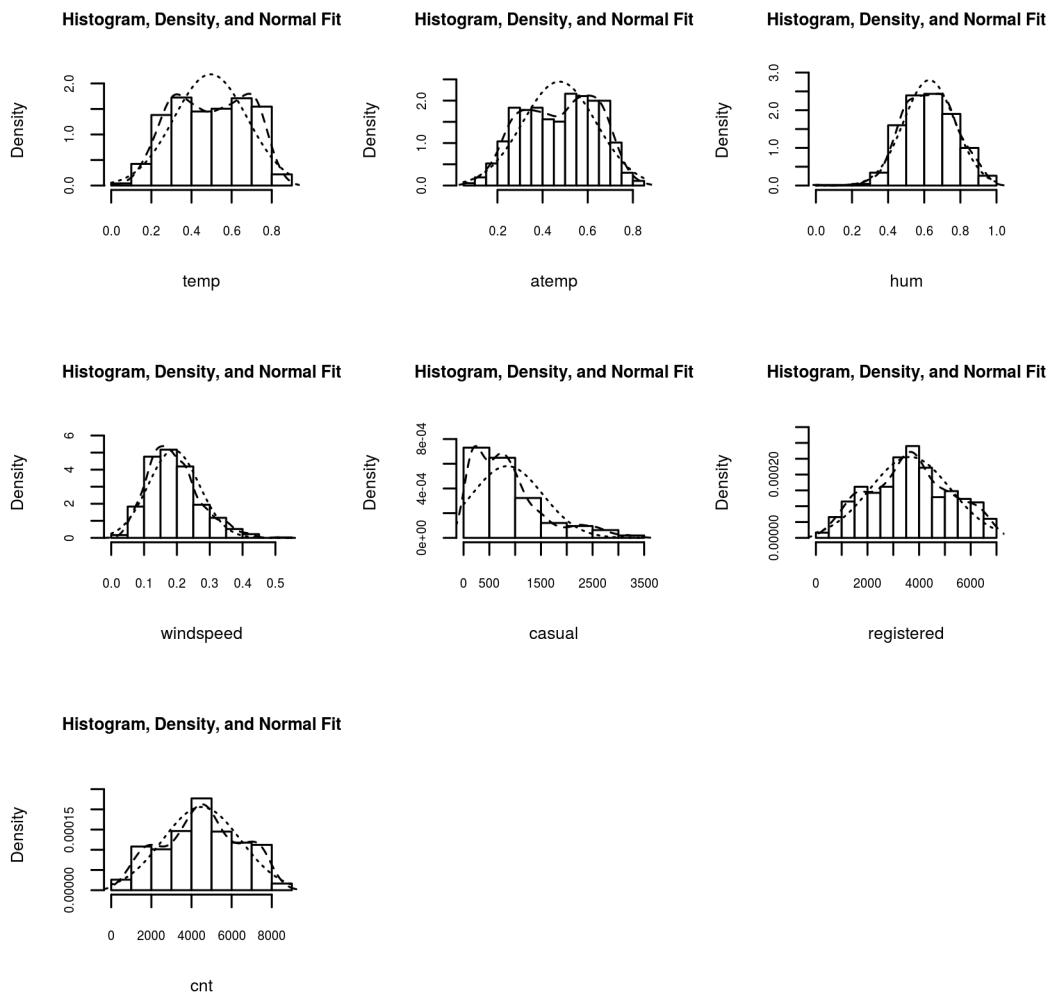
and it may rarely be useful to anyone other than you. Even then, it's a great way to get a sense of the data at a glance.

Create histograms of all numeric variables in one plot

Sometimes you just want to see histograms, but coding and laying out a bunch of them in a grid is more work than it needs to be. The psych package's `multi.hist` function can do this in a single line. Note that if you specify variables that aren't numeric, it will throw an error.

```
require(psych)

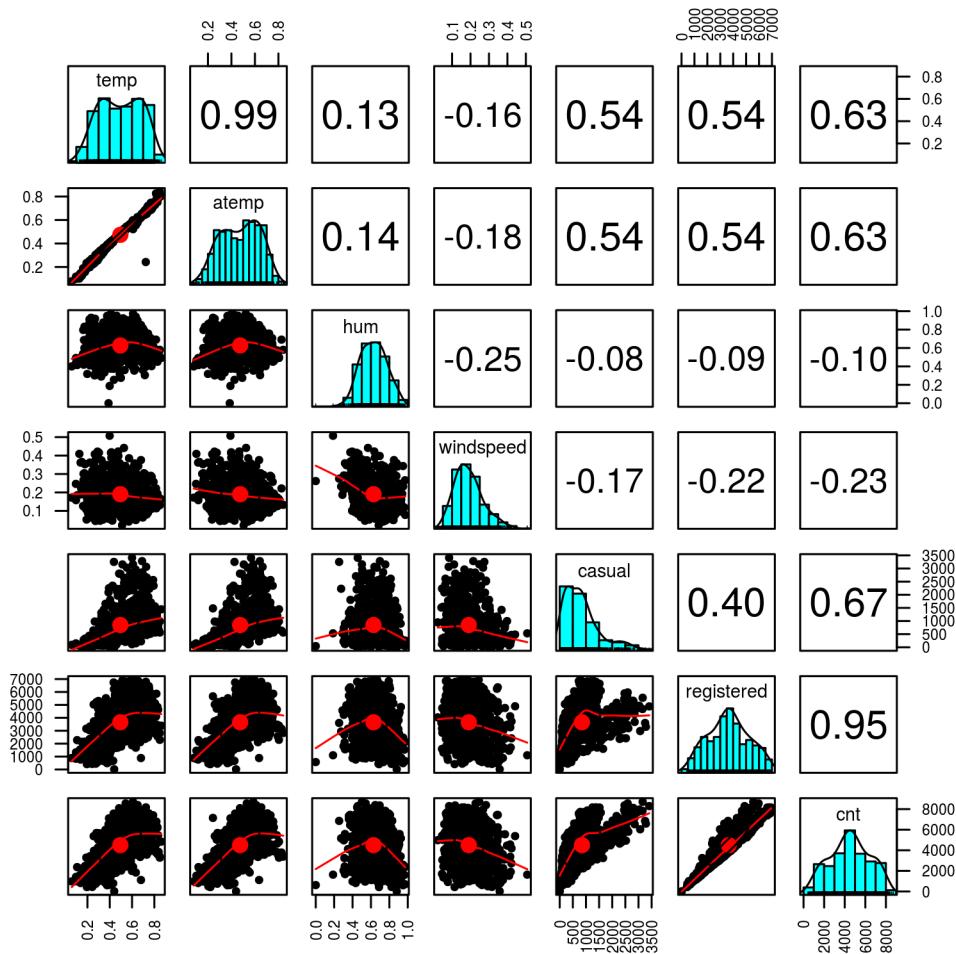
multi.hist(bike_share_daily[, sapply(bike_share_daily, is.numeric)])
```



A better “pairs” plot

The `psych` package also has a modification of the built-in `graphics` package’s `pairs` function that’s more useful out of the box. As mentioned above, until you have evidence that your paired variables are indeed linear in raw form, you should modify the default calculation option to `method="spearman"` for continuous data or `method="kendall"` for ordinal data or continuous data in which you have some known outliers:

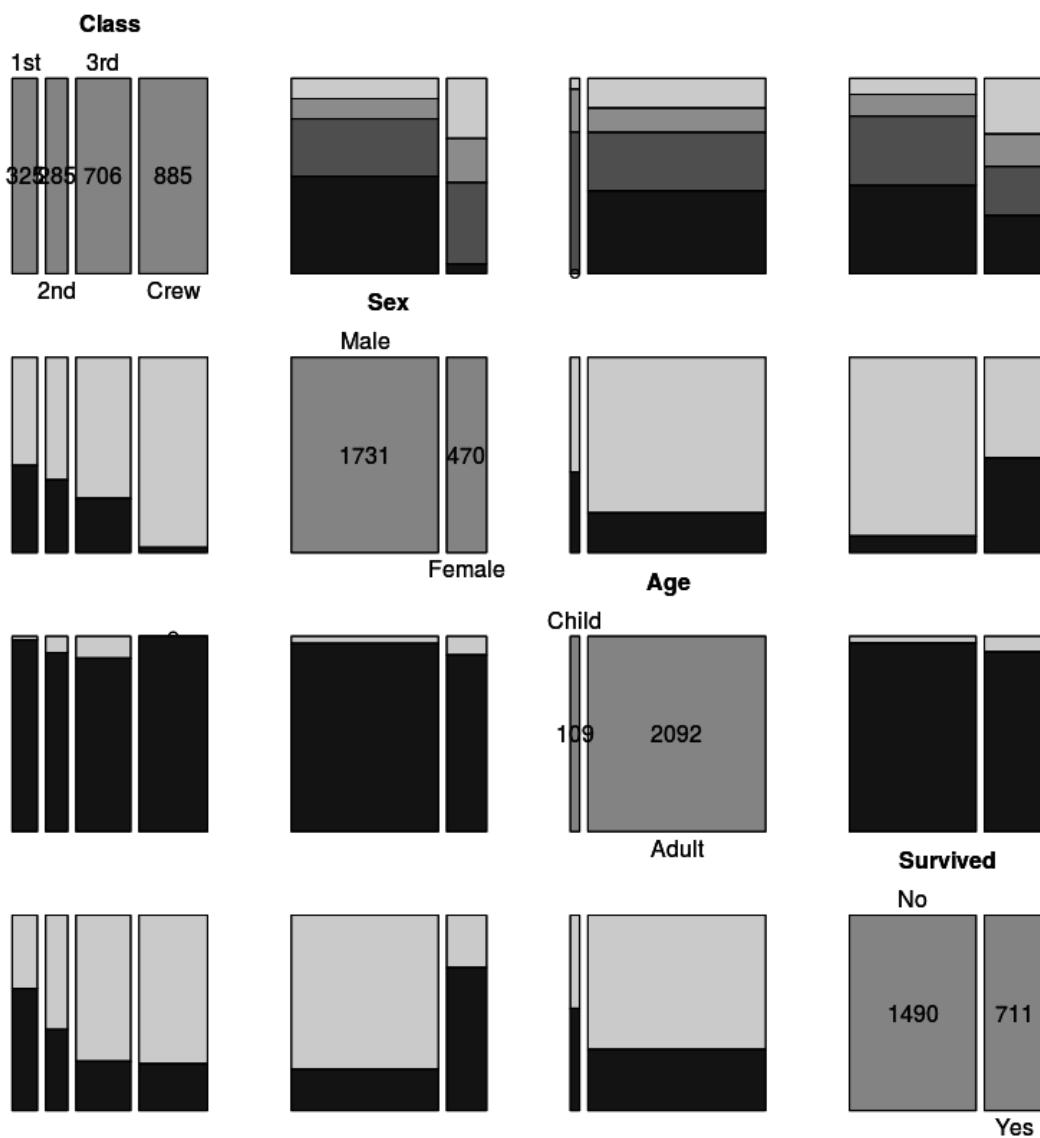
```
pairs.panels(bike_share_daily[, sapply(bike_share_daily, is.numeric)],  
            ellipses=FALSE, pch=". .", las=2, cex.axis=0.7, method="kendall")
```



Mosaic plot matrix: “Scatterplot” matrix for categorical data

The vcd package can extend the pairs function to create a matrix of mosaic plots, either by using the table function on a dataframe, or by referencing a table object. The dataset we’ve used so far isn’t a great one for mosaic plots, so a better example can be seen with the built-in Titanic dataset (?Titanic), using the highlighting option to make the categories more clearly distinct:

```
require(vcd)
pairs(Titanic, highlighting=2)
```



Plotting univariate distributions

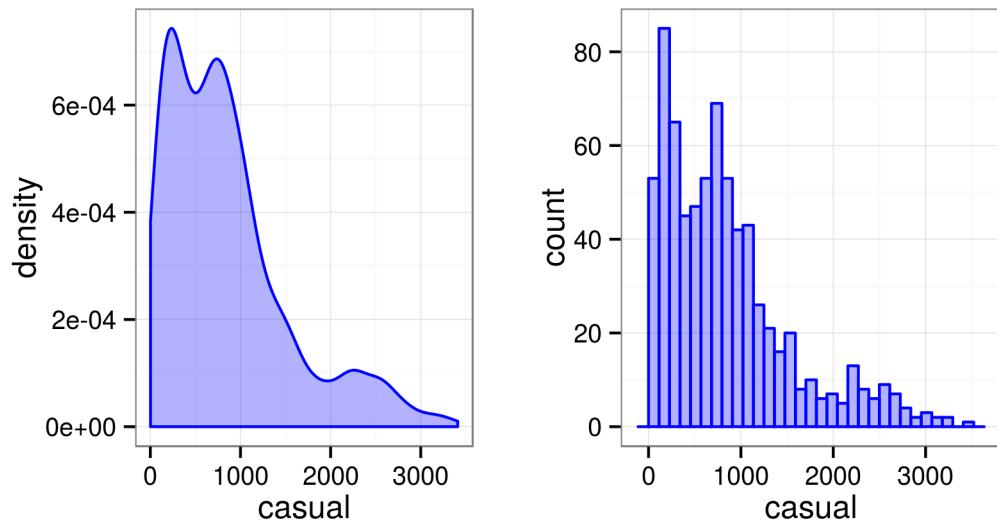
We saw the quick-and-dirty way to plot a bunch of distributions at once above; now we focus on production-ready plots using the `ggplot2` package.

Histograms and density plots

For quantitative data, business users are used to seeing histograms; quants prefer density plots. Here's how to do both, separately...

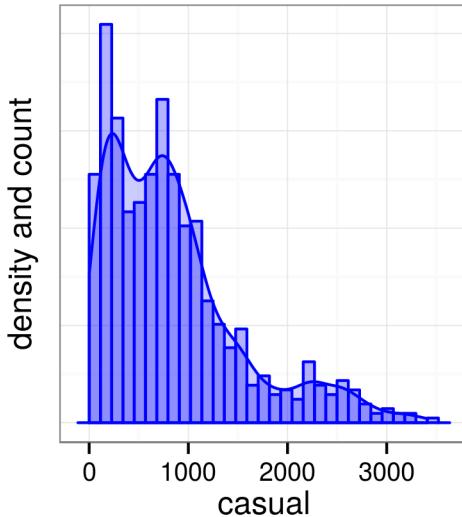
```
ggplot(bike_share_daily, aes(casual)) +
  geom_density(col="blue", fill="blue", alpha=0.3) +
  xlab("Casual Use") +
  theme_bw()

ggplot(bike_share_daily, aes(casual)) +
  geom_histogram(col="blue", fill="blue", alpha=0.3) +
  xlab("Casual Use") +
  theme_bw()
```



...as well as together, with the histogram command set to plot on the density scale instead of by counts:

```
ggplot(bike_share_daily, aes(casual)) +  
  ylab("density and count") +  
  xlab("Casual Use") +  
  geom_histogram(aes(y=..density..), col="blue", fill="blue", alpha=0.3) +  
  geom_density(col="blue", fill="blue", alpha=0.2) +  
  theme_bw() +  
  theme(axis.ticks.y = element_blank(), axis.text.y = element_blank())
```



When asked, I've usually explained density plots as “more objective histograms,” which, while not quite true, is accurate enough for business users. See `?density` for more details on the methods and useful additional references; `?stat_density` provides an overview of calculation methods used in `ggplot2`, while `?geom_density` provides information on graphical options.



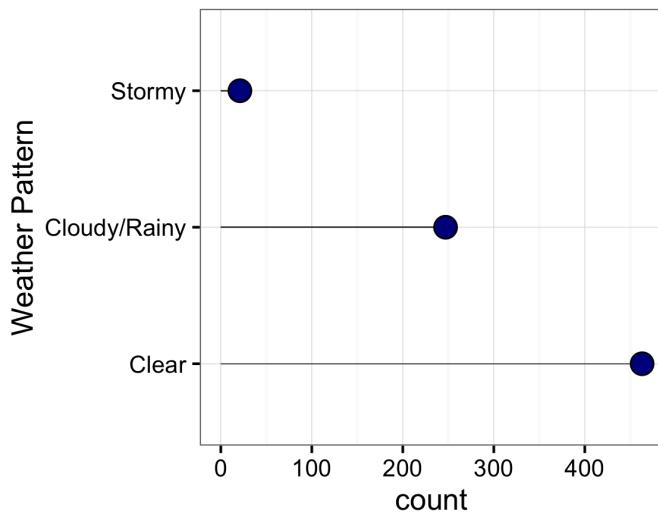
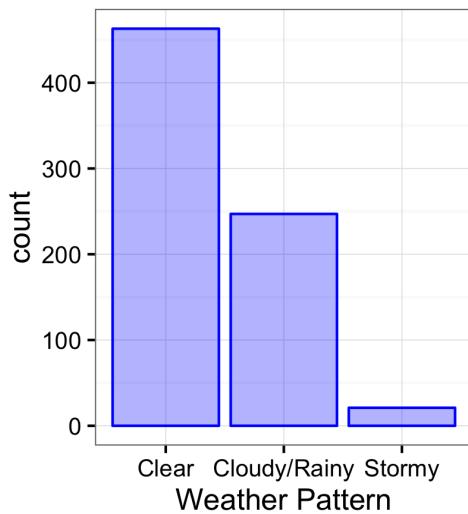
That pattern carries over to many `ggplot2` options—find details of the method by adding the particular stat or geom in `?stat_*` and `?geom_*` for calculation and plotting information, respectively.

Bar and dot plots

When you have categorical data, bar or dot plots serve the same purpose:

```
ggplot(bike_share_daily, aes(weathersit)) +
  geom_bar(col="blue", fill="blue", alpha=0.3) +
  xlab("Weather Pattern") +
  scale_x_discrete(breaks=c(1, 2, 3), labels=c("Clear",
    "Cloudy/Rainy", "Stormy")) +
  theme_bw()

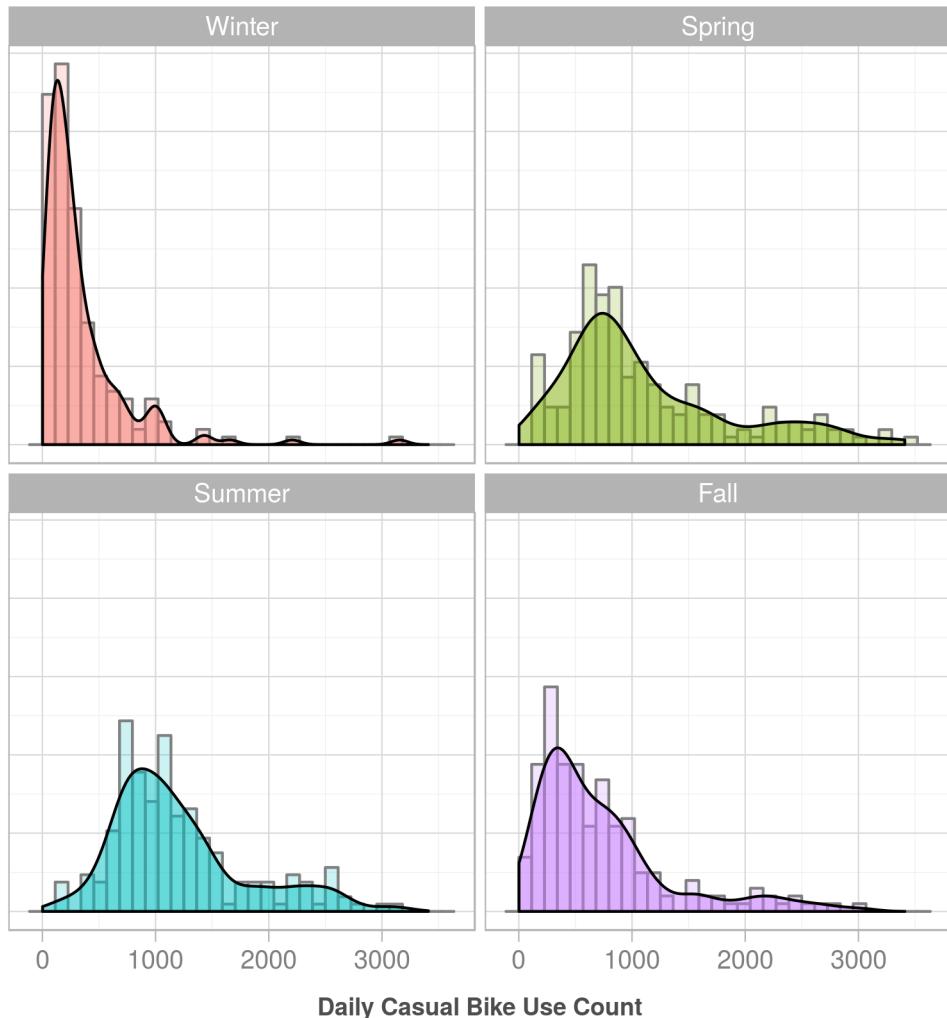
ggplot(bike_share_daily, aes(x=weathersit, y=..count..)) +
  geom_bar(stat="count", width=0.01) +
  geom_point(stat = "count", size=4, pch=21, fill="darkblue") +
  xlab("Weather Pattern") +
  scale_x_discrete(breaks=c(1, 2, 3), labels=c("Clear",
    "Cloudy/Rainy", "Stormy")) +
  coord_flip() +
  theme_bw()
```



Plotting multiple univariate distributions with faceting

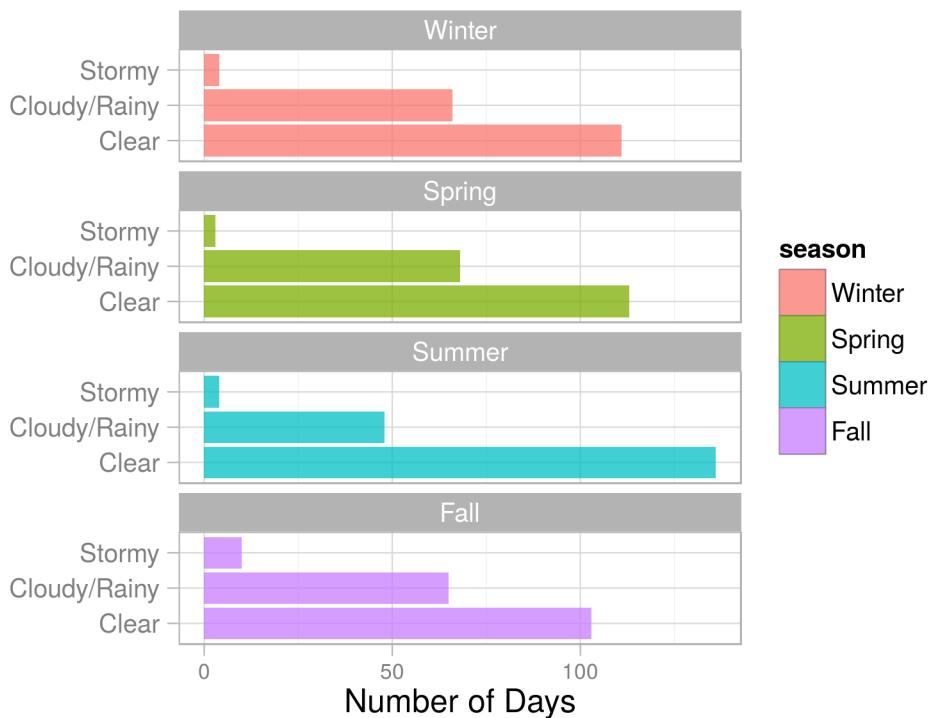
We saw the quick-and-dirty way to plot multiple histograms with `multi.hist` a few recipes ago; here's how to use `ggplot2` to create the same thing more elegantly:

```
ggplot(bike_share_daily, aes(casual, fill=season)) +  
  geom_histogram(aes(y = ..density..), alpha=0.2, color="gray50") +  
  geom_density(alpha=0.5, size=0.5) +  
  facet_wrap(~season) +  
  theme_light() +  
  xlab("Daily Bike Use Count") +  
  ylab("") +  
  theme(legend.position="none") +  
  theme(axis.ticks.y = element_blank(), axis.text.y = element_blank(),  
        axis.title = element_text(size=9, face=2, color="gray30"),  
        axis.title.x = element_text(vjust=-0.5))
```



The same can be done for categorical distributions, too, of course:

```
ggplot(bike_share_daily, aes(weather_sit, fill=season)) +
  geom_bar(alpha=0.5) +
  xlab("") +
  ylab("Number of Days") +
  scale_x_discrete(breaks=c(1, 2, 3), labels=c("Clear",
    "Cloudy/Rainy", "Stormy")) +
  coord_flip() +
  facet_wrap(~season, ncol=1) +
  theme_light()
```



What about pie charts?

With apologies to Hunter S. Thompson, if you use pie charts for anything outside of GraphJam, you deserve whatever happens to you.

Plotting bivariate and comparative distributions

Most of the time, we want to understand how one thing compares to something else. There's no better way to do this other than by plotting each group's distribution against the other's.

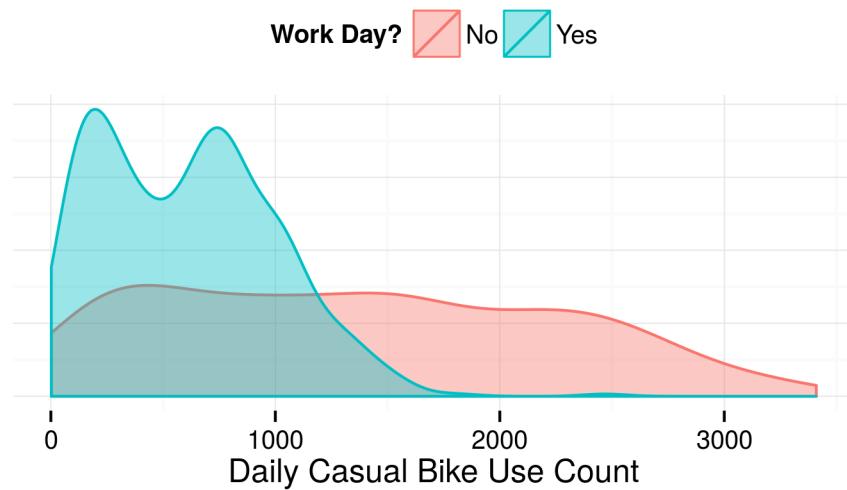
We continue using the bike share data, along with `ggExtra`, `vcd`, and `beanplot`.

```
require(ggExtra)
require(vcd)
require(beanplot)
```

Double density plots

For quantitative data, plotting the two distributions over each other provides a clear, direct comparison:

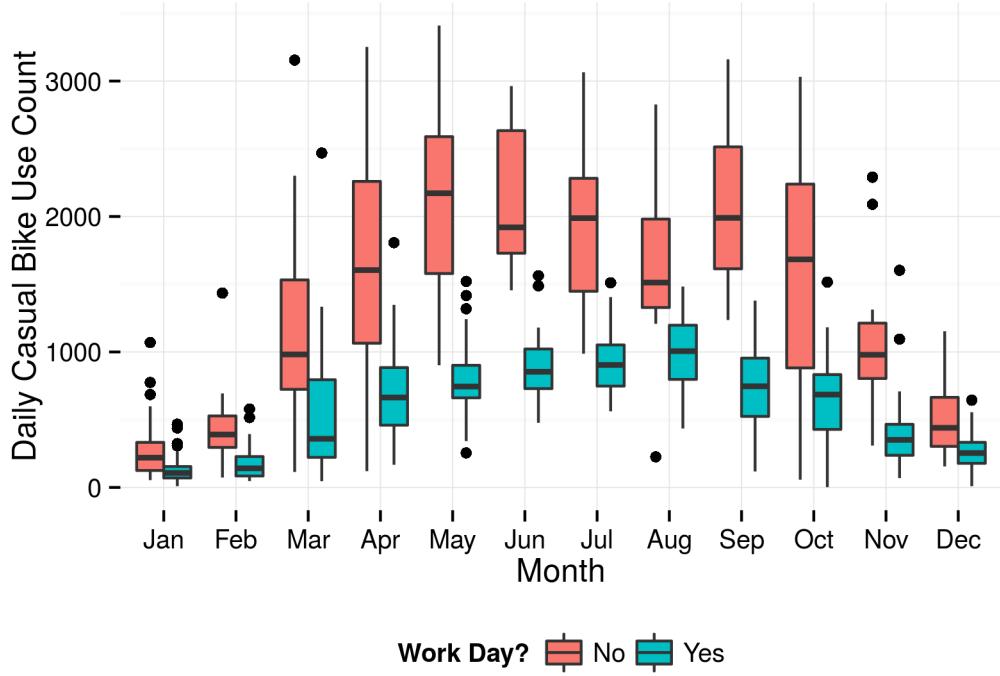
```
ggplot(bike_share_daily, aes(casual, fill=workingday, color=workingday)) +
  geom_density(alpha=0.4) +
  theme_minimal() +
  xlab("Daily Casual Bike Use Count") +
  ylab("") +
  scale_fill_discrete(name="Work Day?") +
  scale_color_discrete(name="Work Day?") +
  theme(axis.ticks.y = element_blank(), axis.text.y = element_blank(),
        legend.position="top")
```



Boxplots

Boxplots can be useful for comparing more than a couple of distributions, and can do so using more than one factor:

```
ggplot(bike_share_daily, aes(mnth, casual, fill=workingday)) +
  xlab("Month") +
  ylab("Daily Casual Bike Use Count") +
  geom_boxplot() +
  theme_minimal() +
  scale_fill_discrete(name="Work Day?") +
  scale_color_discrete(name="Work Day?") +
  theme(legend.position="bottom")
```

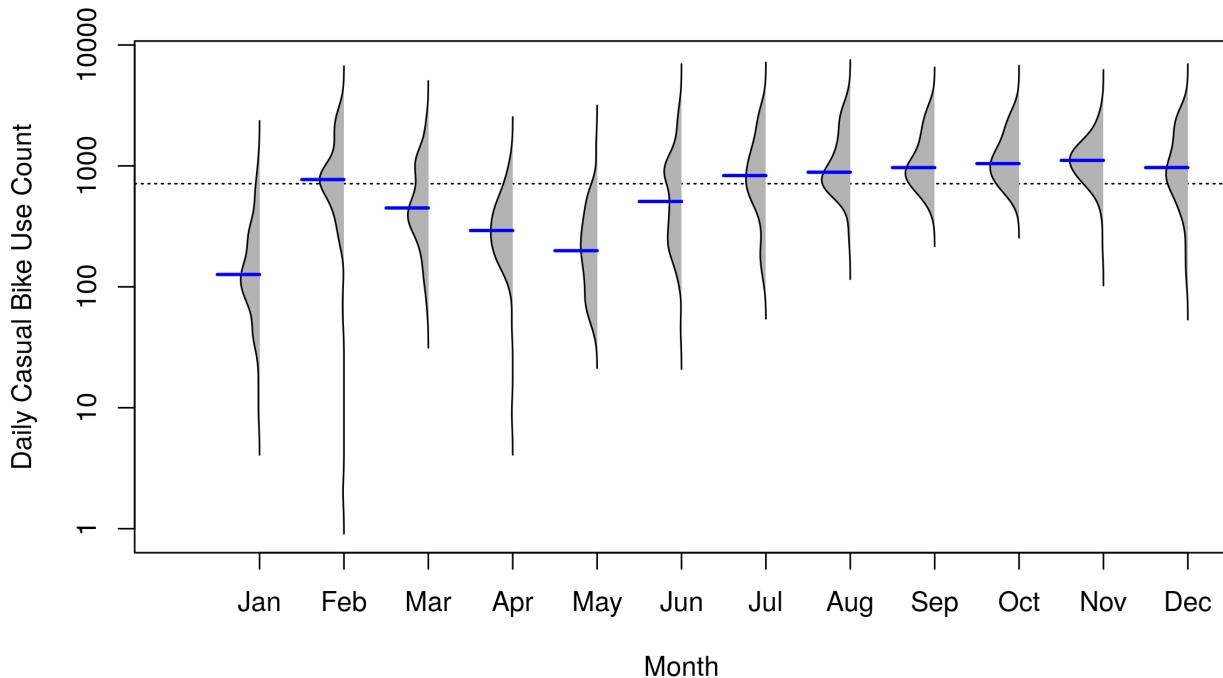


Beanplots

Violin plots have been considered a more useful alternative to boxplots, as you can see the shape of the distribution, not just its major quantiles. Unfortunately, violin plots can sometimes look like a series of genitalia, which pretty much precludes its use in business settings.

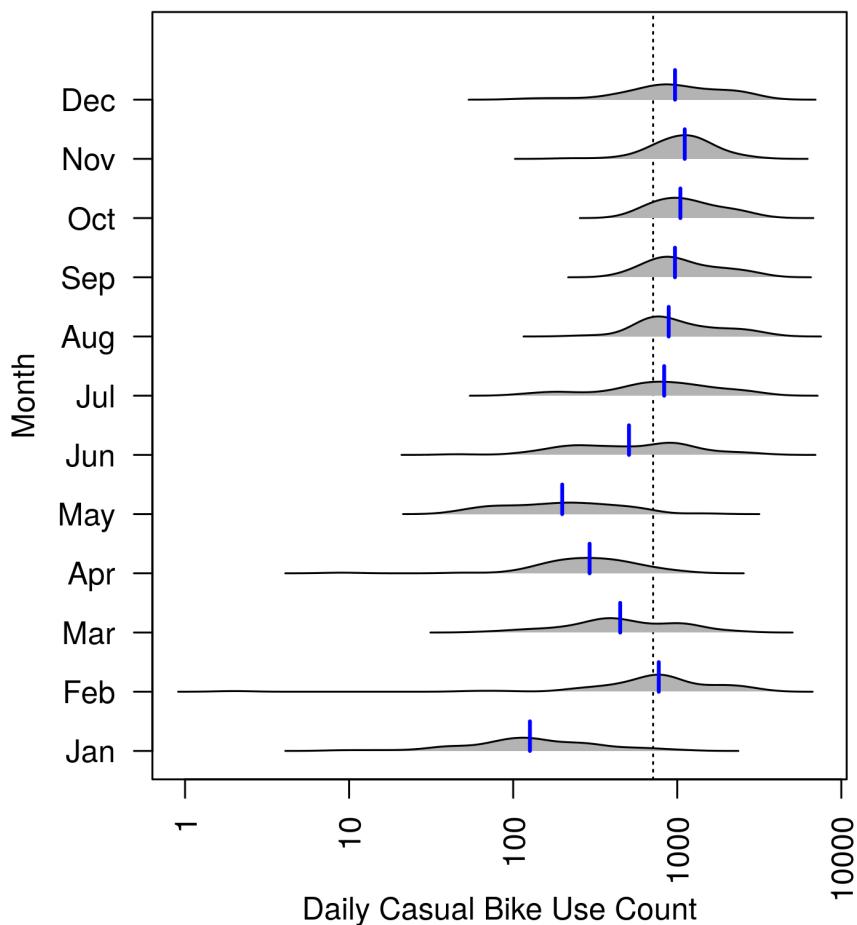
Fortunately, bean plots allow you to create “half” violin plots, or more accurately, comparative density histograms. The `side` option allows you to only plot the density on one side of a line; using `side="first"` has the height of each curve going to the left side of the plot. By playing with the `what` options, you can determine what pieces of the bean plot are shown. One of those pieces is a reference line, which defaults to the mean; `overallline="median"` allows you to use that statistic instead.

```
beanplot(casual ~ mnth, data = bike_share_daily, side="first",
overallline="median", what=c(1,1,1,0), col=c("gray70",
"transparent", "transparent", "blue"), xlab = "Month",
ylab = "Daily Casual Bike Use Count")
```



If you prefer a more traditional look, with the density curves pointing up, you can use `horizontal=TRUE`, change `side` to "second", and flip your axis titles:

```
beanplot(casual ~ mnth, data = bike_share_daily, side="second",
overallline="median", what=c(1,1,1,0), col=c("gray70",
"transparent", "transparent", "blue"), ylab = "Month",
xlab = "Daily Casual Bike Use Count", horizontal=TRUE)
```



Like the boxplot example above, you can also add a comparison for groups within a category, and each group's density is plotted base-to-base. So while you don't have the exact symmetry of violin plots, the result can provide much the same visual impression, so it's probably best to stick with the use of one side of the "bean."

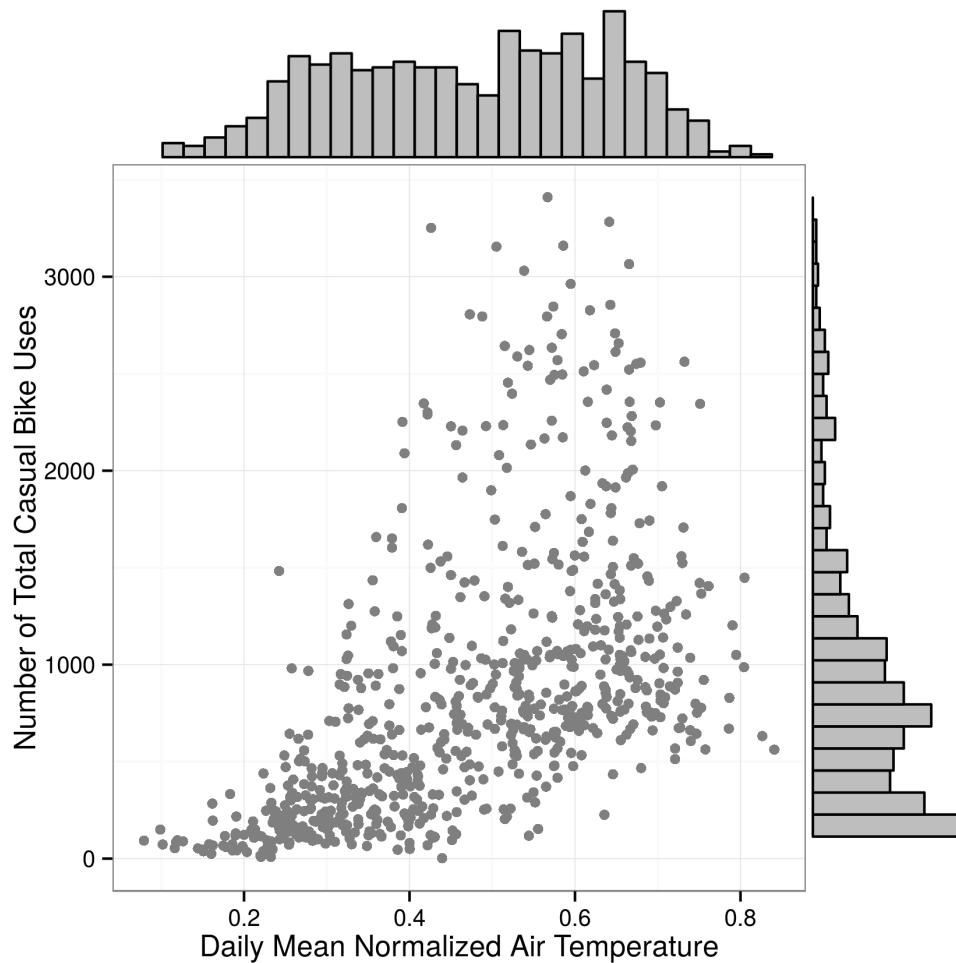
Scatterplots and marginal distributions

The `ggExtra` package provides a simple way to plot scatterplots with marginal distributions with the `ggMarginal` function. It defaults to density curves, but the `histogram` option presents a clearer picture. First, build the scatterplot and assign it to an object.

```
bike_air_temp = ggplot(bike_share_daily, aes(x=atemp, y=casual)) +
  xlab("Daily Mean Normalized Air Temperature") +
  ylab("Number of Total Casual Bike Uses") +
  geom_point(col="gray50") +
  theme_bw()
```

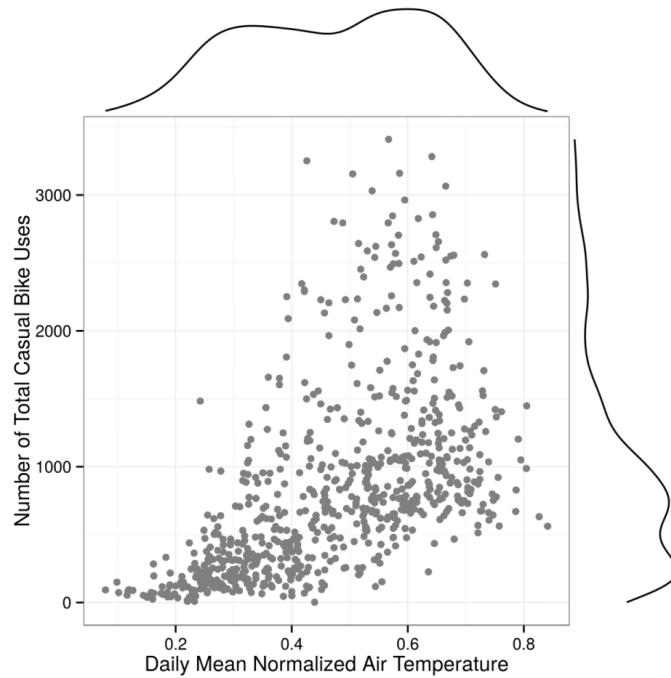
Then, use `ggMarginal` to create the final product:

```
bike_air_temp_mh = ggMarginal(bike_air_temp, type="histogram")
bike_air_temp_mh
```

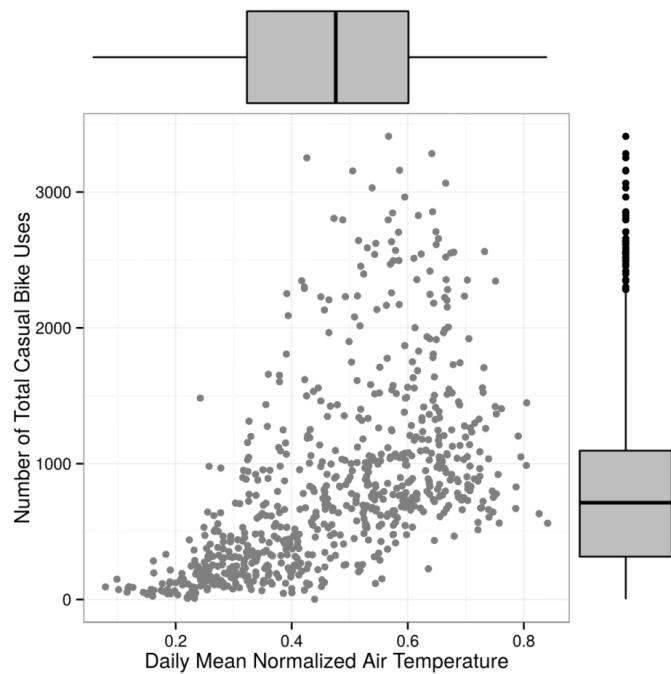


Sometimes density plots (the default) can be useful, or perhaps you want to keep it simple and show a boxplot. Just change the `type` option:

```
ggMarginal(bike_air_temp, type="density")
```



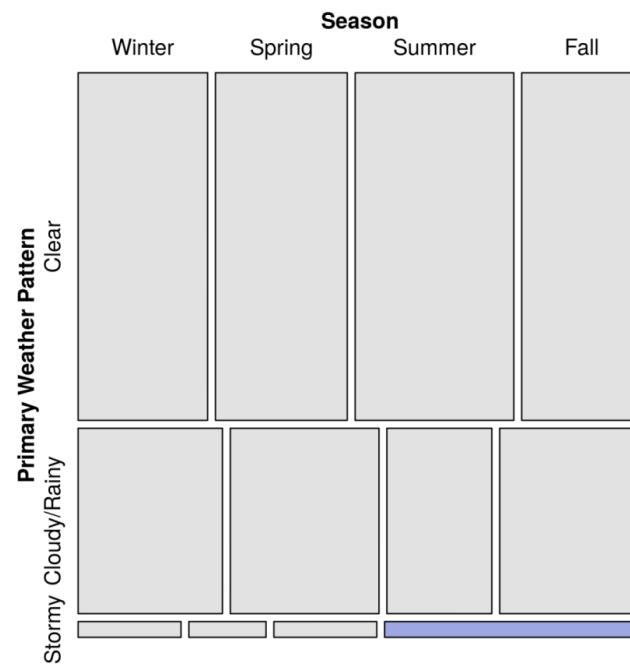
```
ggMarginal(bike_air_temp, type="boxplot")
```



Mosaic plots

The `vcd` package has a nice implementation of mosaic plots for exploring categorical data distributions called, as you might expect, `mosaic`:

```
mosaic(~ weathersit + season, data=bike_share_daily, shade=T,
       legend=F, labeling_args = list(set_varnames = c(season = "Season",
       weathersit = "Primary Weather Pattern"), set_labels = list(weathersit =
       c("Clear", "Cloudy/Rainy", "Stormy"))))
```



Um. What's a mosaic plot?

If you're not already familiar with them, think of mosaic plots as contingency tables made with shapes—the larger the shape, the higher the cell count. For comparison, here's the table that the mosaic plot represents:

	Winter	Spring	Summer	Fall
Clear	111	113	136	103
Cloudy/Rainy	66	68	48	65
Stormy	4	3	4	10

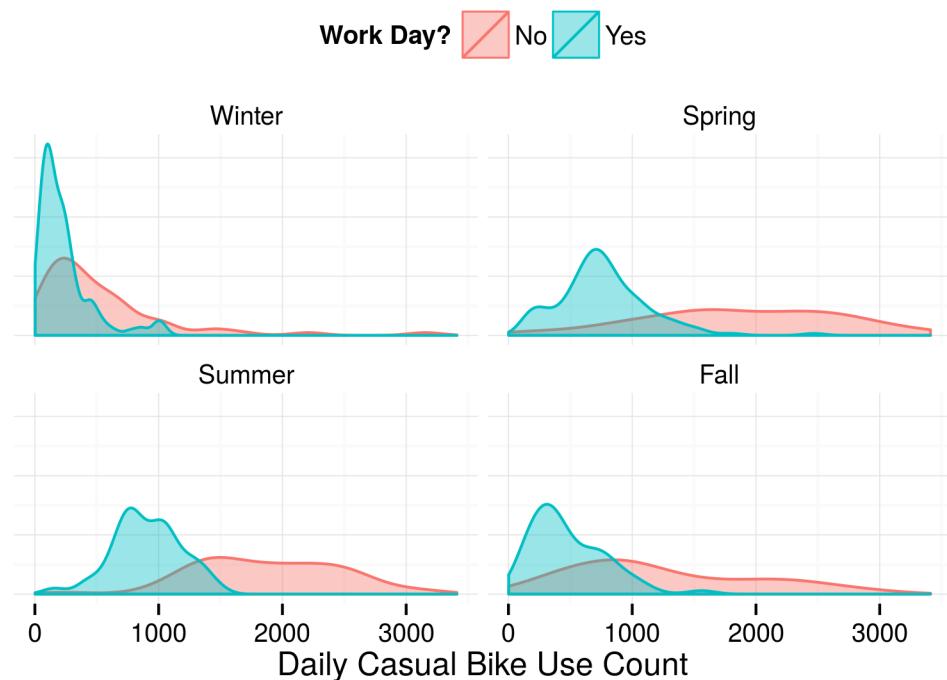
For two dimensions, a table can be more compact, but the mosaic gives you the pattern at a glance;

either is an option depending on what you need to get across. Three or four dimensions is best presented as a mosaic plot.

Multiple bivariate comparisons with faceting

As with univariate plots, we can facet on another variable for comparison's sake; for example, we can take the bike share data shown above and facet by season:

```
ggplot(bike_share_daily, aes(casual, fill=workingday, color=workingday)) +
  geom_density(alpha=0.4) +
  theme_minimal() +
  xlab("Daily Casual Bike Use Count") +
  ylab("") +
  scale_fill_discrete(name="Work Day?") +
  scale_color_discrete(name="Work Day?") +
  facet_wrap(~season, ncol=2) +
  theme(axis.ticks.y = element_blank(), axis.text.y = element_blank(),
        legend.position="top")
```



Pareto charts

Pareto charts are a mainstay of BI work, though the effort to create them can often outweigh the time a decision-maker spends with the result. The qcc library makes their creation simple.

The bike share data doesn't lend itself to Pareto chart use, so we'll use data from unplanned hospital readmissions in Hong Kong (Table 2 in Wong et al. 2011⁴²).

```
require(qcc)

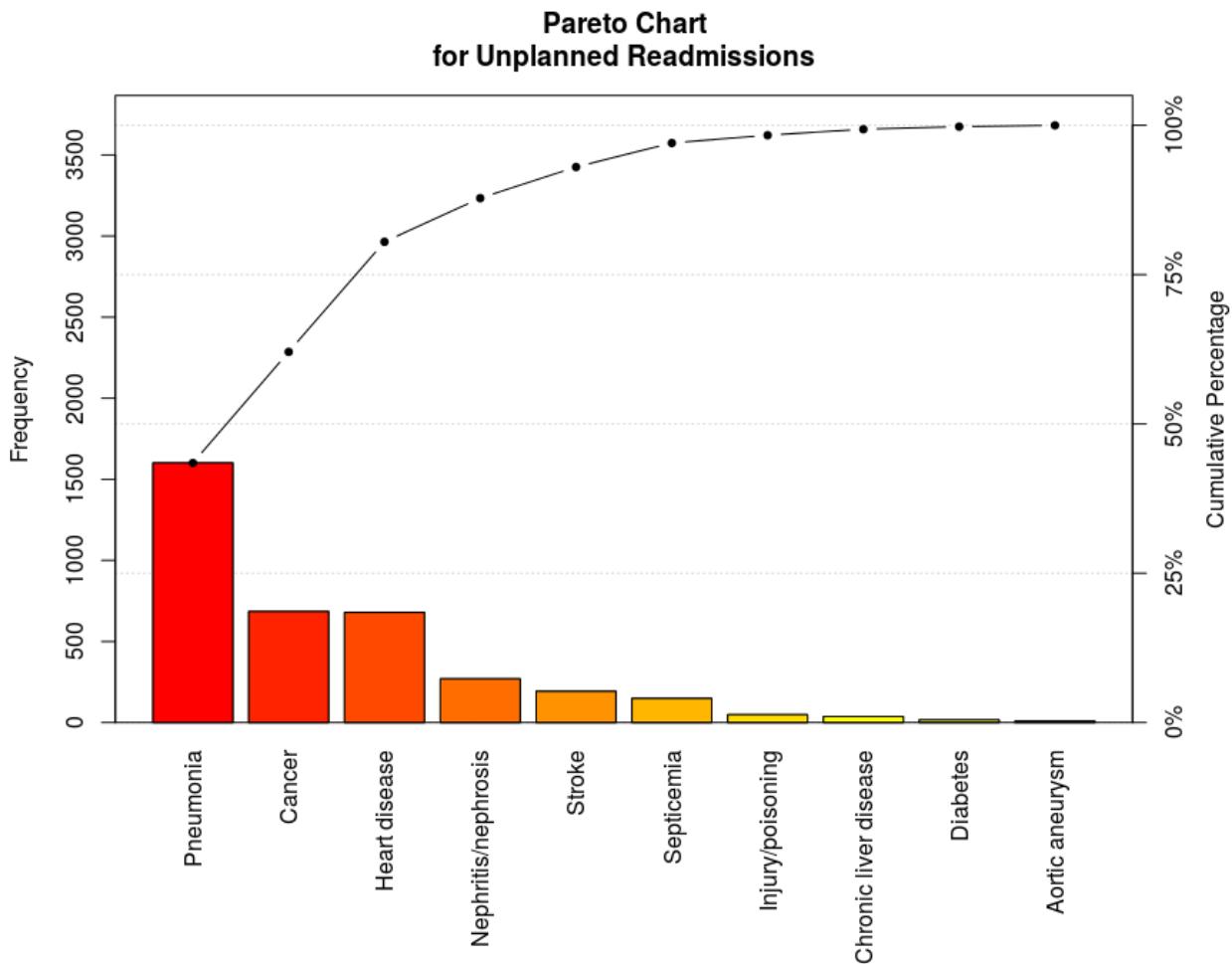
Readmits = c(148, 685, 16, 679, 192, 8, 1601, 37, 269, 48)

Dx = c('Septicemia', 'Cancer', 'Diabetes', 'Heart disease',
      'Stroke', 'Aortic aneurysm', 'Pneumonia',
      'Chronic liver disease', 'Nephritis/nephrosis',
      'Injury/poisoning')

# the xtabs function creates a crosstab table
pareto.chart(xtabs(Readmits ~ Dx), main='Pareto Chart
for Unplanned Readmissions')

Pareto chart analysis for xtabs(Readmits ~ Dx)
      Frequency Cum.Freq. Percentage Cum.Percent.
Pneumonia           1601     1601 43.4699973   43.47000
Cancer              685     2286 18.5989682   62.06897
Heart disease       679     2965 18.4360576   80.50502
Nephritis/nephrosis 269     3234  7.3038284   87.80885
Stroke              192     3426  5.2131415   93.02199
Septicemia          148     3574  4.0184632   97.04046
Injury/poisoning    48     3622  1.3032854   98.34374
Chronic liver disease 37     3659  1.0046158   99.34836
Diabetes             16     3675  0.4344285   99.78279
Aortic aneurysm     8     3683  0.2172142  100.00000
```

⁴²<http://bmchealthservres.biomedcentral.com/articles/10.1186/1472-6963-11-149>



Plotting survey data

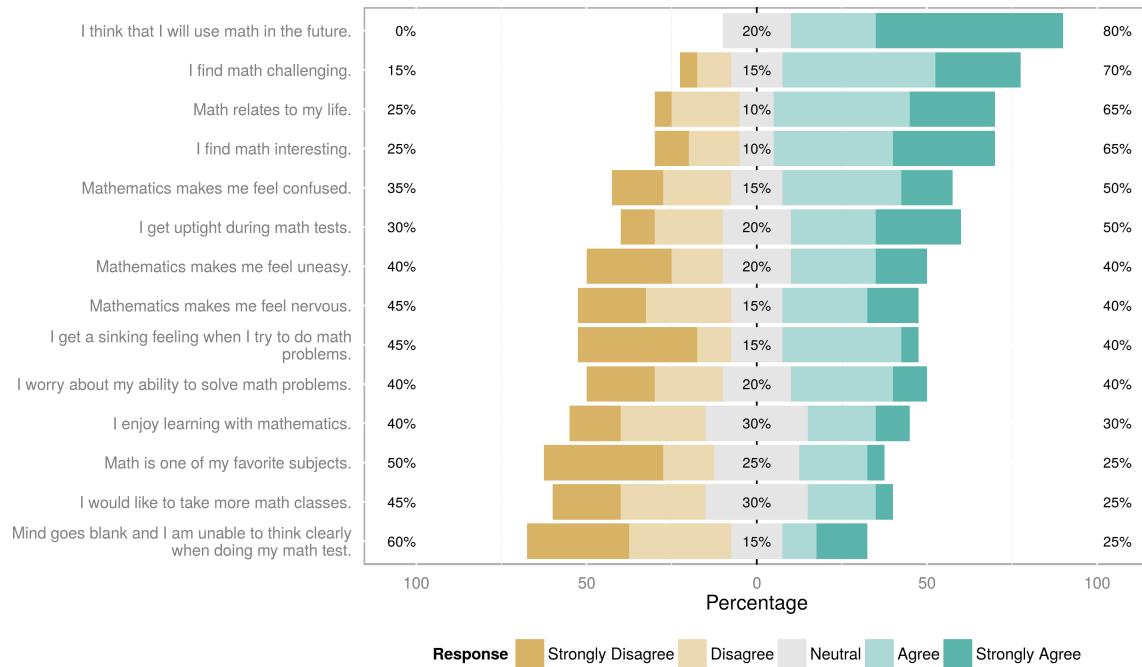
There are a variety of ways you can plot survey data, but the `likert` package makes things really simple. We can use the `mass` dataset (Math Anxiety Scale Survey) built into `likert` to demonstrate:

```
require(likert)

data(mass)

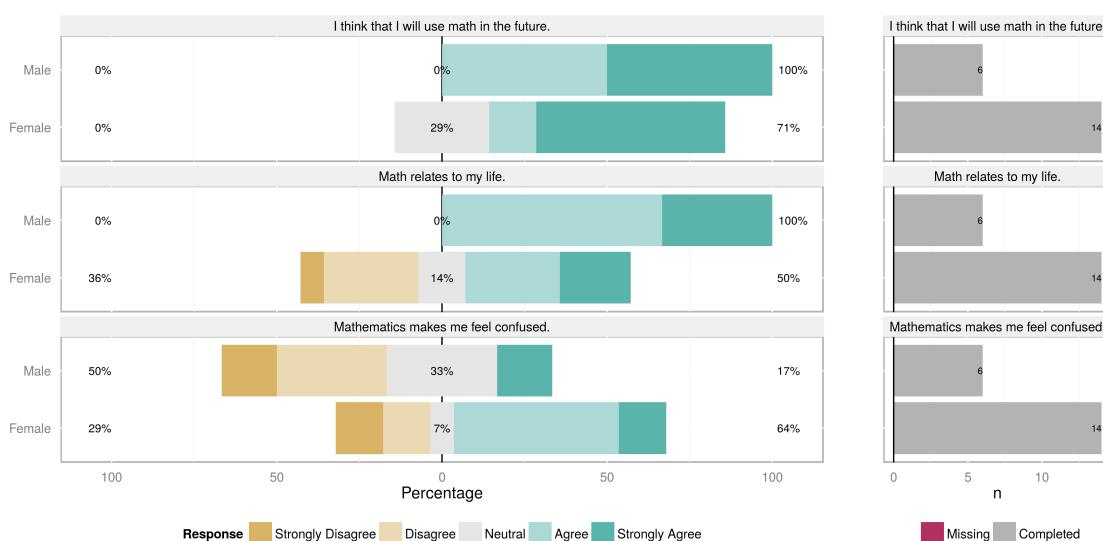
# Create a likert object
mathiness = likert(mass[2:15])

# Plot the likert object
plot(mathiness)
```



```
# Create likert object with a grouping factor
gender_math = likert(items=mass[,c(4,6,15)], drop=FALSE), grouping=mass$Gender)

# Grouped plot
plot(gender_math, include.histogram=TRUE)
```



Obtaining summary and conditional statistics

It's no accident that all the plotting work came before the summary stats in this book, although traditionally it's been the other way around. In the old days, when graphing was hard work, summary stats were used as a workaround to tell the story more simply. Now, the emphasis in both business and academia is finding the graph that tells the story, and summary stats are provided more as points of reference than the focal point. As the old saying goes, “the mean alone means nothing.”

Still, they do provide useful points of reference to help interpret graphical results. Functions to calculate summary statistics are in the base installation and a wide variety of packages, but I've found that the `psych` package probably has the most useful way to obtain them simply, particularly for groupings or when you want to condition on a particular value or threshold.

```
require(psych)
```

For numeric variables, the `describe` and `describeBy` functions do all the work for you, although you do need to specify which columns are quantitative—or which columns you want summary stats on—or it will return (nonsensically) all of them.

Basic `describe`:

```
describe(bike_share_daily[10:16])
```

	vars	n	mean	sd	median	trimmed	mad	min	max	range	skew	kurtosis	se
temp	1	731	0.50	0.18	0.50	0.50	0.23	0.06	0.86	0.80	-0.05	-1.12	0.01
atemp	2	731	0.47	0.16	0.49	0.48	0.20	0.08	0.84	0.76	-0.13	-0.99	0.01
hum	3	731	0.63	0.14	0.63	0.63	0.16	0.00	0.97	0.97	-0.07	-0.08	0.01
windspeed	4	731	0.19	0.08	0.18	0.19	0.07	0.02	0.51	0.49	0.67	0.39	0.00
casual	5	731	848.18	686.62	713.00	744.95	587.11	2.00	3410.00	3408.00	1.26	1.29	25.40
registered	6	731	3656.17	1560.26	3662.00	3641.72	1712.40	20.00	6946.00	6926.00	0.04	-0.72	57.71
cnt	7	731	4504.35	1937.21	4548.00	4517.19	2086.02	22.00	8714.00	8692.00	-0.05	-0.82	71.65

Using `describeBy`:

```
describeBy(bike_share_daily[10:16], bike_share_daily$holiday)
```

```
Console ~/ 
> describeBy(bike_share_daily[10:16], bike_share_daily$holiday)
group: No
    vars   n     mean      sd median trimmed     mad   min     max   range skew kurtosis     se
temp       1 710    0.50    0.18    0.50    0.50    0.23   0.06    0.86    0.80 -0.06    -1.11   0.01
atemp      2 710    0.48    0.16    0.49    0.48    0.20   0.08    0.84    0.76 -0.14    -0.97   0.01
hum        3 710    0.63    0.14    0.63    0.63    0.16   0.00    0.97    0.97 -0.07    -0.08   0.01
windspeed   4 710    0.19    0.08    0.18    0.18    0.07   0.02    0.51    0.49  0.70    0.42   0.00
casual      5 710  841.77  680.53  711.50  739.69  584.14  2.00  3410.00  3408.00  1.27    1.34 25.54
registered   6 710 3685.33 1553.70 3691.00 3672.14 1695.35 20.00 6946.00 6926.00  0.04   -0.71 58.31
cnt         7 710 4527.10 1929.01 4558.00 4541.38 2019.30 22.00 8714.00 8692.00 -0.05   -0.80 72.39
-----
group: Yes
    vars   n     mean      sd median trimmed     mad   min     max   range skew kurtosis     se
temp       1 21    0.47    0.20    0.38    0.46    0.16   0.18    0.79    0.61  0.24    -1.59   0.04
atemp      2 21    0.44    0.18    0.39    0.44    0.20   0.18    0.73    0.56  0.17    -1.58   0.04
hum        3 21    0.61    0.11    0.60    0.62    0.12   0.38    0.79    0.41 -0.13    -1.09   0.03
windspeed   4 21    0.19    0.08    0.19    0.20    0.06   0.04    0.33    0.29 -0.13    -0.74   0.02
casual      5 21 1064.71  860.05  874.00  965.88  658.27 117.00 3065.00 2948.00  0.88   -0.41 187.68
registered   6 21 2670.29 1492.86 2549.00 2604.47 1599.73 573.00 5172.00 4599.00  0.27   -1.26 325.77
cnt         7 21 3735.00 2103.35 3351.00 3685.24 3015.61 1000.00 7403.00 6403.00  0.17   -1.53 458.99
> |
```

The `table` and `prop.table` functions from the `base` package provide the simplest summary stats for categorical and ordinal variables:

```
table(bike_share_daily$holiday)
```

No Yes

710 21

```
prop.table(table(bike_share_daily$holiday))
```

	No	Yes
0.97127223	0.02872777	

You can also subset the stats based on factors or conditions.

Summary stats for bike use during the winter season:

```
describeBy(bike_share_daily[14:16], bike_share_daily$season == "Winter")
```

```
Console ~/ 
> describeBy(bike_share_daily[14:16], bike_share_daily$season == "Winter")
group: FALSE
    vars   n     mean      sd median trimmed     mad   min     max   range skew kurtosis     se
casual      1 550 1017.08  679.84  846.5  929.22  531.51   2 3410  3408  1.14     0.88 28.99
registered   2 550 4112.61 1386.26 3958.0 4116.39 1412.18  20 6946  6926 -0.01    -0.50 59.11
cnt         3 550 5129.69 1662.91 5009.0 5176.35 1773.19  22 8714  8692 -0.16    -0.45 70.91
-----
group: TRUE
    vars   n     mean      sd median trimmed     mad   min     max   range skew kurtosis     se
casual      1 181  334.93  387.66   218  260.04  183.84   9 3155  3146  3.45    17.55 28.81
registered   2 181 2269.20 1200.27  1867 2177.17 1094.16  416 5315  4899  0.64   -0.55 89.22
cnt         3 181 2604.13 1399.94  2209 2475.77 1316.55 431 7836  7405  0.87    0.43 104.06
> |
```

Summary stats for weather on days where there were less than/greater than 1000 casual uses:

```
describeBy(bike_share_daily[10:13], bike_share_daily$casual <= 1000)
```

```
Console ~/ 
> describeBy(bike_share_daily[10:13], bike_share_daily$casual <= 1000)
group: FALSE
  vars   n mean   sd median trimmed mad min max range skew kurtosis   se
temp     1 227 0.61 0.12   0.64    0.62 0.13 0.32 0.86  0.54 -0.47   -0.67 0.01
atemp    2 227 0.58 0.11   0.59    0.58 0.10 0.24 0.80  0.56 -0.53   -0.24 0.01
hum      3 227 0.61 0.13   0.61    0.61 0.15 0.25 0.91  0.66  0.01   -0.44 0.01
windspeed 4 227 0.17 0.07   0.17    0.17 0.07 0.04 0.37  0.33  0.44   -0.20 0.00
-----
group: TRUE
  vars   n mean   sd median trimmed mad min max range skew kurtosis   se
temp     1 504 0.44 0.18   0.41    0.44 0.20 0.06 0.85  0.79  0.35   -0.92 0.01
atemp    2 504 0.43 0.16   0.40    0.42 0.19 0.08 0.84  0.76  0.27   -0.85 0.01
hum      3 504 0.63 0.15   0.63    0.63 0.16 0.00 0.97  0.97 -0.13   -0.04 0.01
windspeed 4 504 0.20 0.08   0.19    0.19 0.08 0.02 0.51  0.49  0.65   0.25 0.00
>
```

Summary stats for bike use on days in which the windspeed was greater than/less than average:

```
describeBy(bike_share_daily$casual, bike_share_daily$windspeed >
  mean(bike_share_daily$windspeed))
```

```
Console ~/ 
> describeBy(bike_share_daily$casual, bike_share_daily$windspeed >
+ mean(bike_share_daily$windspeed))
group: FALSE
  vars   n mean   sd median trimmed mad min max range skew kurtosis   se
1    1 409 914.61 673.18   773  827.61 598.97   9 3410  3401 1.11     0.96 33.29
-----
group: TRUE
  vars   n mean   sd median trimmed mad min max range skew kurtosis   se
1    1 322 763.8 695.26  613.5  642.12 579.7   2 3283  3281 1.5      1.94 38.75
> |
```



Although `describeBy` gives you more versatility, you can also use `sum` for a quick-and-dirty conditional count, e.g.,

```
sum(bike_share_daily$cnt > 2500)
```

587

```
sum(bike_share_daily$workingday == 'Yes')
```

500

You can create two-, three-, or n -way tables by adding variables to the table call, and use `addmargins` to get marginal sums.

A three-way table

```
table(bike_share_daily[c(3,6:7)])
```

		weekday = 0	holiday
season	No Yes		
Winter	27 0		
Spring	26 0		
Summer	26 0		
Fall	26 0		

		weekday = 1	holiday
season	No Yes		
Winter	20 6		
Spring	24 3		
Summer	23 3		
Fall	23 3		

...

A three-way table with marginals

```
addmargins(table(bike_share_daily[c(3,6:7)]))
```

		weekday = 0	holiday	season	No Yes Sum
Winter	27 0	27		Winter	27 0 27
Spring	26 0	26		Spring	26 0 26
Summer	26 0	26		Summer	26 0 26
Fall	26 0	26		Fall	26 0 26
				Sum	105 0 105

		weekday = 1	holiday	season	No Yes Sum
Winter	20 6	26		Winter	20 6 26
Spring	24 3	27		Spring	24 3 27
Summer	23 3	26		Summer	23 3 26
Fall	23 3	26		Fall	23 3 26
				Sum	90 15 105

...

A table of proportions:

```
prop.table(table(bike_share_daily[c(3,9)]))
```

weathersit	1	2	3
season			
Winter	0.151846785	0.090287278	0.005471956
Spring	0.154582763	0.093023256	0.004103967
Summer	0.186046512	0.065663475	0.005471956
Fall	0.140902873	0.088919289	0.013679891

Sometimes you may want the counts to be in a data frame, either for use in plotting or because tables with more than 2 or 3 dimensions can be difficult to understand at a glance. You can quickly create any n -way table as a data frame using `dplyr`:

```
require(dplyr)

summarize(group_by(bike_share_daily, season, holiday, weekday), count=n())

Source: local data frame [37 x 4]
Groups: season, holiday

  season holiday weekday count
1 Winter     No       0     27
2 Winter     No       1     20
3 Winter     No       2     24
4 Winter     No       3     25
...

```

Finding the mode or local maxima/minima

The `which.max` (or `which.min`) function can show you the set of values for a chosen maximum (or minimum) observation:

```
# What were the other variables on the day of minimum casual use?
bike_share_daily[which.min(bike_share_daily[,14]),]
```

The screenshot shows the RStudio console window with the title "Console". The command `> bike_share_daily[which.min(bike_share_daily[,14]),]` is entered, followed by its output. The output shows a single row from the `bike_share_daily` dataset corresponding to the day of minimum casual use (October 29, 2012). The columns displayed are instant, dteday, season, yr, mnth, holiday, weekday, workingday, weathersit, temp, atemp, hum, windspeed, casual, registered, and cnt.

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	cnt
668	668	2012-10-29	Fall	2012	Oct	No	1	Yes	3	0.44	0.4394	0.88	0.3582	2	20	22

This trick can help you find the mode of a distribution (actually, the point of maximum density):

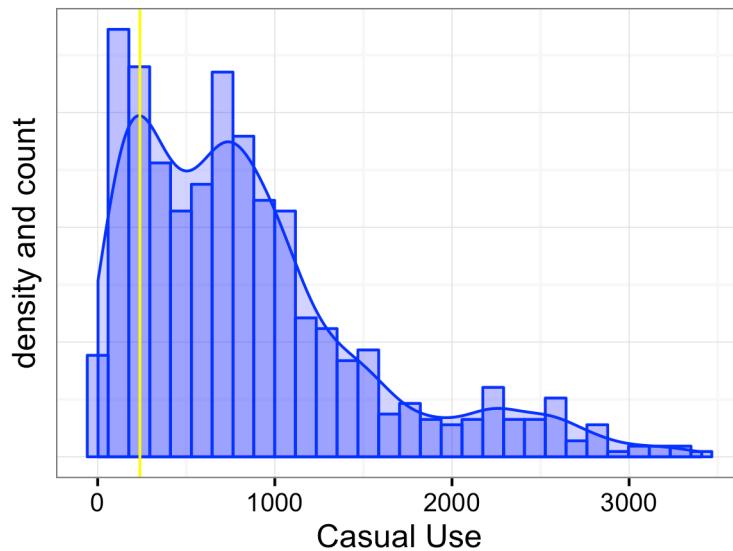
```
# Calculate density of casual bike use
casual_dens = data.frame(casual = density(bike_share_daily$casual)$x,
                         density_value = density(bike_share_daily$casual)$y)

# Mode / maximum density
casual_dens[which.max(casual_dens[,2]),]

  casual density_value
80 238.3353 0.0007430897
```

We can then plot that value on our ggplot from earlier in this chapter by adding it as a `geom_vline`:

```
ggplot(bike_share_daily, aes(casual)) +
  ylab("density and count") +
  xlab("Casual Use") +
  geom_histogram(aes(y=..density..), col="blue", fill="blue", alpha=0.3) +
  geom_density(col="blue", fill="blue", alpha=0.2) +
  theme_bw() +
  theme(axis.ticks.y = element_blank(), axis.text.y = element_blank()) +
  geom_vline(xintercept = dens[which.max(dens[,2]),1], color = "yellow")
```



Inference on summary statistics

Confidence intervals

When we don't have a full population to assess, we often need to determine how uncertain our estimates of the population parameters are—we need confidence intervals on those point estimates. There are base installation functions for most (but not all) of the ones we typically need:

Statistic	Data type	Distribution	Function
Median	Any	Any	asbio::ci.median(x)
Mean	Continuous	Normal, t, “normal enough”	t.test(x)\$conf.int
Proportion	Percentage	Binomial	binom.test(x, n)\$conf.int
Count	Count	Poisson	poisson.test(x)\$conf.int
Rate	Count/n	Poisson	poisson.test(x, n)\$conf.int

```

# 95% CI for median daily casual bike use
asbio::ci.median(bike_share_daily$casual)

95% Confidence interval for population median
Estimate      2.5%     97.5%
    713       668       760

# 95% CI for mean daily casual bike use
t.test(bike_share_daily$casual)$conf.int

798.3192 898.0337

# 95% CI for proportion of casual bike use of all rentals
binom.test(sum(bike_share_daily$casual),
           sum(bike_share_daily$cnt))$conf.int

0.1878795 0.1887244

# Subset data to winter only
bike_share_winter = filter(bike_share_daily, season == "Winter")

# 95% CI for count of winter casual bike use per 1000 rentals
poisson.test(sum(bike_share_winter$casual),
             sum(bike_share_winter$cnt)/1000)$conf.int

127.5923 129.6421

```

Sometimes there isn't a base function for the confidence interval we need, or our data depart considerably from one of the typical distributions. For example, the confidence interval of a standard deviation is very sensitive to departures from normality, and R doesn't have a built-in function for it.

Non-parametric bootstrapping is the way to get these intervals; the `boot` package provides the most options but can be a pain to work with since its syntax is fairly different from most everything else in R. Still, it does what we need it to with a relative minimum of code for most cases. In this example we'll use the `PlantGrowth` dataset that comes with R for simplicity's sake:

```

require(boot)

# Create the boot function
sd_boot_function = function(x,i){sd(x[i])}

```

```
# Run the bootstrapping
sd_boot = boot(PlantGrowth$weight, sd_boot_function, R=10000)

# Bootstrapped sd
sd(PlantGrowth$weight)

0.7011918

# 95% CI for bootstrapped sd
boot.ci(sd_boot, type="bca")$bca[4:5]

0.5741555 0.8772850
```

The bias-corrected and accelerated (BCa) method is generally the appropriate calculation to use unless you have strong reason to believe that another method is best in the particular context. For example, you might want to use `type="norm"`, i.e., use the normal approximation for very small samples that you expect are “normal enough,” realizing that due to sampling error you could be estimating intervals that are wildly incorrect.

We can also bootstrap confidence intervals for any summary statistic. For example, to get the bootstrapped 75th percentile, run the following:

```
q75_function = function(x,i){quantile(x[i], probs=0.75)}

q75_boot = boot(PlantGrowth$weight, q75_function, R=10000)

quantile(PlantGrowth$weight, 0.75)

5.53

boot.ci(q75_boot, type="bca")$bca[4:5]

5.24 5.99
```

You can get quick (gg)plots of means/medians and confidence intervals, standard deviations (`mult`, defaults to 2), or quantiles with the `stat_summary` functions:

Statistic	Interval Type	Function
Mean	Bootstrapped CI	(fun.data = mean_ci_boot, fun.args = list(conf.int=0.95), ...)
Mean	Normal CI	(fun.data = mean_ci_normal, fun.args = list(conf.int=0.95), ...)
Mean	Standard deviation	(fun.data = mean_sdl, fun.args = list(mult=2), ...)
Median	Quantile	(fun.data = median_hilow, fun.args = list(conf.int=0.5), ...)

The following code demonstrates all four options (and provides an example of plotting ggplot2 objects together using the `grid.arrange` function from the `gridExtra` package):

```
require(gridExtra)

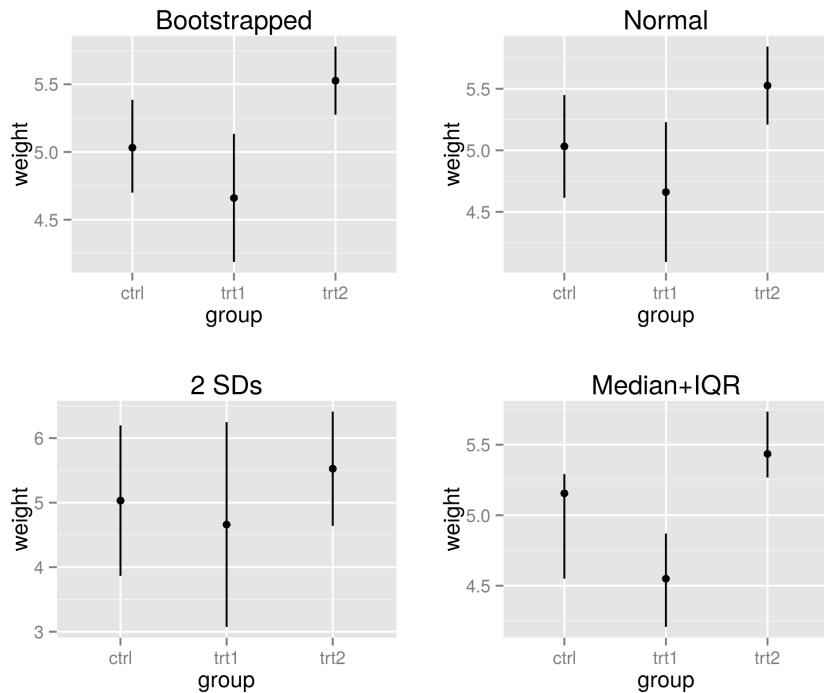
p1 = ggplot(PlantGrowth, aes(group, weight)) +
  ggtitle("Bootstrapped") +
  stat_summary(fun.data = mean_cl_boot, fun.args = list(conf.int = 0.95))

p2 = ggplot(PlantGrowth, aes(group, weight)) +
  ggtitle("Normal") +
  stat_summary(fun.data = mean_cl_normal, fun.args = list(conf.int = 0.95))

p3 = ggplot(PlantGrowth, aes(group, weight)) +
  ggtitle("2 SDs") +
  stat_summary(fun.data = mean_sdl, fun.args = list(mult = 2))

p4 = ggplot(PlantGrowth, aes(group, weight)) +
  ggtitle("Median+IQR") +
  stat_summary(fun.data = median_hilow, fun.args = list(conf.int = 0.5))

grid.arrange(p1, p2, p3, p4, nrow = 2)
```



Tolerance intervals

While confidence or Bayesian credible intervals tend to be more often used in data science, tolerance intervals are more useful in engineering or industrial contexts: there's a profound conceptual and practical difference between, for example, a good estimate of average commuting time to work (using a 95% *confidence* interval) and how long you can expect 95% of those trips will take (using a 95% *tolerance* interval). While the former type of value tends to be used more by statisticians and data miners, engineers and industrial analysts—and sometimes business users—find the latter more useful.

Essentially, a good way to remember the difference is that as the sample size approaches the population size, the confidence interval will approach zero, while a tolerance interval will approach the population percentiles.

The `tolerance` package contains functions for estimating tolerance intervals for distributions typically used in most engineering or industrial processes. As an example, assume you have data on commuting time for 40 randomly chosen workdays. The distribution is oddly-shaped, so we'll use non-parametric tolerance intervals (via the `nptol.int` function) to estimate (at a 95% confidence level) the number of minutes that 75% of commutes will take:

```
require(tolerance)

commute_time = c(68, 42, 40, 69, 46, 37, 68, 68, 69, 38, 51, 36, 50, 37, 41,
  68, 59, 65, 67, 42, 67, 62, 48, 52, 52, 44, 65, 65, 46, 67, 62, 66, 43, 58,
  45, 65, 60, 55, 48, 46)

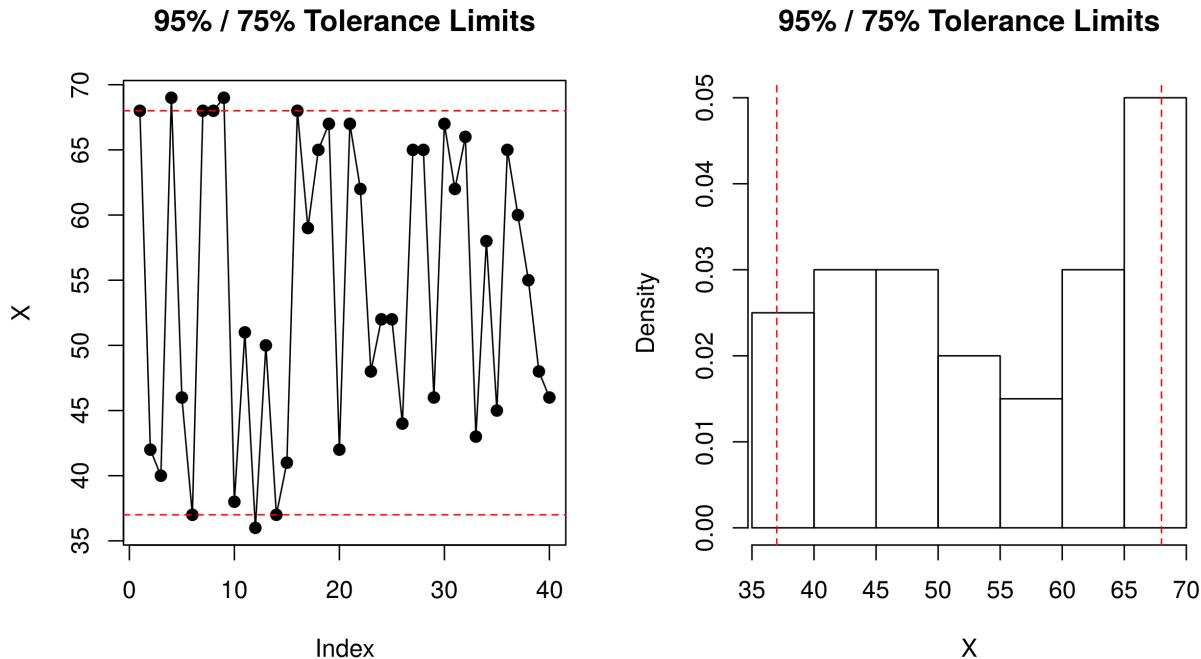
commute_time_npti = nptol.int(commute_time, alpha=0.05, P=0.75, side=2)

commute_time_npti

  alpha      P 2-sided.lower 2-sided.upper
1  0.05  0.75            37              68
```

You can also obtain a graphical summary with the `plottol` function, which shows a control chart on the left and a histogram on the right:

```
plottol(commute_time_npti, commute_time, side="two", plot.type="both")
```



If you plotted both tolerance graphs (`plot.type="both"`), you may need to return `par` to normal manually:

```
par(mfrow=c(1,1))
```

The following table provides a short overview of the primary univariate tolerance interval functions; see `?tolerance` for more details. I recommend you set `alpha` (confidence level), `P` (the coverage proportion, i.e., tolerance interval), and `side` (whether to calculate upper, lower, or both bounds) manually for best results; if not chosen, `alpha` defaults to 0.05, `P` defaults to 0.99, and `side` defaults to 1 (you'll probably want 2 in many cases). Other settings depend on the distribution used, and each distribution sometimes has several algorithms to choose from, so it's worth exploring the documentation before diving in.

Data type	Distribution	Function
Percent	Binomial	<code>bintol.int(x, n, m, ...)</code>
Count or Rate	Poisson	<code>poistol.int(x, n, m, side, ...)</code>
Nonparametric	None	<code>nptol.int(x, ...)</code>
Continuous	Normal, t, “normal enough”	<code>normtol.int(x, side, ...)</code>
Continuous	Uniform	<code>uniftol.int(x, ...)</code>
Lifetime/survival	Exponential	<code>exptol.int(x, type.2, ...)</code>
Score	Laplace	<code>laptol.int(x, ...)</code>
Indicies	Gamma	<code>gamtol.int(x, ...)</code>
Reliability, extreme values	Weibull, Gumbel	<code>exttol.int(x, dist, ...)</code>

Dealing with missing data

Visualizing missing data

The VIM package has some great tools for visualizing missing data patterns. As the bike share data doesn't have any missing values, we'll use a dataset derived from the Tropical Atmosphere Ocean⁴³ project for this recipe.

```
require(VIM)

data(tao)

# Rename the Sea.Surface.Temp column to make label fit on plot
colnames(tao)[4] = "Sea.Temp"

# Look at the data, esp. NAs
summary(tao)
```

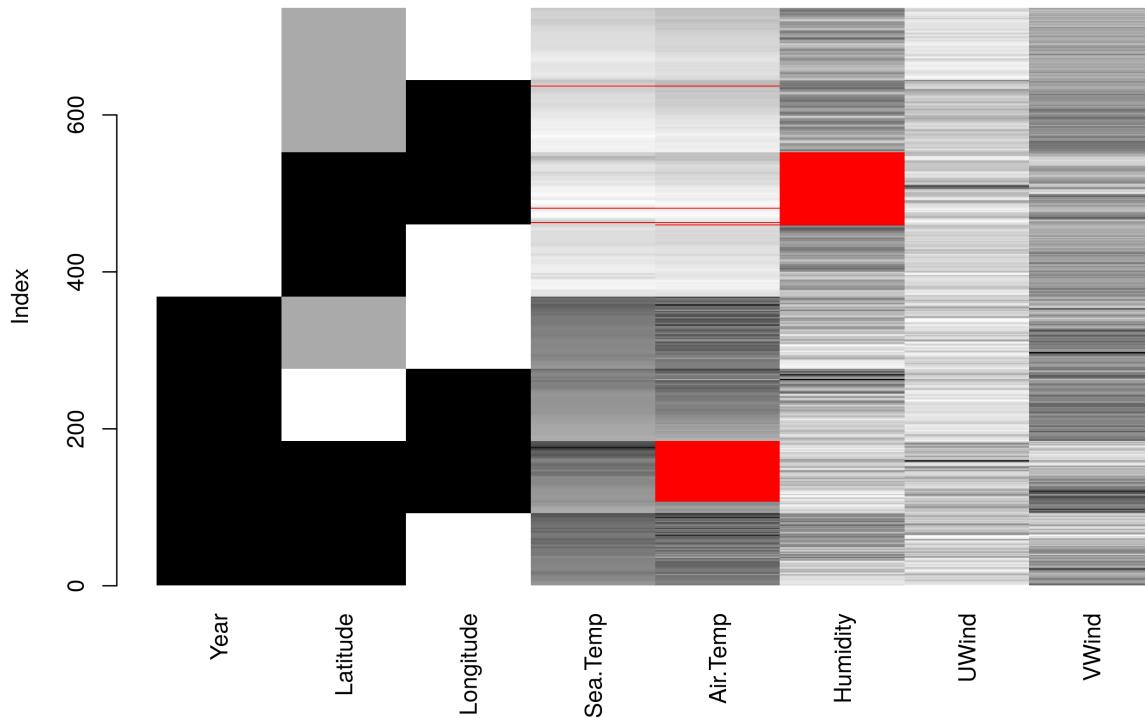
```
Console ~/ ↗
> summary(tao)
   Year      Latitude    Longitude     Sea.Temp     Air.Temp
Min.  :1993  Min.  :-5.000  Min.  :-110.0  Min.  :21.60  Min.  :21.42
1st Qu.:1993  1st Qu.:-2.000  1st Qu.:-110.0  1st Qu.:23.50  1st Qu.:23.26
Median :1995  Median :-1.000  Median :-102.5  Median :26.55  Median :24.52
Mean   :1995  Mean   :-1.375  Mean   :-102.5  Mean   :25.86  Mean   :25.03
3rd Qu.:1997  3rd Qu.: 0.000  3rd Qu.:-95.0  3rd Qu.:28.21  3rd Qu.:27.08
Max.   :1997  Max.   : 0.000  Max.   :-95.0  Max.   :30.17  Max.   :28.50
                           NA's   :3          NA's   :81

   Humidity      UWind      VWind
Min.  :71.60  Min.  :-8.100  Min.  :-6.200
1st Qu.:81.30  1st Qu.:-5.100  1st Qu.: 1.500
Median :85.20  Median :-3.900  Median : 2.900
Mean   :84.43  Mean   :-3.716  Mean   : 2.636
3rd Qu.:88.10  3rd Qu.:-2.600  3rd Qu.: 4.100
Max.   :94.80  Max.   : 4.300  Max.   : 7.300
NA's   :93
> |
```

The `matrixplot` function will display the entire dataset as a set of rectangles to represent each cell. Similar values in the data will have similar grayscale values, and NAs will show up clearly in red:

```
matrixplot(tao)
```

⁴³<http://www.pmel.noaa.gov/tao/>



The error message you may see in the console about gamma was submitted as an issue to the creators of VIM in mid 2016. It doesn't affect the plotting in this package (as the message says, it has no effect), so you can just ignore it.

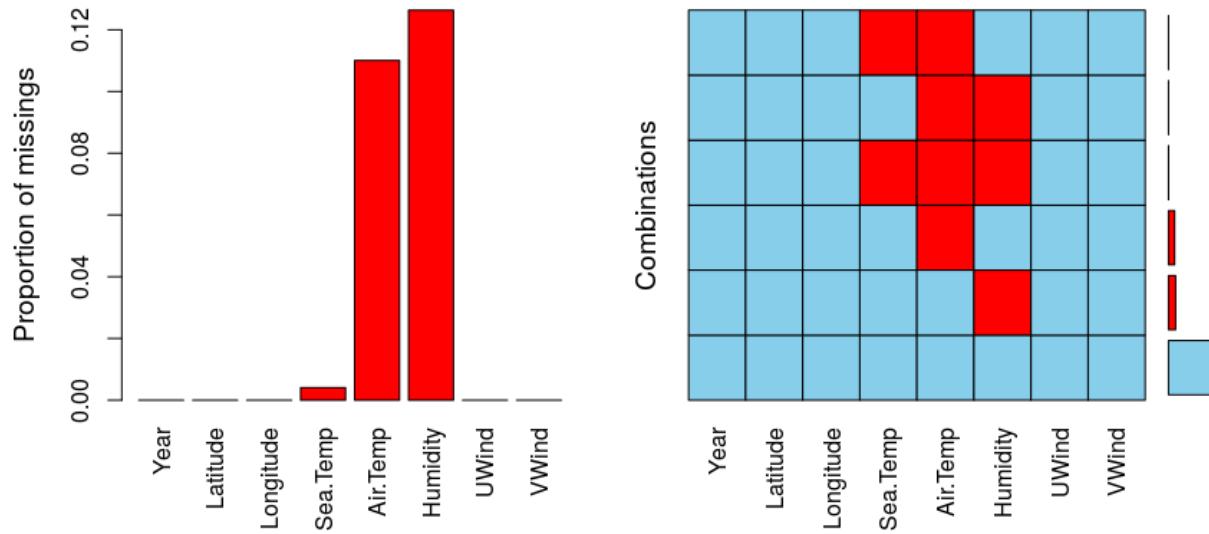
To complement the detailed picture given by `matrixplot`, you can get a numeric and visual summary of missing values with the `aggr` function:

```
tao_aggr = aggr(tao)
```

```
tao_aggr
```

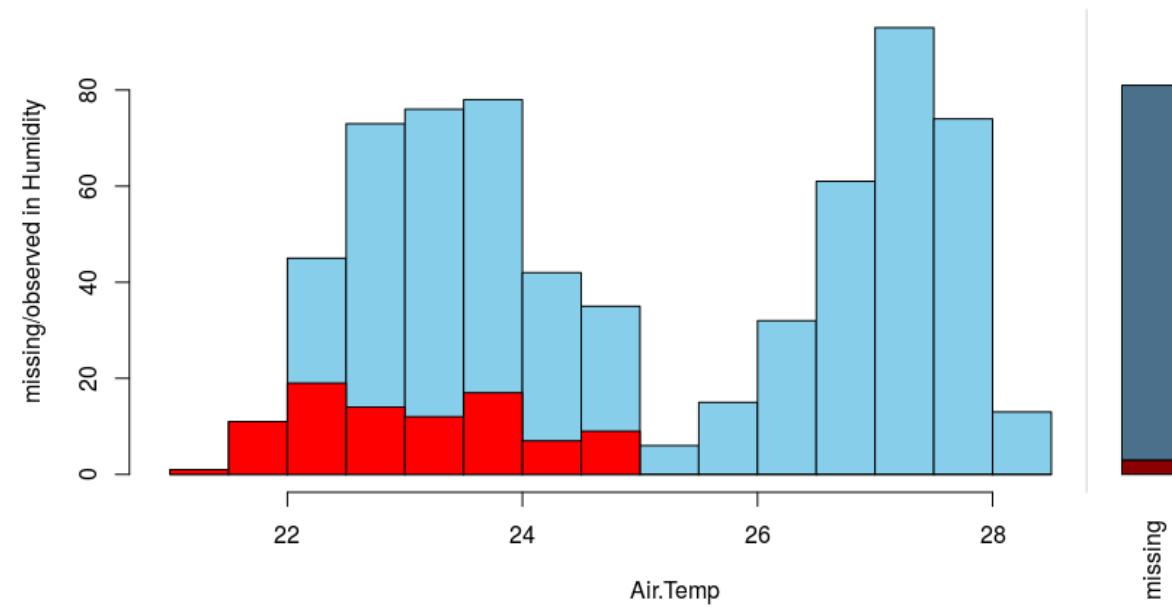
Missings in variables:

	Variable	Count
Sea.Surface.Temp		3
Air.Temp		81
Humidity		93



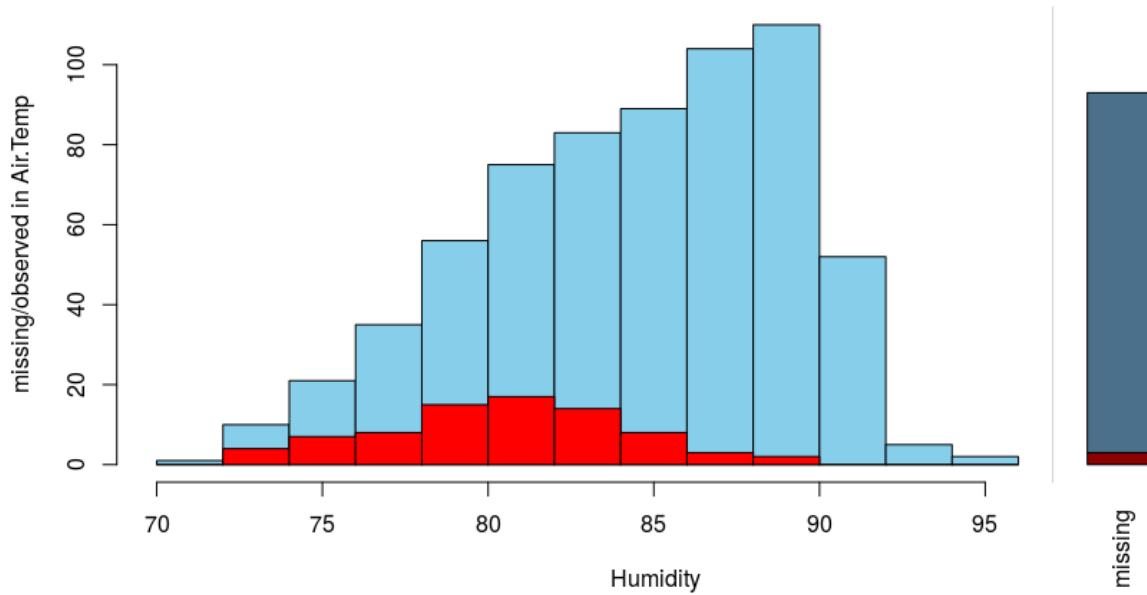
You can compare missingness across fields with histograms, such as the distribution of missing humidity values by corresponding air temperature:

```
histMiss(tao[5:6])
```



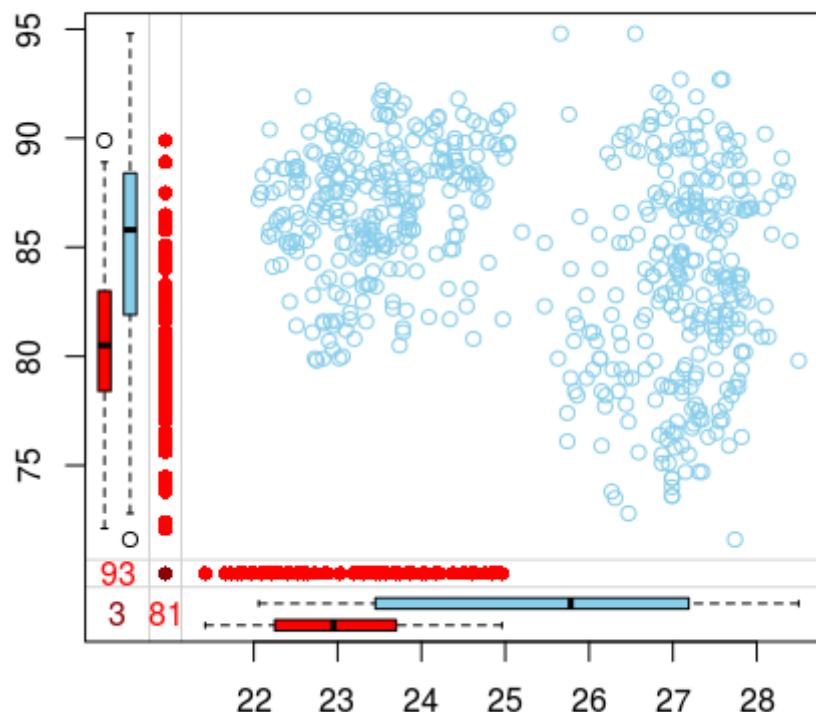
You can flip the variables to see the opposite distribution:

```
histMiss(tao[c(6,5)])
```



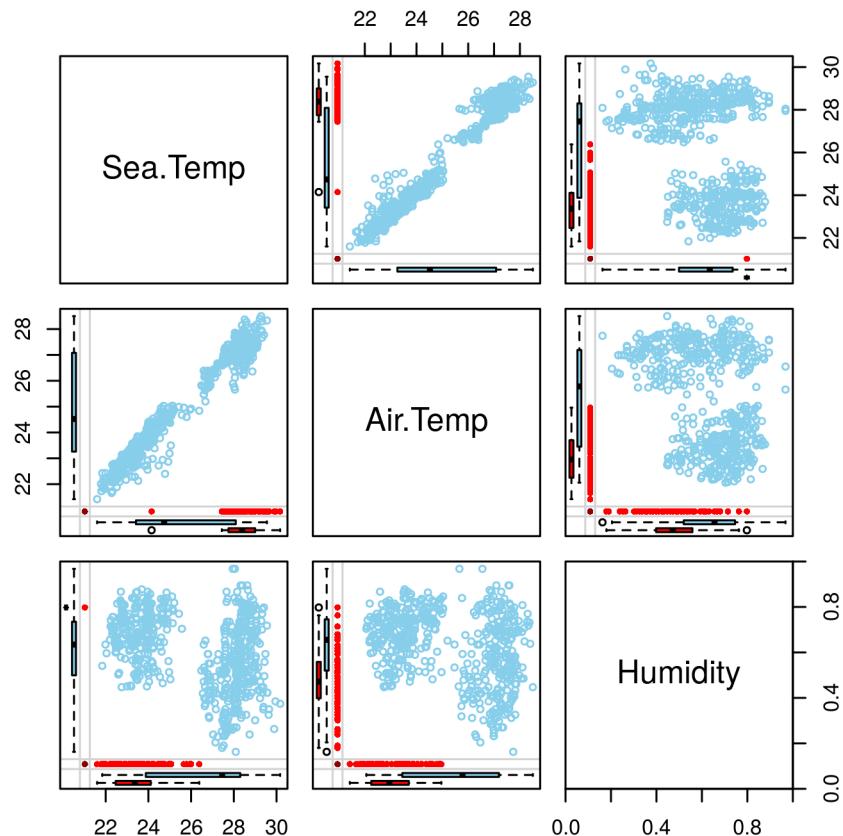
You can show the same data in a scatterplot for more details:

```
marginplot(tao[5:6])
```



This can be expanded to visualizing pairwise comparisons with the `marginmatrix` function:

```
marginmatrix(tao[4:6])
```



`VIM` also has a separate GUI for interactive/on-the-fly exploration called `VIMGUI`. Even if you prefer coding, it contains a [vignette⁴⁴](#) from which these examples are drawn, so it's worth installing to learn more about how it works and the many options you can control in the imputation process.

Imputation for missing values

`VIM` also performs imputation on missing values using a variety of methods. Generally, replacing values with a constant is a bad idea, as it will bias the distribution (see below for an example).

⁴⁴<http://cran.r-project.org/web/packages/VIMGUI/vignettes/VIM-Imputation.pdf>

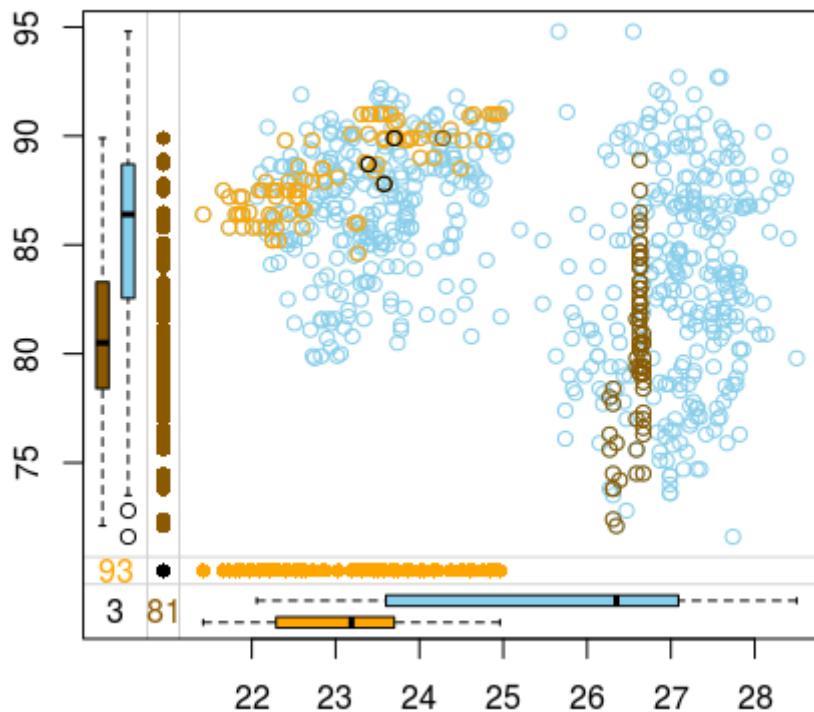
Using a method that accounts for more variables or adds some noise or randomness is often a better approach. But like all things in stats, the best approach will depend on the entire analytics context, from the variables' data types to the business question itself. There is no best way, so proceed with caution whenever you need to impute data.

The *k*-nearest neighbors approach can provide reasonable imputed values for many situations. Using the same data as above:

```
# Perform k Nearest Neighbors imputation
# Result is new dataframe with imputed values
tao_knn = kNN(tao)
```

This function appends a set of indicator variables to the data that show TRUE/FALSE for whether a particular observation was imputed. This becomes useful in the subsequent plotting of the results, e.g., for the Air.Temp and Humidity variables:

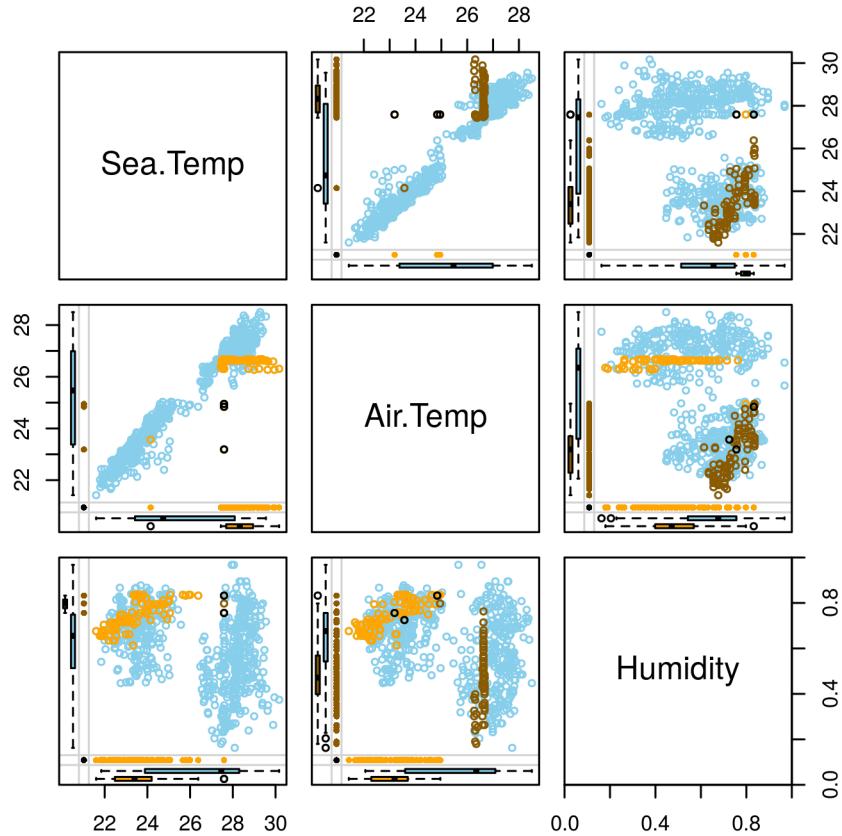
```
marginplot(tao_knn[c(5:6, 13:14)], delimiter="_imp")
```



Light and dark orange show results imputed for the variable on that axis; black shows observations for which both were imputed. Blue continues to show the distribution of the non-imputed values.

Viewing a matrix works here as well:

```
marginmatrix(tao_knn[c(4:6, 12:14)], delimiter="_imp")
```



There are other methods in VIM, including a model-based approach:

```
# Perform standard Iterative Robust Model-based Imputation
tao_irmi = irmi(tao)

# Perform robust Iterative Robust Model-based Imputation
tao_irmi_robust = irmi(tao, robust=TRUE)
```

By subsetting the results of different methods, you can plot their densities to see how each approach might influence subsequent analysis. In addition to the original data and the three methods above, we'll add a mean imputation to show why a constant can be a bad choice for imputation.

```

# Create a mean-imputed air temp variable
tao$tao_airtemp_mean = ifelse(is.na(tao$Air.Temp), mean(tao$Air.Temp,
  na.rm=TRUE), tao$Air.Temp)

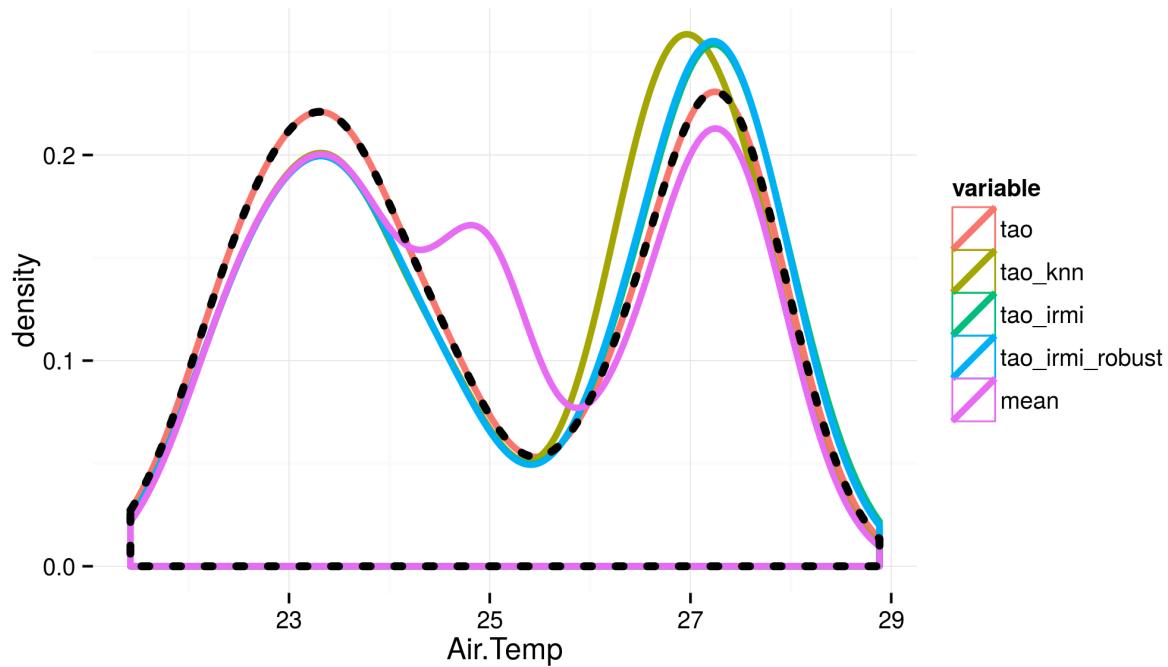
# Make a data frame of each air temp result
tao_compare_airtemp = data.frame(tao=tao[,5], tao_knn=tao_knn[,5],
  tao_irmi=tao_irmi[,5], tao_irmi_robust=tao_irmi_robust[,5], mean=tao[,9])

# Melt the various air temp results into a long data frame
require(reshape2)

tao_compare_melt = melt(tao_compare_airtemp, value.name="Air.Temp")

# Plot density histograms of each option and
# add black dotted line to emphasize the original data
ggplot(tao_compare_melt, aes(Air.Temp, color=variable)) +
  geom_density(lwd=1.25) +
  geom_density(data=subset(tao_compare_melt, variable=="tao"),
    aes(Air.Temp, lty=3, lwd=1.5, color="black")) +
  theme_minimal()

```





More imputation in R: other packages

Since imputation is a complex subject, there are a lot of different R packages to do it; explore the *Imputation* section of the CRAN Task View [Official Statistics & Survey Methodology⁴⁵](#) for a short overview of those packages and their methods.

⁴⁵<http://cran.r-project.org/web/views/OfficialStatistics.html>

Chapter 5: Effect Sizes

- Overview
- Measuring *differences* between groups
- Measuring *similarities* between groups

Overview

The use of effect sizes has been gaining in popularity in research for the last decade or so, but unfortunately has made little headway into business. But effect sizes provide the results that business users *really* want to know—not just whether two groups are different or the same, but **how** different or similar are they?

The following tables summarize major effect size statistics (ESSs) by type of effect size (differences in tendency, differences in variation, and relationships), and some packages and functions I've found useful to calculate them. After the tables are recipes and examples of these ESSs. There are a variety of options inside R, of course, but the functions in this chapter should provide most of the essential values you'll need.

Differences in tendency

Type	Statistic	Package	Function
Difference in means	D	base	<code>t.test(...)\$estimate</code> <code>t.test(...)\$conf.int</code>
	D	bootES	<code>bootES</code> with <code>effect.type="unstandardized"</code>
	D	BayesianFirstAid	<code>bayes.t.test</code>
Difference in medians		simpleboot	<code>two.boot(..., median)\$t0</code> <code>boot.ci</code> of <code>two.boot</code> object
		BayesianFirstAid	<i>not yet implemented</i>
Difference in quantiles		simpleboot	<code>two.boot(..., quantile,</code> <code>probs=0.75)\$t0</code> <code>boot.ci</code> of <code>two.boot</code> object

Type	Statistic	Package	Function
Standardized mean difference	Cohen's d	bootES	bootES with effect.type="cohens.d"
	robust Cohen's d	bootES	bootES with effect.type="akp.robust.d"
	Hedge's g	bootES	bootES with effect.type="hedges.g"
Difference between proportions		base	prop.test(...)\$estimate, prop.test(...)\$conf.int
		BayesianFirstAid	bayes.prop.test
Difference between counts or rates	rate ratio	base	poisson.test(...)\$estimate, poisson.test(...)\$conf.int
	rate ratio	BayesianFirstAid	bayes.prop.test
Standardized group differences	Cliff's Δ	orrdom	dmes.boot with theta.es="dc"
	Vargha-Delaney's A	orrdom	dmes.boot with theta.es="Ac"

Differences in variability

Type	Package	Function
Variance ratio	base	var.test(...)\$estimate, var(...).test\$conf.int
	BayesianFirstAid	<i>not yet implemented</i>
Difference between variances	asympTest	asymp.test(...)\$estimate, asymp.test(...)\$conf.int with parameter="dVar"

Similarities and Associations

Type	Statistic	Package	Function
Correlation	Pearson's r	base	<code>cor, cor.test(...)\$conf.int</code>
		BayesianFirstAid	<code>bayes.cor.test</code>
		bootES	<code>bootES</code> with <code>effect.type="r"</code>
Kendall's τ -b		pysch	<code>cor.ci</code> with <code>method="kendall"</code>
		boot	<i>function in recipe below</i>
	Spearman's r_s	pysch	<code>cor.ci</code> with <code>method="spearman"</code>
Partial correlation		boot	<i>function in recipe below</i>
		psych	<code>corr.test, partial.r</code>
		psych	<code>polychoric</code>
Polyserial correlation (numeric/ordinal)		polycor	<code>polyserial</code>
Odds ratio	OR	psych	<code>oddsratio</code>
Standardized odds ratio	Yule's Q	psych	<code>Yule</code>
Comparisons of agreement	Cohen's κ	psych	<code>cohen.kappa</code>
Regression coefficient	β	base	<code>lm, confint</code>

Effect sizes: Measuring *differences* between groups

Perhaps the most common question an analyst will get from a business decision maker is whether or not there is a difference between groupings. There are a lot of ways to answer that, of course, but the bread-and-butter answer should always be based on effect sizes.

Why not use *p*-values?

If I had a dollar for every time a clinician or manager asked for “statistical significance,” I’d be rich. No doubt you have had this same request numerous times as well. But even when properly used—which almost never happens—*p*-values give a trivial answer to a question they don’t care about.

[Aschwanden 2015^a](#) provides a nice general-audience overview of this problem, while Intel engineer Charles Lambdin wrote a scathing yet entertaining take-down of the details of typical mindsets around *p*-values in a 2012 article called [*Significance tests as sorcery: Science is empirical-significance tests are not^b*](#). Both are well worth the read, as are many of the the references in the latter.

Effect sizes (and confidence intervals) get at what decision-makers really need to know. The following table shows why these, and not *p*-values, provide that knowledge:

Experiment	Prefer A (<i>n</i>)	Prefer B (<i>n</i>)	<i>p</i> -value	Effect size (for A)	95% CI
1	15	5	0.04	75%	(50%, 90%)
2	114	86	0.04	57%	(50%, 64%)
3	1,046	954	0.04	52%	(50%, 55%)
4	1,001,455	998,555	0.04	50%	(50%, 50%)

In the first half of this chapter, we’ll go through all the major effect size metrics for differences; the remainder will explore effect sizes for comparisons of agreement between groups.

Effect size measures that estimate a population value should be accompanied by an appropriate confidence or credible interval whenever possible. This is usually best done by bootstrap unless you can meet distributional assumptions, although at larger sample sizes the differences will be trivial. We’ll look at several ways to obtain CIs in this section.

We’ll continue to use the bike share data, but we’ll reshape and filter some of it for this section. We’ll use `bootES`, `orddom`, `BayesianFirstAid`, and `asympTest` to acquire confidence intervals for some of the contrasts we’ll consider.

```

# Load libraries
require(simpleboot)
require(bootES)
require(orddom)
require(asympTest)
require(reshape2)
require(dplyr)
# devtools::install_github("rasmusab/bayesian_first_aid")
require(BayesianFirstAid)

# Reshape the data
casual_workingday_use = dcast(bike_share_daily, yr~workingday,
  value.var="casual", sum)

casual_workingday_use$sum = casual_workingday_use$Yes + casual_workingday_use$No

# Filter the data into subsets
casual_notworkingday = filter(bike_share_daily, workingday == "No" &
  season == "Spring" | workingday == "No" & season == "Fall")

casual_notworking_Spring = filter(casual_notworkingday, season == "Spring")

casual_notworking_Fall = filter(casual_notworkingday, season == "Fall")

```

Basic differences

Proportions

For a difference between two proportions, we'll look at the effect of it being a working day on casual bike use in 2011 versus 2012:

```

workday_diff = prop.test(casual_workingday_use$Yes, casual_workingday_use$sum)

round(workday_diff$estimate[1] - workday_diff$estimate[2], 2)

-0.02

round(workday_diff$conf.int, 2)

-0.02 -0.01

```

Means

When we wish to describe differences in central tendency, we'll want to know whether two means or medians are different from each other. For example, to see if there's a difference in average casual bike use between Spring and Fall non-working days, we can use `t.test`:

```
# Difference in means, t-test version
casual_notworkingday_mean = t.test(casual~season, data=casual_notworkingday)

abs(casual_notworkingday_mean$estimate[1] -
  casual_notworkingday_mean$estimate[2])

636.6031

casual_notworkingday_mean$conf.int

350.6546 922.5516
```

Because the distribution isn't really normal, getting bootstrapped CIs on the difference of means is probably a better option in this case. Using the `bootES` package with 10,000 replications:

```
# Difference in means, bootstrapped version
# Your results will vary unless you set a seed
bootES(casual_notworkingday, data.col="casual", group.col="season",
       contrast=c("Fall", "Spring"), effect.type="unstandardized", R=10000)

User-specified lambdas: (Fall, Spring)
Scaled lambdas: (-1, 1)
95.00% bca Confidence Interval, 2000 replicates
Stat      CI (Low)    CI (High)   bias      SE
636.603   332.203    908.222    3.116    149.380
```

Medians

To look at a difference between medians (with a CI), use `two.boot` from the `simpleboot` package. Unfortunately, `simpleboot` doesn't use formula notation so we'll have to reuse some intermediate data frames created above:

```
# Difference in medians
diff_medians = two.boot(casual_notworking_Spring$casual,
  casual_notworking_Fall$casual, median, R=2000)

diff_medians$t0

834

diff_medians_ci = boot.ci(diff_medians, conf=0.95, type='bca')

diff_medians_ci

BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 2000 bootstrap replicates

CALL :
boot.ci(boot.out = diff_medians, conf = 0.95, type = "bca")

Intervals :
Level      BCa
95%   ( 480.0, 1277.8 )
Calculations and Intervals on Original Scale
```

Sometimes you may want the median difference instead of the difference between medians. Use `wilcox.test` for that:

```
# Median difference
median_diff = wilcox.test(casual~season,
  data=casual_notworkingday, conf.int=TRUE)

median_diff$estimate

684.0001

median_diff$conf.int

399 997
```

Any summary statistic

`simpleboot` allows you to compare any two univariate statistics. For example, you could compare the 75th percentile of the two groups:

```
# Difference between 75th percentiles
diff_75 = two.boot(casual_notworking_Spring$casual,
  casual_notworking_Fall$casual, quantile, probs=0.75, R=10000)

diff_75$t0

75%
731.25

diff_75_ci = boot.ci(diff_medians, conf=0.95, type='bca')

diff_75_ci

BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 2000 bootstrap replicates

CALL :
boot.ci(boot.out = diff_medians, conf = 0.95, type = "bca")

Intervals :
Level      BCa
95%   ( 480.0, 1277.8 )
Calculations and Intervals on Original Scale
```

Variance

We also often want to know whether two groups differ in terms of their variances, i.e., is one group more variable than another? If both groups are normally distributed, `var.test` is appropriate to obtain the variance ratio:

```
# Variance ratio
var.test(casual_notworkingday$casual ~ casual_notworkingday$season)$estimate

ratio of variances
1.105087

var.test(casual_notworkingday$casual ~ casual_notworkingday$season)$conf.int

0.6498105 1.8817627
```

More typically, our groups are not normally distributed, and when that's the case, the `var.test` will give misleading results because it is very sensitive to departures from normality. The `asymp.test` function in the `asympTest` package provides a suitable alternative:

```
# Robust variance ratio
asymp.test(casual_notworkingday$casual ~ casual_notworkingday$season,
  parameter = "dVar")$estimate

difference of variances
  58696.86

asymp.test(casual_notworkingday$casual ~ casual_notworkingday$season,
  parameter = "dVar")$conf.int

-197619.6 315013.4
```

Standardized differences

While not always easily explained to business users, standardized effect sizes are often very useful for analysts. When comparing means, Hedge's g can be used for most cases (as it is an improvement on Cohen's d , its more famous predecessor), while a robust version of Cohen's d can be acquired by setting the `effect.type` option to "akp.robust.d".

```
bootES(casual_notworkingday, data.col="casual", group.col="season",
  contrast=c("Fall", "Spring"), effect.type="hedges.g")
```

```
User-specified lambdas: (Fall, Spring)
Scaled lambdas: (-1, 1)
95.00% bca Confidence Interval, 2000 replicates
Stat      CI (Low)    CI (High)   bias      SE
0.825     0.374       1.252       0.004     0.219
```

```
bootES(casual_notworkingday, data.col="casual", group.col="season",
  contrast=c("Fall", "Spring"), effect.type="akp.robust.d")
```

```
User-specified lambdas: (Fall, Spring)
Scaled lambdas: (-1, 1)
95.00% bca Confidence Interval, 2000 replicates
Stat      CI (Low)    CI (High)   bias      SE
0.866     0.433       1.406       0.039     0.250
```

When you don't want to make any distributional assumptions, Cliff's Δ and Vargha-Delaney's A (which is the same as the AUC statistic applied to a two-group contrast) are the best options, and, in fact, are simply linear transformations of each other. Both can be obtained from the `orddom` package, with CIs obtained via BCa by default:

```
dmes.boot(casual_notworking_Fall$casual, casual_notworking_Spring$casual,  
theta.es="dc")$theta
```

dc

0.4555138

```
dmes.boot(casual_notworking_Fall$casual, casual_notworking_Spring$casual,  
theta.es="dc")$theta.bci.lo
```

0.2493734

```
dmes.boot(casual_notworking_Fall$casual, casual_notworking_Spring$casual,  
theta.es="dc")$theta.bci.up
```

0.6265664

```
dmes.boot(casual_notworking_Fall$casual, casual_notworking_Spring$casual,  
theta.es="Ac")$theta
```

Ac

0.7277569

```
dmes.boot(casual_notworking_Fall$casual, casual_notworking_Spring$casual,  
theta.es="Ac")$theta.bci.lo
```

0.6246867

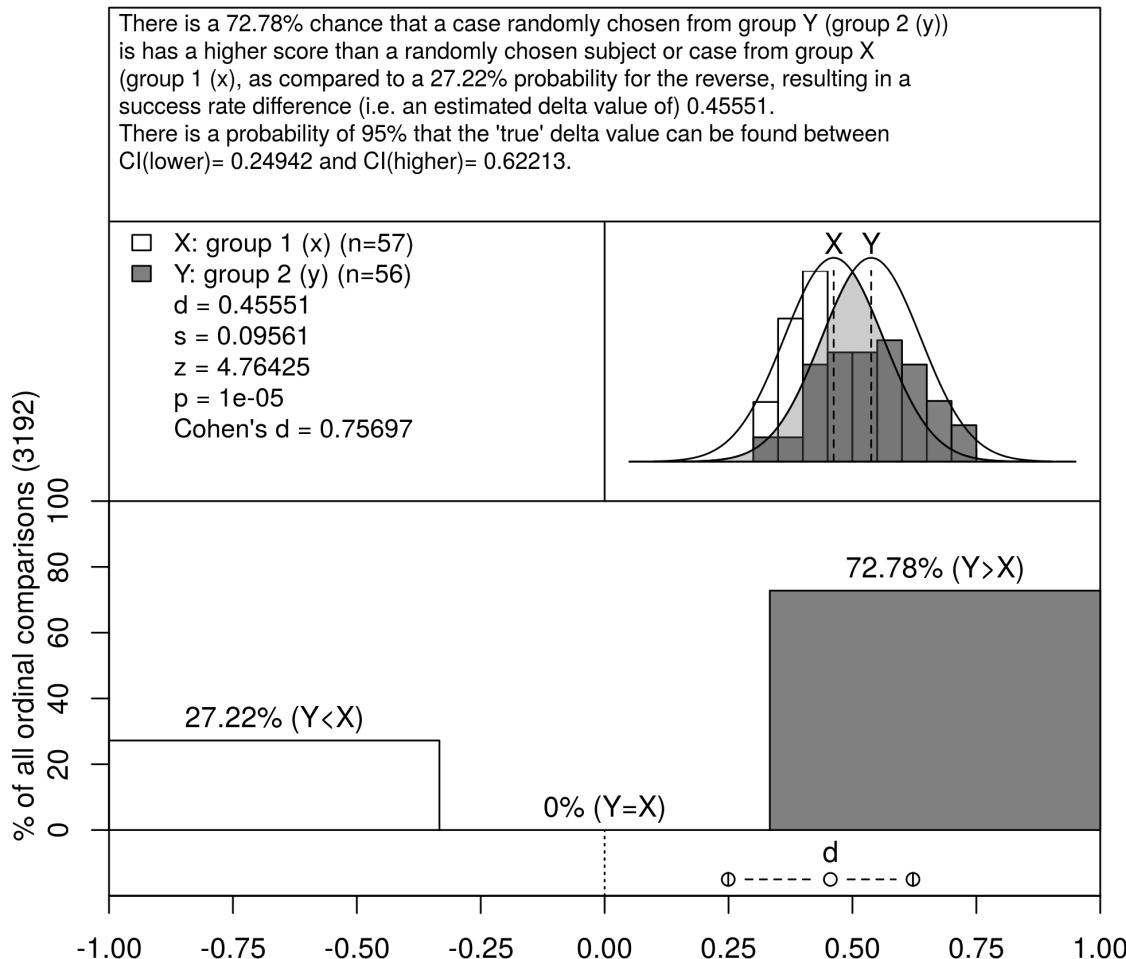
```
dmes.boot(casual_notworking_Fall$casual, casual_notworking_Spring$casual,  
theta.es="Ac")$theta.bci.up
```

0.8132832

This package also has a neat feature in that it will plot the results and summary of a Cliff's Δ assessment with the delta_gr function:

```
delta_gr(casual_notworking_Fall$casual, casual_notworking_Spring$casual,  
x.name="Fall", y.name="Spring")
```

Cliff's delta & 95% Confidence Interval (2-tailed)



Determining the probability of a difference

The rapid increase in computing power in recent years has put previously-intractable analytics tools at anyone's fingertips. Markov-chain Monte Carlo (MCMC) is one of the breakout stars of this revolution, which provides the mathematical basis for the resurgence of interest in Bayesian methods.

You'll need to install [JAGS⁴⁶](#) to your system before starting, and also install the R package `rjags` to allow R to converse with JAGS.

`BayesianFirstAid` provides one-line Bayesian alternatives to common R functions for CIs ("credible intervals") and basic effect sizes. It's in development at the time of this writing and needs to be installed from GitHub if you haven't already done so:

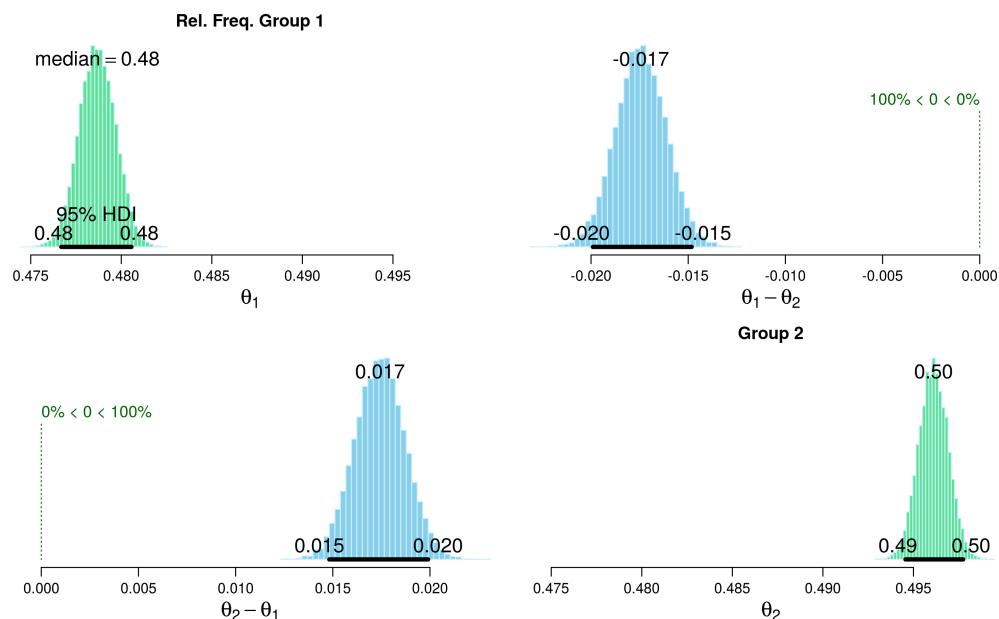
⁴⁶<http://mcmc-jags.sourceforge.net/>

```
devtools::install_github("rasmusab/bayesian_first_aid")
require(BayesianFirstAid)
```

Using the earlier example above for the differences in proportions, we get the same sort of results (i.e., the effect size and CIs), as well as—most importantly—a plot of the posterior distributions:

```
workday_diff_bayes = bayes.prop.test(casual_workingday_use$Yes,
casual_workingday_use$sum)
```

```
plot(workday_diff_bayes)
```



```
workday_diff_bayes
```

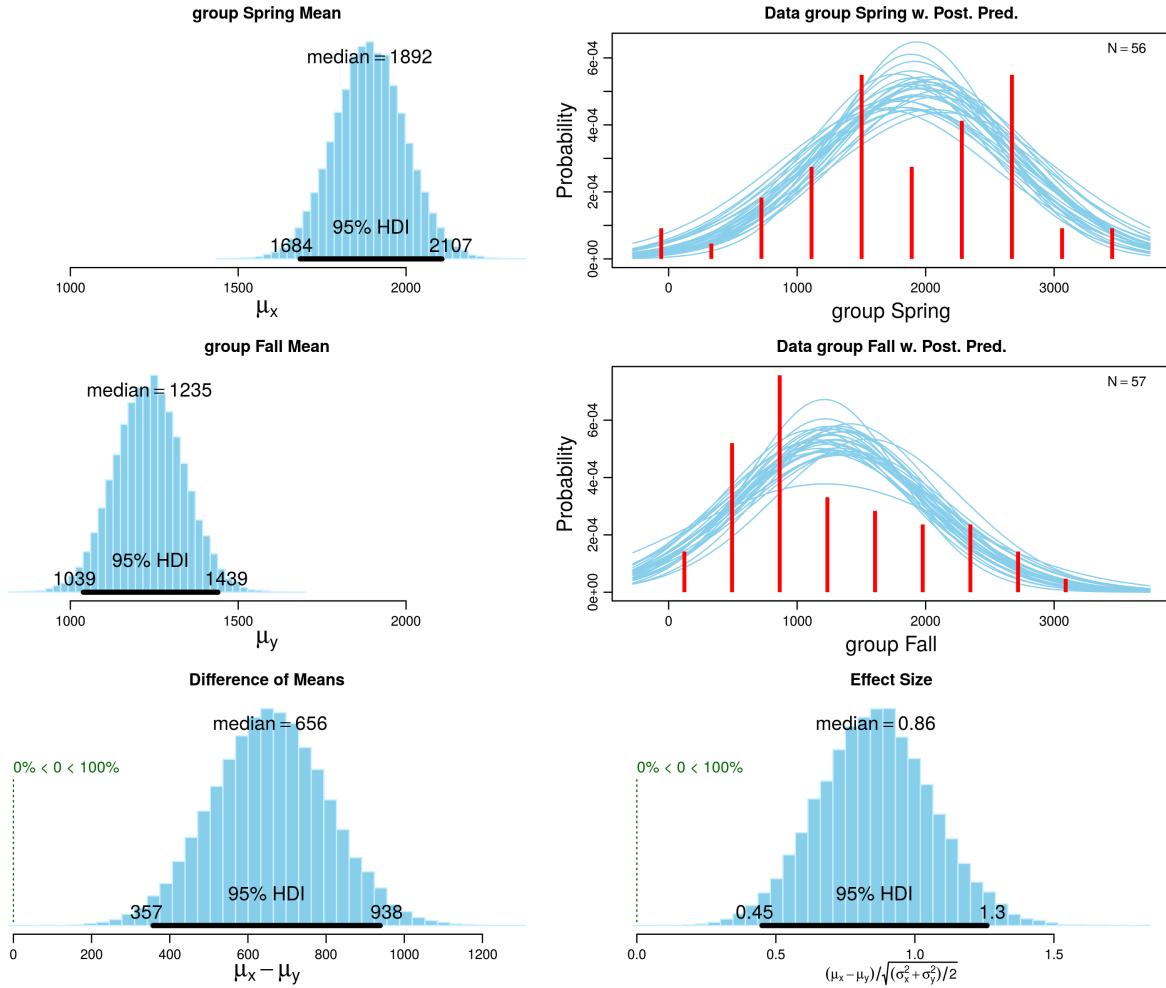
Bayesian First Aid proportion test

```
data: casual_workingday_use$Yes out of casual_workingday_use$sum
number of successes: 118354, 184931
number of trials: 247252, 372765
Estimated relative frequency of success [95% credible interval]:
Group 1: 0.48 [0.48, 0.48]
Group 2: 0.50 [0.49, 0.50]
Estimated group difference (Group 1 - Group 2):
-0.02 [-0.02, -0.015]
The relative frequency of success is larger for Group 1 by a probability
of <0.001 and larger for Group 2 by a probability of >0.999 .
```

To look at the difference between means of two groups, we can use the `bayes.t.test` function:

```
casual_notworkingday_mean_bayes = bayes.t.test(casual~season,
data=casual_notworkingday)
```

```
plot(casual_notworkingday_mean_bayes)
```



```
casual_notworkingday_mean_bayes
```

```
Bayesian estimation supersedes the t test (BEST) - two sample

data: group Spring (n = 56) and group Fall (n = 57)

Estimates [95% credible interval]
mean of group Spring: 1892 [1684, 2107]
mean of group Fall: 1235 [1039, 1439]
difference of the means: 656 [357, 938]
sd of group Spring: 780 [637, 949]
sd of group Fall: 742 [609, 904]
```

The difference of the means is greater than 0 by a probability of >0.999
and less than 0 by a probability of <0.001

The package allows the use of `summary` for its output; with it we can see the CIs for each statistic as well as the standardized effect size (`eff_size`, d) on this contrast:

```
summary(casual_notworkingday_mean_bayes)
```

Measures

	mean	sd	HDIlo	HDIup	%<comp	%>comp
mu_x	1891.896	107.961	1684.426	2107.064	0.000	1.000
sigma_x	784.985	80.819	636.610	949.354	0.000	1.000
mu_y	1235.311	103.061	1038.706	1439.451	0.000	1.000
sigma_y	747.461	76.146	609.178	904.010	0.000	1.000
mu_diff	656.585	148.684	357.229	938.414	0.000	1.000
sigma_diff	37.523	109.254	-181.929	249.049	0.360	0.640
nu	46.196	32.717	5.557	110.800	0.000	1.000
eff_size	0.860	0.206	0.450	1.259	0.000	1.000
x_pred	1891.952	827.714	307.633	3559.847	0.012	0.988
y_pred	1231.742	781.921	-290.477	2794.578	0.056	0.944

'HDIlo' and 'HDIup' are the limits of a 95% HDI credible interval.

'%<comp' and '%>comp' are the probabilities of the respective parameter being smaller or larger than 0.

Quantiles

	q2.5%	q25%	median	q75%	q97.5%
mu_x	1680.655	1820.233	1891.884	1963.547	2103.933
sigma_x	642.575	728.117	779.837	835.775	958.585

	1035.024	1165.506	1235.177	1304.515	1437.405
mu_y	614.269	694.245	741.870	794.897	912.781
sigma_y	365.089	556.231	656.493	756.053	947.740
mu_diff	-178.064	-33.623	37.084	108.491	253.580
sigma_diff	9.355	23.275	37.863	60.040	128.710
nu	0.461	0.720	0.859	0.996	1.271
eff_size	264.184	1346.469	1892.044	2432.797	3520.282
x_pred	-309.647	721.162	1225.480	1743.096	2781.779
y_pred					

Evaluations of the difference in median and variance contrasts (among others) are planned but not yet implemented. Follow it on [GitHub⁴⁷](#) to check for updates.

Finally, you can review the MCMC diagnostics for any of these functions by using `diagnostics`:

```
diagnostics(casual_notworkingday_mean_bayes)
```

```
Iterations = 601:10600
Thinning interval = 1
Number of chains = 3
Sample size per chain = 10000
```

```
Diagnostic measures
      mean        sd mcmc_se n_eff  Rhat
mu_x     1891.532 109.652  0.815 18162 1.000
sigma_x    786.383  81.211  0.681 14276 1.000
mu_y     1237.637 103.310  0.781 17532 1.000
sigma_y    747.915  76.431  0.618 15346 1.001
mu_diff   653.895 150.729  1.113 18408 1.001
sigma_diff 38.468 110.067  0.868 16083 1.000
nu        46.731  32.070  0.389  6879 1.002
eff_size    0.855  0.208  0.002 16886 1.001
x_pred     1884.072 827.166  4.776 29999 1.000
y_pred     1240.818 784.492  4.552 29712 1.000
```

`mcmc_se`: the estimated standard error of the MCMC approximation of the mean.

`n_eff`: a crude measure of effective MCMC sample size.

`Rhat`: the potential scale reduction factor (at convergence, `Rhat=1`).

Model parameters and generated quantities

`mu_x`: the mean of group Spring

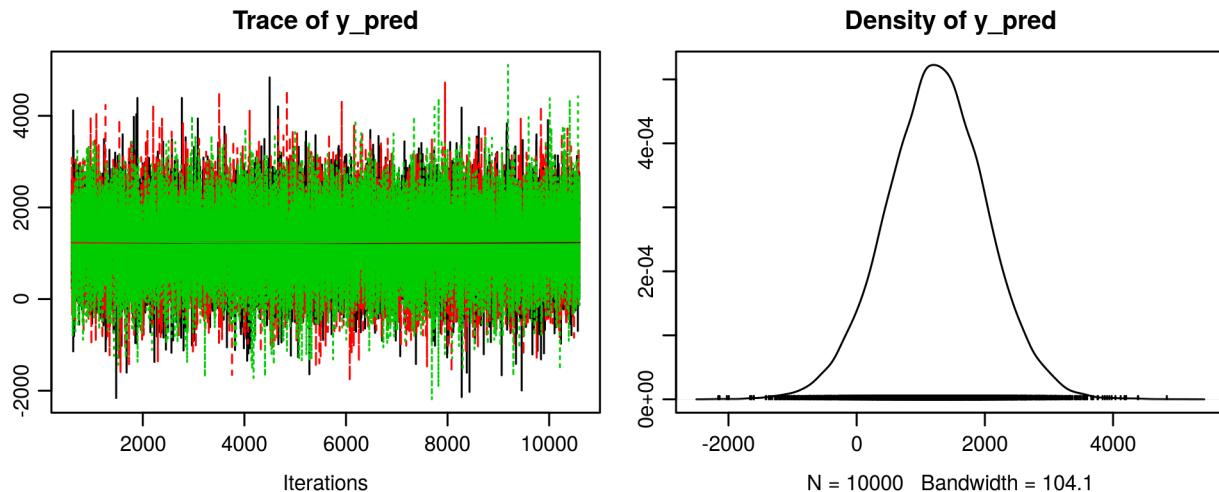
`sigma_x`: the scale of group Spring , a consistent

⁴⁷https://github.com/rasmusab/bayesian_first_aid

estimate of SD when nu is large.
`mu_y`: the mean of group Fall
`sigma_y`: the scale of group Fall
`mu_diff`: the difference in means ($\mu_x - \mu_y$)
`sigma_diff`: the difference in scale ($\sigma_x - \sigma_y$)
`nu`: the degrees-of-freedom for the t distribution
 fitted to casual by season
`eff_size`: the effect size calculated as

$$(\mu_x - \mu_y) / \sqrt{(\sigma_x^2 + \sigma_y^2) / 2}$$

`x_pred`: predicted distribution for a new datapoint
 generated as group Spring
`y_pred`: predicted distribution for a new datapoint
 generated as group Fall



Effect sizes: Measuring *similarities* between groups

The other basic analytics question aims to understand relationships: how similar are two groups? Correlation is the primary (standardized) tool here, although a wide variety of other approaches exist; of particular use are regression coefficients to answer the question: what's the incremental effect of one variable on another?

Correlation

Correlation is one of the oldest and most well-known statistical tools, and for good reason: it provides a lot of detail in a single metric. While Pearson's correlation (r) is often the default approach by

many analysts, it's not always the best—in many cases, Spearman's rho (r_s), Kendall's tau-b (τ -b), or polyserial/polychoric approaches are better choices.

When you have categorical data, using association coefficients like the odds ratio, Yule's Q, or Cohen's kappa (κ) are appropriate tools, and are just as useful (and old) as correlation. We'll explore how you can acquire each of these values with R in this section.

As before, every effect size measure should be accompanied by an appropriate credible or confidence interval whenever possible, which is usually best done by bootstrap unless you can meet the method's assumptions.

Although correlation coefficients are easy to acquire with R's base installation, anything other than confidence/credible intervals are not, so we'll use the `psych` package for the non-parametric correlation part of this recipe and `boot/boot.ci/bootES` and `BayesianFirstAid` for those intervals. We'll use a different piece of the bike share dataset to illustrate these methods.

```
require(psych)
require(bootES)

# Use count and air temp variables from bike share data
bike_use_atemp = data.frame(air_temp = bike_share_daily$atemp,
  count = bike_share_daily$cnt)
```

Traditional Pearson's correlation (r) is easy to acquire:

```
cor(bike_use_atemp$air_temp, bike_use_atemp$count)
```

```
0.6310657
```

```
cor.test(bike_use_atemp$air_temp, bike_use_atemp$count)$conf.int
```

```
0.5853376 0.6727918
```

You can get BCa bootstrap CIs for Pearson's r by using:

```
bootES(c(bike_use_atemp$air_temp, bike_use_atemp$count), effect.type="r")
```

```
95.00% bca Confidence Interval, 2000 replicates
Stat      CI (Low)    CI (High)   bias      SE
0.650     0.633      0.667      -0.000    0.009
```

However, it's pretty clear that this relationship isn't truly linear if you plot it. Kendall's and Spearman's can be acquired with the `cor.ci` function; use the `*.emp`—empirical—CI values to obtain the percentile confidence interval values, although the differences will be trivial with large sample sizes:

```
# Kendall's (which should generally be preferred)
cor.ci(bike_use_atemp, method="kendall", n.iter = 10000, plot=FALSE)

Coefficients and bootstrapped confidence intervals
      ar_tm count
air_temp 1.00
count     0.43  1.00

scale correlations and bootstrapped confidence intervals
      lower.emp lower.norm estimate upper.norm upper.emp p
ar_tm-count      0.39       0.39      0.43      0.47      0.47 0

# Spearman's
cor.ci(bike_use_atemp, method="spearman", n.iter = 10000, plot=FALSE)

Coefficients and bootstrapped confidence intervals
      ar_tm count
air_temp 1.00
count     0.62  1.00

scale correlations and bootstrapped confidence intervals
      lower.emp lower.norm estimate upper.norm upper.emp p
ar_tm-count      0.57       0.57      0.62      0.67      0.67 0
```

Kendall's vs. Spearman's: What's the difference?

In short, they measure different things, although both can be used on numeric as well as ordinal scales. Spearman's is meant to measure the monotonic association between two variables, and is often used in place of Pearson's when the relationship is non-linear (but still monotonic) and where outliers or other factors are skewing the relationship away from *bivariate* normality (the individual variables can be any distribution). Kendall's has a huge advantage in that it can be used when an association has a trend that is not monotonic, and also has some superior statistical properties we'll get to in a moment.

Historically, Spearman's was often used because it was easier to calculate and it mimicked the least-squares approach of Pearson's. Perhaps somewhat cynically, I'd also bet Spearman's was often used instead of Kendall's because it gives you higher values, which when you're used to interpreting r makes your result look better. Finally, explaining Spearman's "as simply an extension of Pearson's r " can be far easier than having to explain Kendall's in a business meeting.

Computers have since rendered the first reason irrelevant, and statisticians have determined the second one to be statistically naive: Spearman's is less robust and less efficient than Kendall's approach, especially when trying to compute standard errors—there is still no widely accepted

way to do it with Spearman's coefficient. Kendall's coefficient estimates a population value, while Spearman's does not. Spearman's approach also breaks down in the presence of only a few large deviations between pairs of values, even when most of the pairs in the data are fairly close. In addition, there is no intuitive interpretation of what Spearman's coefficient *really* means (only that “further from zero is better”), while Kendall's can be directly interpreted as a probability.

As for the third and fourth reasons... well, as analysts we should be honest and not use Spearman's or any other statistic just because it looks better (you're just *asking* for trouble down the road). Off-the-cuff explanation of Kendall's in business settings *is* tough, though a simple transformation can help: using the formula: $\tau + (0.5 * (1 - \tau))$. For example, using the Kendall's τ value above— $0.43 + (0.5 * (1 - 0.43))$ —we can determine that there is a positive association between 71% of all possible pairs of data points. Still not as easy as dropping Spearman's coefficient and leaving it to the audience, but hey, it's your choice as an analyst when it's worth it to “move the needle”^a of managerial stats literacy...

^aI can't believe I just wrote that without irony (or quadrature).

Bootstrapping BCa CIs for non-parametric correlation

If you want to use the `boot` package to calculate BCa confidence intervals on Spearman's or Kendall's correlation coefficients, here's how to acquire them for each method:

```
# Boot function for Kendall's CI
rt_function = function(x,i){cor(x[i,1], x[i,2], method="kendall")}

# Run boot function for Kendall's CI
rt_boot = boot(bike_use_atemp, rt_function, R=10000)

# Kendall's CI
boot.ci(rt_boot, type="bca")$bca[4:5]

0.3931281 0.4665541

# Spearman's
rs_function = function(x,i){cor(x[i,1], x[i,2], method="spearman")}

rs_boot = boot(bike_use_atemp, rs_function, R=10000)

boot.ci(rs_boot, type="bca")$bca[4:5]

0.5748000 0.6669792
```

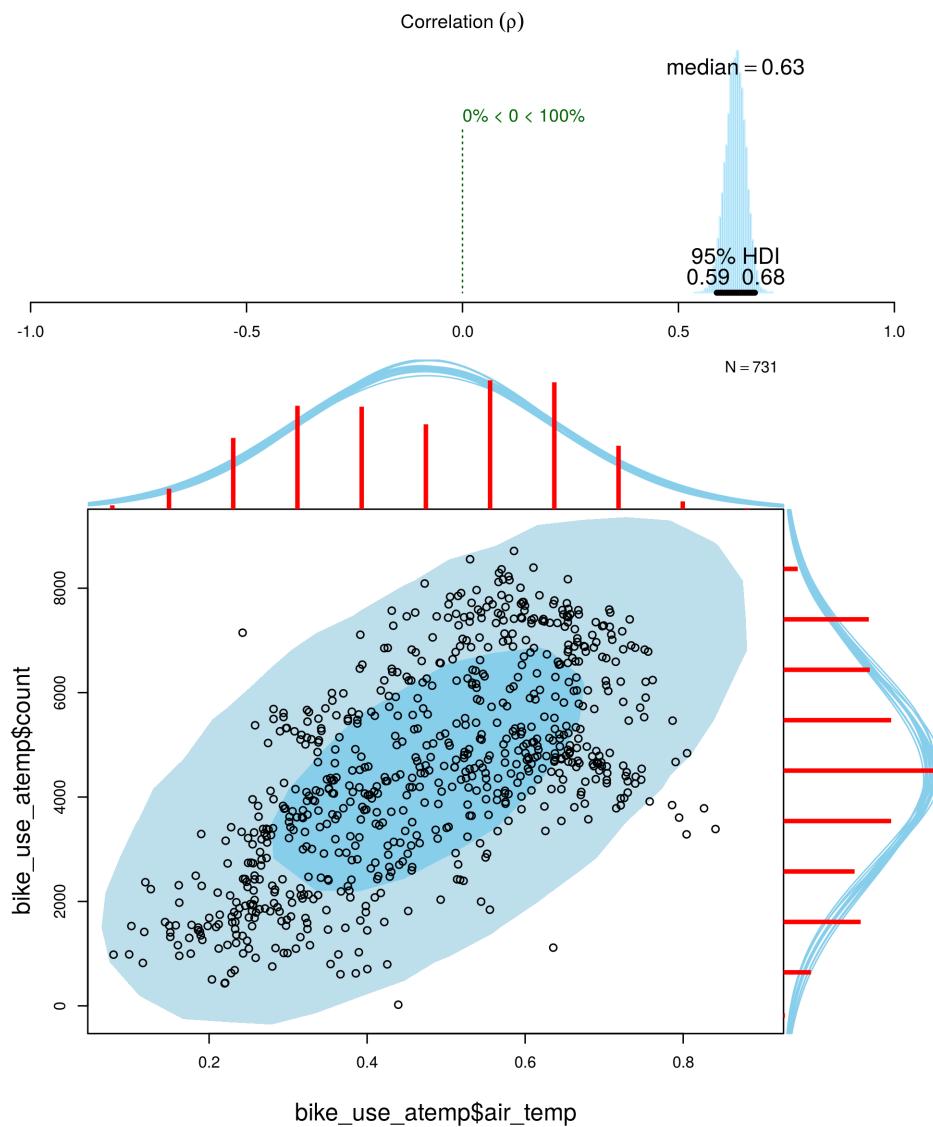
Determining the probability of a correlation

We saw the Bayesian approach to difference-based effect size calculations earlier in this chapter; here's the approach for correlation:

```
require(BayesianFirstAid)
```

```
atemp_bike_cor_bayes = bayes.cor.test(bike_use_atemp$air_temp,
                                         bike_use_atemp$count)
```

```
plot(atemp_bike_cor_bayes)
```



```
atemp_bike_cor_bayes
```

```
Bayesian First Aid Pearson's Correlation Coefficient Test

data: bike_use_atemp$air_temp and bike_use_atemp$count (n = 731)
Estimated correlation:
  0.63
95% credible interval:
  0.59 0.68
The correlation is more than 0 by a probability of >0.999
and less than 0 by a probability of <0.001
```

Partial correlations

If you have variables you want to control for in a correlation, we can turn again to the psych package. For example, if we wanted to evaluate the correlation between air temperature and bike use while controlling for windspeed, we could do it by creating a correlation matrix with the `corr.test` function:

```
# Subset bike share data data
bike_use_atemp_wind = data.frame(temp = bike_share_daily$temp,
                                   cnt = bike_share_daily$cnt, windspeed = bike_share_daily$windspeed )

# Acquire correlation matrix
atemp_wind_count = corr.test(bike_use_atemp_wind, method="kendall")

# Review matrix
atemp_wind_count$r

      temp        cnt     windspeed
temp 1.00000000  0.4320092 -0.09511844
cnt   0.43200921  1.0000000 -0.14711015
windspeed -0.09511844 -0.1471101  1.00000000
```

Once we have a matrix, we can plug it into the `partial.r` function, specifying that the first two variables of the matrix are the correlation of interest, and the third is the variable we wish to control for (the second and third inputs, respectively):

```
# Obtain partial r
partial.r(as.matrix(atemp_wind_count$r), c(1:2), 3)

partial correlations
    temp   cnt
temp 1.00 0.42
cnt   0.42 1.00
```

Polychoric and polyserial correlation for ordinal data

Polychoric (ordinal-ordinal) and polyserial (numeric-ordinal) correlation allows you to perform correlation when some or all of your variables of interest are on an ordinal scale, particularly if they're likert-type variables. Again, psych provides the functions we'll use. To illustrate, we'll use the math attitudes survey (mass) data seen in the last chapter, converted from ordered factor to numeric for the polychoric function to work:

```
# Get math attitudes data
data(mass, package="likert")

# Subset and convert to numeric
poly_math = data.frame(as.numeric(mass[,7]), as.numeric(mass[,14]))

# Name columns
colnames(poly_math) = c("worry", "enjoy")

# Obtain polychoric correlation
polychoric(poly_math)

Call: polychoric(x = poly_math)
Polychoric correlations
    worry enjoy
worry  1.00
enjoy -0.77  1.00

with tau of
      1     2     3     4
worry -0.84 -0.25  0.25  1.3
enjoy -1.04 -0.25  0.52  1.3
```

To perform a polyserial correlation, we'll add a fake variable representing each student's pre-college math assessment test score to see whether it relates to their stated enjoyment. At the time of this writing, there's a bug in the psych package's implementation, but the polycor package (which is

an indirect suggested dependency of psych, so they probably installed together) provides a working version.

```
# Made up math scores
math_score = c(755, 642, 626, 671, 578, 539, 769, 614, 550, 615, 749, 676, 753,
509, 798, 783, 508, 767, 738, 660)

# Obtain polyserial correlation using the polycor package
polycor::polyserial(math_score, poly_math$enjoy)

0.2664201
```

Associations between categorical variables

Perhaps the most famous association metric is the odds ratio, which is really useful but can be misleading for those not used to its mathematical properties. Yule's Q has more desirable properties in many cases, especially in explaining results to business users: it provides the same type of information as correlation does for quantitative data, that is, a value scaled to [-1,1]. Cohen's κ is probably the best tool when you want to compare *agreement* instead of *association*.

Since the bike share data doesn't lend itself well to exploring the association of categorical variables, we'll explore "categorical correlation" with the Aspirin data from the abd package. The functions we'll use include the oddsratio function from the epitools package, and the Yule (Yule's Q) function from psych:

```
require(epitools)
require(psych)

data(Aspirin, package="abd")

# Obtain the odds ratio and CI
oddsratio(table(Aspirin))$measure

      odds ratio with 95% C.I.
treatment estimate    lower    upper
  Aspirin  1.0000000    NA      NA
  Placebo  0.9913098  0.9187395 1.069632

# Obtain Yule's Q
Yule(table(Aspirin))

-0.004352794
```

Cohen's kappa for comparisons of agreement

Ratings on ordinal scales require special treatment, since they aren't true numbers, and comparisons of agreement make it even more necessary to choose a statistically-appropriate tool. Cohen's κ (cohen.kappa in the psych package) provides a way to compare ordinal ratings, e.g., a doctor's ratings for a set of patients potentially eligible for care management as compared with the same output from a predictive model:

```
# Doctor ratings
doctor = c("yes", "no", "yes", "unsure", "yes", "no", "unsure", "no", "no",
          "yes", "no", "yes", "yes")

# Model ratings
model = c("yes", "yes", "unsure", "yes", "no", "no", "unsure", "no", "unsure",
          "no", "yes", "yes", "unsure")

# Obtain Cohen's kappa
cohen.kappa(x=cbind(doctor, model))
```

Cohen Kappa and Weighted Kappa correlation coefficients and confidence boundaries

	lower	estimate	upper
unweighted kappa	-0.15	0.22	0.59
weighted kappa	-0.45	0.15	0.75
Number of subjects	= 12		

The \$agree output value shows the proportion of agreement for each category:

```
# Category proportion of agreement
cohen.kappa(x=cbind(doctor, model))$agree

x2f
x1f      no    unsure    yes
no    0.15384615 0.07692308 0.15384615
unsure 0.00000000 0.07692308 0.07692308
yes   0.15384615 0.15384615 0.15384615
```

Regression coefficient

Although technically the use of regression implies causality, there are times when you just wish to know the impact of one variable on its relationship with another. When the “predictor” is categorical, the regression coefficient is simply the difference in means, seen at the start of this

chapter (`lm(casual ~ season, data=casual_notworkingday)`). When both variables are numeric, the coefficient is the change in the y-variable for each unit of the x-variable, so regardless of which variable you choose to be the descriptor, you can see the effect size.

We'll use the `tao` dataset seen in the previous chapter to explore the effect of air temperature on sea surface temperature using the `lm` function.

```
data(tao, package="VIM")

# run the linear model
effect_air_on_sea = lm(Sea.Surface.Temp ~ Air.Temp, data=tao)

# review model coefficients
effect_air_on_sea

Call:
lm(formula = Sea.Surface.Temp ~ Air.Temp, data = tao)

Coefficients:
(Intercept)      Air.Temp
              -3.867          1.176

# get 95% confidence interval
confint(effect_air_on_sea)

             2.5 %    97.5 %
(Intercept) -4.320256 -3.412817
Air.Temp     1.157677  1.193817
```

We can see that for every 1 degree increase in air temperature, we get an increase of about 1.18 degrees in sea surface temperature (95% CI [1.16, 1.19]):

R²: Proportion of variance explained

The R^2 effect size is simply the correlation coefficient squared, of course, but oddly enough there does not seem to be a built-in function to readily obtain the CIs for it. We can use the `boot` package to create one, thanks to a function by Rob Kabacoff⁴⁸:

⁴⁸<http://www.statmethods.net/advstats/bootstrapping.html>

```
library(boot)

# R-squared boot function
rsq = function(formula, data, indices) {
  d = data[indices,] # allows boot to select sample
  fit = lm(formula, data=d)
  return(summary(fit)$r.square)
}

# bootstrap R2 with 10k replications
air_temp_R2 = boot(data=tao, statistic=rsq, R=10000,
  formula=Sea.Surface.Temp ~ Air.Temp)

# view bootstrapped R2 results
air_temp_R2

ORDINARY NONPARAMETRIC BOOTSTRAP

Call:
boot(data = tao, statistic = rsq, R = 10000, formula = Sea.Surface.Temp ~
  Air.Temp)

Bootstrap Statistics :
      original     bias   std. error
t1* 0.9615353 4.891461e-05 0.003223984

# get 95% confidence interval
boot.ci(air_temp_R2, type="bca")

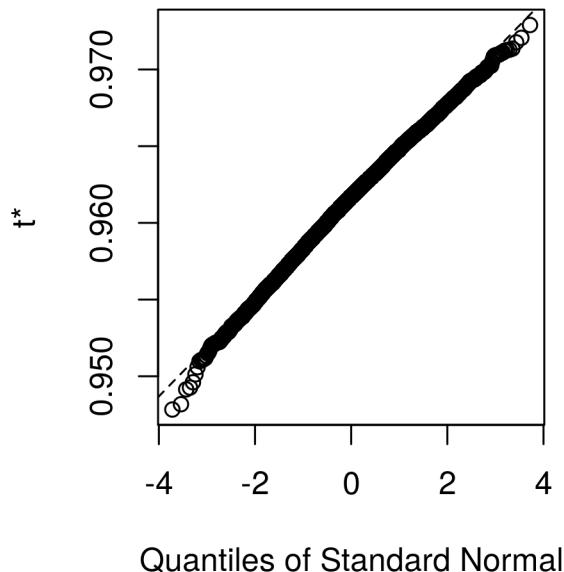
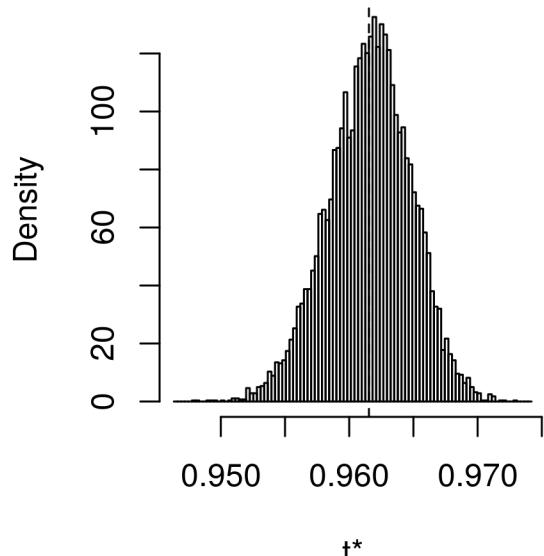
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 10000 bootstrap replicates

CALL :
boot.ci(boot.out = air_temp_R2, type = "bca")

Intervals :
Level      BCa
95%  ( 0.9541,  0.9669 )
Calculations and Intervals on Original Scale

# plot results
plot(air_temp_R2)
```

Histogram of t



R^2 by itself can be misleading; you can get a high value from pure noise, as this example illustrates:

```
set.seed(123)
y = rnorm(10)
x = sapply(rep(10,8), rnorm)
noise = lm(y ~ x)
summary(noise)$r.squared
```

0.9736913

Chapter 6: Trends and Time

- Describing trends in non-temporal data
- The many flavors of regression
- Plotting regression coefficients
- Working with temporal data
- Calculate a mean or correlation in circular time (clock time)
- Plotting time-series data
- Detecting autocorrelation
- Plotting monthly patterns
- Plotting seasonal adjustment on the fly
- Decomposing time series into components
- Using spectral analysis to identify periodicity
- Plotting survival curves
- Evaluating quality with control charts
- Identifying possible breakpoints in a time series
- Exploring relationships between time series: cross-correlation
- Basic forecasting

Trends are of fundamental interest throughout business intelligence work, but can be tricky to deal with statistically. And dealing with temporal data presents its own set of unique challenges. R lessens that burden considerably—as long as your data are in the right form.

Describing trends in non-temporal data

If you’re in the data exploration phase, and want to get a loose sense of the trend in the data, starting with a loess curve is probably the best first step. Quantile regression provides tools for evaluating trends in data with increasing or decreasing variance, or any data with a monotonic trend regardless of variance. Ordinary least squares is probably the *last* tool you want to use in the exploratory phase, though it can sometimes be what you start with when doing model building.

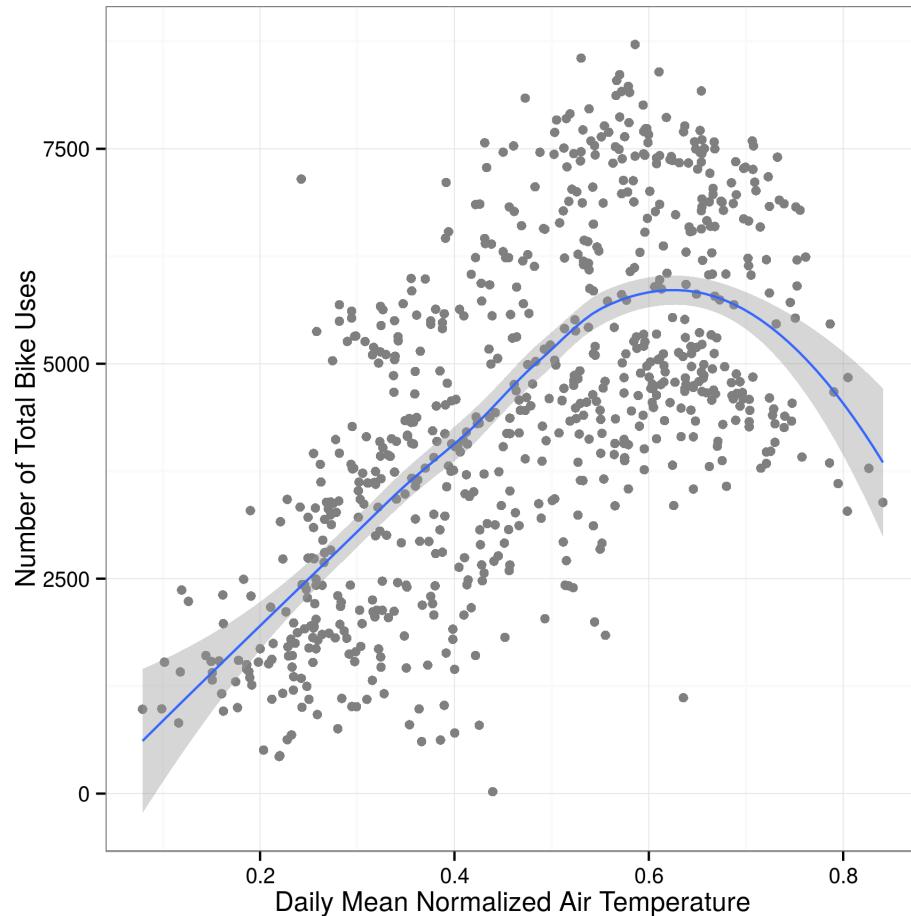
We’ll continue to use the bike share data, and plot the results with `ggplot2`. We’ll also need the `quantreg` package in order to plot the quantile regression trend lines, and the `mgcv` package to do the same with GAMs:

```
require(ggplot2)
require(quantreg)
require(mgcv)
```

Smoothed trends

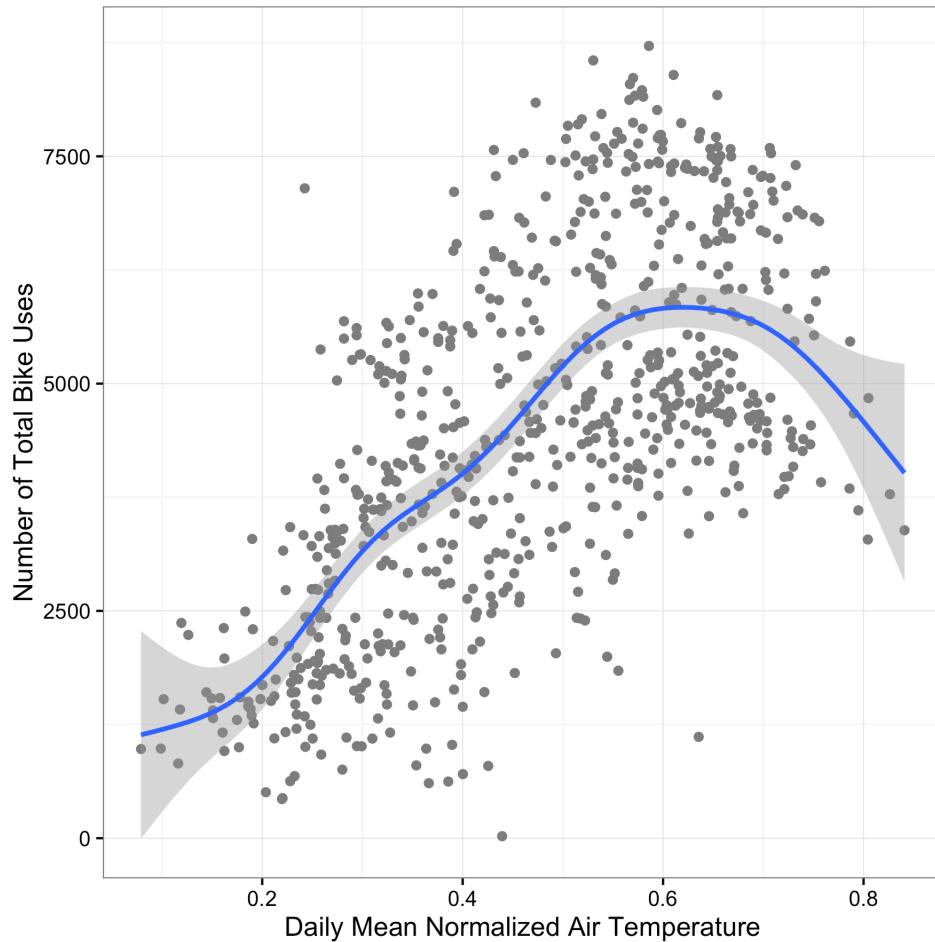
To see a smoothed average trend, use `method="loess"` in the `geom_smooth` function:

```
ggplot(bike_share_daily, aes(x=atemp, y=cnt)) +
  xlab("Daily Mean Normalized Air Temperature") +
  ylab("Number of Total Bike Uses") +
  geom_point(col="gray50") +
  geom_smooth(method="loess") +
  theme_bw()
```



`ggplot2` automatically smooths via a GAM if you have > 1,000 points. If you have fewer, and want to see the trend a GAM can plot, you'll need to change the the `geom_smooth` method *and* include the option, written just like it is here: `formula = y ~ s(x)`. Don't replace the `y` and `x` with the variable names; you'll get an error. Other GAM options can be included in the `geom_smooth` call inside the formula (e.g., `formula = y ~ s(x), k=10`).

```
ggplot(bike_share_daily, aes(x=atemp, y=cnt)) +
  xlab("Daily Mean Normalized Air Temperature") +
  ylab("Number of Total Bike Uses") +
  geom_point(col="gray50") +
  geom_smooth(method="gam", formula=y~s(x)) +
  theme_bw()
```

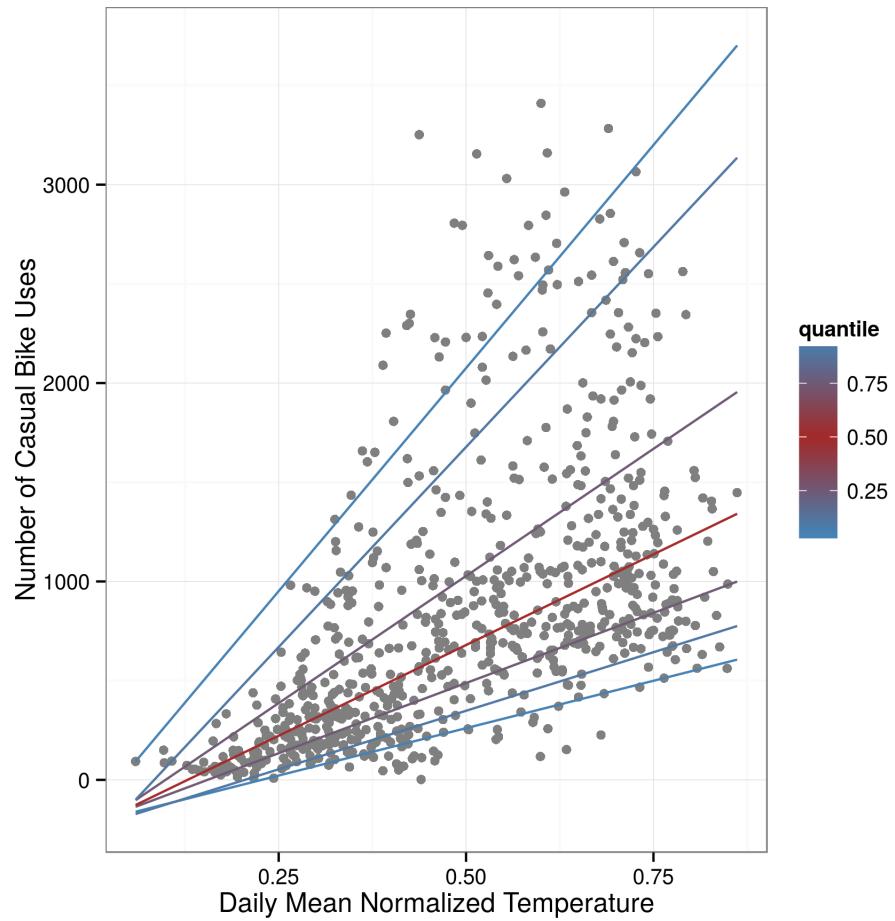


Quantile trends

To show trends based on quantiles, use the `stat_quantile` option, and choose which quantiles you want to show, either specifically (as below) or sequentially (e.g., `quantiles = seq(0.05, 0.95,`

by=0.05):

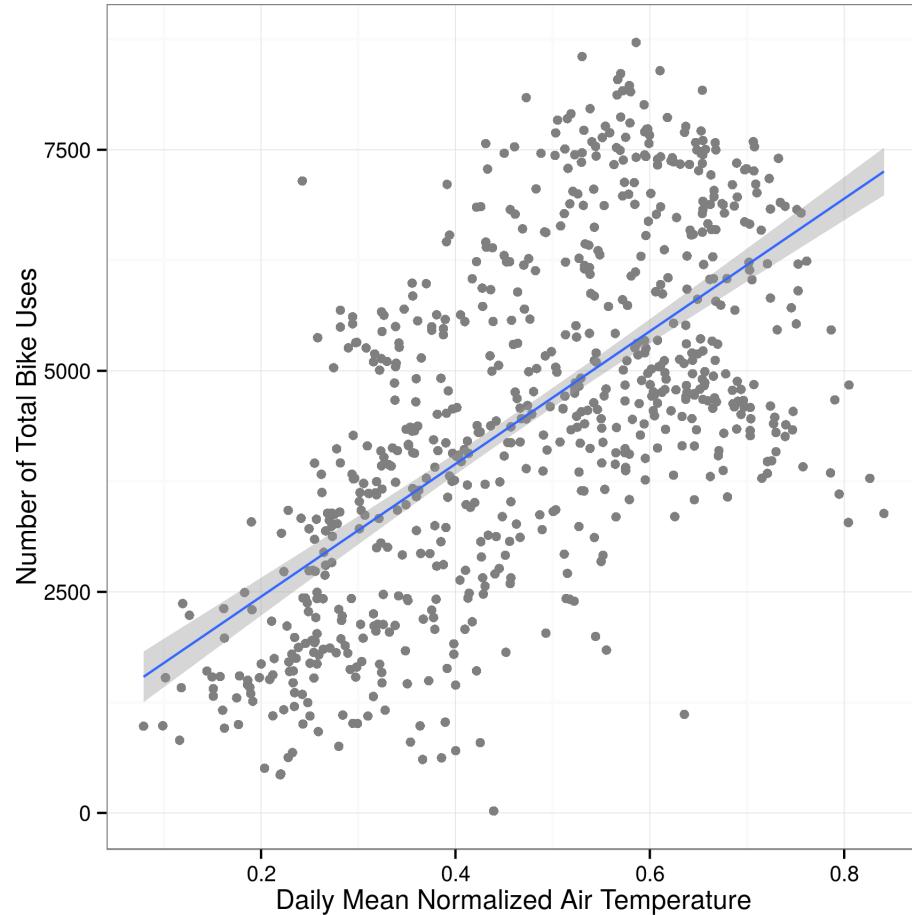
```
ggplot(bike_share_daily, aes(x=temp, y=casual)) +
  xlab("Daily Mean Normalized Temperature") +
  ylab("Number of Casual Bike Uses") +
  geom_point(col="gray50") +
  stat_quantile(aes(color = ..quantile..), quantiles = c(0.05, 0.1, 0.25,
  0.5, 0.75, 0.9, .95)) +
  scale_color_gradient2(midpoint=0.5, low="steelblue", mid="brown",
  high="steelblue ") +
  theme_bw()
```



Simple linear trends

Finally, to get a simple linear trend, use the `lm` method with `geom_smooth`:

```
ggplot(bike_share_daily, aes(x=atemp, y=cnt)) +
  xlab("Daily Mean Normalized Air Temperature") +
  ylab("Number of Total Bike Uses") +
  geom_point(col="gray50") +
  geom_smooth(method="lm") +
  theme_bw()
```



If you're exploring trends, it's good habit to *not* start with an OLS/linear trend, as in real data exploration you usually don't have any *a priori* knowledge about the functional relationship or trend. Only using this approach after you've tried the others helps prevent starting your thinking into a (linear) rut.

Segmented linear trends

Sometimes you do want to see a linear trend, but know (or suspect) that there are changes in the trend. We'll explore ways to evaluate changepoints later, but for now we'll just quickly plot

a segmented linear trend line. We'll allow the `segmented` package to choose the optimal breakpoint for us, after we specify approximately where we think it is with the `psi` option:

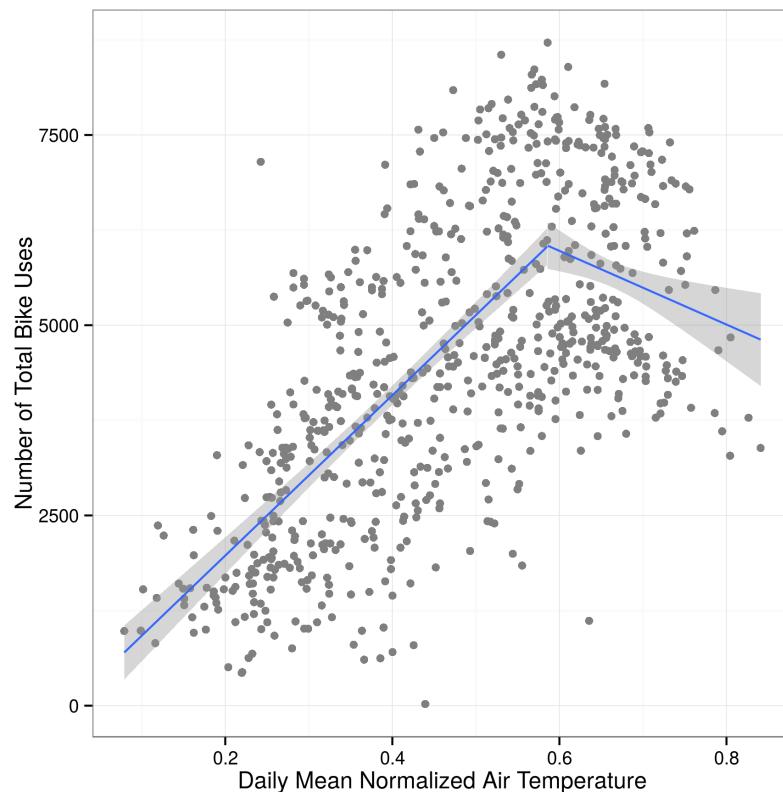
```
require(segmented)

bike_segment = segmented(lm(cnt~atemp, data=bike_share_daily), ~atemp, psi=0.6)
bike_segment$psi

      Initial      Est.      St.Err
psi1.atemp    0.6  0.585762  0.01352518

psi = bike_segment$psi[2]

ggplot(bike_share_daily, aes(x=atemp, y=cnt, group = atemp > psi)) +
  xlab("Daily Mean Normalized Air Temperature") +
  ylab("Number of Total Bike Uses") +
  geom_point(col="gray50") +
  geom_smooth(method="lm") +
  theme_bw()
```



The `psi` option is pretty robust to starting values when you have a large sample size and there's at least a hint of a real break in the trend. For example, you can run the above code using a `psi` of 0.2—where there's clearly no change in trend—and the breakpoint estimate is exactly the same. However, with fewer data points and/or a lot of variance, the choice of `psi` could change the results, so try a few different values.

The many flavors of regression

There are many ways to model trends in data, but the majority of cases can be addressed with a few types. Still, there are more flavors of regression than a Ben & Jerry's warehouse after a tornado.

The coefficients, of course, are effect sizes.

Response type	Response error distribution	Predictor type	Model type	Code**
Continuous	Normal	Continuous	Regression	<code>lm(y ~ x1 + x2 ... + xn)</code>
Continuous	Normal	Categorical	ANOVA	<code>lm(y ~ x1 + x2 ... + xn)</code>
Continuous	Normal	Mixed	ANCOVA	<code>lm(y ~ x1 + x2 ... + xn)</code>
Binary	Logistic	Mixed	Logistic regression	<code>glm(y ~ x1 + x2 ... + xn, family=binomial(link=logit))</code>
Ordinal	Logistic	Mixed	Proportional odds logistic regression	<code>polr(y ~ x1 + x2 ... + xn, Hess=TRUE)</code>
Nominal	Logistic	Mixed	Multinomial regression	<code>multinom(y ~ x1 + x2 ... + xn)</code>
Count	Poisson	Mixed	Poisson regression	<code>glm(y ~ x1 + x2 ... + xn, family=poisson(link=log))</code>
Rate	Overdispersed Poisson	Mixed	Negative binomial regression	<code>glm.nb(y ~ x1 + x2 ... + xn)</code>
	Poisson	Mixed	Poisson regression	<code>glm(y ~ x1 + x2 ... + xn, offset=log(t), family=poisson(link=log))</code>
Counts with lots of 0s	Overdispersed Poisson	Mixed	Negative binomial regression	<code>glm.nb(y ~ x1 + x2 ... + xn, offset=log(t))</code>
	Overdispersed Poisson	Mixed	Zero-inflated Poisson regression	<code>require (pscl); zeroinfl(y ~ x1 + x2 ... + xn, dist="poisson")</code>

Response type	Response error distribution	Predictor type	Model type	Code**
Counts where 0 is not possible and mean < 5	Overdispersed Poisson	Mixed	Zero-truncated Poisson regression	<pre>require(gamlss.tr); gamlss(y ~ x1 + x2 ... + xn, family=PO) library(mgcv); gam(y ~ x1 + x2 ... + xn, family=...)</pre>
Any	Same as glm	Smoothed	GAM	

** change + to * to create interactions

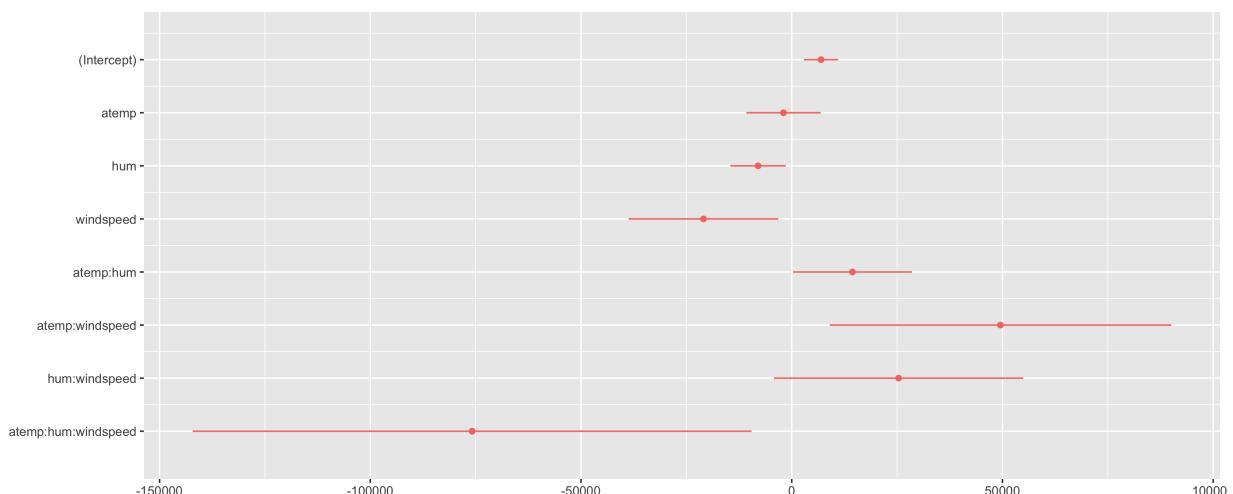
Plotting regression coefficients

The dotwhisker package has a simple function to graph regression results using ggplot2. dwplot shows the coefficients and their confidence intervals (95% by default):

```
require(dotwhisker)

# Create a model object
bad_model = lm(cnt ~ atemp * hum * windspeed, data=bike_share_daily)

# Plot coefficients and CIs
dwplot(bad_model)
```



Working with temporal data

Date and time formats are the bane of coders everywhere. No matter which coding language you use, you'll eventually run into the need to convert dates or times and end up wanting to punch your screen.

Time series analysis requires specific data types or structures—we'll explore different types of those structures below, and the [CRAN Time Series task view⁴⁹](#) is worth spending time exploring if you need to evaluate time series in any depth.

The informal consensus is that in R, `as.Date` is best for dates and `as.POSIXct` is best for times because these are the simplest ways to work with each data type. The `lubridate` package provides a more intuitive wrapper to `as.POSIXct` and is very useful for date conversions between types.

It's useful to have a cheat-sheet of the major date and time formats; you can get a complete list with `?strptime` but the major ones are as follows:

Major date formats

Date step	Symbol	Format	Alternate symbols	Alternate format
Format Date (ISO)	%F	Y-m-d	%x, %D	y/m/d, m/d/y
Year	%Y	4-digit year (2014)	%y	2-digit year (14)
Month	%m	Decimal month (10)	%b, %B	Abbreviated month (Oct.), Full month (October)
Day	%d	Decimal date (1-31)	%e	Decimal date (01-31)
Weekday	%w	Decimal weekday (0-6, starts with Sunday)	%a, %A, %u	Abbreviated weekday (Sun), Full weekday (Sunday), Decimal with Monday start (7)
Weekyear (ISO)	%V	Sunday start (0-53)	%U, %W	Sunday start (USA), Monday start (UK)
Day of the year	%j	Decimal day of the year (1-366)		

⁴⁹<http://cran.r-project.org/web/views/TimeSeries.html>

Major time formats

Time step	Symbol	Format
Date and time	%c	a b e H:M:S Y
Time	%T	H:M:S
Time (short)	%R	H:M
Hours (24 hour clock)	%H	
Hours (12 hour clock)	%I	
Minutes	%M	
Seconds	%S	
AM or PM	%p	
Offset from GMT	%z	
Time zone	%Z	

There just isn't the space to cover all (or even a small subset) of date/time conversions and temporal math, so we'll just illustrate a few very common date conversion needs in this recipe.

In addition to the `base` function `as.Date`, we'll use the `lubridate` package in this recipe.

```
require(lubridate)
```

The ISO standard for a date format is 4-digit year, 2-digit month, and 2-digit day, separated by hyphens. We'll often aim to convert to that format, but as we'll see, it's not always necessary as long as you specify the format type in the conversion process.

Perhaps the most common date conversion is a character to a date. If it's not in a clear *year-month-day* format, you'll need to specify the format it's in so that R can properly convert it to a date (a table of the major formats appears below).

```
india_national_holidays = c("Jan 26, 2014", "Apr 13, 2014", "Aug 15, 2014",
                           "Oct 02, 2014", "Dec 25, 2014")

india_national_holidays = as.Date(india_national_holidays, format="%b %d, %Y")

str(india_national_holidays)

Date[1:5], format: "2014-01-26" "2014-04-13" "2014-08-15"
"2014-10-02" "2014-12-25"
```

Another common date conversion need is to calculate age based on today's date (or any reference date):

```

birthdays = c("20000101", "19991201", "19760704", "20140314")

ymd(birthdays)

"2000-01-01 UTC" "1999-12-01 UTC" "1976-07-04 UTC" "2014-03-14 UTC"

age = ymd(today()) - ymd(birthdays)

round(age/dyears(1), 1) # dyears is a lubridate function, not a typo!

14.9 15.0 38.4 0.7

age = ymd(20141201) - ymd(birthdays)

round(age/dyears(1), 0)

15 15 38 1

```

The basic thing to remember is that even when you don't have a real day value (e.g., your data is binned into months), many functions in R generally expect one if you want to use date functions. See the next recipe for an example of this issue in action.

Calculate a mean or correlation in circular time (clock time)

Sometimes you want to know the average time at which something happens, or the relationship between times, but if the data occur near the end of a time period, the use of `mean()` will give you incorrect results. The `psych` package comes to the rescue with the `circadian.mean` and `circadian.cor` functions:

```

require(psych)

wake_time = c(7.0, 7.5, 6.5, 6.25, 7.0, 7.25, 7.25)

sleep_time = c(23.5, 0.5, 23.75, 0.25, 23.25, 23.5, 0.75)

wake_sleep = data.frame(wake_time, sleep_time)

circadian.mean(wake_sleep$sleep_time)

23.92808

circadian.cor(wake_sleep)

```

```
wake_time sleep_time
wake_time 1.0000000 0.1524331
sleep_time 0.1524331 1.0000000
```

Plotting time-series data

While having dates in the dataset can allow easy plotting via `ggplot2`, there are a few plotting functions that provide one-line insight into time series data. They may not be graphically elegant, but they provide a great way to understand your data more thoroughly.

We'll use a dataset of 10 years of monthly births in the UK (2003-2012), acquired from the EU's *EuroStat* program.

```
require(eurostat)

demo_fmonth = get_eurostat('demo_fmonth')
```

R can be a little particular about date values, and if your data aren't in a format it recognizes, it can choke. Case in point is using `ggplot2` to plot this data, which contains a year field and a month field but not a day field—which makes sense, seeing as how the data are monthly values. But you'll have to tie yourself in knots to make those two work together as time elements in a `ggplot`, so it's actually easier if you just gave each month a fake day (i.e., the 1st), mush 'em together as a character via the `paste` function, and then specify that you want the final result to be in `Date` format.

```
demo_fmonth$date = as.Date(paste(substr(demo_fmonth$time, 1, 4),
substr(demo_fmonth$month, 2, 3), "01", sep="-"))
```

Now we can use `dplyr` to filter and arrange the data...

```
require(dplyr)

UK_births = filter(demo_fmonth, geo == "UK" & month != 'TOTAL' &
month != 'UNK' & date >= '2003-01-01' & date <= '2012-12-01')

UK_births = arrange(UK_births, date)
```

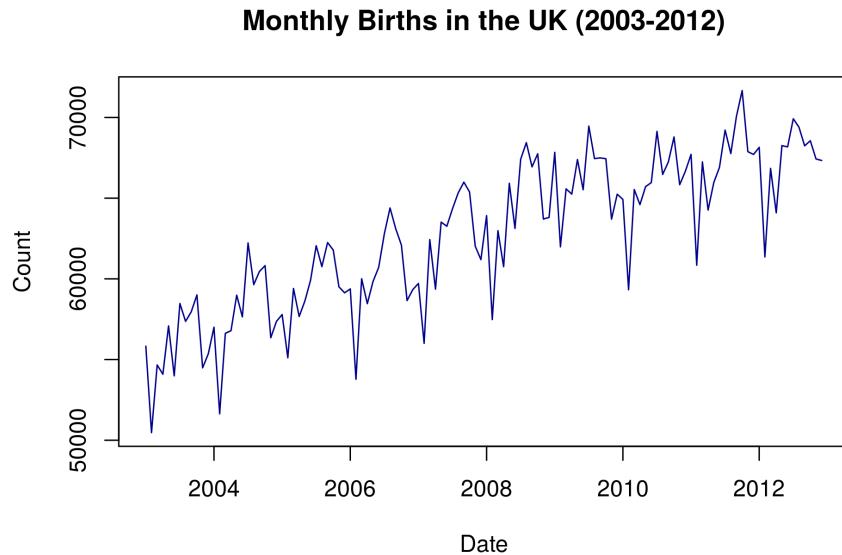
...and then we can convert the data into a `ts` (time series) object:

```
UK_births_ts = ts(UK_births$values, start=c(2003, 1), frequency=12)
```

The `frequency` option is related to how many time steps make up a “whole” unit in your data; for example, if your data is hourly and you expect seasonality on the daily level, frequency equals 24. Obviously, the above setting tells R that the data are monthly and we expect annual “seasonality.”

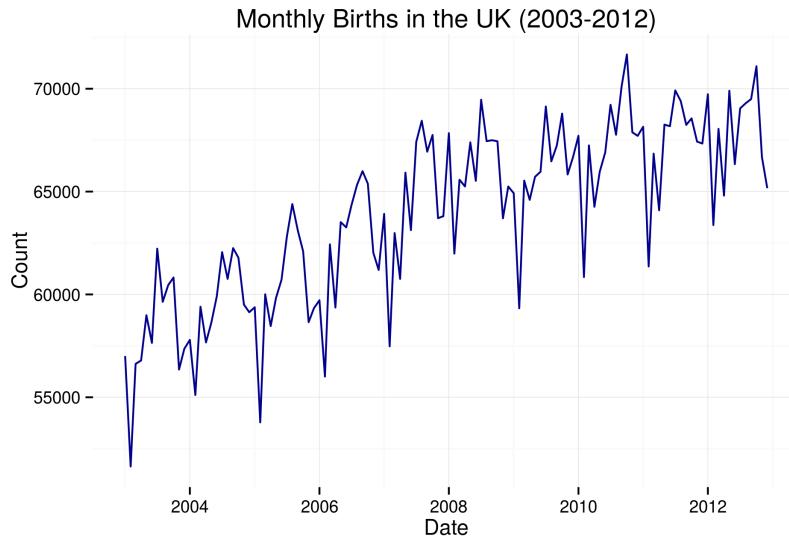
Once we have a `ts` object, we can plot it; if all you need to do is look at it, you can simply use `plot(my_ts_object)`. But if you want to use the plot for more than just your own edification, you’ll need to add some decoration manually:

```
plot(UK_births_ts, xlab="Date", ylab="Count", main="Monthly Births in the UK  
(2003-2012)", col="darkblue")
```



And we can use `ggplot2` to make it prettier:

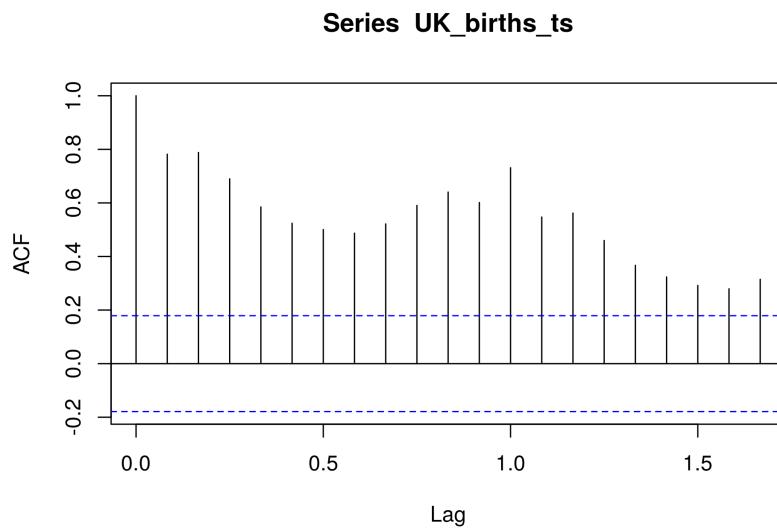
```
ggplot(UK_births, aes(x=date, y=values)) +  
  geom_line(col="darkblue") +  
  ylab("Count") +  
  theme_minimal()
```



Detecting autocorrelation

Autocorrelation can be present in non-temporal data as well as in time series; the `acf` function works the same for either type of data.

```
acf(UK_births_ts)
```

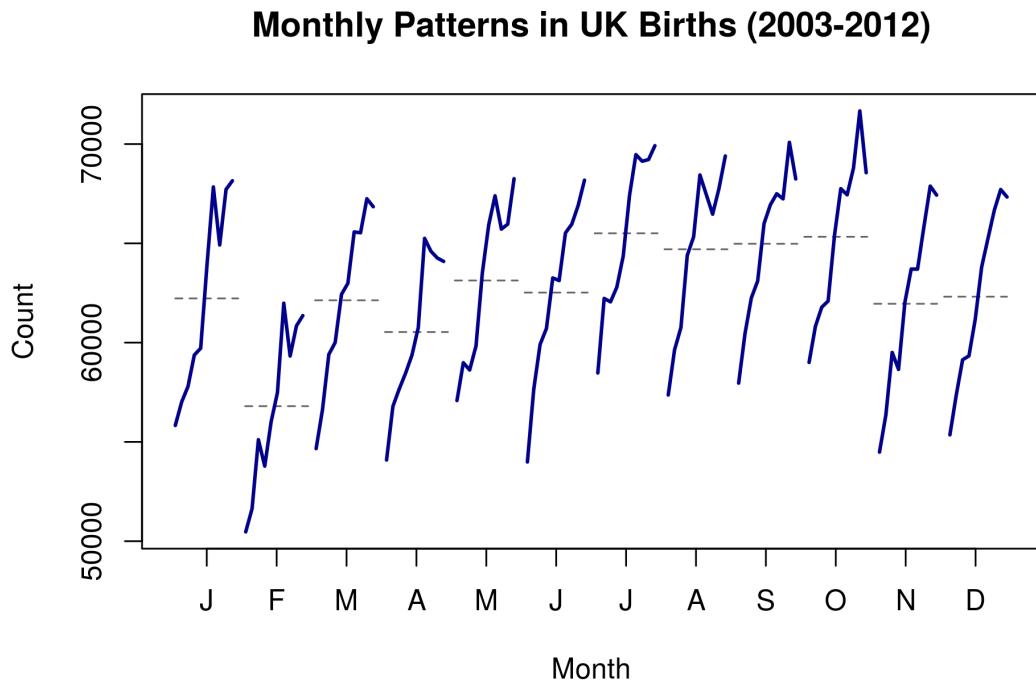


What's often under-appreciated is that autocorrelated data, temporal or not, will make standard confidence intervals *too small*. In other words, if you want to plot a trend from autocorrelated data and use the default confidence intervals, you'll underestimate the actual uncertainty associated with that trend.

Plotting monthly and seasonal patterns

If your data are monthly (or if you aggregate them to months), the `monthplot` function in the base installation breaks out each month's data and plots them separately, along with a horizontal line for each month's mean (the default) or median (`base = median`) value over the time span. A few manual tweaks help show the information in this plot more clearly:

```
monthplot(UK_births_ts, main="Monthly Patterns in UK Births (2003-2012)",
  xlab="Month", ylab="Count", col="darkblue", lwd=2, lty.base=2, lwd.base=1,
  col.base="gray40")
```

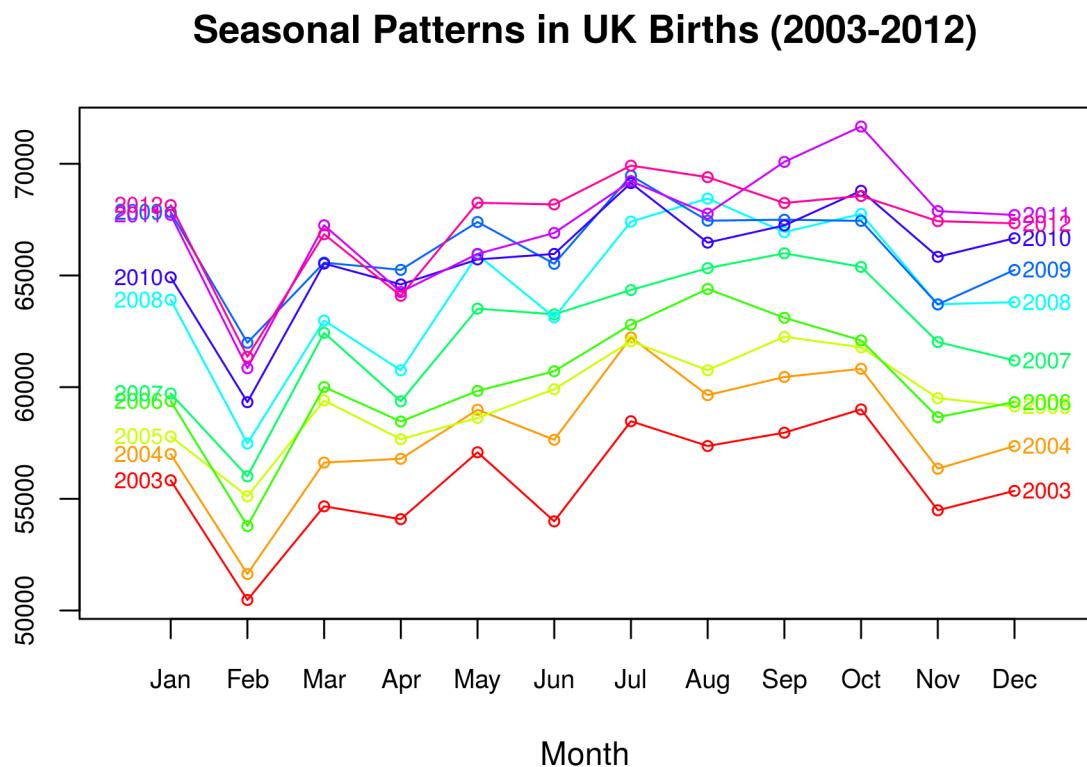


You can plot out the seasonal, trend, or random components in a `monthplot` with the `choice` option, e.g., `choice = "seasonal"`).

The `forecast` package contains another useful plot function that shows each year's trend as a separate line, allowing you to discern whether there's a seasonal or monthly effect at a glance:

```
require(forecast)

seasonplot(UK_births_ts, main="Seasonal Trends in UK Births (2003-2012)",
  col=rainbow(10), year.labels=TRUE, year.labels.left=TRUE, cex=0.7,
  cex.axis=0.8)
```



Plotting seasonal adjustment on the fly

The ggseas package provides an easy way to plot seasonally-adjusted data in ggplot2. There are a variety of ways to adjust for seasonality, so review its GitHub site⁵⁰ for more details and options that come with this package.

⁵⁰<https://github.com/ellisp/ggseas>

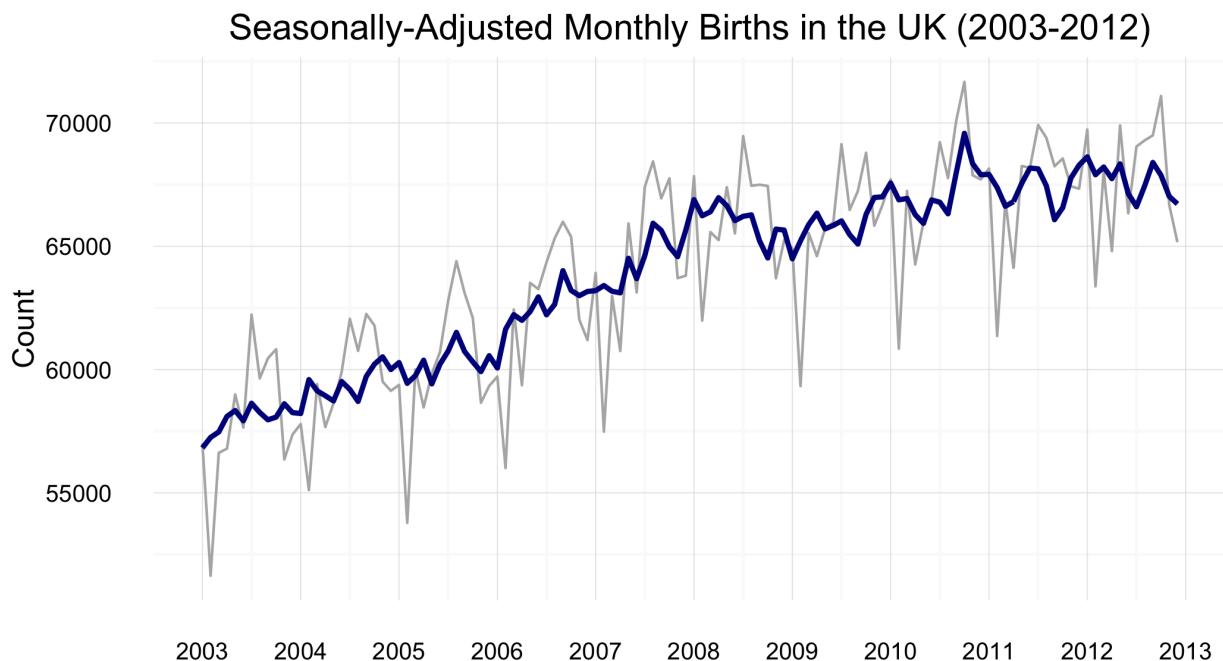
```

require(ggseas)

# Convert the time series object to a dataframe for stat_seas
UK_births_ts_df = tsdf(UK_births_ts)

# Plot seasonally-adjusted data over actual data
ggplot(UK_births_ts_df, aes(x=x, y=y)) +
  ggtitle("Seasonally-Adjusted Monthly Births in the UK (2003-2012)") +
  geom_line(color="gray70") +
  stat_seas(color="darkblue", size=1) +
  scale_x_continuous(breaks=seq(2003, 2013, 1)) +
  ylab("Count") +
  xlab("") +
  theme_minimal()

```



Decomposing time series into components

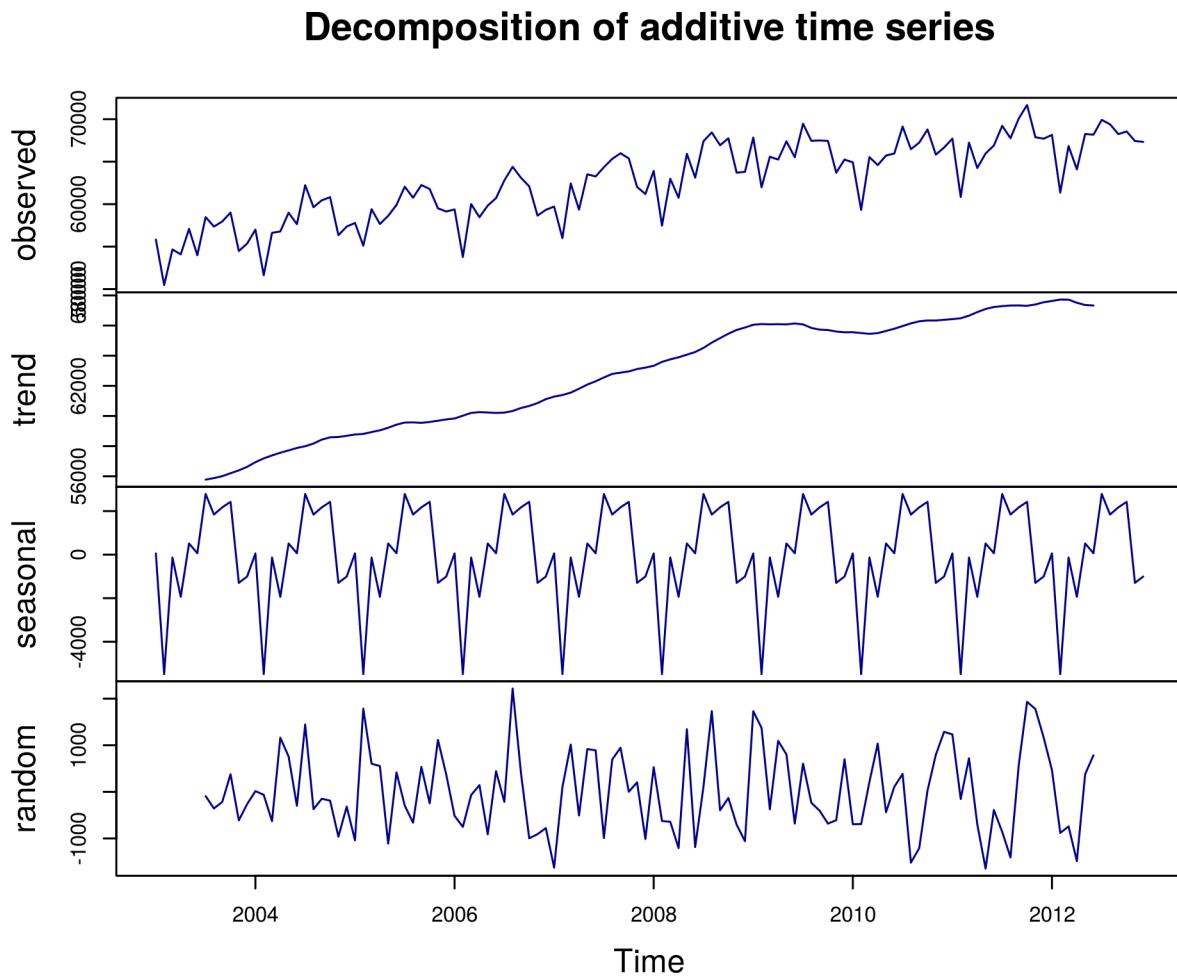
If you need to explore a time series in more depth, you'll probably want to decompose it into seasonal, trend, and random/remainder components.

Additive decomposition

We'll continue with the UK monthly births data in its time series format. Most functions we'll use in this recipe are in the base installation's stats package, but one is in the forecast package, so make sure it's loaded with the data.

The decompose function uses a moving average for simplicity:

```
plot(decompose(UK_births_ts), col="darkblue")
```



Decomposing when the variance changes

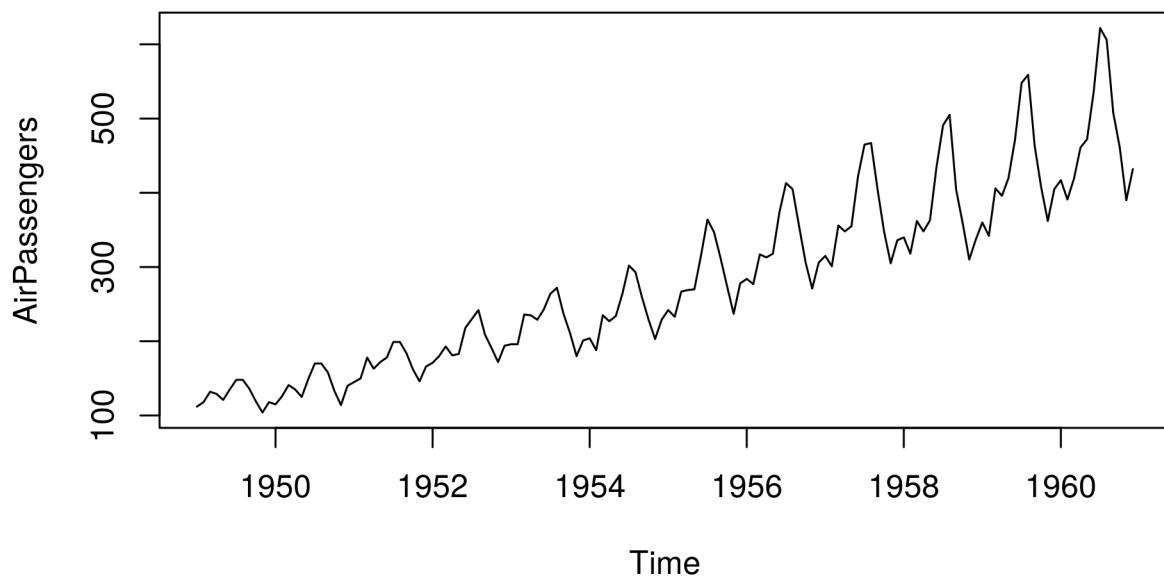
Of course, simple time series decomposition only works correctly if the variance is more or less stable. When it's not—as in the AirPassengers dataset, below—you have to transform the values before you decompose the series into its components.

There are two ways to do this: by using type = "multiplicative" in the decompose function as above, or by using the BoxCox.lambda function in the forecast package to determine an optimal transformation.

Look at the data to see the change in variance over time:

```
data(AirPassengers)
```

```
plot(AirPassengers)
```



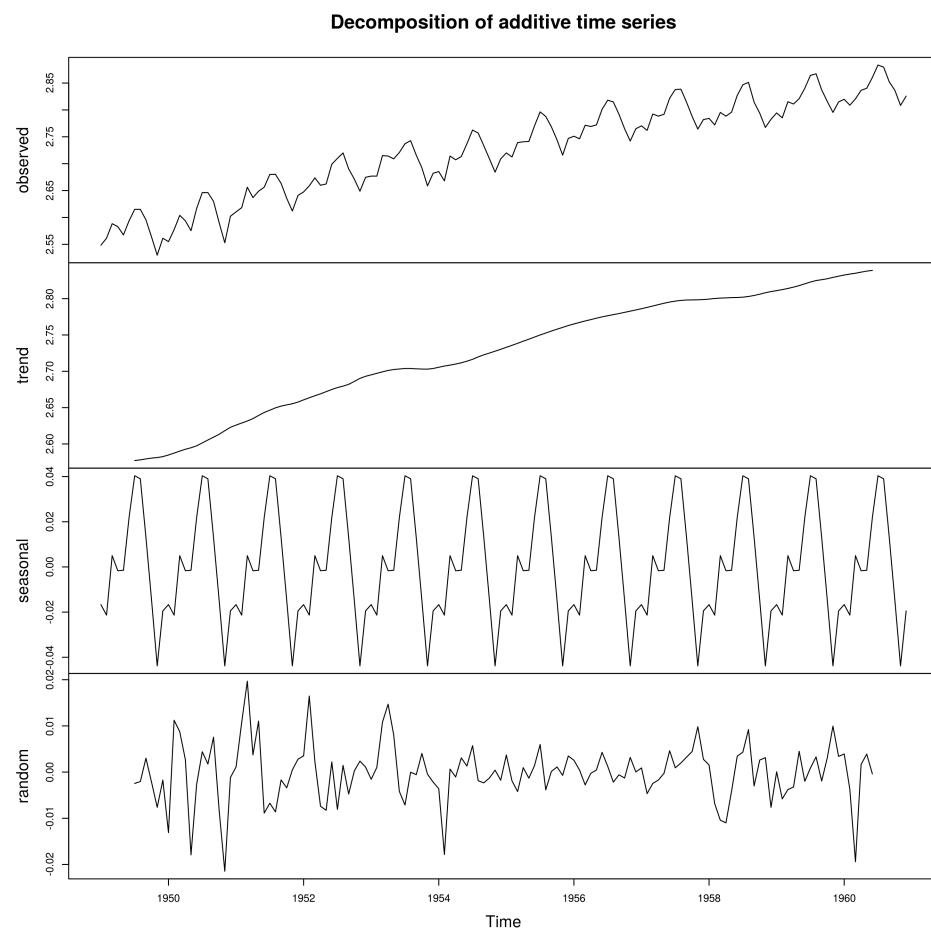
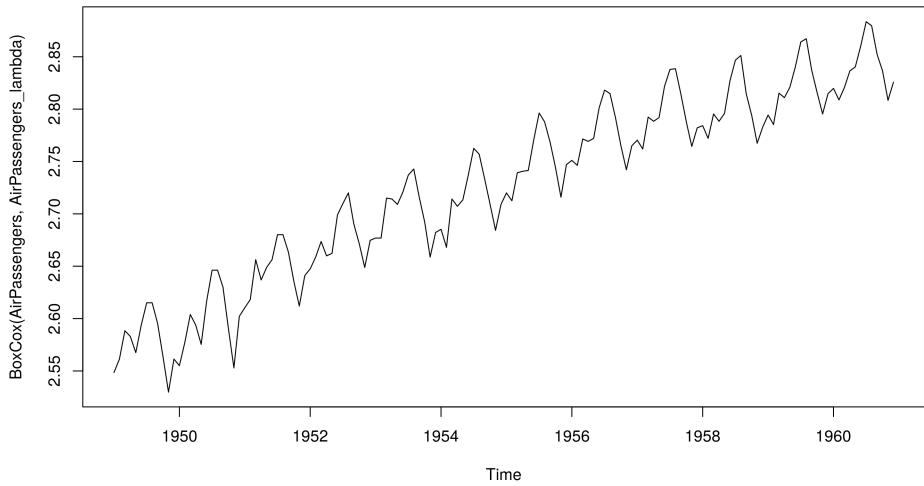
Use the BoxCox.lambda function and view the results:

```
require(forecast)

# Calculate the Box-Cox transformation
AirPassengers_lambda = BoxCox.lambda(AirPassengers)

# Plot the Box-Cox transformation results
plot(BoxCox(AirPassengers, AirPassengers_lambda))

# Plot the transformed decomposition
plot(decompose(BoxCox(AirPassengers, AirPassengers_lambda)))
```



Using spectral analysis to identify periodicity

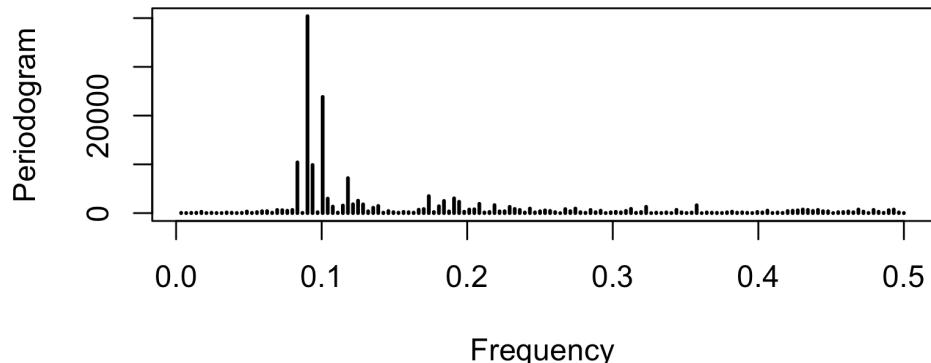
Spectral analysis provides a simple way to evaluate whether there are any cyclical patterns in your time series. The TSA package's `periodogram` function creates a periodogram, where the highest peak(s) in the plot represent(s) possible cycling times. To do spectral analysis, you need to ensure the data are detrended first.

We'll use the built-in annual sunspot dataset in this recipe, as sunspots have a well-known 11-year cycle.

```
data(sunspot.year)
```

To find the cycle times, look for a clear peak (or peaks) in the plot.

```
sunny = TSA::periodogram(diff(sunspot.year))
```



You can also extract the `$freq` and `$spec` values from the spectrum object into a data frame as `x` and `y`, respectively, to get the exact frequency where the spectrum is maximized, or if you want to create a prettier plot with `ggplot2`.

```
sunny_df = data.frame(freq = sunny$freq, spec = sunny$spec)
```

```
sunny_df[which.max(sunny_df[,2]),]
```

	freq	spec
26	0.09027778	40400.47

Frequency is the reciprocal of the time period. This data is annual, and since the the highest spectral peak is at a frequency of 0.09, it suggests that there is a \sim 11 year cycle ($1/0.09=11$).

Plotting survival curves

The `survminer` package makes a beautiful all-in-one survival plot and risk table. We'll use the tongue cancer data from the `KMsurv` package to demonstrate.

```

require(survival)
require(survminer)

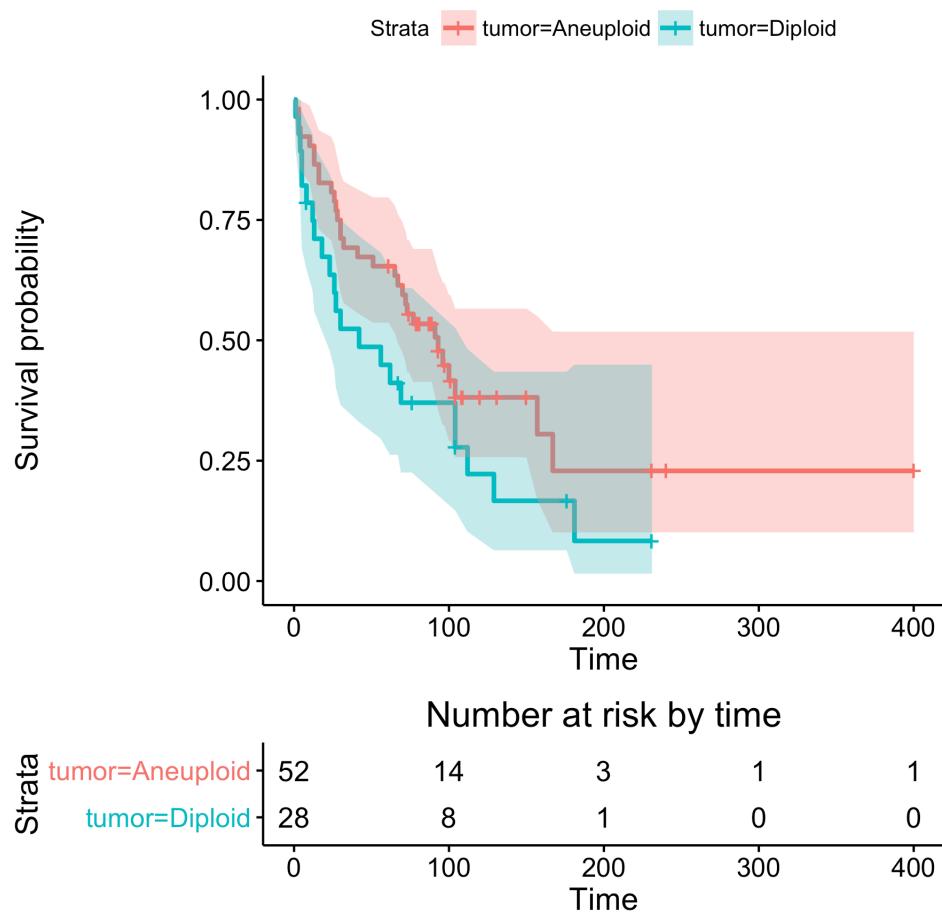
# Get data and add names of cancer type
data(tongue, package = "KMsurv")

tongue$tumor = ifelse(tongue$type==1, "Aneuploid", "Diploid")

# Create survival model
tongue_survival = survfit(Surv(time = time, event = delta) ~ tumor,
                           data = tongue)

# Plot the curves and table
ggsurvplot(tongue_survival, risk.table = TRUE, conf.int = TRUE)

```



Evaluating quality with control charts

Companies with quality control processes or activities often use control charts (aka “Shewhart charts”) to evaluate whether a **stable** process is moving “out of control” over time and thus requires intervention. The `qcc` package provides a way to plot all major control chart types simply and quickly.

As an example, we’ll create some fake data on hospital-acquired-infections (HAI) for a 25-month period to illustrate the creation of a u-chart, a control chart that plots changes in a rate over time:

```
require(qcc)

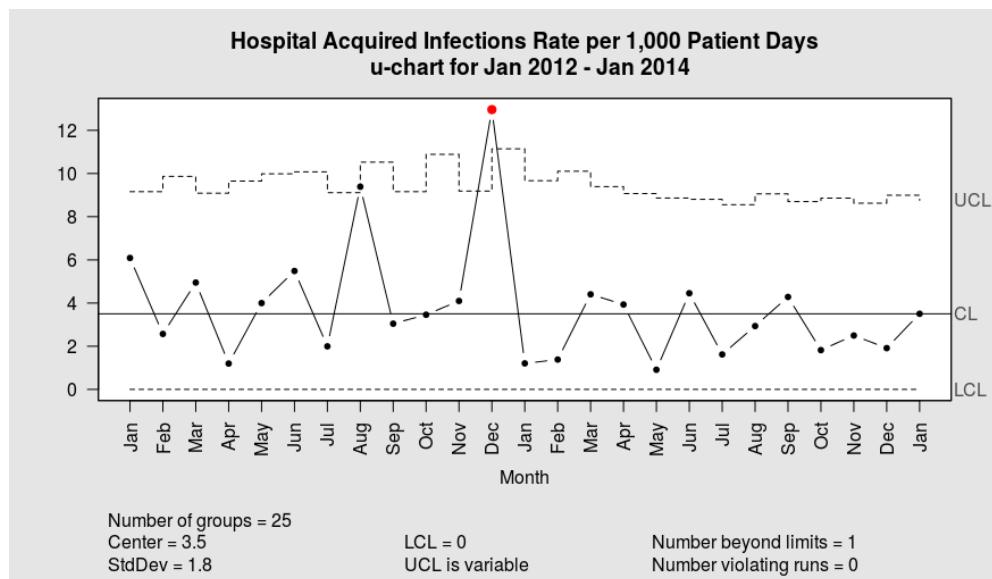
infections = c(6, 2, 5, 1, 3, 4, 2, 6, 3, 2, 4, 7, 1, 1, 4, 4, 1, 5, 2, 3, 5,
2, 3, 2, 4)

patient_days = c(985, 778, 1010, 834, 750, 729, 1002, 639, 985, 578, 976, 540,
829, 723, 908, 1017, 1097, 1122, 1234, 1022, 1167, 1098, 1201, 1045, 1141)

# These are just labels for qcc, not real dates
month_name = month.abb[c(1:12, 1:12, 1:1)]
```

We’ll use 1,000 patient days for the rate baseline. The `qcc` function takes care of everything:

```
infection_control = qcc(infections, sizes=patient_days/1000, type="u",
labels=month_name, axes.las=2, xlab="Month", ylab="", digits=2,
title="Hospital Acquired Infections Rate per 1,000 Patient Days\n
u-chart for Jan 2012 - Jan 2014")
```

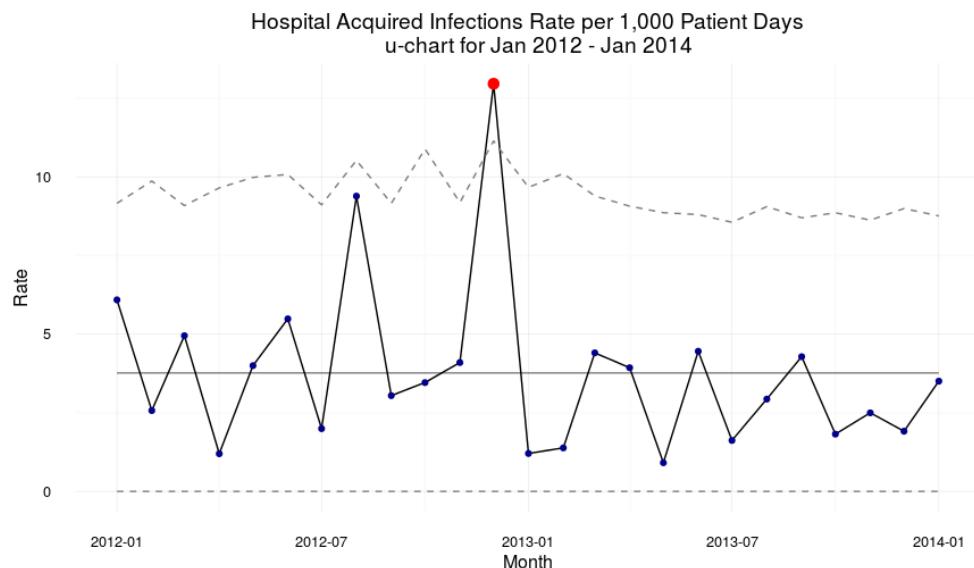


The control-chart default plot in qcc is fairly ugly. You can remove the components of the resulting R object for use elsewhere, such as with ggplot2:

```
# Create a real date and move limits and rate into a data frame
ic_qcc = data.frame(Month = seq(as.Date("2012-01-01"),
  as.Date("2014-01-01"), "months"), infection_control$limits,
  Rate = (infections / patient_days)*1000)

# Create a factor for the "special causes" points
ic_qcc$Violations = factor(ifelse(row.names(ic_qcc)
  == infection_control$violations$beyond.limits, "Violation", NA))

# Plot a cleaner control chart
ggplot(ic_qcc, aes(x=Month, y=Rate)) +
  geom_line(aes(y=mean(ic_qcc$Rate)), color="gray50") +
  geom_line() +
  geom_point(color="darkblue") +
  geom_point(data=filter(ic_qcc, Violations=="Violation"),
    color="red", size=3) +
  geom_line(aes(y=LCL), linetype="dashed", color="gray50") +
  geom_line(aes(y=UCL), linetype="dashed", color="gray50") +
  xlab("Month") +
  ylab("Rate") +
  ggtitle("Hospital Acquired Infections Rate per 1,000 Patient Days
  u-chart for Jan 2012 - Jan 2014") +
  theme_minimal()
```





Many people use control charts incorrectly on trending and/or autocorrelated data. When you need something like a control chart on these types of data, use a GAM instead. Morton et al.'s book [Statistical Methods for Hospital Monitoring with R](#)⁵¹ has R code and some good advice on this issue.

Identifying possible breakpoints in a time series

The `strucchange` package provides a simple way to identify changes in the statistical structure of a time series.

We'll look at total monthly United States CO₂ emissions between February 2001 and June 2015 ([data from the US Energy Information Administration](#)⁵²). Load the `strucchange` package, download and filter the data, and convert the data to a time series object:

```
require(strucchange)
US_co2 = read.table("http://www.eia.gov/totalenergy/data/browser/
  csv.cfm?tbl=T12.01", sep=",", header=T)

US_co2 = filter(US_co2, Column_Order == 14 & YYYYMM >= 200102 &
  substr(YYYYMM, 5, 7) != 13)

US_co2_ts = ts(US_co2[,3], freq=12, start=c(2001,2))
```

`strucchange` will pick the optimal number of breakpoints as determined by BIC, which can be subsequently plotted over the time series. As there does not seem to be a clear indication of generally trending rises or declines in mean tendency, we can use ~ 1 in the formula in place of a dependent variable to assess whether there is a step change (or changes) in the time series:

```
# Obtain breakpoints estimate
breakpoints(US_co2_ts ~ 1)

Optimal 2-segment partition:
Call:
breakpoints.formula(formula = us_co2_ts ~ 1)
Breakpoints at observation number:
96
Corresponding to breakdates:
2009(1)
```

⁵¹<https://www.amazon.com/Statistical-Methods-Hospital-Monitoring-R/dp/1118596307>

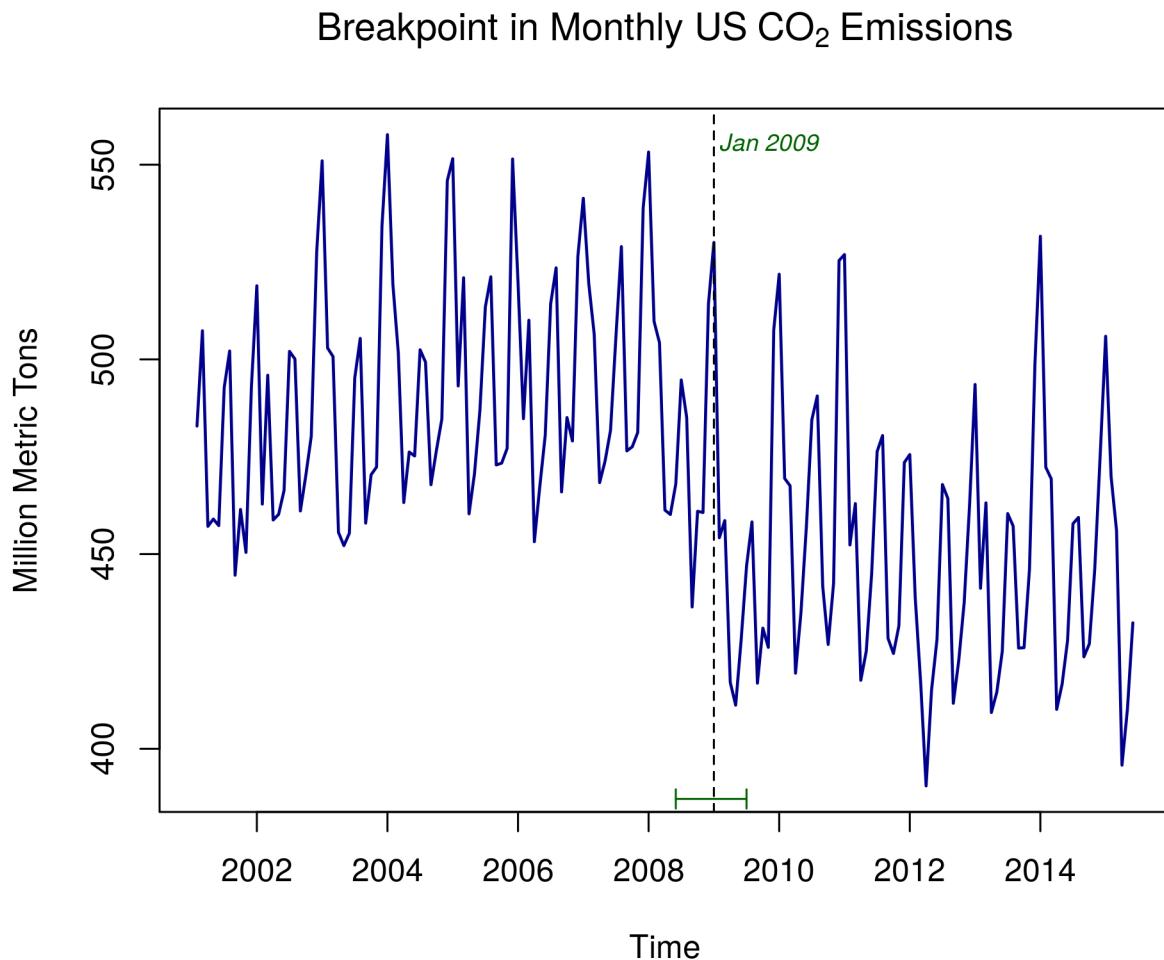
⁵²<http://www.eia.gov/beta/MER/index.cfm?tbl=T12.01#/?f=M&start=200101&end=201506&charted=0-1-13>

```
# Plot the time series
plot(US_co2_ts, main=expression(paste("Breakpoint in Monthly US"
~CO[2]~Emissions)), ylab="Million Metric Tons", col="darkblue", lwd=1.5)

# Plot the line at the optimal breakpoint
lines(breakpoints(US_co2_ts ~ 1), col="darkgreen")

# Plot a 90% confidence interval
lines(confint(breakpoints(US_co2_ts ~ 1), level=0.90), col="darkgreen")

# Add breakpoint location text
text(2008.8, 555, "Jan 2009", cex=0.75, col="darkgreen", pos=4, font=3)
```



Exploring relationships between time series: cross-correlation

Cross-correlation is extremely useful for helping identify leading and lagging indicators. The `ccf` function makes this easy.

We'll use data from the US Bureau of Labor Statistics to demonstrate cross-correlation.

```
# Load dplyr to subset the BLS data
require(dplyr)

# Download CPI time series data from US BLS
CPI_xport = read.table("http://download.bls.gov/pub/time.series/cu/
  cu.data.14.USTransportation", header=T, sep="\t", strip.white=T)

# CPI for motor fuel, 2000-2014
fuel = filter(CPI_xport, series_id == "CUUR0000SETB" &
  year >= 2000 & year <= 2014 & period != "M13")

# CPI for airline fare, 2000-2014
fare = filter(CPI_xport, series_id == "CUUR0000SETG01" &
  year >= 2000 & year <= 2014 & period != "M13")
```

Since `ccf` assumes that the time series is already detrended, we'll do that first.

```
# Create time series
fuel_ts = ts(fuel$value, start=c(2000, 1), freq=12)

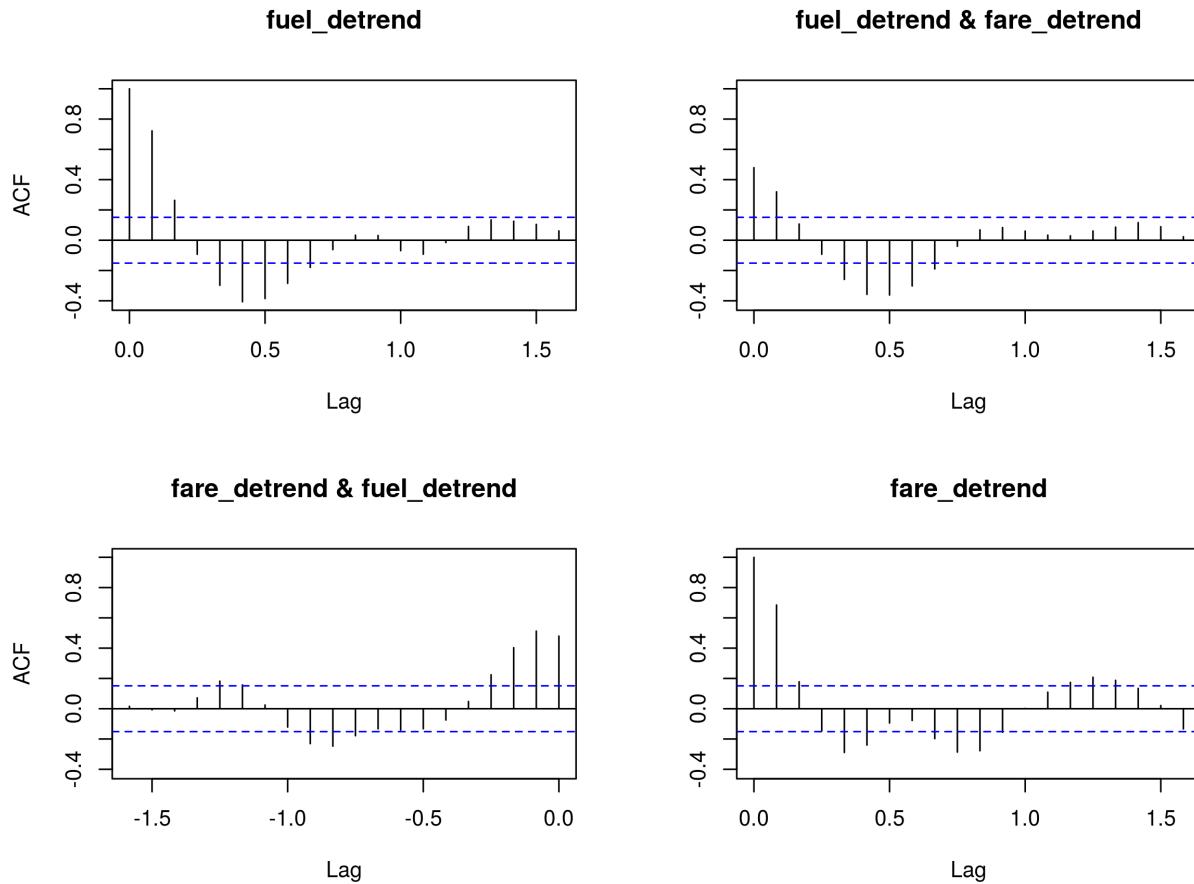
fare_ts = ts(fare$value, start=c(2000, 1), freq=12)

# CCF *requires* detrended data
# Get detrended values of each time series
fuel_detrend = na.omit(decompose(fuel_ts)$random)

fare_detrend = na.omit(decompose(fare_ts)$random)
```

An ACF plot shows the pattern of each variable separately and together for an 18 month time span:

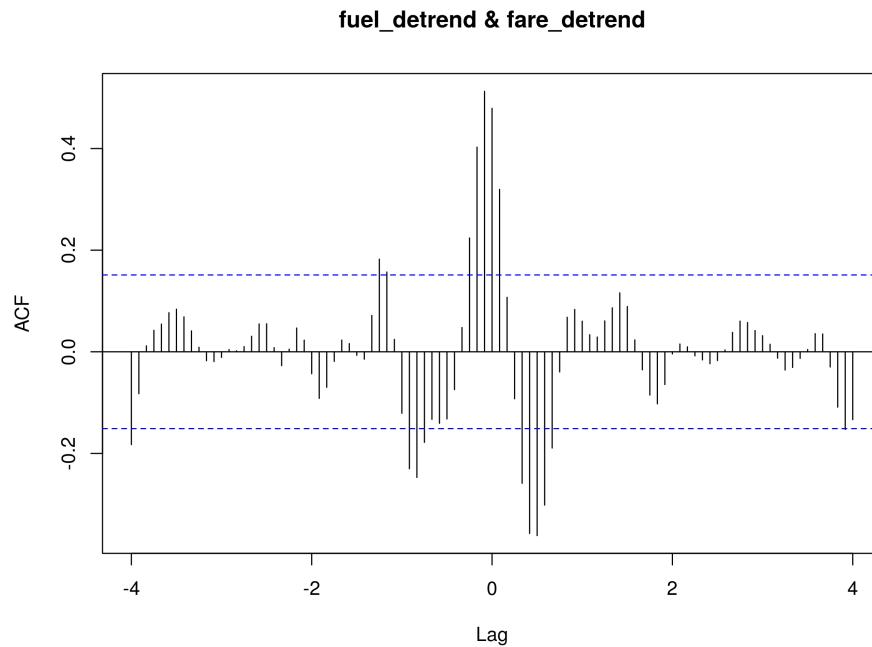
```
acf(ts.union(fuel_detrend, fare_detrend))
```



You can combine these views into a single plot with the `ccf` function:

```
# Calculate CCF with +/- 4 year span
ccfvalues = ccf(fuel_detrend, fare_detrend, lag.max=48)

ccfvalues
```



We can see that the maximum occurs just to the left of zero, we can infer that fuel costs lead airline fares slightly. Since it's hard to read values off a plot, we can use a small function from [nvogen](#) on Stack Overflow⁵³ that can find the maximum correlation value:

```
# Function to find CCF
Max_CCF = function(a, b) {
  d = ccf(a, b, plot = FALSE, lag.max = length(a)-5)
  cor = d$acf[,1]
  abscor = abs(d$acf[,1])
  lag = d$lag[,1]
  res = data.frame(cor,lag)
  absres = data.frame(abscor,lag)
  absres_max = res[which.max(absres$abscor),]
  return(absres_max)
}

# Determine maximum CCF value
Max_CCF(fuel_detrend, fare_detrend)
      cor          lag
163 0.5126287 -0.08333333
```

The maximum cross-correlation value is 0.51, which occurs between airline fares (at time 0) and fuel prices in the prior month (i.e., as $1/12 = 0.08333333$).

⁵³<http://stackoverflow.com/a/20133091>

Basic forecasting

We saw forecasting as a section within the *Chapter 1* example. For the sake of keeping ideas together, we'll replay that forecasting example here.

An excellent companion to *doing* forecasting is *understanding* it: for that, I recommend Rob Hyndman's excellent online text, *Forecasting: Principles and Practice*⁵⁴.

```
require(forecast)
require(dplyr)

### Data download and prep, same as in Chapter 1 ####
download.file("http://archive.ics.uci.edu/ml/machine-learning-databases
/00235/household_power_consumption.zip", destfile =
 "~/BIWR/Chapter1/Data/household_power_consumption.zip")
unzip("~/BIWR/Chapter1/Data/household_power_consumption.zip", exdir="Data")
power = read.table("~/BIWR/Chapter1/Data/household_power_consumption.txt",
 sep=";", header=T, na.strings=c("?", ""), stringsAsFactors=FALSE)

power$Date = as.Date(power$Date, format="%d/%m/%Y")
power$Month = format(power$Date, "%Y-%m")
power$Month = as.Date(paste0(power$Month, "-01"))

power$Global_active_power_locf = na.locf(power$Global_active_power)

power_group = group_by(power, Month)
power_monthly = summarize(power_group,
  Total_Use_kWh = sum(Global_active_power_locf)/60)
power_monthly = power_monthly[2:47,]

total_use_ts = ts(power_monthly$Total_Use_kWh, start=c(2007,1), frequency=12)
```

The main function of the `forecast` package is `forecast`; you can also be more specific by calling `forecast.ets` or `auto.arima` for explicitly forecasting the time series based on the exponential smoothing model family or ARIMA models, respectively.

Once you have a time series object, forecasting is a breeze:

```
# Automatically obtain the forecast for the next 6 months
total_use_fc = forecast(total_use_ts, h=6)
```

⁵⁴<https://www.otexts.org/fpp>

```
# View the forecast model results
summary(total_use_fc)
plot(total_use_fc)
```

Console ~/BIWR/Chapter1/ >

```
> summary(total_use_fc)

Forecast method: ETS(A,N,A)

Model Information:
ETS(A,N,A)

Call:
ets(y = object, lambda = lambda)

Smoothing parameters:
alpha = 0.0613
gamma = 3e-04

Initial states:
l = 829.9356
s=257.3091 165.2708 45.671 -82.8744 -375.8192 -255.0943
-144.7483 -20.4648 -23.0432 97.3446 88.7408 247.708

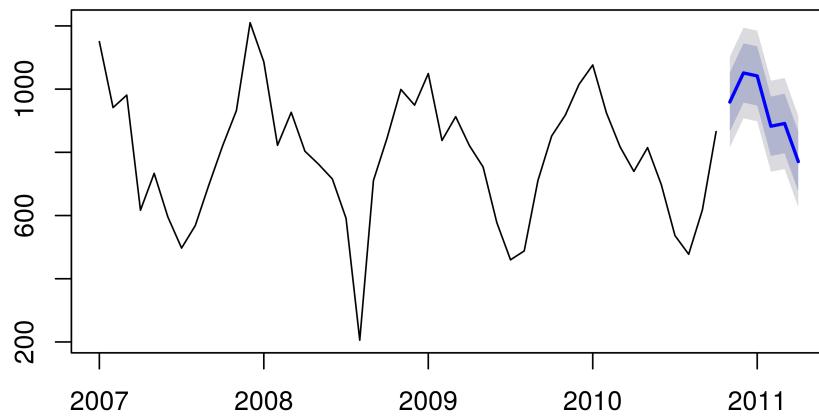
sigma: 72.9475

      AIC     AICc      BIC
598.7736 612.3220 624.3746

Error measures:
      ME     RMSE      MAE      MPE      MAPE      MASE      ACF1
Training set -12.79661 72.94752 54.94647 -3.944092 9.466253 0.653263 -0.09482945

Forecasts:
      Point Forecast    Lo 80     Hi 80    Lo 95     Hi 95
Nov 2010     959.1288 865.6428 1052.6148 816.1543 1102.1033
Dec 2010    1051.1827 957.5214 1144.8439 907.9402 1194.4252
Jan 2011    1041.6224 947.7862 1135.4586 898.1124 1185.1324
Feb 2011     882.5943 788.5836 976.6051 738.8173 1026.3714
Mar 2011     891.2219 797.0369 985.4069 747.1783 1035.2655
Apr 2011     770.7830 676.4241 865.1420 626.4734 915.0926
> |
```

Forecasts from ETS(A,N,A)



Chapter 7: A Dog's Breakfast of Dataviz

- Plotting multivariate data
- Interactive data visualizations
- Making maps in R

Plotting multivariate distributions

We've already explored summary plots of a set of variables earlier in this chapter, but sometimes you want to consider a set of variables all at once in a consistent way. Heatmaps and parallel coordinate plots (PCPs) are two common ways to do this (in the next chapter we'll explore a few others, such as cluster dendograms); heatmaps do a good job of showing three dimensions clearly, while PCPs can work for a large number of variables, as long as there aren't too many observations in the data.

Visualizing multidimensional data is difficult to do in two dimensions. Heatmaps can provide an extra dimension or two via color scale and/or intensity, while parallel coordinates lays out all of your variables as a series of 1-dimensional axes. Bubbleplots provide you the ability to use color and point size to include additional dimensions. These approaches shouldn't substitute for a set of univariate displays of the data (e.g., as a set of density plots presented in facets, as seen in the *Plotting univariate data* recipe in *Chapter 4*), but they're an essential additional tool you can use when exploring the dimensions of your data. Further, interactive versions of these plots (later this chapter) can be great tools for decision makers, and exploring layout and design with static versions before creating interactive ones can save time later.

Heatmaps

In this recipe, we'll return to the bike share data and use ggplot2 and to create a heatmap.

```
require(ggplot2)
require(dplyr)
require(gridExtra)
```

We'll first use dplyr's group_by function to group the casual bike use values by month and weekday, then use summarize to calculate mean and sd values for each month/weekday combination. Once we have that, we'll be able to use ggplot's geom_tile option to create heatmaps:

```
bike_share_grp = group_by(bike_share_daily, weekday, mnth)

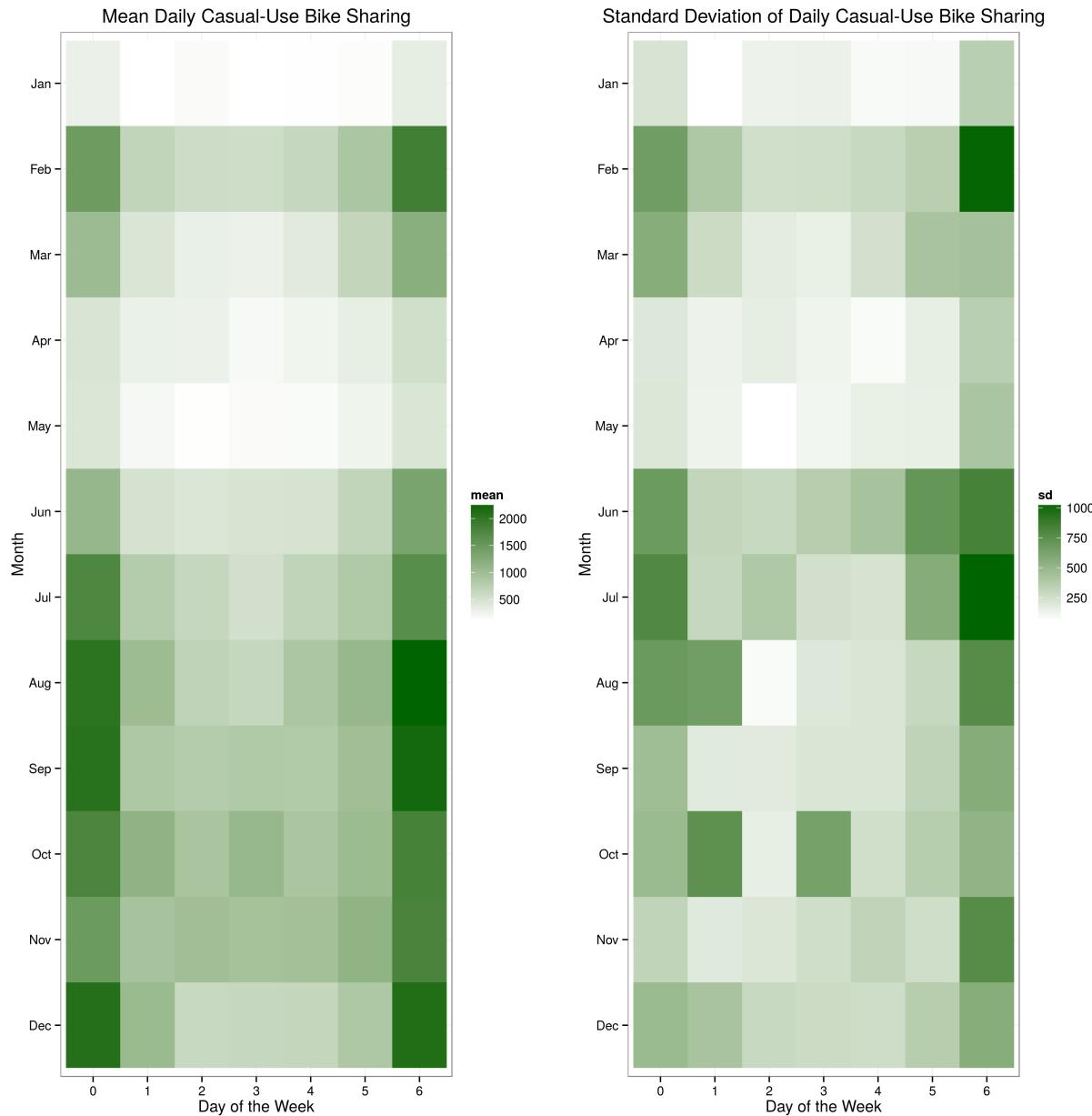
bike_share_mean = summarize(bike_share_grp, mean=mean(casual))

bike_share_sd = summarize(bike_share_grp, sd=sd(casual))

p1 = ggplot(bike_share_mean, aes(weekday, mnth)) +
  geom_tile(aes(fill = mean)) +
  scale_fill_gradient(low = "white", high = "darkgreen") +
  xlab("Day of the Week") +
  ylab("Month") +
  ggtitle("Mean Daily Casual-Use Bike Sharing") +
  scale_y_discrete(limits = rev(levels(bike_share_mean$mnth))) +
  theme_bw() +
  theme(plot.title = element_text(vjust = 1))

p2 = ggplot(bike_share_sd, aes(weekday, mnth)) +
  geom_tile(aes(fill = sd)) +
  scale_fill_gradient(low = "white", high = "darkgreen") +
  xlab("Day of the Week") +
  ylab("Month") +
  ggtitle("Standard Deviation of Daily Casual-Use Bike Sharing") +
  scale_y_discrete(limits = rev(levels(bike_share_sd$mnth))) +
  theme_bw() +
  theme(plot.title = element_text(vjust = 1))

grid.arrange(p1, p2, ncol=2)
```



Creating calendar heatmaps

We covered temporal data in *Chapter 6*, but since a calendar heatmap is just another heatmap at heart, we'll cover it here.

There are a variety of functions and scripts floating around the web that create calendar heatmaps, so those interested in custom plots might consider a search to explore those in depth. But if you just need a simple, to-the-point function to create a calendar heatmap, Paul Bleicher's `calendarHeat` R script posted on the Revolution R blog is probably all you need.

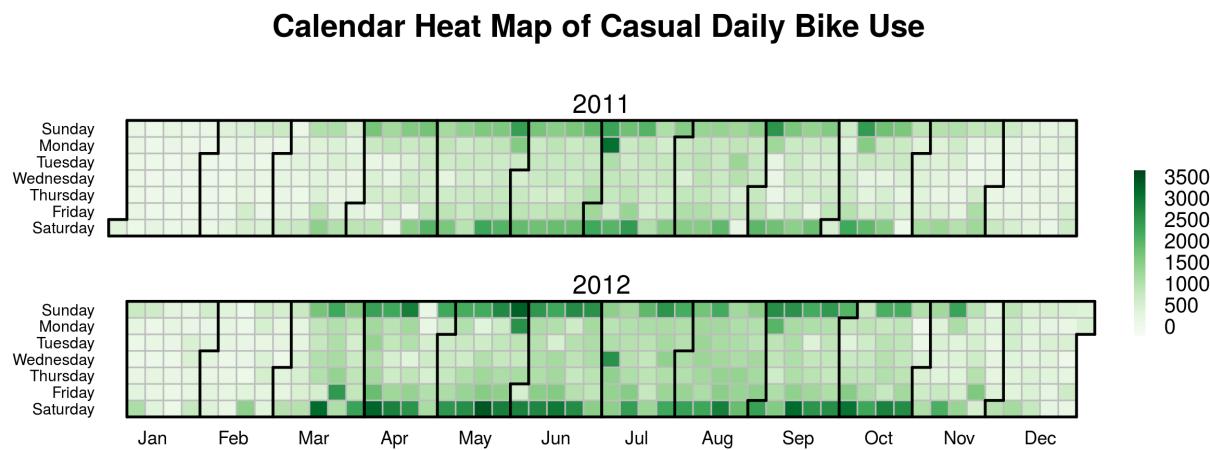
There are three color schemes built into Paul's script: red to blue (`color=r2b`), white to blue (`color=w2b`), and the color-blind unfriendly red to green (`color=r2g`), which is unfortunately the default. Opening the script via `fix(calendarHeat)` and scrolling down to lines 38-41 shows those schemes, which can be readily modified, extended, or added to using your own color ramp preferences. You can also put a different ramp into an object and call it from within the function. For example:

```
require(RColorBrewer)

source("http://blog.revolutionanalytics.com/downloads/calendarHeat.R")

green_color_ramp = brewer.pal(9, "Greens")

calendarHeat(bike_share_daily$dteday, bike_share_daily$casual,
  varname="Casual Daily Bike Use", color="green_color_ramp")
```



Parallel coordinates plots

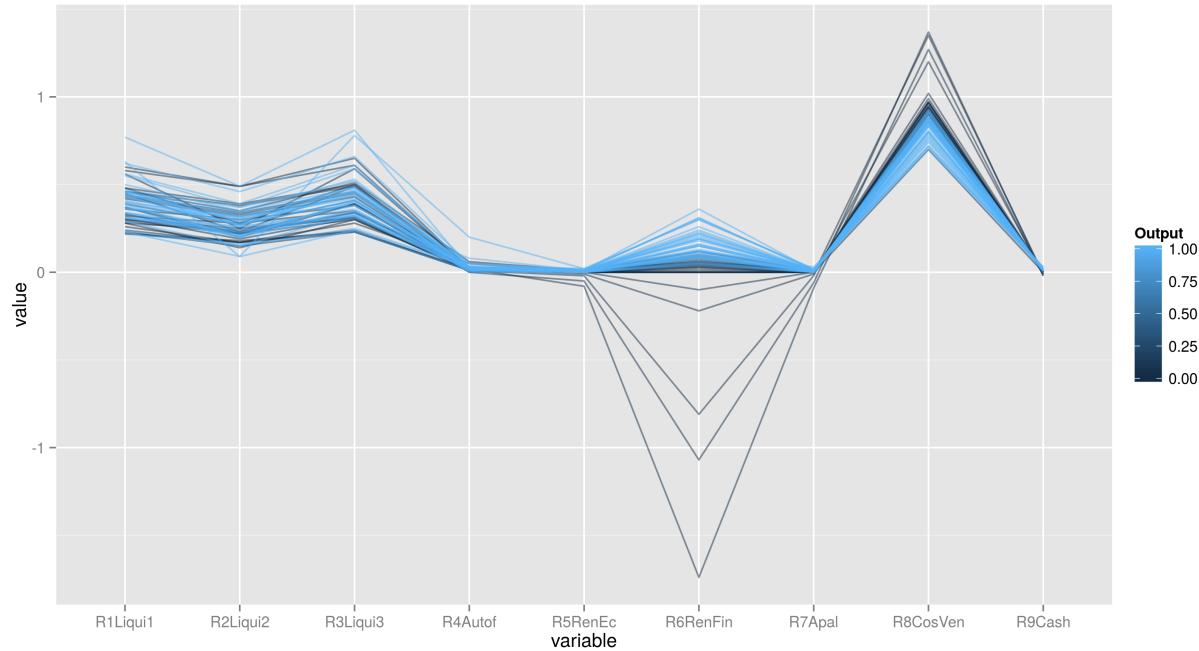
To demonstrate parallel coordinate plots, we'll use a dataset on a set of financial ratios for some banks associated with the Spanish banking crisis of the late 1970s and early 1980s. We'll see this dataset again, so it's useful to see the data here for comparison with dimension reduction techniques we'll explore in *Chapter 8*.

```
require(GGally)

banks = read.csv("https://raw.githubusercontent.com/Rmadillo/
  business_intelligence_with_r/master/manuscript/code/quiebra.csv",
  encoding = "UTF-8", stringsAsFactors = FALSE, header = TRUE)
```

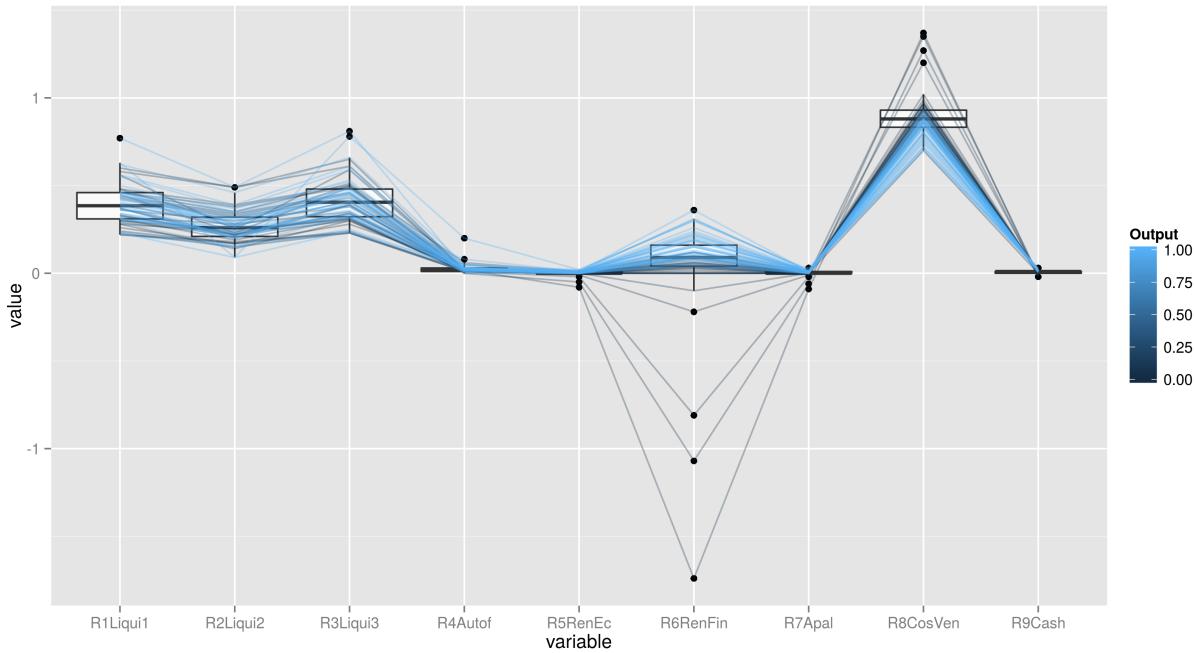
GGally has a parallel coordinates plot function that's simple to use, albeit difficult to customize. Still, it gets the job done; here are each of the nine dimensions (variables) plotted, colored by whether the bank remained solvent (light blue) or not (dark blue).

```
ggparcoord(data=banks, columns=c(3:11), groupColumn=12,
  scale="globalminmax", alphaLines=0.5)
```



A neat feature of `ggparcoord` is the ability to put boxplots on the background of the plot:

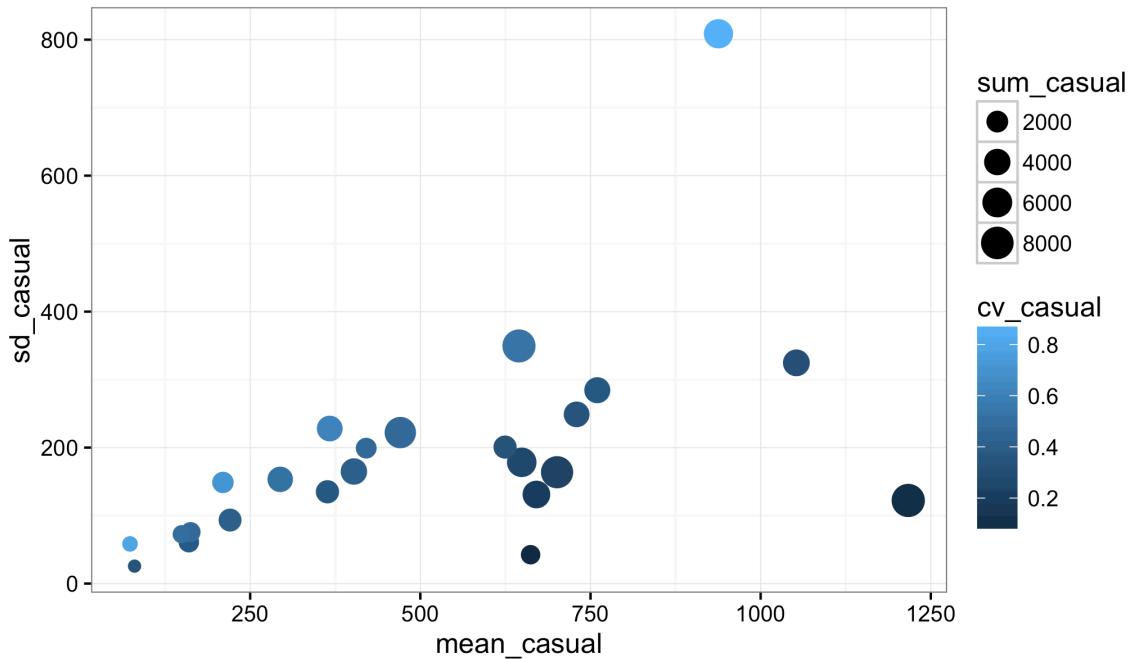
```
ggparcoord(data=banks, columns=c(3:11), groupColumn=12,
  scale="globalminmax", boxplot=TRUE, alphaLines=0.3)
```



Peeking at multivariate data with dplyr and a bubblechart

In *Chapter 3*, we used `dplyr` to string together commands with the `%>%` (“then”) operator to do some grouping and summarization to create a new data frame from the fake customer service data. The following code uses the bike share data to perform essentially the same operations as we saw in *Chapter 3*, but outputs a `ggplot2` bubblechart instead of a data frame:

```
bike_share_daily %>%
  filter(workingday == "Yes" & weathersit != 1) %>%
  group_by(yr, mnth) %>%
  summarize(mean_casual = mean(casual, na.rm=T),
           sd_casual = sd(casual, na.rm=T),
           sum_casual = sum(casual)) %>%
  mutate(cv_casual = round(sd_casual / mean_casual, 2)) %>%
  ggplot(aes(mean_casual, sd_casual, color=cv_casual, size=sum_casual)) +
  geom_point() +
  scale_size(range=c(2,6)) +
  coord_equal() +
  theme_bw()
```



Plotting a table

If you want a pretty table, the combination of `gridExtra` and `ggplot2` can provide an *image* of a table, which is useful if you want to match the style of `ggplot` graphs. We'll first use the `reshape2` package to cast the data into how we want the table to appear, and then plot it. You'll have to remove all the usual plot elements to make it look like a table, instead of a table inside a graph. The rather long `theme` function below accomplishes this.

```
require(reshape2)
require(gridExtra)

bike_share_grp = group_by(bike_share_daily, yr, mnth)

bike_share_mean = summarize(bike_share_grp, mean=mean(casual))

bike_share_mean$mean = round(bike_share_mean$mean, 0)

bike_share_mean_wide = dcast(bike_share_mean, yr~mnth, value.var="mean")

bike_share_mean_wide = rename(bike_share_mean_wide, Year = yr)
```

```
ggplot(bike_share_mean_wide, aes(Year, Jan)) +
  annotation_custom(tableGrob(bike_share_mean_wide, rows=NULL)) +
  ggtitle("Mean Daily Casual Bike Share Use, by Month (2011-2012)") +
  theme_minimal() +
  theme(panel.border = element_blank(), panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(), axis.ticks = element_blank(),
        axis.text = element_blank(), axis.title = element_blank())
```

Mean Daily Casual Bike Share Use, by Month (2011-2012)

Year	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2011	99	814	520	273	223	414	745	1002	1020	1176	930	885
2012	289	1114	700	427	301	1020	1282	1427	1443	1345	1393	1459

Interactive dataviz

For the most part, tooltips and chart interaction is overkill except in the case of multi-group scatterplots. A well-crafted static graph can convey the important messages to decision makers without suggesting the need for time-wasting interaction. However, in some cases interactivity is essential, by saving time, enhancing interpretability, or even if it's just because the boss asked for it. (What the boss wants, the boss gets!)

Most interactive graphs in R are built behind the scenes with [javascript⁵⁵](#) (often d3) using R syntax. Detailed customization is often somewhat difficult, but most packages' defaults are sensible enough that such customization isn't needed. `htmlwidgets` is the primary means by which people integrate interactive data viz with R; we'll use a few of the packages built around that in this section.

We'll start with some basic interactive visualizations by recalling some of the graphs built in earlier chapters, showing the commands that render approximately the same graphs. We'll also add a few new visualizations that demonstrate cases where static plots don't work and interactivity is essential. Of course, the packages shown below have many more options and possibilities, and you should explore their websites and use cases for more information.

Basic interactive plots

We'll use the `taucharts` package to recreate several of the daily bike share data plots in *Chapter 4*. There, `ggplot` did the counting/aggregation work, but here we'll need to use `dplyr` to summarize it first.

⁵⁵<http://shiny.rstudio.com/tutorial/js-lesson1/>

```
# devtools::install_github("hrbrmstr/taucharts")
require(taucharts)
require(dplyr)

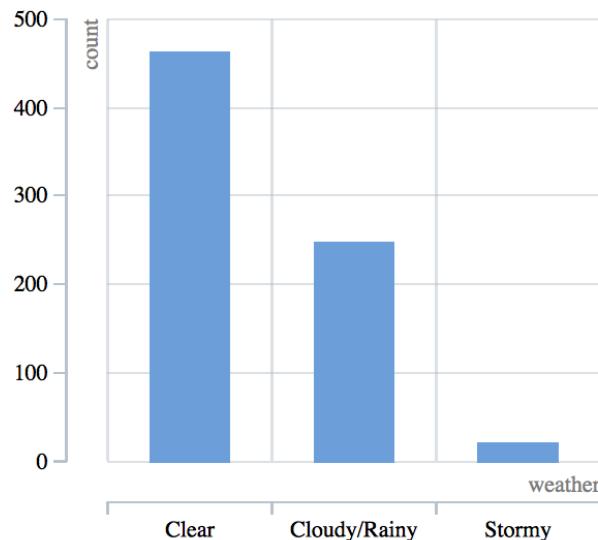
bike_share_daily$weather = as.factor(ifelse
  (bike_share_daily$weathersit == 1, "Clear",
   ifelse(bike_share_daily$weathersit == 2, "Cloudy/Rainy",
   "Stormy")))

bike_share_count = bike_share_daily %>%
  group_by(weather) %>%
  summarize(count=n())

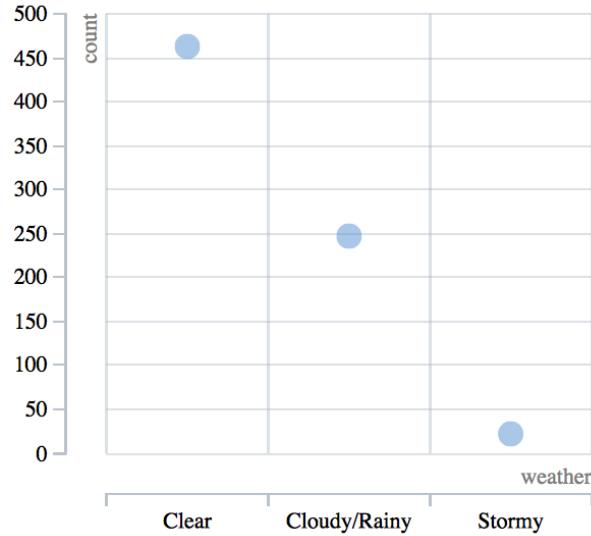
bike_share_count_season = bike_share_daily %>%
  group_by(weather, season) %>%
  summarize(count=n())
```

Now we can start making plots:

```
tauchart(bike_share_count) %>%
  tau_bar("weather", "count") %>%
  tau_tooltip()
```

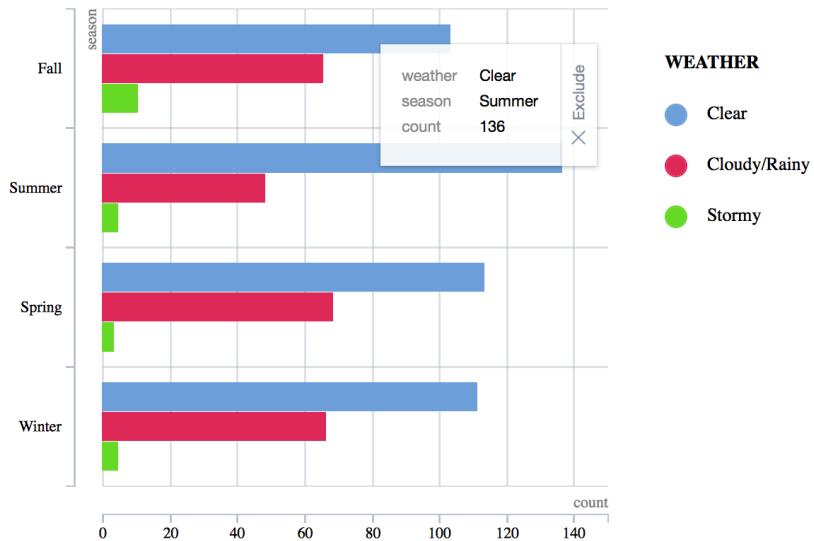


```
tauchart(bike_share_count) %>%
  tau_point("weather", "count") %>%
  tau_tooltip()
```

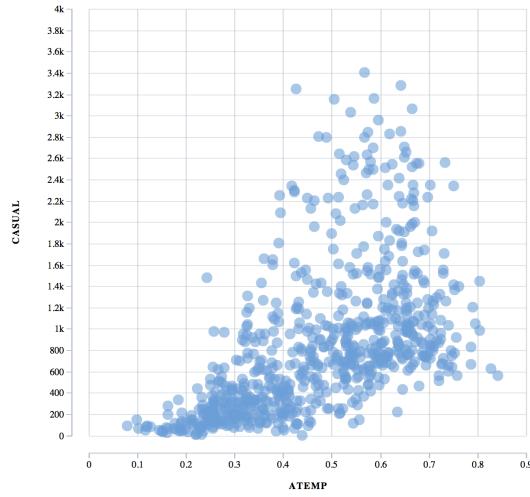


Bivariate and faceted plots are similarly simple (showing what a tooltip looks like here):

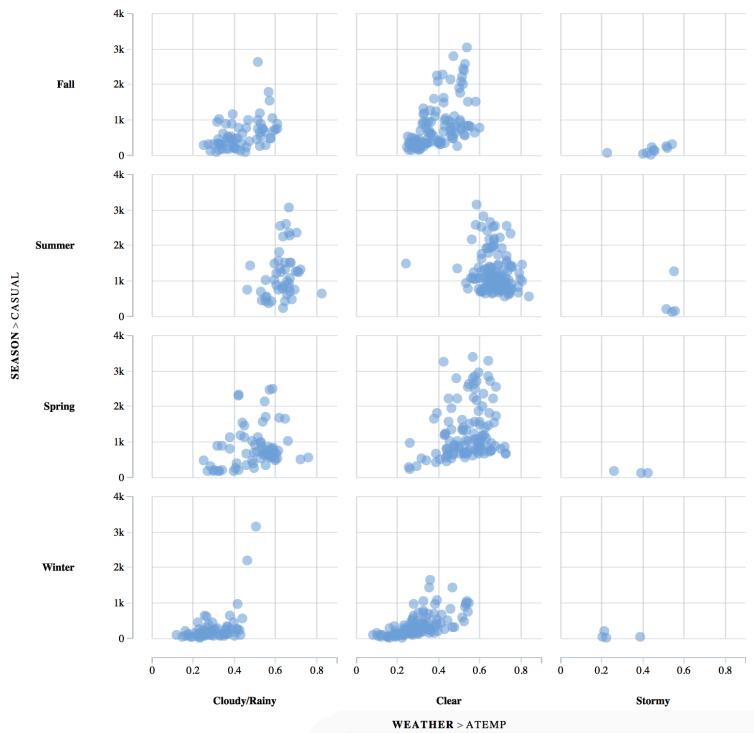
```
tauchart(bike_share_count_season) %>%
  tau_bar("count", "season", "weather", horizontal=T) %>%
  tau_legend() %>%
  tau_tooltip()
```



```
tauchart(bike_share_daily) %>%
  tau_point("atemp", "casual") %>%
  tau_tooltip()
```



```
tauchart(bike_share_daily) %>%
  tau_point(c('weather', 'atemp'), c('season', 'casual')) %>%
  tau_tooltip()
```



Scatterplot matrix

The `pairsD3` package can create interactive scatterplots matrices, with the ability to brush across plots. If you use the group option, note that the default color scheme is not color-blind friendly, so change them to something more clearly separable by all viewers.

```
require(pairsD3)

pairsD3(bike_share_daily[,11:14], group = bike_share_daily[,3],
        opacity = 0.7, tooltip = paste("Season: ", bike_share_daily$season,
        "<br/> Casual use count: ", bike_share_daily$casual),
        col = c("#0571b0", "#92c5de", "#ca0020", "#f4a582"))
```



Motionchart: a moving bubblechart

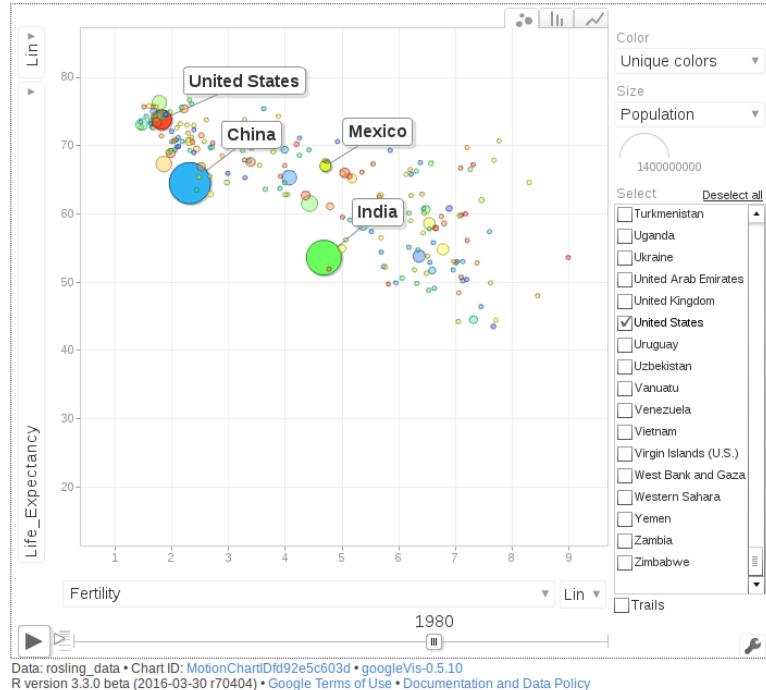
In 1996, Hans Rosling made dataviz history with his [famous TED talk⁵⁶](#) that used a motionchart to challenge the assumptions of global development.

We'll use the `googleVis` package to create a mimic of his famous chart.

```
# load the googleVis package
library(googleVis)

# load the data (an excerpt of data from gapminder.org)
rosling_data = read.csv("https://raw.githubusercontent.com/Rmadillo/
  business_intelligence_with_r/master/manuscript/code/rosling_data.csv")

# create the motion plot using larger plot size than default
plot(gvisMotionChart(rosling_data,
  idvar = "Country",
  timevar = "Year",
  sizevar = "Population",
  options=list(width = 700, height = 600)))
```



⁵⁶https://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve Ever_seen

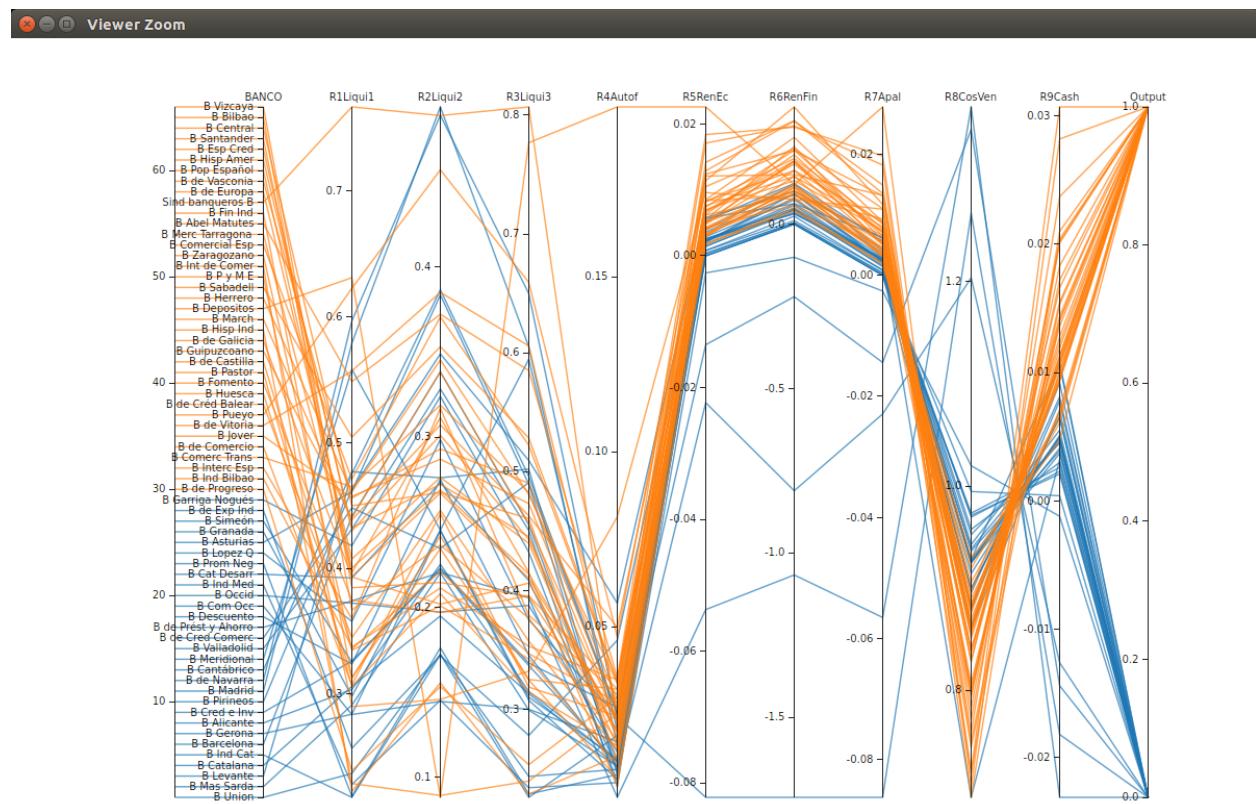
Interactive parallel coordinates

Parallel coordinates plots aren't always useful when you have lots of dimensions or if you want to explore outcomes relative to variables. That's where interactivity becomes essential. Using the `parcoords` `htmlwidget` on the Spanish banks data set, we can set colors based on whether the bank remain solvent, and use interactivity to explore the associated patterns within the 9 financial ratio variables.

```
# devtools::install_github("timelyportfolio/parcoords")

require(parcoords)

parcoords(banks[2:12], reorderable = T, color = list(colorScale=
  htmlwidgets::JS('d3.scale.category10()' ), colorBy="Output" ) ,
brushMode = "1D-axes-multi")
```



Interactive tables with DT

Interactive tables, courtesy of the `DT` package, are made with a single line of code:

```
require(DT)

datatable(rosling_data, filter = "top")
```

		Country	Year	Fertility	Life_Expectancy	Population
		All	1980 ... 2015	All	40.00 ... 70.00	All
7262	Colombia		1982	3.74	70	8051 1376048943
7373	Romania		1982	2.16	70	
7443	Belize		1983	5.34	70	155820
7490	Georgia		1983	2.28	70	5194091
7878	Ecuador		1985	4.22	70	9045977
8014	Ukraine		1985	2.05	70	50920778
8124	South Korea		1986	1.75	70	41059473
8580	Russia		1988	2.14	70	146040116
8783	St. Lucia		1989	3.5	70	135954
8826	West Bank and Gaza		1989	6.51	70	2019486

Showing 1 to 10 of 3,472 entries (filtered from 13,869 total entries)

Previous 1 2 3 4 5 ... 348 Next

Making maps in R

If you need to do real map work, you shouldn't use R, it's just not very good at it. That being said, sometimes all you need is a quick view of a spatial distribution, or perhaps you just need a simple map and you're already in R.

There are several packages that do or support map work, many of which show promise for future use, such as `ggmap`, a mapping package built to fit over `ggplot2` and its coherent graphics approach. But most are still too clunky or buggy when compared against other open source options like QGIS, Leaflet, and others. In addition, Python is the lingua franca language among professional GISers, not R.

Probably the hardest part of making simple maps in R can just be getting set up. There are a variety of mapping and map-support packages, as a tour of the [CRAN spatial task view](#)⁵⁷ shows. Many of them access different APIs as well, some of which require keys. Later in this recipe we'll explore a few more detailed examples, but for now we'll just stick with mapping simple point and area attributes within an R work session.

Since there are different ways to do mapping in R, I've placed the package loading within the steps below instead of at the top of this section, which helps show which package is needed for each step.

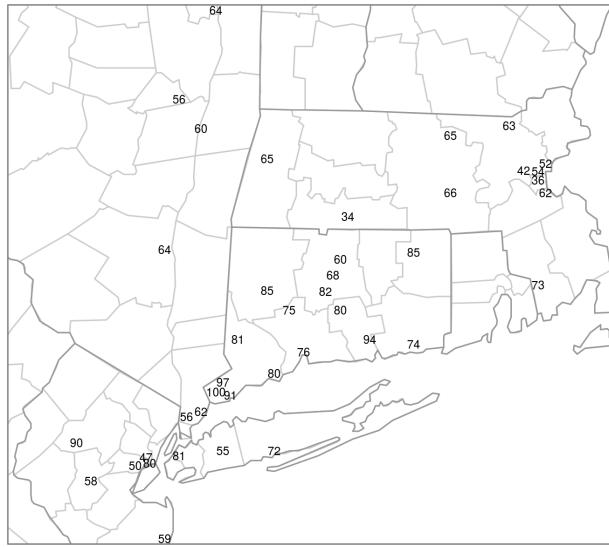
⁵⁷<http://cran.r-project.org/web/views/Spatial.html>

Basic point maps

Creating a map with point-location data plotted on it is deceptively simple with the maps package. To plot this packages example data of ground level ozone over states and counties in southern New England:

```
require(maps)

map("county", xlim=c(min(ozone$x-0.5),max(ozone$x+0.5)), ylim=range(ozone$y),
  col="gray80")
map("state", xlim=c(min(ozone$x-0.5),max(ozone$x+0.5)), ylim=range(ozone$y),
  col="gray60", add=TRUE)
box(col="gray50")
text(ozone$x, ozone$y, ozone$median, cex=0.5)
```



It's "deceptively simple" because as long as you don't need anything fancy or if the boundaries you intend to map haven't changed in a while, you're fine—just use a map base and a dataframe with x (longitude), y (latitude), and data values. But what if the boundaries have changed? Let's look at Europe from the world map within the map package:

```
map('world', xlim=c(-12,45), ylim=c(35,60), col="gray90", fill=T)
box()
```



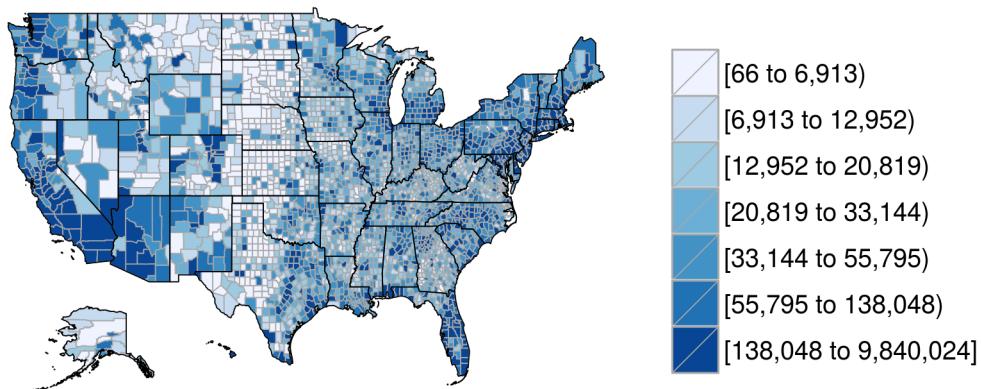
A little outdated, to say the least. A look at the database option inside the `?map` help file tells us why—it's from an old CIA database that predates the collapse of the USSR (the `mapdata` package, which contains more maps for the `maps` package, has the same problem). To get newer boundaries, you'll need to download and convert a spatial data file from an online source, something we'll see an example of below.

Choropleth maps

A second typical type of map is a choropleth, and again, it can appear deceptively simple if we want to, say, plot the 2012 population by county across the United States. Using the `choroplethr` package:

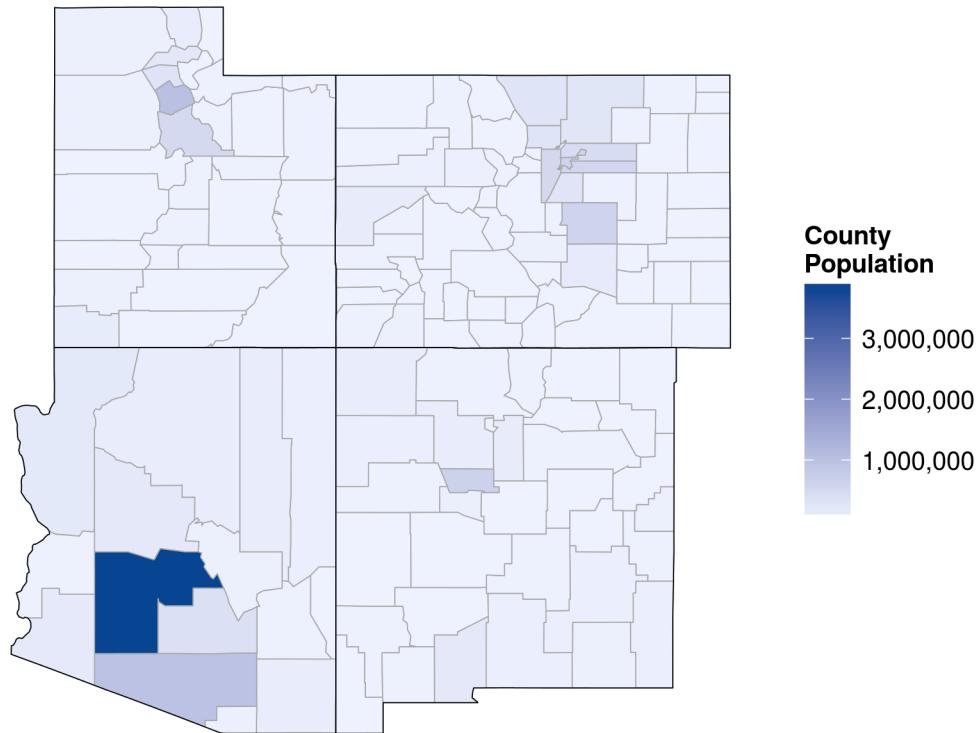
```
require(choroplethr)
require(choroplethrMaps)
data(df_pop_county)
```

```
county_choropleth(df_pop_county)
```



You can subset this to states or counties with the zoom options, and display a continuous scale instead of breaks with the `num_colors=1` option:

```
county_choropleth(df_pop_county,
  legend="County\nPopulation", num_colors=1,
  state_zoom=c("arizona", "colorado", "new mexico", "utah"))
```



The world map data that `choroplethr` and `choroplethrMaps` includes is fairly up to date. But if you want to subset on Europe? Unfortunately, you'll have to do that manually, either by extracting the countries you need by name or ISO code, since `x-` and `y-lim` options don't work in this package (see `?choroplethr_wdi` or `?df_pop_country` for more details on names/ISO codes), or by acquiring map boundaries by different means. We'll explore the latter option below, as it takes considerably less effort.

As you can see from the first example, for a simple point map all you need is longitude (x), latitude (y), and data values. For choropleths (the second example), all you need are a geographical id and data values.

But both of these approaches assume that the map boundaries and scale you need are available in the package(s) you're using, of course. If you understand the geographic projection problem—how

to fit a curved surface to a flat plot—you'll be able to navigate the creation of maps in other spatial packages in ways that ensure the points and the real world line up correctly. If not... you either have to learn a variety of sometimes-contradictory packages in R and their various approaches as well as how they work together, or (probably the better option) you can just use a tool better suited to spatial analyses.

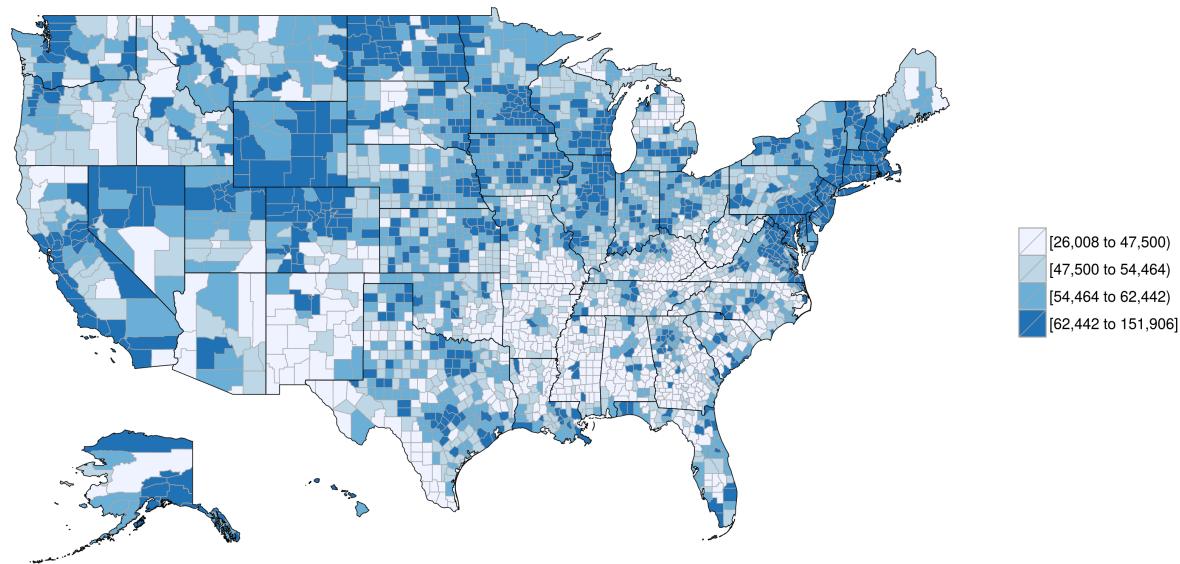
Choropleth mapping with the *American Community Survey*

If you're fortunate enough to need to map American Community Survey (ACS) data, choroplethr connects directly to it via the `acs` package. You'll need to acquire a key for the [ACS API⁵⁸](#) before you begin, but that key will be emailed to you within a few minutes of requesting it. And then you can make one-line maps directly from the ACS tables:

```
require(acs)
api.key.install("YOUR KEY HERE")

choroplethr_acs(tableId="B19113", map="county", buckets=4, endyear=2012)
```

Median Family Income: Median family income in the past 12 months (in 2012 inflation-adjusted dollars)



If you don't know which tables you need, the US Census provides more information [here⁵⁹](#).

⁵⁸<http://www.census.gov/data/developers/about/terms-of-service.html>

⁵⁹http://factfinder2.census.gov/faces/affhel/jsf/pages/metadata.xhtml?lang=en&type=survey&id=survey.en.ACS_ACS

Using shapefiles and raw data

It says a lot about mapping in R that what we'll explore next is one of the simpler ways to get (updated) spatial boundaries into R. Using a boundary shapefile from EuroGeographics—and having looked at the metadata first to determine its projection—we can create a map of modern European boundaries using the `maptools` package's shapefile translation functions:

```
download.file("http://epp.eurostat.ec.europa.eu/cache/GISCO
/geodatafiles/CNTR_2014_03M_SH.zip", destfile="~/world2014.zip")

unzip("~/Downloads/world2014.zip", exdir "~/Downloads/world2014",
junkpaths = TRUE)

require(maptools)

worldmap2014 = readShapeLines("~/Downloads/world2014/CNTR_BN_03M_2014.shp",
proj4string = CRS("+proj=longlat +datum=WGS84"))

plot(worldmap2014, xlim=c(7,16), ylim=c(35,60), col="gray60")
box(col="gray30")

# © EuroGeographics for the administrative boundaries
```



We plotted ground level ozone from the `maps` example data pretty easily—but what does it take to create a real one? The EU air quality database requires human intervention to [download the data⁶⁰](#), so for convenience it's already available in this book's GitHub repo (subset to ozone pollution data and linked to each station's geographic coordinates; code to do so yourself is [here⁶¹](#)). We'll plot the annual daily median ozone values with the modern EU boundaries obtained above:

⁶⁰<http://www.eea.europa.eu/data-and-maps/data/airbase-the-european-air-quality-database-8>

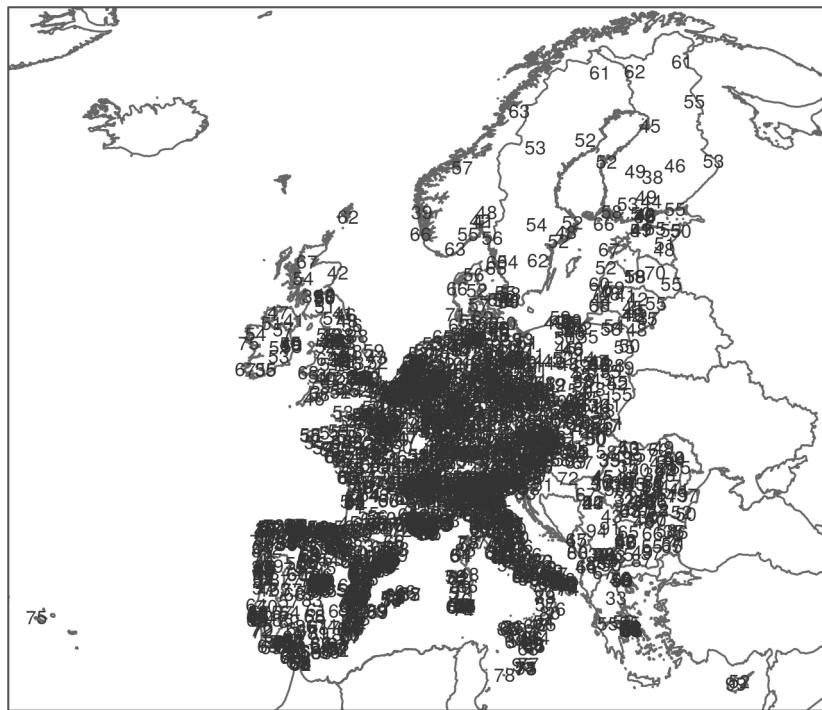
⁶¹https://raw.githubusercontent.com/Rmadillo/business_intelligence_with_r/master/manuscript/code/eu_o3.R

```

eu_o3 = read.table("https://raw.githubusercontent.com/Rmadillo/
  business_intelligence_with_r/master/manuscript/code/eu_o3.csv",
  sep=",", header=T)

plot(worldmap2014, xlim=c(2,12), ylim=c(35, 70), col="gray60")
box(col="gray30")
text(eu_o3$longitude, eu_o3$latitude, round(eu_o3$statistic_value,0), cex=0.5,
  col="gray20")

```



Too much data! Turning the values into points with a color scale takes a little more effort, but allows us to see the data:

```

require(RColorBrewer)

o3_colors = brewer.pal(5, "OrRd")

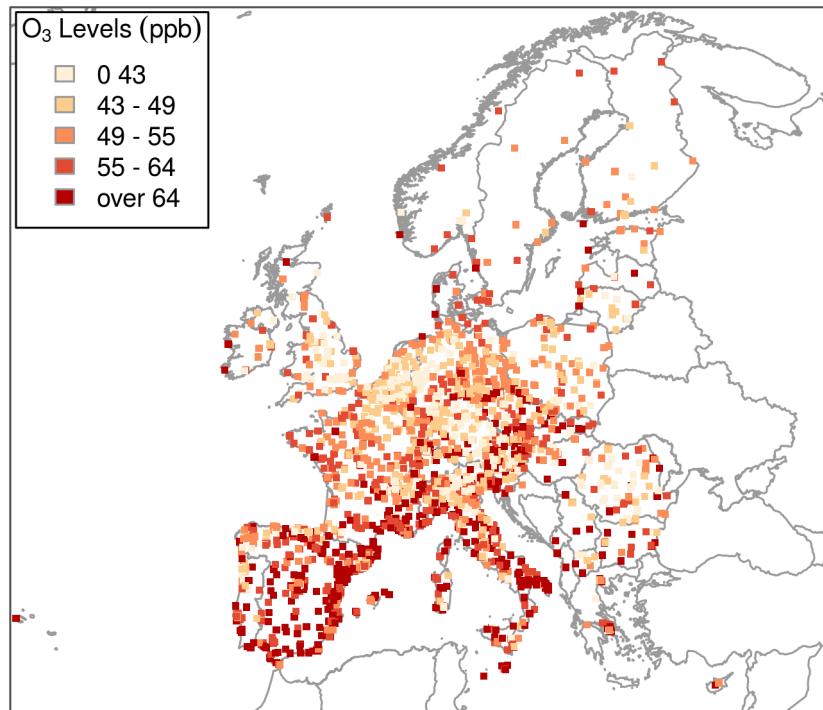
require(classInt)

o3_breaks = classIntervals(sort(eu_o3$statistic_value), n=5, style="quantile")

plot(worldmap2014, xlim=c(2,12), ylim=c(35, 70), col="gray60")
box(col="gray30")

```

```
points(eu_o3$longitude, eu_o3$latitude, pch=15, cex=0.5,  
       col=o3_colors[findInterval(eu_o3$statistic_value, o3_breaks$brks,  
       all.inside=TRUE)])  
legend("topleft", title=expression(0[3]~Levels~(ppb)), inset=0.007, cex=0.8,  
      legend=leglabs(round(o3_breaks$brks),0), fill=o3_colors, bg="white",  
      border="gray60")
```



Using ggmap for point data and heatmaps

The ggmap package makes it relatively easy to map point-location data, as well as to convert those points into “heatmaps” (actually, a density map).

We can use a Socrata connection to download data from the [Seattle Police Department](#)⁶² for this recipe.

⁶²<https://data.seattle.gov>

```

require(ggmap)
require(dplyr)

# SPD Police Reports
spd = read.csv("https://data.seattle.gov/api/views/7ais-f98f/rows.csv",
               header=T, strip.white = T)

# Filter to 2 years with full data
spd = filter(spd, Year >= 2014 & Year < 2016)

# Filter to bike thefts only
bike_theft = filter(spd, Summarized.Offense.Description == "BIKE THEFT")

```

You can set up the base map using the `get_map` function and include the data you want to overplot as the base layer:

```

## Set up base map
seattle_map = ggmap(get_map('Seattle, Washington',
                           zoom=11, source='google', maptype='terrain'),
                     base_layer = ggplot(aes(x = Longitude, y = Latitude),
                                         data = bike_theft))

```

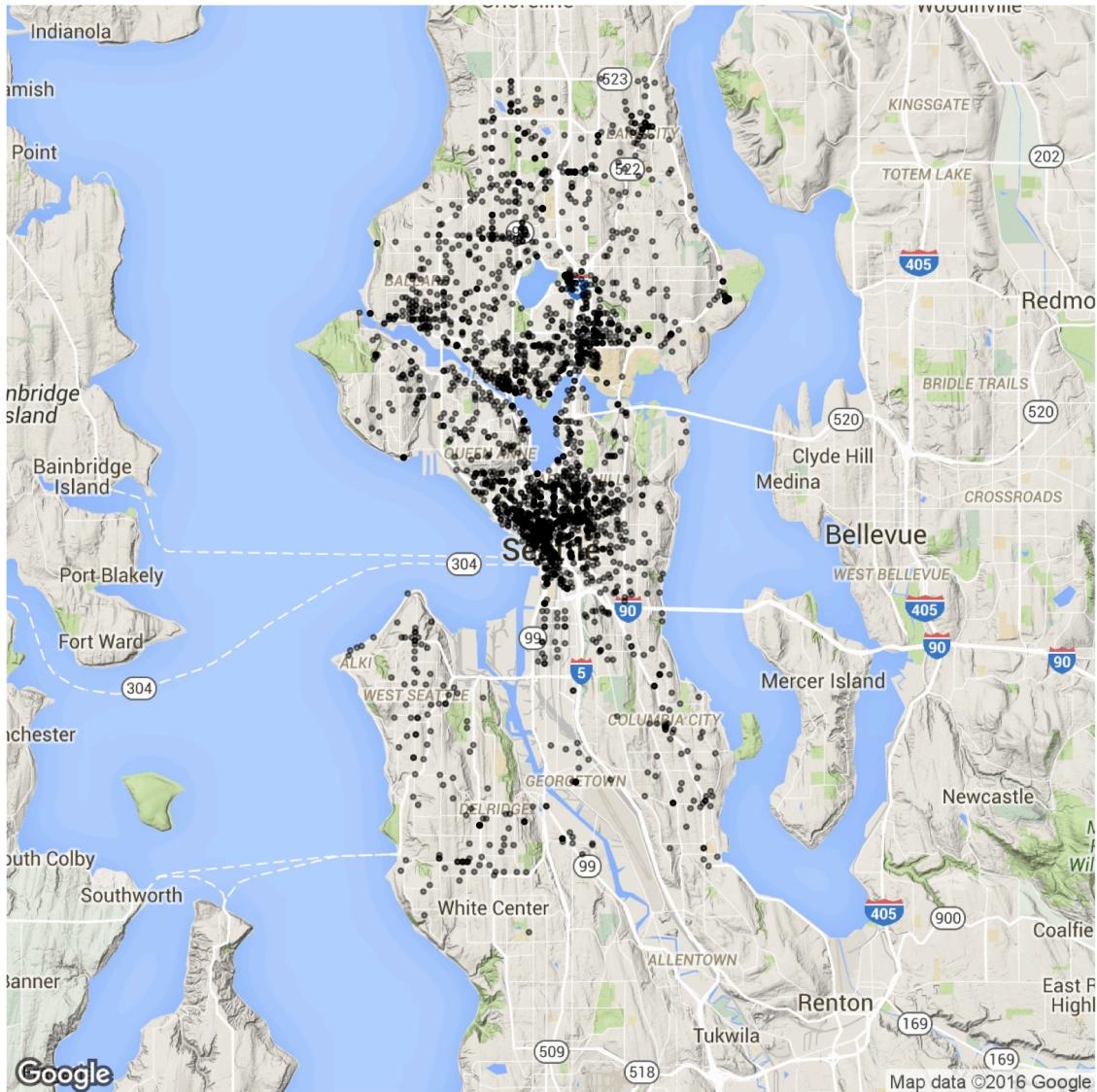
Once the base map is set up, you can plot it a variety of ways, including as individual points...

```

# Point map of bike theft
seattle_map +
  geom_point(alpha=0.5, size=0.5) +
  scale_alpha(guide = FALSE) +
  ggtitle('Seattle Bike Thefts (2014-2015)') +
  xlab('') +
  ylab('') +
  theme(axis.ticks=element_blank(), axis.text=element_blank(),
        legend.position='none')

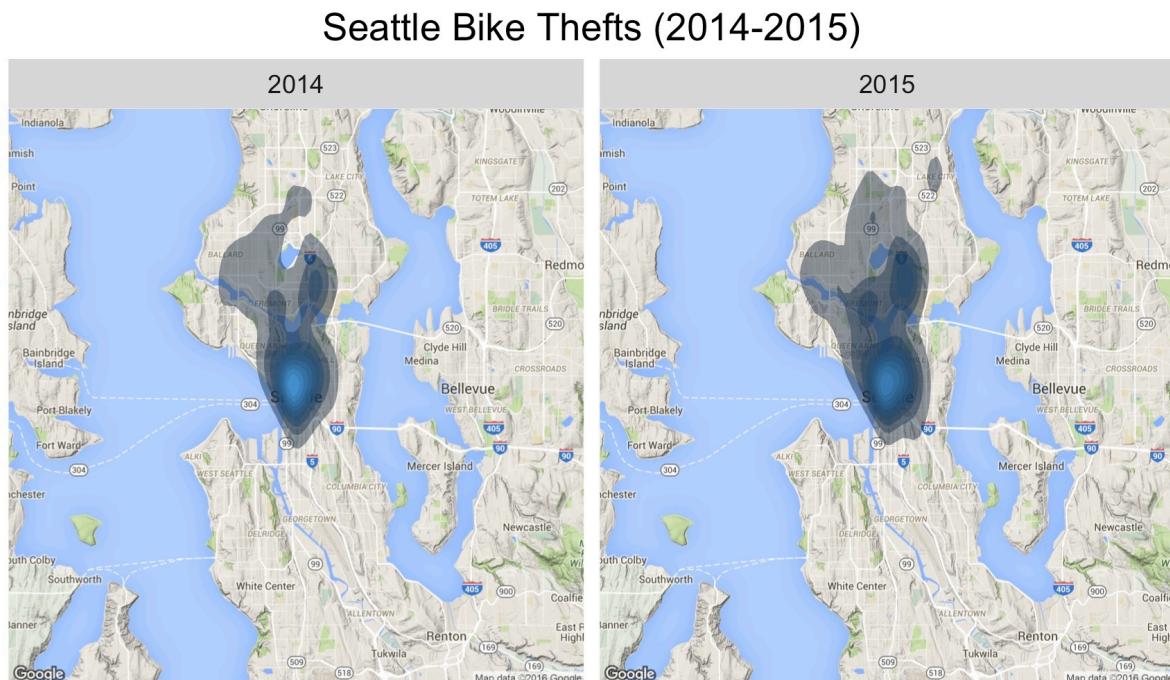
```

Seattle Bike Thefts (2014-2015)



...or as faceted heatmaps:

```
# Density map of bike theft by year
seattle_map +
  stat_density2d(aes(fill=..level..),
    alpha=0.5,
    bins = 10,
    geom = 'polygon') +
  scale_alpha(guide = FALSE) +
  ggtitle('Seattle Bike Thefts (2014-2015)') +
  xlab('') +
  ylab('') +
  facet_wrap(~Year) +
  theme(axis.ticks=element_blank(), axis.text=element_blank(),
    legend.position='none')
```



Interactive maps with leaflet

The `leaflet` package makes basic interactive maps fairly easy to build. We'll use a subset of the bike theft data to avoid cluttering the map. We'll also add a few options to show some customization

possibilities; walking through the [leaflet documentation](#)⁶³ provides examples of a wide variety of map object and customization options.

```
require(leaflet)

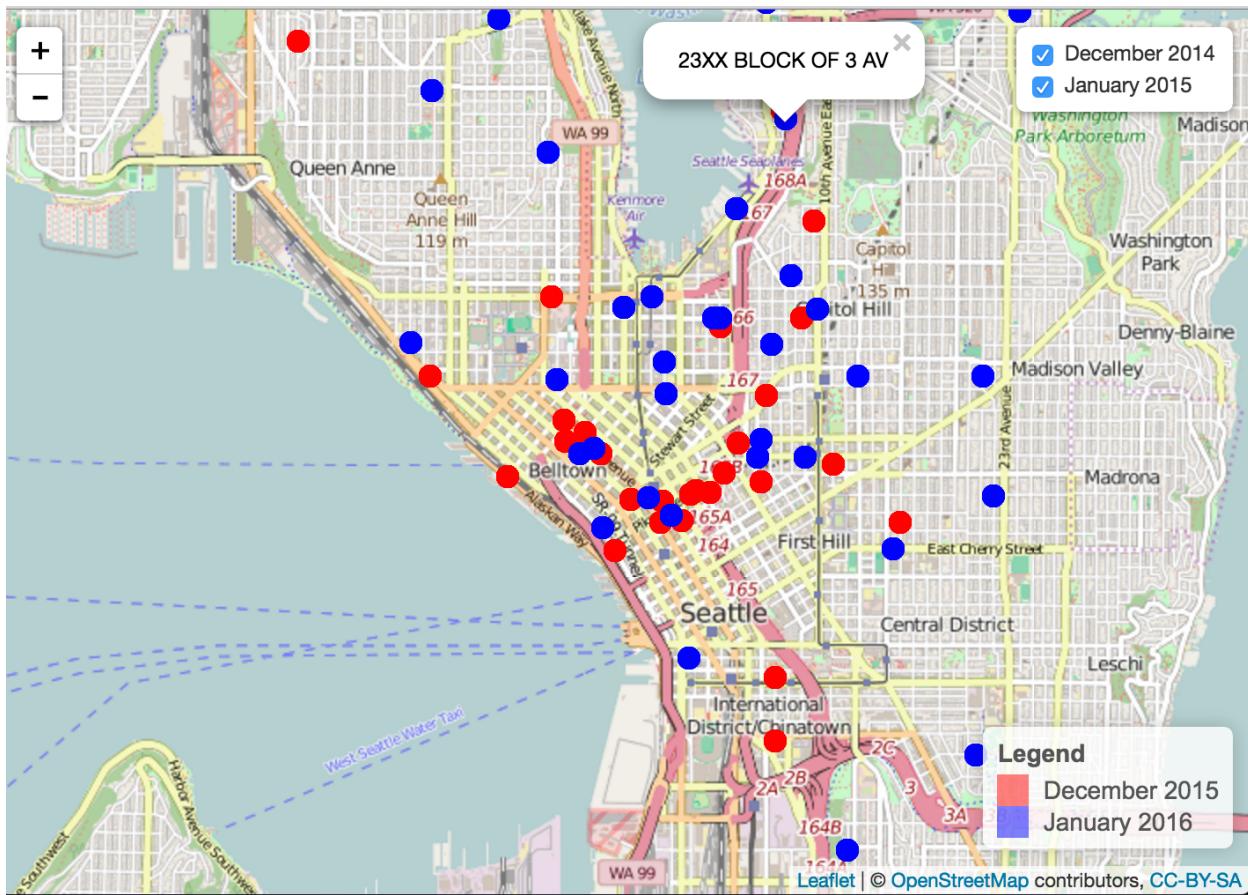
bike_theft_dec = filter(bike_theft, Year==2014 & Month==12)

bike_theft_jan = filter(bike_theft, Year==2015 & Month==1)

bike_theft_interactive = leaflet() %>%
  addTiles() %>%
  setView(-122.3397, 47.6144, zoom = 13) %>%
  addCircleMarkers(data = bike_theft_dec,
    lat = ~ Latitude,
    lng = ~ Longitude,
    popup = bike_theft$Hundred.Block.Location,
    group="December 2014",
    fillOpacity = 0.25,
    color="red",
    radius=4) %>%
  addCircleMarkers(data = bike_theft_jan,
    lat = ~ Latitude,
    lng = ~ Longitude,
    popup = bike_theft$Hundred.Block.Location,
    group="January 2015",
    fillOpacity = 0.25,
    color="blue",
    radius=4) %>%
  addLayersControl(overlayGroups = c("December 2014", "January 2015"),
    options = layersControlOptions(collapsed = FALSE)) %>%
  addLegend( "bottomright", title = "Legend",
    colors=c("red","blue"), labels=c("December 2014", "January 2015"))

bike_theft_interactive
```

⁶³<https://rstudio.github.io/leaflet/>



Why map with R?

Mapping in R is relatively easy *if* the maps, boundaries, and scale you need are available in the package(s) you're using. If not, you'll have to do it manually, at considerably more time and mental expense than you'd have spent doing it on a platform more amenable to spatial analysis. The problem only gets harder if you're not comfortable working with geographic projections.

My recommendation: don't use R for mapping unless your need is solved by existing functionality. If you do need to work with spatial data in R, one of the best resources to learn some ways to create beautiful and useful maps in R is James Cheshire's spatial.ly blog⁶⁴.

⁶⁴<http://spatial.ly/>

Chapter 8: Pattern Discovery and Dimension Reduction

- Mapping multivariate relationships with non-metric multidimensional scaling (nMDS), principal components analysis (PCA), and correspondence analysis (CA)
- Grouping observations with hierarchical clustering
- How to partition the results of a hierarchical cluster analysis
- Identifying and describing group membership with kMeans and PAM
- Determining optimal numbers of groups with model-based clustering
- Identifying group membership with irregular clusters
- Variable selection in cluster analysis
- Error checking cluster results when you have known outcomes
- Exploring outliers
- Finding associations in shopping carts

Humans like to put things in boxes. “What things are most like each other?” is one of the most common questions you face in the analysis of multivariate data. Technically, we can think of these tasks as dimension reduction—how can we take a mass of data and reduce it to something that provides insight? Techniques to take many variables and cluster observations or detect outliers have been around for decades, though it’s been only recently where desktop computing power was sufficient for more advanced similarity-analysis tool development.

Mapping multivariate relationships

Non-metric multidimensional scaling (nMDS)

Non-metric multidimensional scaling (nMDS) is one such tool. It’s similar to principal components analysis (PCA), but does not suffer from the assumption that the multivariate relationships must be linear. As a result, it has already replaced PCA as the primary descriptive and dimension reduction tool for multivariate data in some fields, and is moving rapidly into others. But although nMDS works better for nearly all applied problems and is easier to explain to business clients, PCA is still widely used.

Non-metric multidimensional scaling (nMDS) is a useful way to identify groups, outliers, and influential variables in multivariate space. In essence, nMDS collapses multiple dimensions into two or three so that relationships based on multivariate proximity can be visualized. nMDS models

a similarity matrix in low dimensional space in much the same way as a map can be created from scattered GPS coordinates.

The data in this example we first saw in *Chapter 7*—nine financial ratios for 66 Spanish banks, 29 of which failed during the Spanish banking crisis ([Serrano-Cinca 2001⁶⁵](#)).

The nMDS analysis is largely automated in the *vegan* package using the `metaMDS` function (see `?metaMDS` for all the details). This function defaults to expecting count data; for continuous data, setting the distance metric to `euclidean` is more appropriate. There are dozens of distance metrics (see `?vegdist` for the options available in *vegan*), but only two are really needed for most applications: `Bray-Curtis` for count data (the default) and `Euclidean` for continuous data. `metaMDS` automatically performs transformations when given count data. Manual transformations are sometimes needed if the variables are continuous and have considerably different scales (e.g., scale to mean 0/sd 1, see `?scale`).

```
require(vegan)

banks = read.csv("https://raw.githubusercontent.com/Rmadillo/
  business_intelligence_with_r/master/manuscript/code/quiebra.csv",
  encoding = "UTF-8", stringsAsFactors = FALSE, header = TRUE)

banks_mds = metaMDS(banks[,3:11], distance="euclidean", autotransform=FALSE,
  noshare=FALSE, wascores=FALSE)

Run 0 stress 0.03123018
Run 1 stress 0.03122783
... New best solution
... procrustes: rmse 0.0007384291 max resid 0.004514075
*** Solution reached
```

The plot that is built from this algorithm is the heart of nMDS analysis. Categorical information can be added as well.

Axis values are meaningless in nMDS except as x/y coordinates; only spatial pattern matters for interpretation. Much like a map, points that lie close together on the plot are more similar, while points that are further away are more dissimilar:

⁶⁵<http://www.tandfonline.com/doi/abs/10.1080/13518470122202>

```

# You may need to mess with par for the margins. Note: Clearing the
# plots in RStudio (broom icon) returns par to defaults
# par(mar=c(1.5,0,1,0))

# Save the stress value for the legend
banks.stress = round(banks_mds$stress, digits=2)

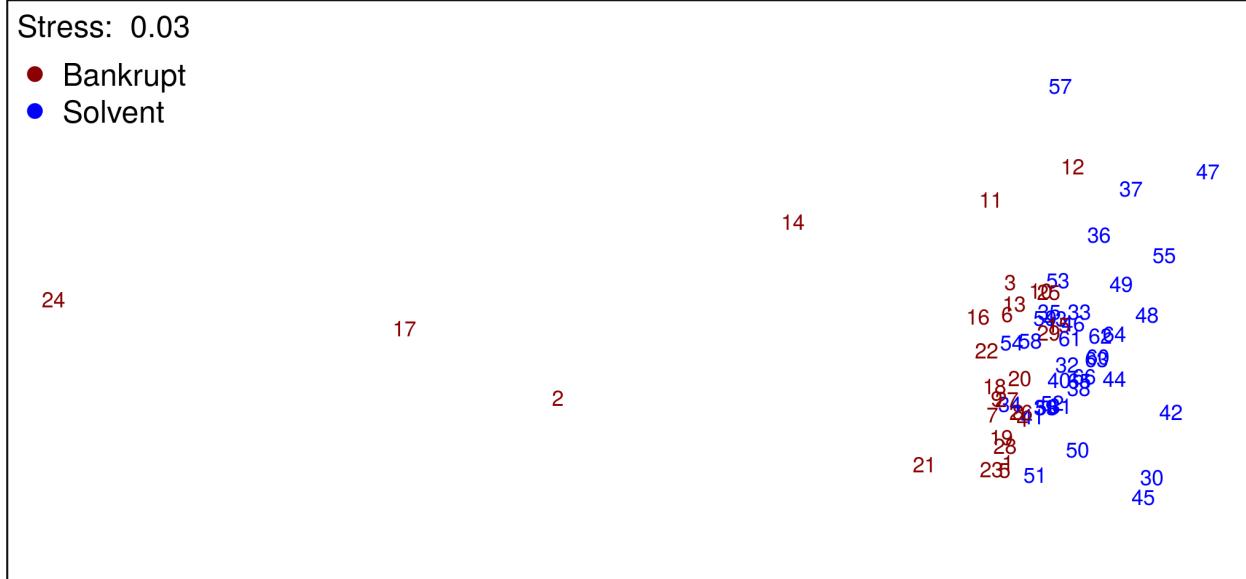
# Set up color scheme
groups = ifelse(banks$output == 0, "darkred", "blue")

# Create an empty plot
ordiplot(banks_mds, type="n", xaxt="n", yaxt="n", xlab=" ", ylab=" ",
display="sites")

# Add banks by number and color (solvency)
orditorp(banks_mds, display="sites", col=groups, air=0.01, cex=0.75)

# Add legend
legend("topleft", legend=c("Bankrupt", "Solvent"), bty="n",
col=c("darkred", "blue"), title=paste0("Stress: ", banks.stress),
pch=21, pt.bg=c("darkred", "blue"), cex=0.75)

```

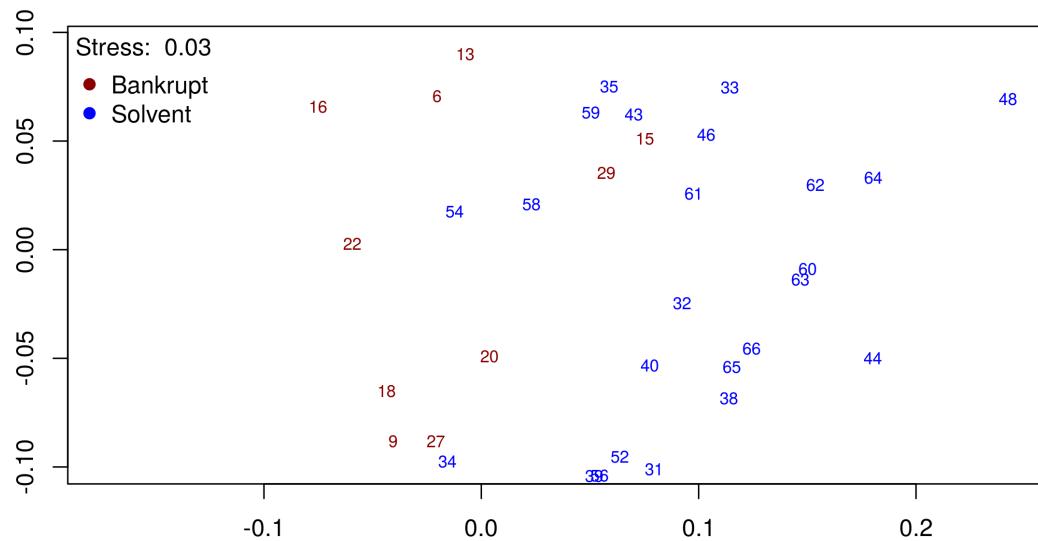


You also can easily explore regions within the plot in more detail by using `xlim` and `ylim` to zoom in on an area of interest, such as the right-side cluster (axes included here to show how to set zoom limits):

```
# Zoom in to the cluster
ordiplot(banks_mds, type="n", xlab=" ", ylab=" ", display="sites",
  xlim=c(0,0.07), ylim=c(-0.1, 0.095))

orditorp(banks_mds, display="sites", col=groups, air=0.01, cex=0.75)

legend("topleft", legend=c("Bankrupt", "Solvent"), bty="n", col=c("darkred",
  "blue"), title=paste0("Stress: ", banks.stress), pch=21, pt.bg=c("darkred",
  "blue"), cex=0.75)
```



nMDS does not use the absolute values of the variables, but rather their rank orders. While this loses some information contained in the raw data, the use of ranks omits some of the issues associated with using absolute distance (e.g., sensitivity to transformation and linearity), and as a result is a very flexible technique that works well with a variety of types of data. It's also particularly useful when there are a large number of predictors relative to the number of entities, where using actual values could easily lead to overfitting.

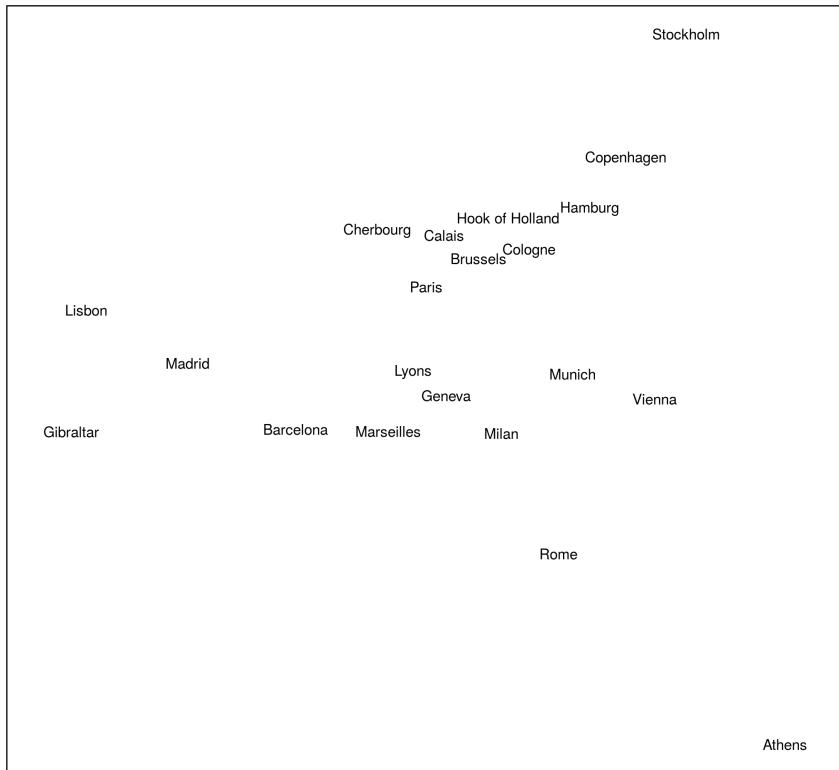
In short, nMDS creates a map of a similarity matrix. For example, a matrix of European inter-city distance (see ?eurodist) creates an nMDS plot that closely reflects the distances and spatial direction between these cities:

```
# You may need to mess with par for the margins
# par(mar=c(1.5,4,1,4))

euromap = metaMDS(eurodist, distance="euclidean", autotransform=FALSE,
noshare=FALSE, wascores=FALSE)

euromap$points[,1] = -euromap$points[,1]

ordiplot(euromap, display="sites", xaxt="n", yaxt="n", xlab=" ", ylab=" ",
type="t")
```



Diagnostics for nMDS results

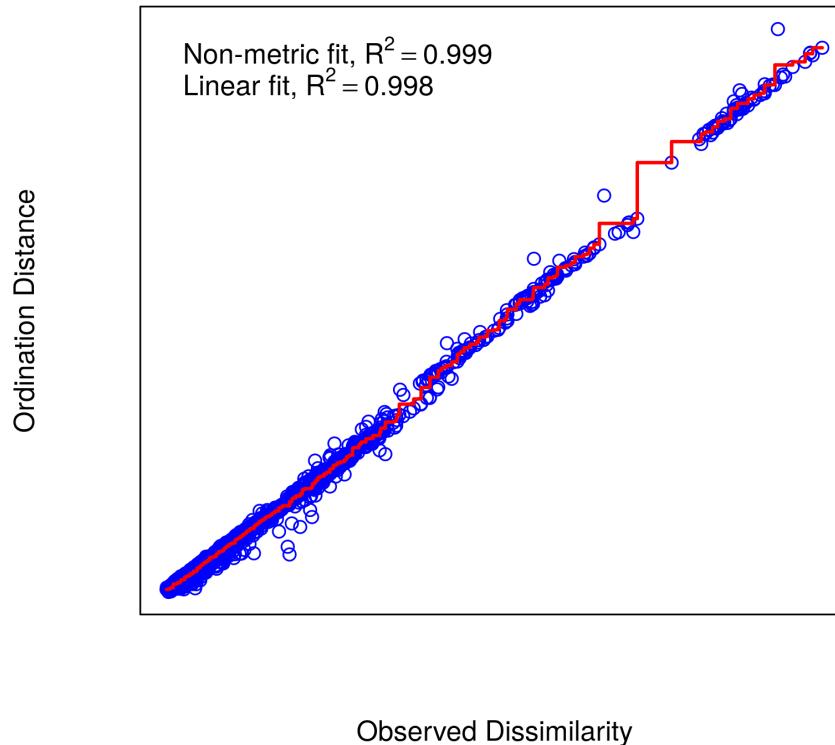
Diagnostics for nMDS results are also simple. A Shepard plot (also called a stress plot) shows the relationship between the similarity matrix and how well the nMDS algorithm represented that matrix in the plot. The greater the scatter around the line, the higher the stress, which would imply that the matrix similarities are not as well represented in the plot.

```
banks_stress = round(banks_mds$stress, digits=2)
```

```
banks_stress
```

```
0.03
```

```
stressplot(banks_mds, xaxt="n", yaxt="n")
```



The stress value itself is a single value that shows how well the plot represents the underlying similarity matrix. It can be roughly interpreted as follows:

Stress Score	How well the plot represents the matrix
< 0.05	Excellent
> 0.05-0.10	Very Good
> 0.10-0.15	Good
> 0.15-0.20	Fair
> 0.20-0.30	Poor
> 0.30	No representation

Based on the diagnostics, this plot represents the similarity matrix quite well, so we can be confident in the nMDS representation of this dataset.

Note: in general, stress values rise with the size of the dataset. Thus, interpreting the final stress score requires considering the size of the dataset in addition to considering the scatter in the Shepard plot and the patterns revealed in the nMDS plot. When stress is high, sometimes it can be brought down by `metaMDS` options `k` and `trymax`, but the guidelines above should not be considered definitive—sometimes clear patterns can be seen even with relatively higher levels of stress.

The final configuration may differ depending on the noise in the data, the initial configuration, and the number of iterations, so running the nMDS multiple times and comparing the interpretations from the lowest stress results is a good practice. `metaMDS` does this automatically, however, so multiple restarts are only necessary when running nMDS manually.

Vector mapping influential variables over the nMDS plot

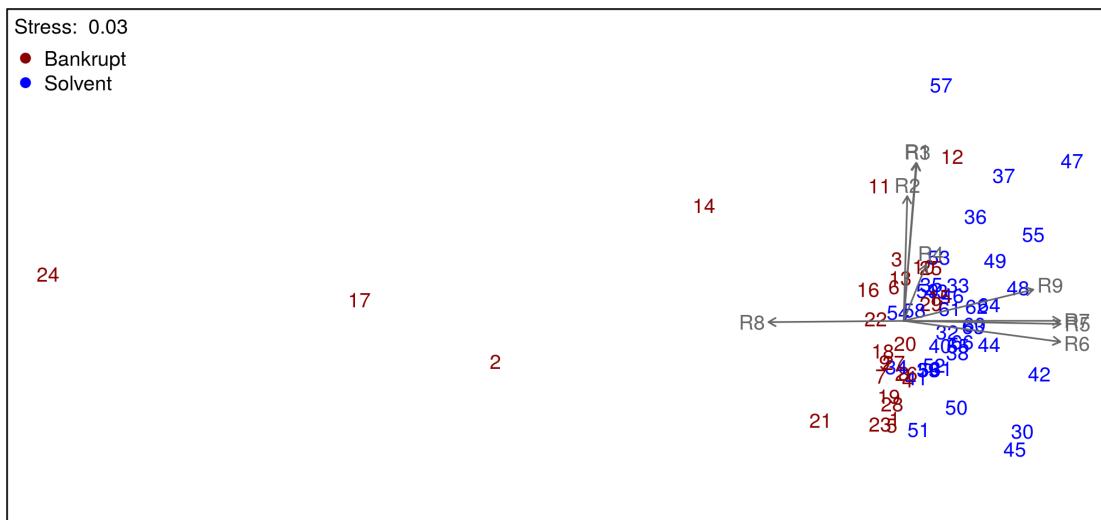
You can add vectors of the variables used to construct the matrix over the nMDS plot to explore how they influenced the final configuration. The length and direction of each vector gives a relative sense of each variable's influence on the “map”:

```
banks_vectors = envfit(banks_mds, banks[,3:11])

ordiplot(banks_mds, type="n", xaxt="n", yaxt="n", xlab=" ", ylab=" ",
         display="sites")

orditorp(banks_mds, display="sites", col=groups, air=0.01, cex=0.75)

plot(banks_vectors, col="gray40", cex=0.75)
legend("topleft", legend=c("Bankrupt", "Solvent"), bty="n",
       col=c("darkred", "blue"), title=paste0("Stress: ", banks$stress),
       pch=21, pt.bg=c("darkred", "blue"), cex=0.75)
```



Contour mapping influential variables over the nMDS plot

Since there is a clear left-right gradient, and the R8 variable (the ratio of Cost of Sales to Sales) seems to be one of the major reasons for that. A contour of that variable can be mapped in the nMDS plot; contours are plotted in each variable's original scale. Other contours can be mapped as well; this example shows R8 and R4 contours:

```

# par(mar=c(1.5,0,1,0))

# Empty plot
ordiplot(banks_mds, type="n", xaxt="n", yaxt="n", xlab=" ", ylab=" ",
  display="sites")

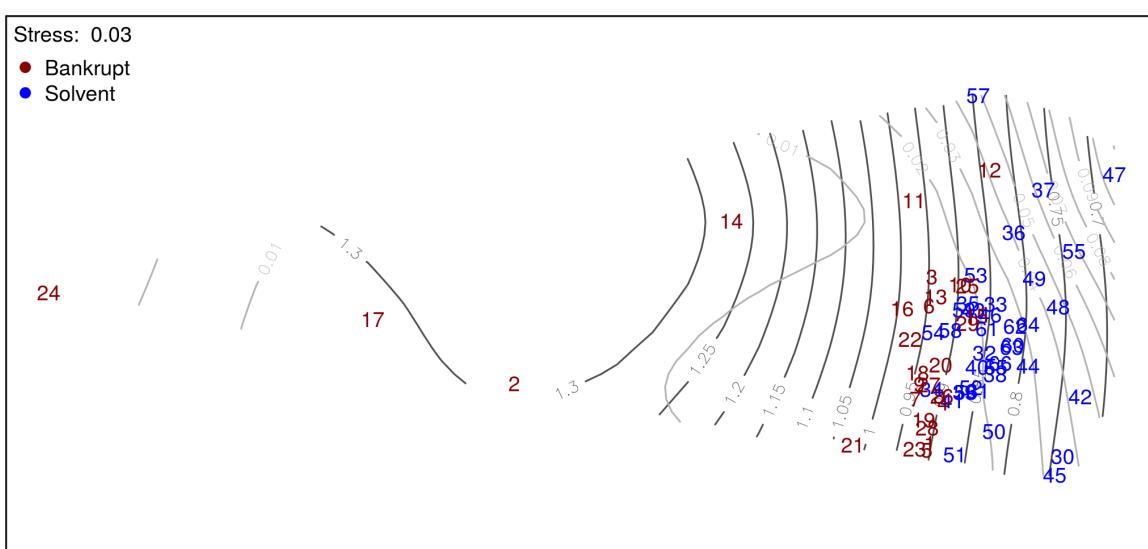
# Add contour of variable R8 (Cost of Sales:Sales ratio)
# Putting it in tmp suppresses console output
tmp = ordisurf(banks_mds, banks$R8, add=TRUE, col="gray30")

# Add contour of variable R4 (Reserves:Loans ratio)
tmp = ordisurf(banks_mds, banks$R4, add=TRUE, col="gray70")

# Plot nMDS solution
orditorp(banks_mds, display="sites", col=groups, air=0.01, cex=0.75)

legend("topleft", legend=c("Bankrupt", "Solvent"), bty="n",
  col=c("darkred", "blue"), title=paste0("Stress: ", banks$stress),
  pch=21, pt.bg=c("darkred", "blue"), cex=0.75)

```



While technically any number of contours can be plotted, in practice more than two renders the plot essentially unreadable.

Principal Components Analysis (PCA)

PCA is often used as an exploratory technique in similar ways to nMDS, and in some cases where your data have similar scales and there are linear combinations between the variables, it can sometimes prove more powerful. If the assumptions can be justified, PCA has the added benefit of being able to provide dimension reduction through linear combinations of the original variables, in which case it can be used as input to other tools, such as regression.

```
banks_pca = princomp(banks[,3:11], cor=TRUE)
```

```
summary(banks_pca, loadings=TRUE)
```

Importance of components:

	Comp.1	Comp.2	Comp.3	Comp.4	Comp.5	\
Comp.6						
Standard deviation	2.1811785	1.5982195	1.0433711	0.62850818	0.33908817	0.248\
945162						
Proportion of Variance	0.5286155	0.2838117	0.1209581	0.04389139	0.01277564	0.006\
885966						
Cumulative Proportion	0.5286155	0.8124273	0.9333854	0.97727679	0.99005243	0.996\
938395						
	Comp.7	Comp.8	Comp.9			
Standard deviation	0.161143044	0.0392016948	7.113040e-03			
Proportion of Variance	0.002885231	0.0001707525	5.621704e-06			
Cumulative Proportion	0.999823626	0.9999943783	1.000000e+00			

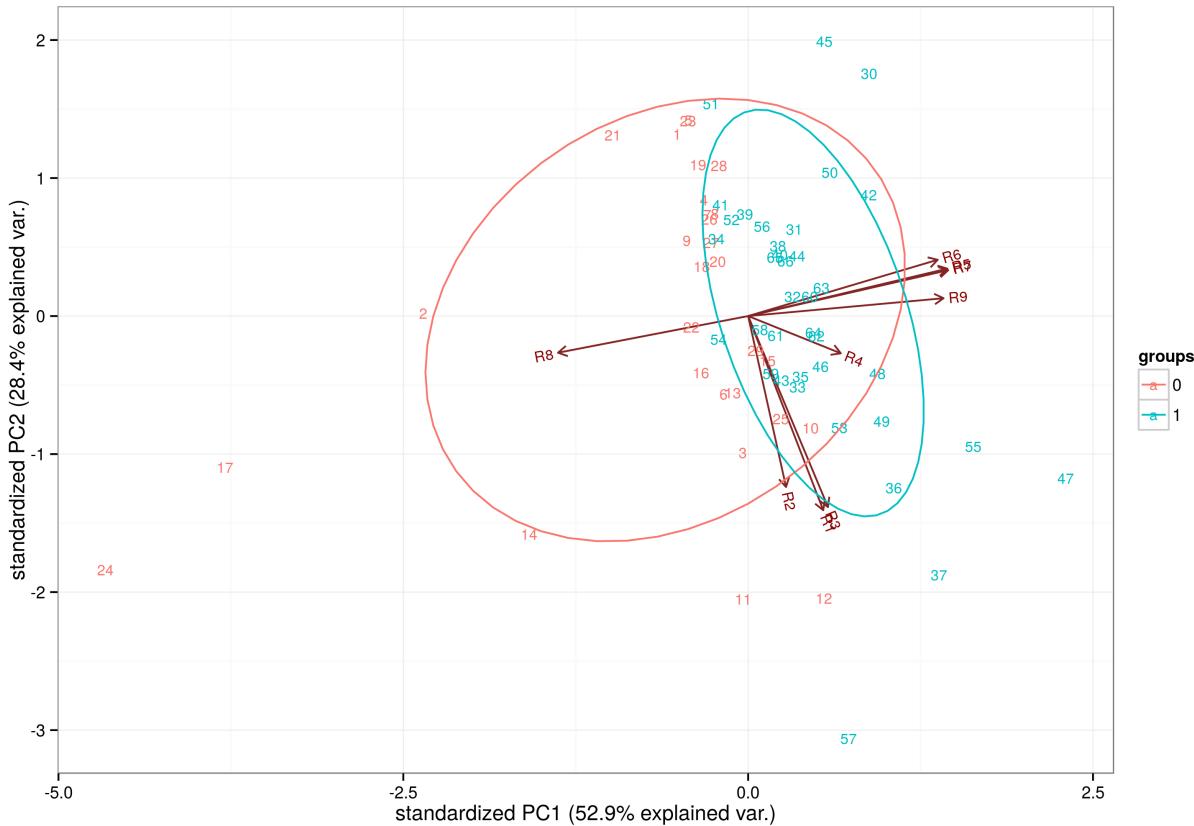
Loadings:

	Comp.1	Comp.2	Comp.3	Comp.4	Comp.5	Comp.6	Comp.7	Comp.8	Comp.9
R1	0.163	-0.575		-0.190	-0.328	0.115		0.693	
R2		-0.506	-0.452	0.342	0.601	-0.207	-0.105		
R3	0.173	-0.566	0.112	-0.200	-0.254	0.143	0.146	-0.698	
R4	0.201	-0.110	0.831		0.445		-0.228		
R5	0.433	0.141	-0.114	-0.306	0.196		0.344	0.118	-0.715
R6	0.412	0.167	-0.254	-0.300			-0.790	-0.120	
R7	0.434	0.136		-0.308	0.205		0.404		0.693
R8	-0.413	-0.109		-0.570		-0.695			
R9	0.424			0.451	-0.430	-0.648			

The basic `biplot` function provides the essential details (e.g., `biplot(banks_pca, col=c("black", "gray40"))`) but a function in the `ggbio` package does the same, only via `ggplot2` for a prettier image:

```
# devtools::install_github("vqv/ggbio")
require(ggbio)

ggbio(banks_pca, labels=rownames(banks), ellipse=T,
      groups=as.factor(banks$Output)) +
  theme_bw()
```



Variance plots usually accompany PCA work; here's a way to do it with ggplot:

```

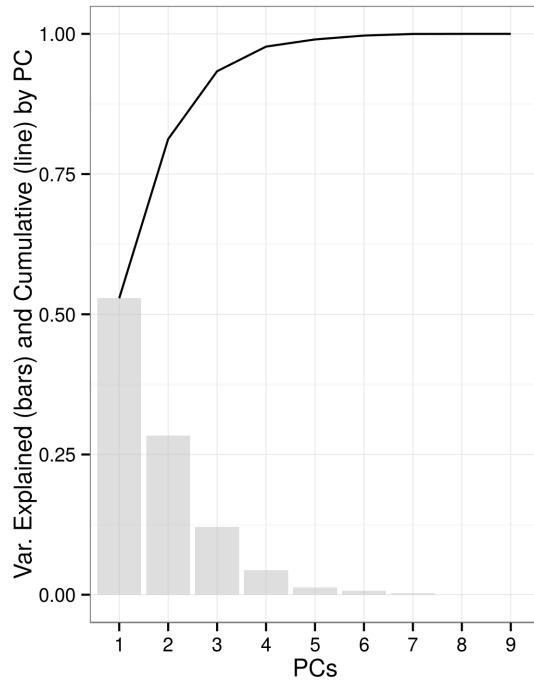
# Get variances
vars = banks_pca$sdev^2

# Get proportion of variance
var_exp = vars/sum(vars)

# Get cumulative variance and make into data frame
variances = data.frame(vars = vars, var_exp = var_exp,
                       var_cumsum = cumsum(var_exp), PCs = 1:length(banks_pca$sdev))

# Plot variance explained and cumulative variance
ggplot(variances, aes(PCs, var_cumsum)) +
  scale_x_discrete(limits=c(1:9)) +
  ylab("Var. Explained (bars) and Cumulative (line) by PC") +
  geom_bar(aes(y=var_exp), stat="identity", fill="gray", alpha=.5) +
  geom_line() +
  theme_bw()

```



nMDS for Categories: Correspondence Analysis

Correspondence analysis (CA) is a special case of nMDS, serving the same purpose for categorical variables. It's very useful for visualizing relationships within contingency tables, particularly when

mosaic plots are too noisy. There are several packages that can perform CA, but the aptly named `ca` package provides a good, simple way to do it. We'll use the `smoke` dataset built into `ca`, which is a fake dataset on smoking habits (none, light, medium, heavy) among senior managers (SM), junior managers (JM), senior employees (SE), junior employees (JE), and secretaries (SC).

```
require(ca)
```

```
smoke_ca = ca(smoke)
```

```
summary(smoke_ca)
```

Principal inertias (eigenvalues):

dim	value	%	cum%	scree plot
1	0.074759	87.8	87.8	*****
2	0.010017	11.8	99.5	***
3	0.000414	0.5	100.0	
<hr/>				
Total: 0.085190 100.0				

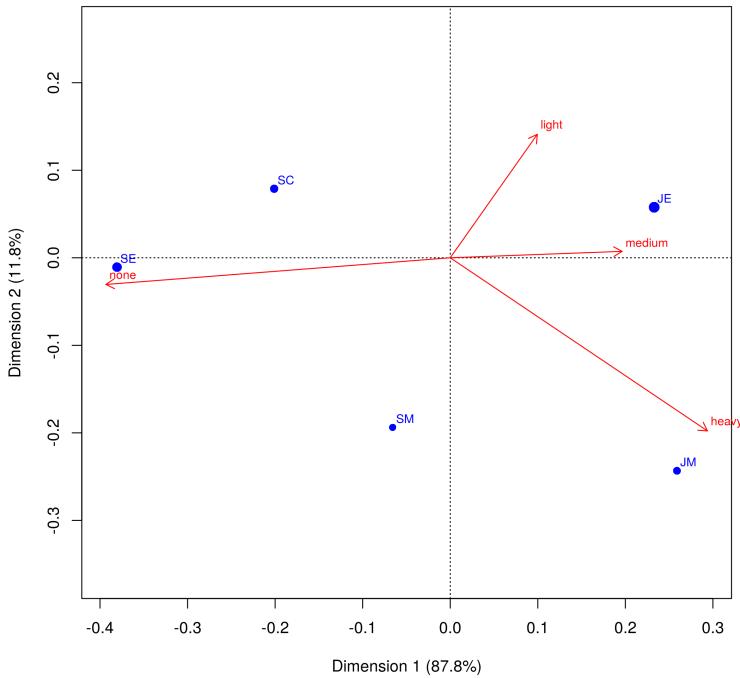
Rows:

	name	mass	qlt	inr	k=1 cor	ctr	k=2 cor	ctr			
1	SM	57	893	31	-66	92	3	-194	800	214	
2	JM	93	991	139	259	526	84	-243	465	551	
3	SE	264	1000	450	-381	999	512	-11	1	3	
4	JE	456	1000	308	233	942	331	58	58	152	
5	SC	130	999	71	-201	865	70	79	133	81	

Columns:

	name	mass	qlt	inr	k=1 cor	ctr	k=2 cor	ctr			
1	none	316	1000	577	-393	994	654	-30	6	29	
2	lght	233	984	83	99	327	31	141	657	463	
3	medm	321	983	148	196	982	166	7	1	2	
4	hevy	130	995	192	294	684	150	-198	310	506	

```
plot(smoke_ca, arrows=c("F","T"), mass = c(TRUE, TRUE))
```



Cluster analysis

Although it's a relatively old tool, cluster analysis has advanced with the development of techniques such as clustering irregular groupings by density via DBSCAN, and being able to make inferential claims on cluster membership with model-based clustering. Still, the old standbys of kMeans, PAM, and hierarchical clustering all remain excellent first-order descriptive tools for multivariate data.

Grouping observations with hierarchical clustering

Hierarchical clustering can be a useful way to summarize high-dimensional data, but results can be strongly dependent on the clustering method. Ward's, average, and centroid methods tend to be the best places to start. Hierarchical cluster analysis tools are built into the base installation, so no specific packages are required here.

Typically, variables to cluster on are scaled (usually to mean 0 and sd 1), a distance matrix is created on the scaled data, and a clustering method is applied to the distance matrix. Since the set of descriptive variables in the Spanish banks data set are all on the same scale (i.e., a set of ratios), we don't need to scale this data, so we'll start by creating the distance matrix. We'll use Ward's clustering method here, which aims to create compact clusters by minimizing variance.

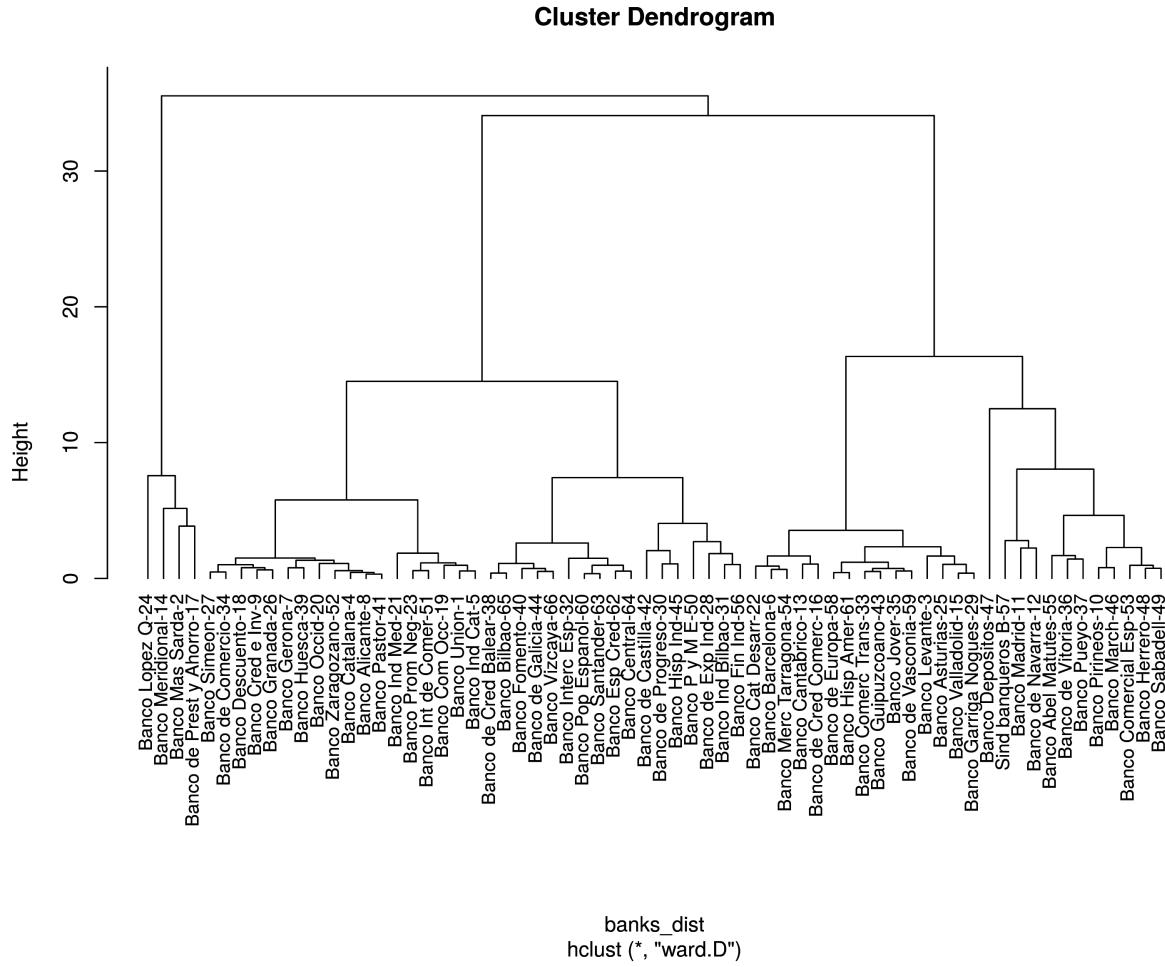
```

banks_dist = dist(banks[,3:11])

banks_hclust = hclust(banks_dist, method="ward.D")

plot(banks_hclust, hang=-1, labels=paste(banks$BANCO, banks$Numero, sep="-"))

```



See `?hclust` for a short overview of the clustering methods available. The method and approach you use are important—changes in either will change the result (see below for an example).

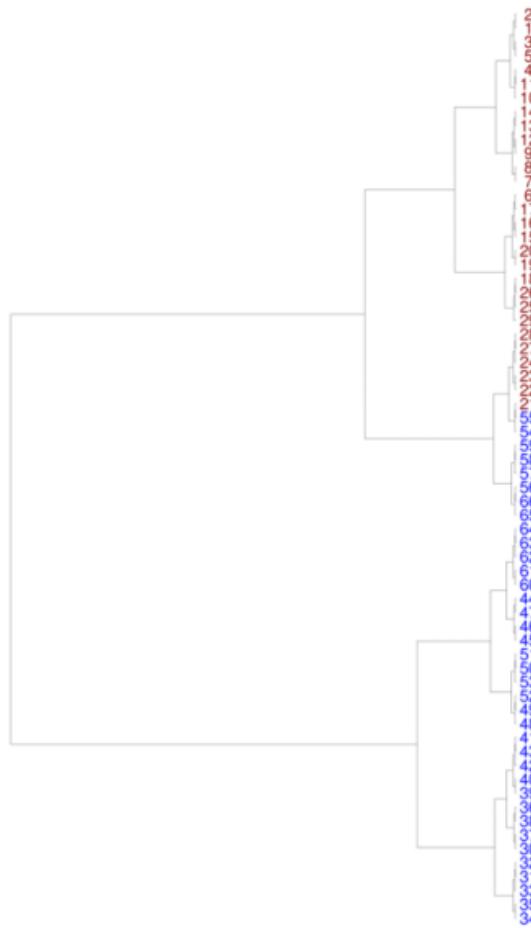
Plotting a cluster dendrogram with ggplot

The `NeatMap` package has a helper function (`draw.dendrogram`) that allows you to place an `hclust` object in `ggplot2`, where you can style or label it from there however you wish, e.g.:

```
require(NeatMap)

ggplot() +
  draw.dendrogram(banks_hclust) +
  scale_color_manual(name="Status: ", values=c("darkred", "blue"),
    labels=c("Bankrupt ", "Solvent ")) +
  geom_text(aes(x=0.75, y=banks_hclust$order, label=banks$Numero,
    color=factor(banks$Output)), size=4) +
  ggtitle("Cluster Dendrogram (Ward's) - Spanish Banking Crisis") +
  theme(panel.grid.major = element_blank(), panel.grid.minor = element_blank(),
    panel.border = element_blank(), panel.background = element_blank(),
    axis.title = element_blank(), axis.text = element_blank(),
    axis.ticks = element_blank(), legend.position="top")
```

Cluster Dendrogram (Ward's) - Spanish Banking Crisis

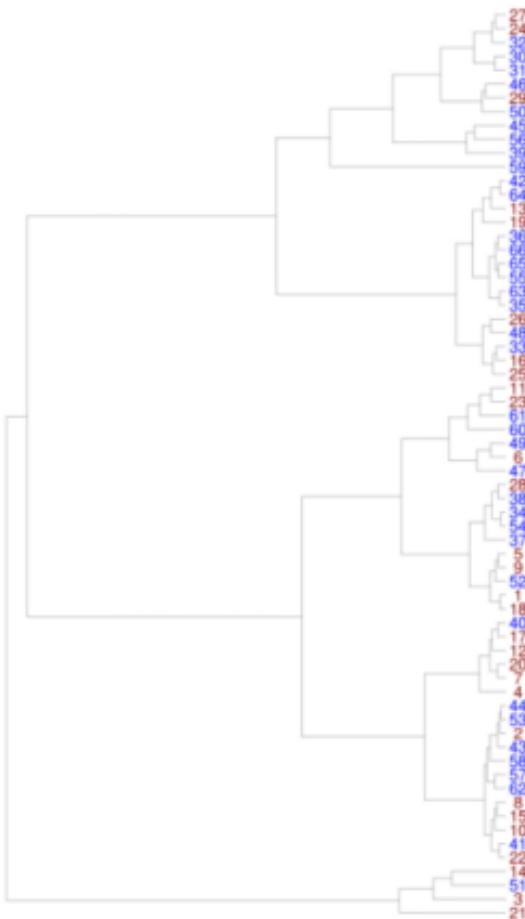
Status: ■ Bankrupt ■ Solvent

As we mentioned above, a small change in approach can completely alter the result. Say we had scaled the data first.

```
banks_scaled = scale(banks[,3:11])  
  
banks_dist = dist(banks_scaled)  
  
banks_hclust = hclust(banks_dist, method="ward.D")  
  
# Now rerun the ggplot commands as above
```

Cluster Dendrogram (Ward's) - Spanish Banking Crisis

Status: ■ Bankrupt ■ Solvent



The clear structure seen above in the unscaled data is now somewhat muddled.

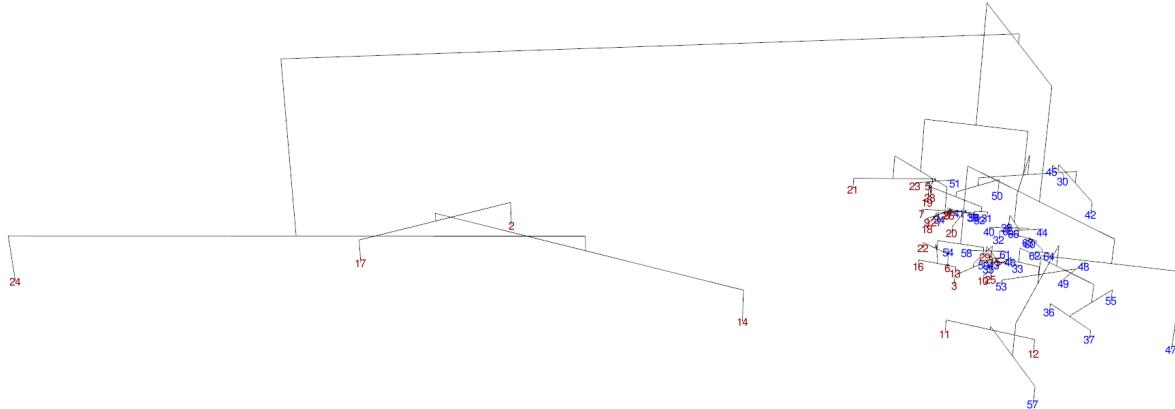
Exploring hierarchical clustering of nMDS results

NeatMap's `draw.dendrogram3d` function allows you to plot an `hclust` object over your nMDS results, and explore it interactively in 3D in a new window; a screenshot of this view is shown below the code:

```
require(NeatMap)

banks_pos = banks_mds$points # nMDS results from above

draw.dendrogram3d(banks_hclust, banks_pos, labels=banks$Numero,
  label.colors=banks$Output+3, label.size=1)
```



How to partition the results of a hierarchical cluster analysis

The `cutree` function will cut a hierarchical cluster dendrogram into a user-supplied number of groups, which can be useful when combined with the results of a `clusGap`-based PAM or kMeans analysis (see the recipes later in this chapter). Since we know there are two natural clusters—bankrupt and solvent—we'll use $k=2$ here. Note that since it is cutting the tree based on the clustering method's distance values, results from `cutree` may not match the clusters obtained with `pam` or `kmeans`.

Using our original `hclust` object:

```
# Run cutree for 2 groups
bank_clusters = cutree(banks_hclust, k=2)

# View results
table(bank_clusters)

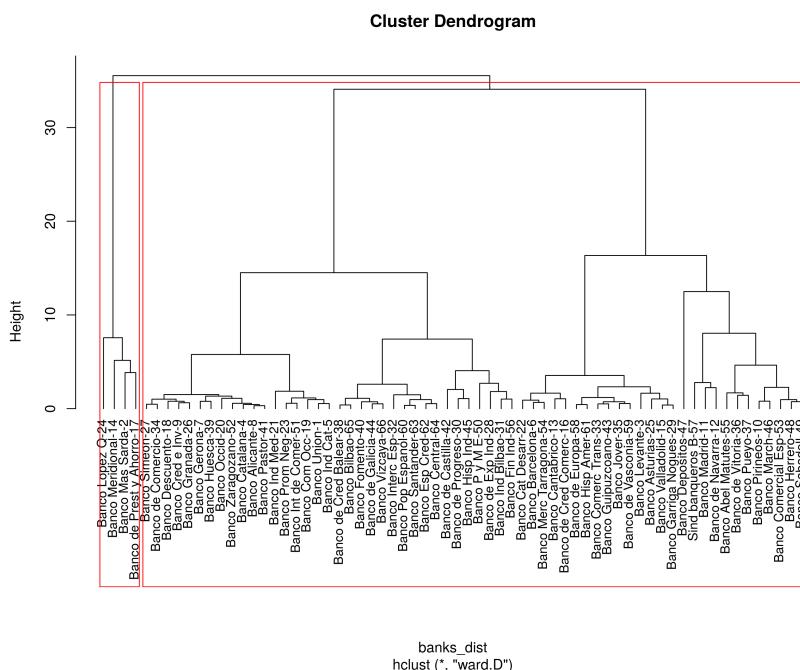
bank_clusters
 1  2
62  4

# Show variable medians for each group
aggregate(banks[,3:11], by=list(cluster=bank_clusters), median)
```

cluster	R1	R2	R3	R4	R5	R6	R7	R8	R9
1	0.39265	0.26210	0.40725	0.0203	0.0042	0.1002	0.0044	0.87650	0.00955
2	0.36585	0.23295	0.38455	0.0152	-0.0380	-0.9390	-0.0397	1.30815	-0.01630

```
# Plot dendrogram
plot(banks_hclust, hang=-1, labels=paste(banks$BANCO, banks$Numero, sep="-"),
      cex=0.9)

# Add cluster boxes to dendrogram
rect.hclust(banks_hclust, k=2)
```



Identifying and describing group membership with kMeans and PAM

kMeans and Partitioning Around Medoids (PAM) are two methods that will divide your observations into k groups. kMeans works with quantitative data only, and can be susceptible to outliers, while PAM works with both quantitative as well as categorical data. As medoids are simply the “most representative” member of a cluster, outliers don’t really affect PAM. As a result, it is a more robust version of kMeans. If you have a really large data set, look at the `clara` package, which performs PAM by iteratively sampling the data to speed up analysis time.

The `cluster` package contains the basic PAM function, so load that first. We'll continue to use the Spanish banking data.

```
require(cluster)
```

To run either analysis, you start with the number of groups (k) in which you wish to partition the data. We know there are two natural categories in this data set—banks that went bankrupt and those that remained solvent, so we can use those factors to check the results of the analysis. Often you don't have natural groups in your data, and you're not sure how many groups there should be. That topic is briefly touched on in the next recipe with `clusGap` and at more length later in this chapter in the recipes on bootstrapping and model-based clustering.

To run PAM with $k=2$:

```
banks_pam = pam(banks[,3:11], k=2)
```

banks pam

Medoids:

ID R1Liqui1 R2Liqui2 R3Liqui3 R4Autof R5RenEc R6RenFin R7Apal R8CosVen

```
[1,] 58   0.4074   0.2764   0.4397   0.0156   0.0043   0.0582   0.0046   0.8846
```

17 0.

R9Cash

[1,] 0.0095

[2,] -0.0231

Clustering vector:

[38] 1 1 1 1 1 1 1

Objective function

build swap

0.219742 0.219742

Available components:

```
[1] "medoids"      "id.med"       "clustering"   "objective"    "isolation"  
[6] "clusinfo"     "silinfo"      "diss"        "call"         "data"
```

To run kMeans with $k=2$:

```
banks_kmeans = kmeans(banks[,3:11], centers=2, nstart=50)
```

banks_kmeans

K-means clustering with 2 clusters of sizes 63, 3

Cluster means:

	R1Liqui1	R2Liqui2	R3Liqui3	R4AutoF	R5RenEc	R6RenFin
1	0.3952127	0.2674238	0.4173810	0.02471746	0.005095238	0.1117937
2	0.3409333	0.2465000	0.3561333	0.01186667	-0.052733333	-1.2071667
	R7Apal	R8CosVen	R9Cash			
1	0.005436508	0.8728571	0.009780952			
2	-0.055200000	1.2808000	-0.017966667			

Clustering **vector**:

Within cluster sum of squares by cluster:

```
[1] 3.595726 0.495665  
(between_SS / total_SS = 57.3 %)
```

Available components:

```
[1] "cluster"      "centers"       "totss"        "withinss"      "tot.withinss"  
[6] "betweenss"    "size"          "iter"         "ifault"
```

If you're satisfied with your results, you can add them to the original data frame:

```
banks$pam2 = banks_pam$clustering
```

```
banks$km2 = banks kmeans$cluster
```

PAM and kMeans work by assigning observations into a set of discrete categories, the number of which (k) is assigned by the user. Like `hc1ust`, there are several algorithms to choose from, so explore those options with `?kmeans` and `?pam`.

If you do have known outcome values, a table of actual vs. modeled clusters can give you a sense of each algorithm's performance on your data.

```
table(banks$Output, banks$pam2)
```

1	2
0	26 3
1	37 0

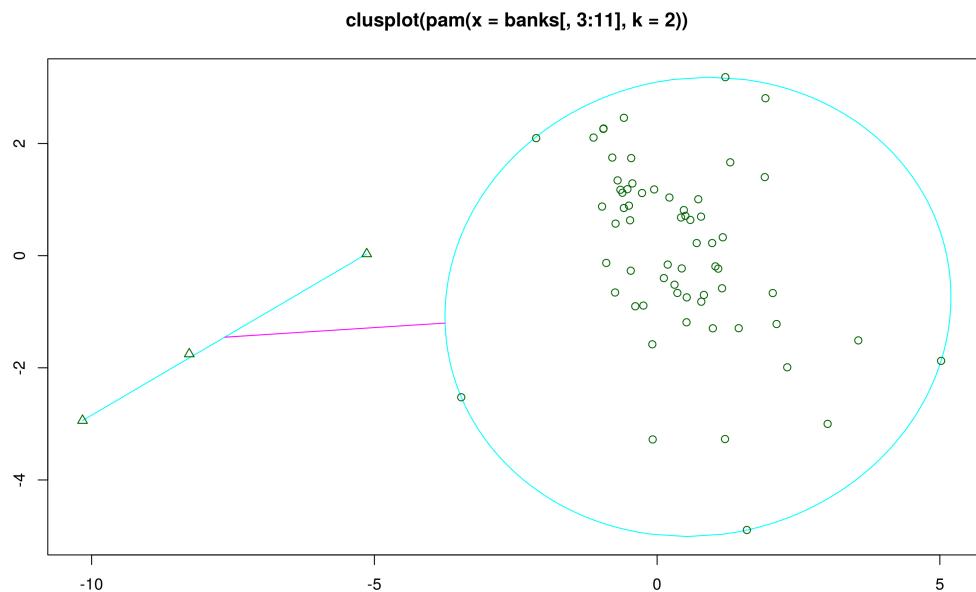
```
table(banks$Output, banks$km2)
```

1	2
0	26 3
1	37 0

Both `pam` and `kmeans` objects have ready-to-use plotting functions that allow you to compare the results of the cluster with actual values on an nMDS plot. If for some reason you're looking for a production plot, you're probably better off doing it manually in `ggplot2`, but both of these functions work fine for quick validation work.

For `pam` results, the `plot` function provides two plots, the first of which has what we're looking for. Note that the scale along the y-axis is reversed as compared to the other nMDS plots we've seen, but that doesn't change the interpretation:

```
plot(banks_pam, which.plot=1, xlab="", ylab "")
```

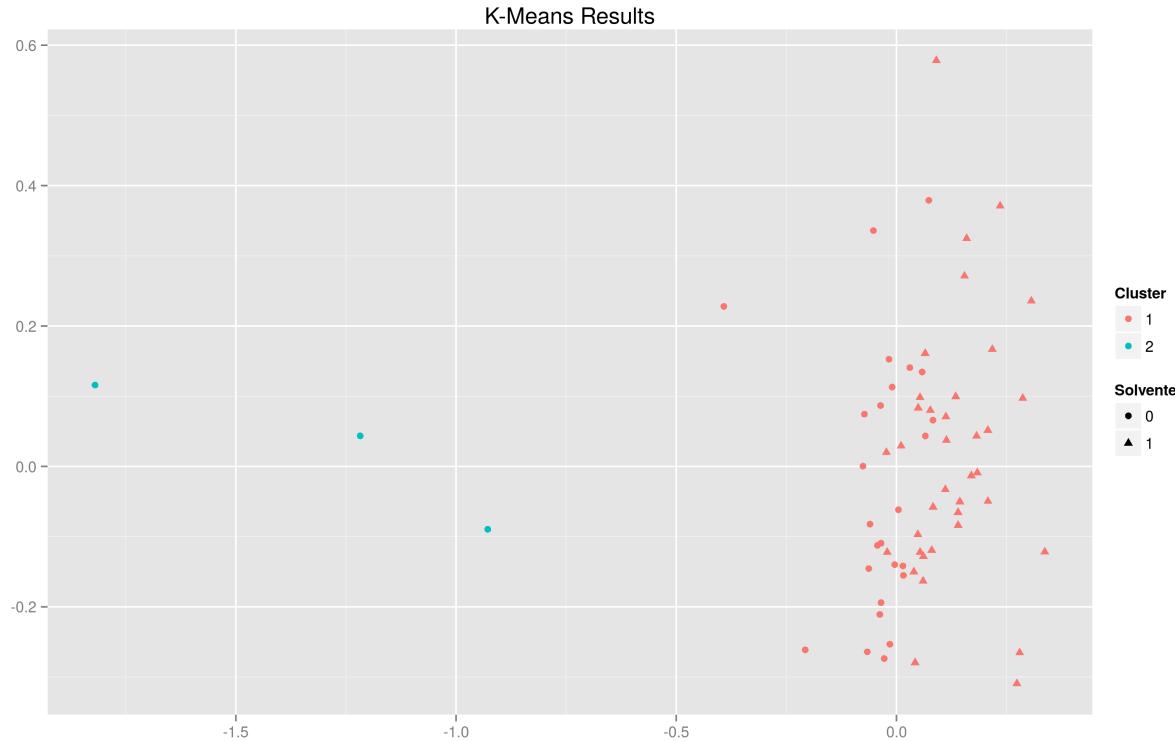


These two components explain 81.24 % of the point variability.

The `useful` package allows you to plot a `kmeans` object via a simplified `ggplot`:

```
require(useful)

plot(banks_kmeans, data=banks, class="Output", xlab="", ylab="")
```



How to choose an optimal number of clusters with bootstrapping

The `clusGap` function in the `cluster` package compares a set of observed cluster sizes with a bootstrapped sample to evaluate the difference between actual and expected for a given number of clusters. That distance is the Gap statistic; the mathematically optimal number of clusters according to this approach is the smallest number within one standard error (`SE.sim`) of the smallest gap's value (`gap`). You can specify whether to use `kmeans` or `pam` in the `FUNcluster` option.

From these results, we can see that the two cluster solution is clearly the best option.

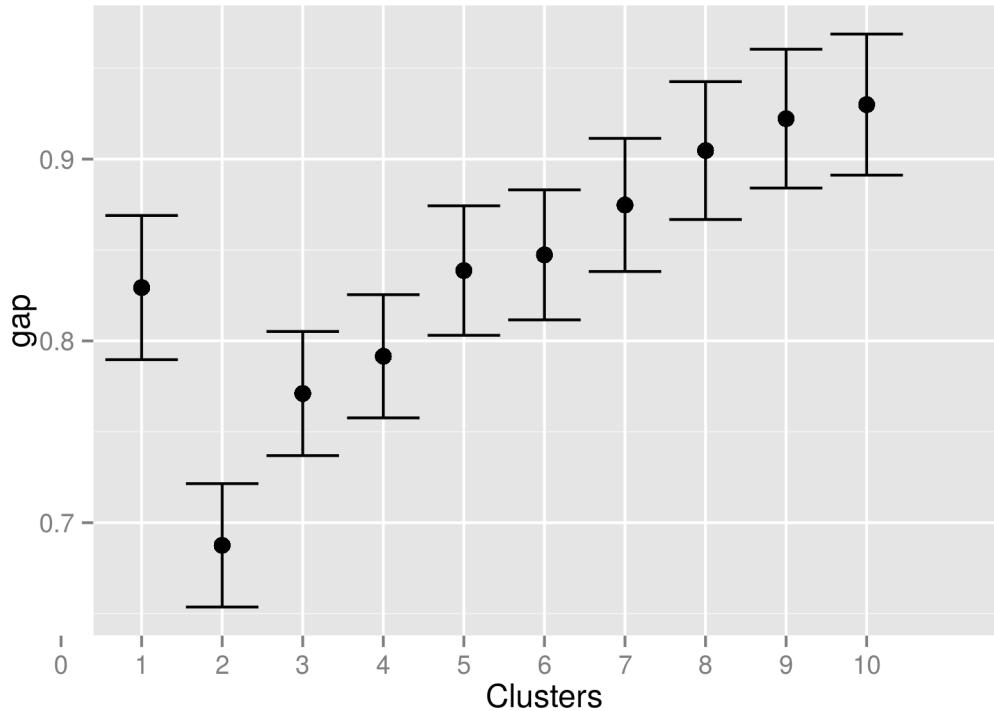
```

banks_gap = clusGap(banks[,3:11], FUNcluster=pam, K.max=10, B=500)

banks_gap_df = as.data.frame(banks_gap$Tab)

ggplot(banks_gap_df, aes(x=1:nrow(banks_gap_df))) +
  scale_x_discrete(limits=c(1:10)) +
  xlab("Clusters") +
  geom_point(aes(y=gap), size=3) +
  geom_errorbar(aes(ymin=gap-SE.sim, ymax=gap+SE.sim))

```



Determining optimal numbers of clusters with model-based clustering

Sometimes you actually need to be able to justify your cluster analysis results with more than verbal arguments and intuition. Model-based clustering lets you do formal inference on cluster grouping results using finite mixture density models. Even better, it can pick up clusters that traditional kmeans or hierarchical clustering fail at, because of the flexibility of its clustering shapes.

The `mclust` package provides a simple interface to model-based clustering. For the sake of example, we'll use the `diabetes` dataset that comes with the `mclust` package.

```

require(mclust)
data("diabetes")

diabetes_mclust = Mclust(diabetes[,2:4])

summary(diabetes_mclust)

-----
Gaussian finite mixture model fitted by EM algorithm
-----

Mclust VVV (ellipsoidal, varying volume, shape, and orientation) model with 3 co\
mponents:

log.likelihood    n df      BIC      ICL
-2307.883 145 29 -4760.091 -4776.086

```

Clustering table:

1	2	3
82	33	30

Running the main function, `Mclust`, is easy. Interpreting the results takes more work. There are currently 20 possible model types in this package, which allows for a huge amount of flexibility in cluster shapes, something essential for oddly-distributed data (which is exactly why we have used the `diabetes` dataset here). These are summarized below (for more details, run `?mclustModelNames`).

Model Name	Model Type
Univariate	
“E”	equal variance (one-dimensional)
“V”	variable variance (one-dimensional)
Multivariate	
“EII”	spherical, equal volume
“VII”	spherical, unequal volume
“EEI”	diagonal, equal volume and shape
“VEI”	diagonal, varying volume, equal shape
“EVI”	diagonal, equal volume, varying shape
“VVI”	diagonal, varying volume and shape
“EEE”	ellipsoidal, equal volume, shape, and orientation
“EVE”	ellipsoidal, equal volume and orientation
“VEE”	ellipsoidal, equal shape and orientation
“VVE”	ellipsoidal, equal orientation

Model Name	Model Type
“EEV”	ellipsoidal, equal volume and equal shape
“VEV”	ellipsoidal, equal shape
“EVV”	ellipsoidal, equal volume
“VVV”	ellipsoidal, varying volume, shape, and orientation
Single Component	
“X”	univariate normal
“XII”	spherical multivariate normal
“XXI”	diagonal multivariate normal
“XXX”	ellipsoidal multivariate normal

Note: the `mclust` package reverses the typical BIC value scale/sign relationship where lowest is better; here, the higher value suggests the better model(s).

By looking at the BIC table as well as plotting the modeling results, we can see why the algorithm chose the VVV model type—the BIC value of -4760 under VVV for 3 clusters is the best value in the table.

```
diabetes_mclust$BIC
```

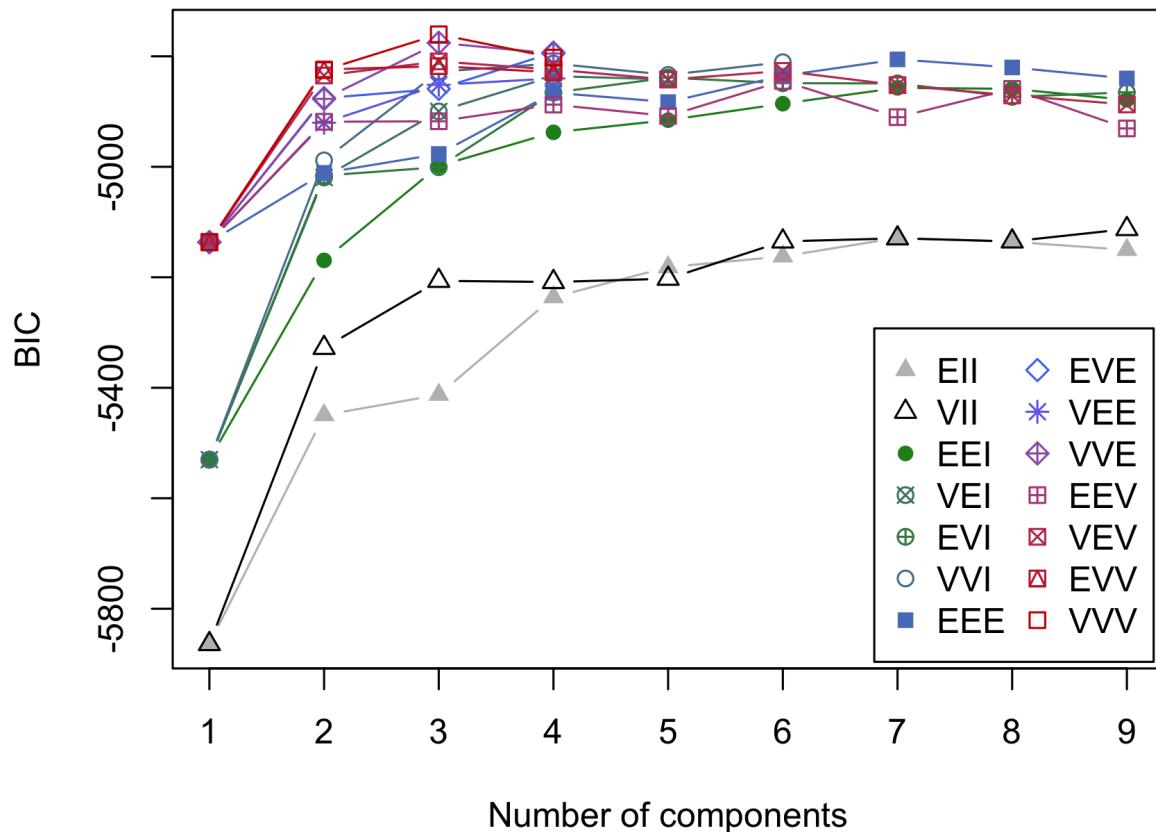
Bayesian Information Criterion (BIC):

	EII	VII	EEI	VEI	EVI	VVI	EEE
1	-5863.923	-5863.923	-5530.129	-5530.129	-5530.129	-5530.129	-5136.446
2	-5449.518	-5327.719	-5169.399	-5019.350	-5015.884	-4988.322	-5010.994
3	-5412.588	-5206.399	-4998.446	-4899.759	-5000.661	-4827.818	-4976.853
4	-5236.008	-5208.512	-4937.627	-4835.856	-4865.767	-4813.002	-4865.864
5	-5181.608	-5202.555	-4915.486	-4841.773	-4838.587	-4833.589	-4882.812
6	-5162.164	-5135.069	-4885.752	NA	-4848.623	-4810.558	-4835.226
7	-5128.736	-5129.460	-4857.097	NA	-4849.023	NA	-4805.518
8	-5135.787	-5135.053	-4858.904	NA	-4873.450	NA	-4820.155
9	-5150.374	-5112.616	-4878.786	NA	-4865.166	NA	-4840.039
	EVE	VEE	VVE	EEV	VEV	EVV	VVV
1	-5136.446	-5136.446	-5136.446	-5136.446	-5136.446	-5136.446	-5136.446
2	-4875.633	-4920.301	-4877.086	-4918.500	-4834.727	-4823.779	-4825.027
3	-4858.851	-4851.667	-4775.537	-4917.567	-4809.225	-4817.884	-4760.091
4	-4793.261	-4840.034	-4794.892	-4887.406	-4823.882	-4828.796	-4802.420
5	NA	NA	NA	-4908.030	-4842.077	NA	NA
6	NA	NA	NA	-4844.584	-4826.457	NA	NA
7	NA	NA	NA	-4910.155	-4852.182	NA	NA
8	NA	NA	NA	-4858.974	-4870.633	NA	NA
9	NA	NA	NA	-4930.535	-4887.206	NA	NA

Top 3 models based on the BIC criterion:

VVV,3 VVE,3 EVE,4
-4760.091 -4775.537 -4793.261

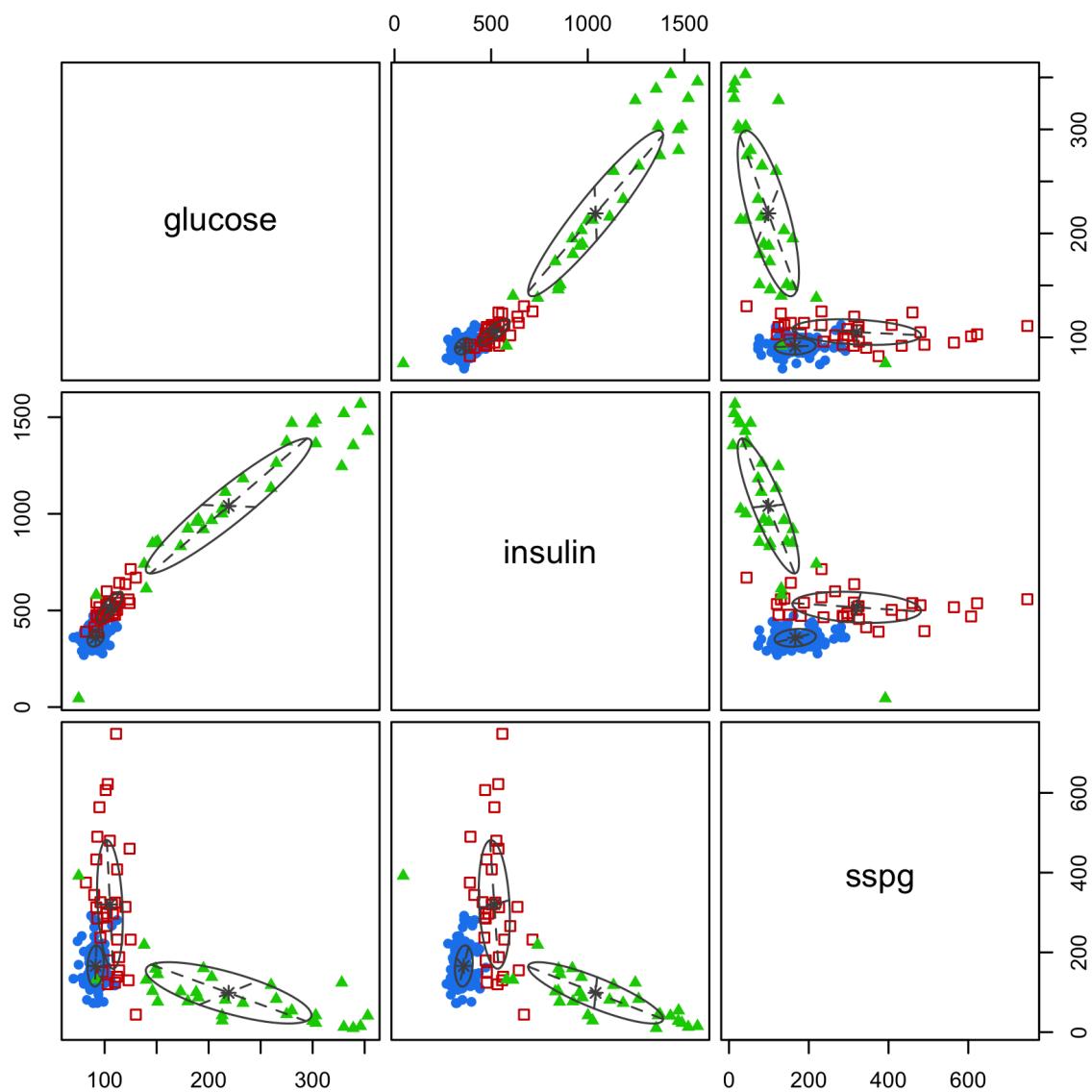
```
# Model comparison plot
plot(diabetes_mclust, what="BIC")
```



We can see that the VVV model is definitely the best among the top three.

Now we can plot the clusters over a pairs plot of the variables:

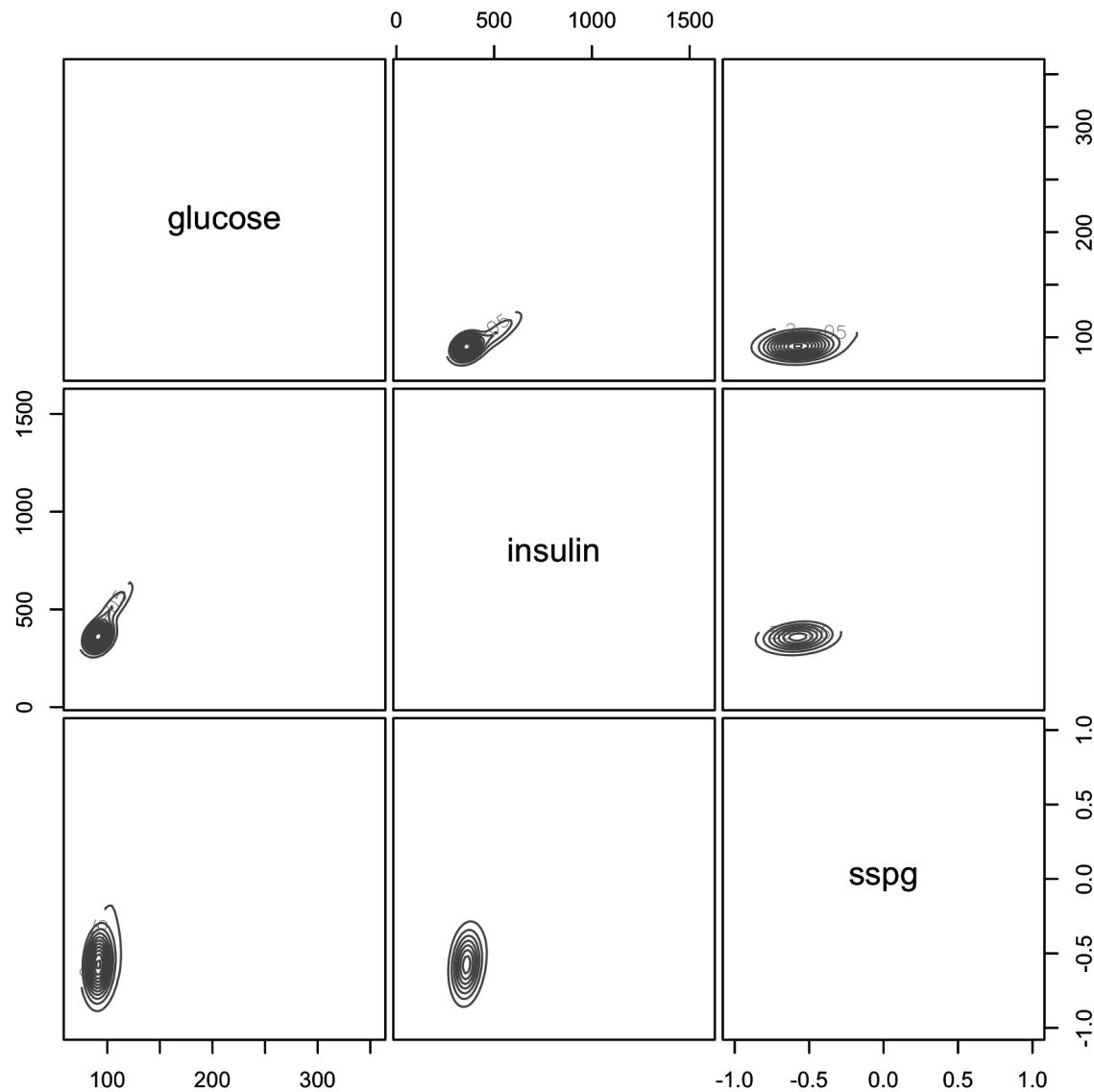
```
# Cluster results
plot(diabetes_mclust, what="classification")
```

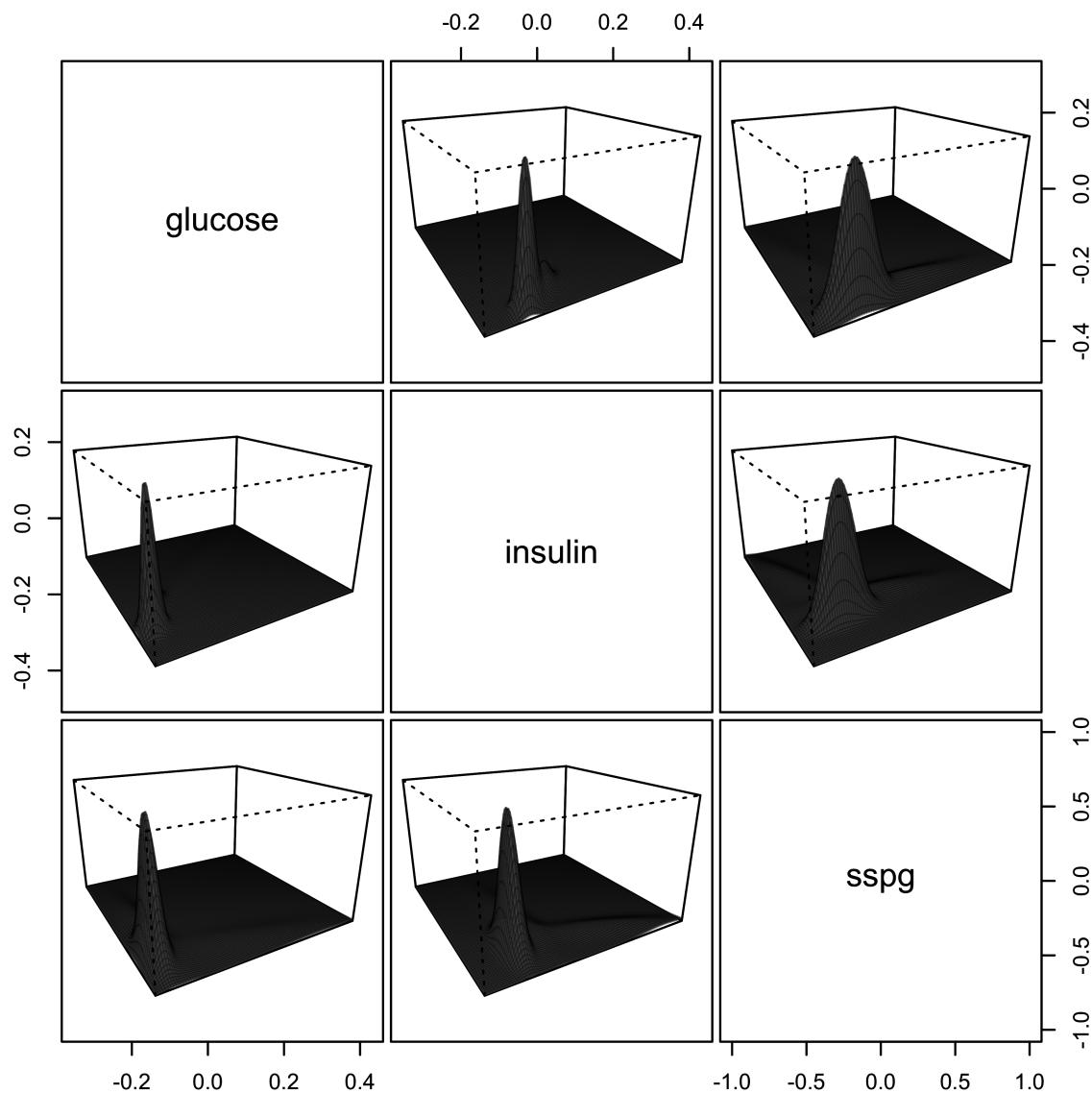


We can also explore the bivariate density of the original data through this package in both 2- and 3-dimensions:

```
# Bivariate density (basic contour plot)
plot(diabetes_mclust, what="density")

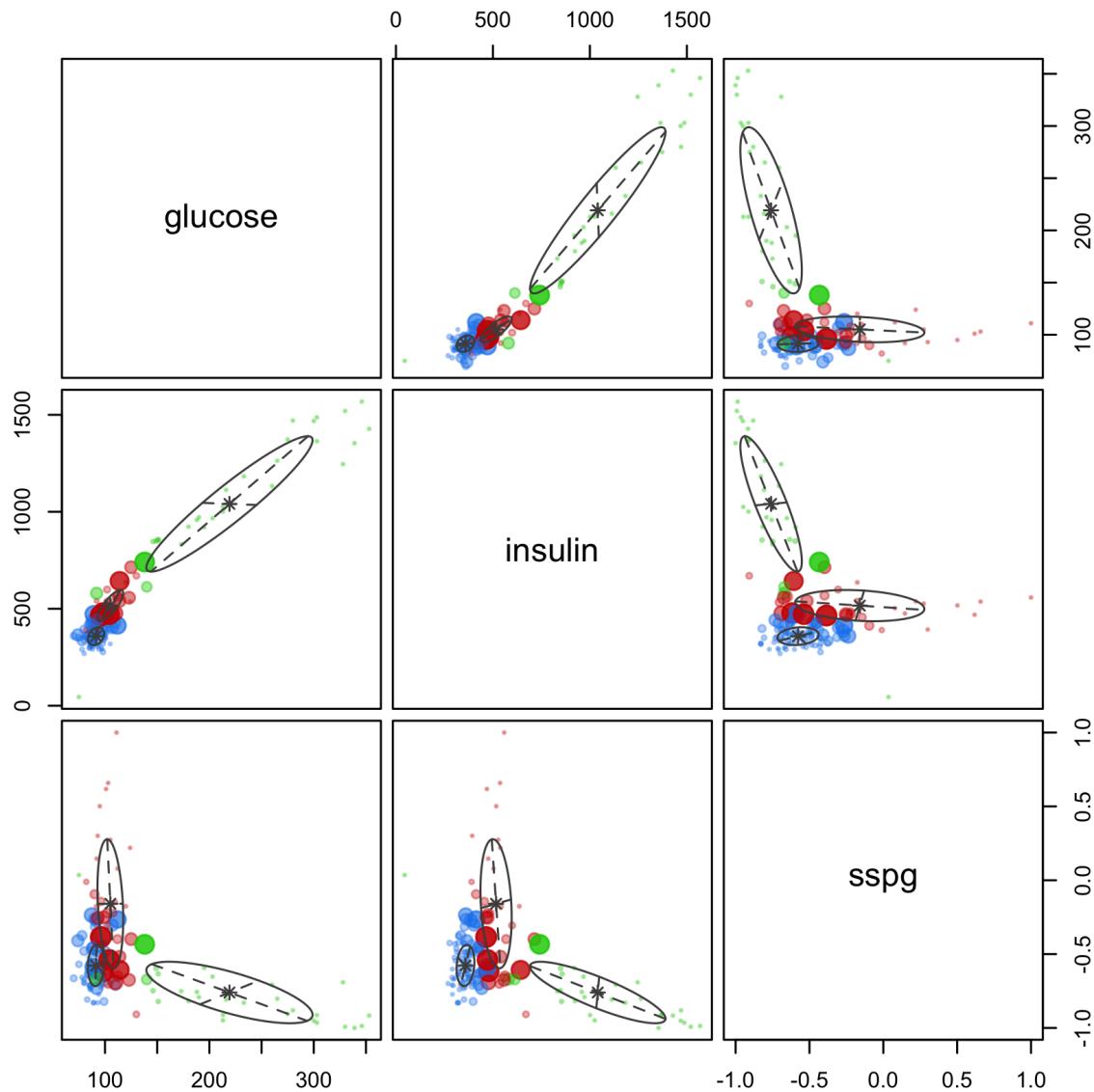
# Bivariate density (perspective plot)
plot(diabetes_mclust, what = "density", type = "persp",
border = adjustcolor(grey(0.01), alpha.f = 0.15))
```





Understanding the uncertainty associated with specific observations and their cluster assignment is a useful diagnostic tool as well.

```
# Cluster uncertainty plot  
plot(diabetes_mclust, what="uncertainty")
```



This graph shows us *where* the uncertainty occurs, but it doesn't tell us which observations are the most uncertain. We can plot the uncertainty values in an interactive setting for more directed exploration. We'll use the `googleVis` package here:

```
# Add classification and uncertainty to data set
diabetes$mclust_cluster = diabetes_mclust$classification
diabetes$mclust_uncertainty = diabetes_mclust$uncertainty

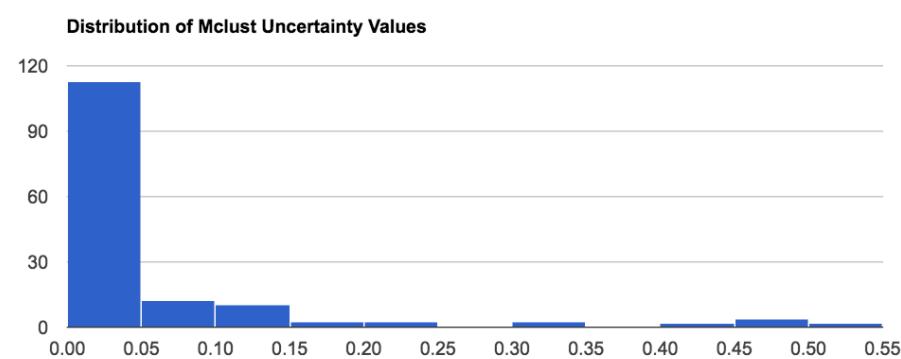
# Plot the uncertainty as a histogram and table
require(googleVis)

diabetes_histogram = gvisHistogram(data.frame(diabetes$mclust_uncertainty),
  options=list(width=800, height=300, legend="none",
  title="Distribution of Mclust Uncertainty Values"))

diabetes_table = gvisTable(diabetes, options=list(width=800, height=300))

HT = gvisMerge(diabetes_histogram, diabetes_table)

plot(HT)
```



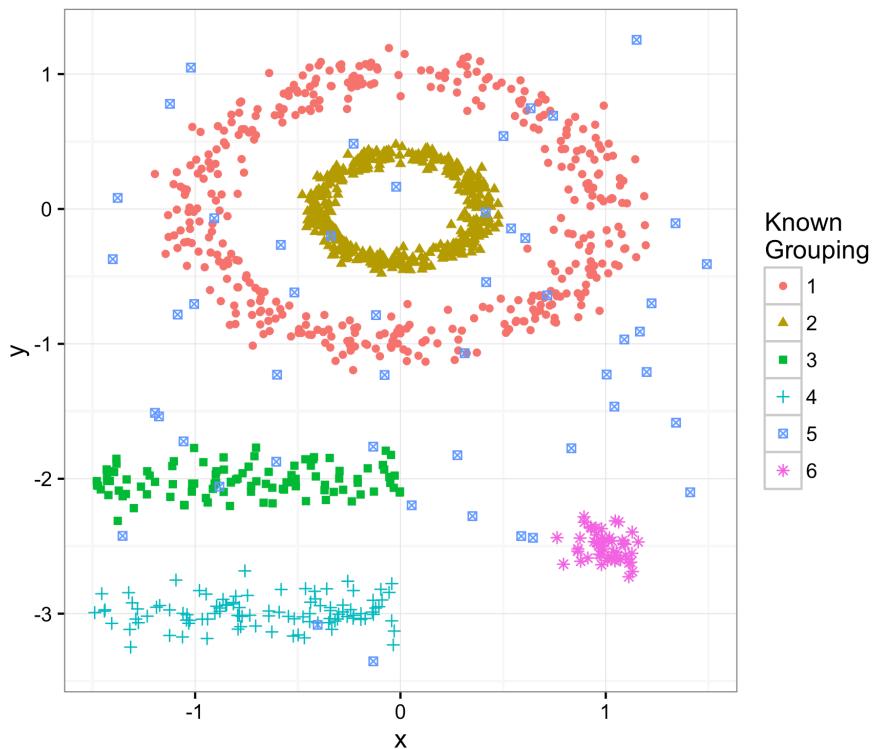
class	glucose	insulin	sspg	mclust_cluster	mclust_uncertainty
Chemical	96	465	237	2	0.512
Chemical	104	472	180	2	0.496
Chemical	98	478	151	2	0.494
Overt	138	741	219	3	0.452
Chemical	114	643	155	2	0.412
Normal	112	414	281	1	0.327
Chemical	89	472	162	1	0.322
Chemical	110	477	124	2	0.217
Normal	87	360	292	1	0.209
Normal	92	386	279	1	0.16
Chemical	93	472	285	2	0.153
Overt	125	714	232	2	0.147
Overt	123	557	130	2	0.146
Chemical	88	439	208	1	0.143
Chemical	92	442	109	1	0.143

Data: various • Chart ID: MergedID34cbdf23365 • googleVis-0.5.10
R version 3.3.0 (2016-05-03) • Google Terms of Use • Data Policy: See individual charts

Identifying group membership with irregular clusters

Sometimes the clusters we want to identify have clear patterns but consist of non-linear or irregular groupings—things we can identify readily by eye but cannot cluster with the usual methods. The DBSCAN algorithm was built for this purpose. We’ll use a synthetic dataset from the `multishapes` package to demonstrate exactly what this means.

```
data("multishapes", package="factoextra")  
  
ggplot(multishapes, aes(x, y)) +  
  geom_point(aes(shape=as.factor(shape),  
                 color=as.factor(shape))) +  
  scale_shape_discrete(name="Known\nGrouping", labels =  
    levels(as.factor(multishapes$shape))) +  
  scale_color_discrete(name="Known\nGrouping", labels =  
    levels(as.factor(multishapes$shape))) +  
  theme_bw()
```



The patterns are clear to your eye, but the methods we’ve explored above completely fail in identifying them. The `dbSCAN` function in the package of the same name *can* identify these groupings.

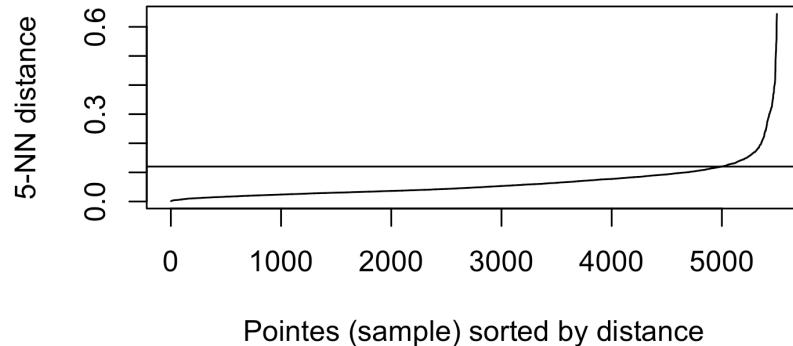
The algorithm requires you choose a value for the ϵ (*epsilon*) parameter. The `kNNdistplot` function can help you choose this value; you should try a variety of values close to the bend in the resulting

line to explore how changes in this value affect the outcome. k is usually taken to be the number of variables you have plus one, though you can also play with this option as a way to explore the sensitivity of any solution.

```
require(dbSCAN)

# Estimate eps with kNN distance plot
kNNdistplot(multishapes[,1:2], k=3)

# Try an eps value of 0.15
abline(h=0.15, lty=3)
```



Once you have chosen `eps`, run it through the main `dbSCAN` function:

```
multishapes_dbSCAN = dbSCAN(multishapes[,1:2], eps = 0.15, minPts = 3)
```

```
multishapes_dbSCAN
```

```
DBSCAN clustering for 1100 objects.
Parameters: eps = 0.15, minPts = 3
The clustering contains 5 cluster(s) and 25 noise points.
```

```
0   1   2   3   4   5
25  411  406  106  100  52
```

Available fields: `cluster`, `eps`, `minPts`

We can add the modeled cluster assignment to the original data, and compare the DBSCAN assignment (columns) against the known groupings in the data (`multishapes$shape`, here designated in the rows):

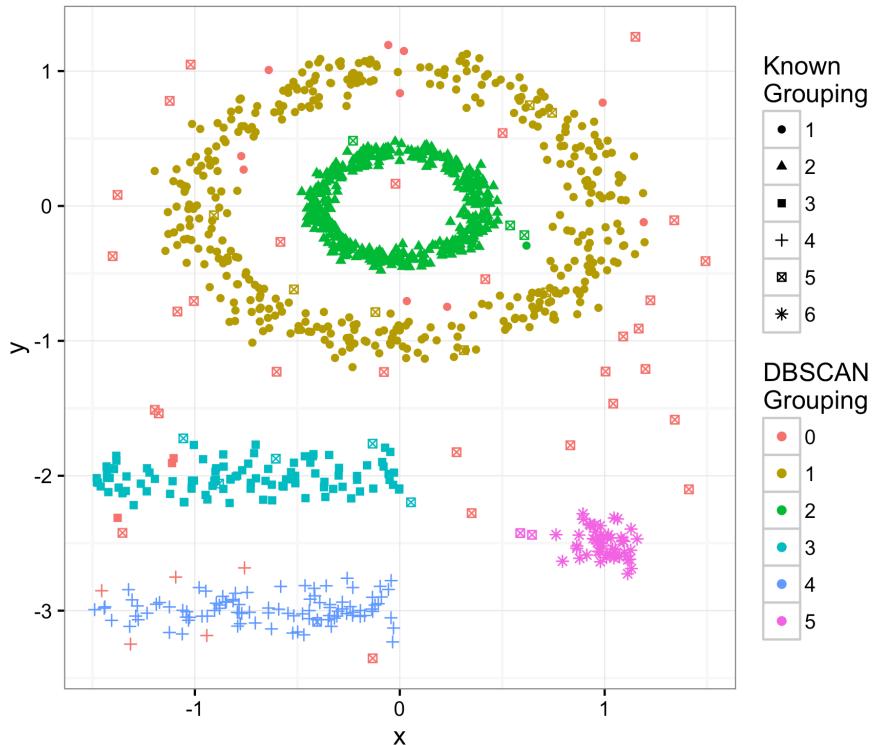
```
multishapes$dbscan = multishapes_dbSCAN$cluster
```

```
table(multishapes$shape, multishapes$dbscan)
```

	0	1	2	3	4	5
1	1	398	1	0	0	0
2	0	0	400	0	0	0
3	0	0	0	100	0	0
4	1	0	0	0	99	0
5	23	13	5	6	1	2
6	0	0	0	0	0	50

We can see the outcome and the known values together graphically:

```
ggplot(multishapes, aes(x, y)) +
  geom_point(aes(color=as.factor(dbSCAN), shape=as.factor(shape))) +
  labs(color="DBSCAN\nnGrouping", shape="Known\nnGrouping") +
  theme_bw()
```



Variable selection in cluster analysis

We often find that we have many, many variables, and are not sure of the extent to which they add value or noise to the clustering process. The `clustvsel` package—created by the same folks who created `mclust`—provides a way to assess this.

We'll use the famous Wisconsin Breast Cancer data set, which consists of nine diagnostic cell features measured on a 1-10 ordinal scale that was automatically computed from biopsy tissue images (see [this paper⁶⁶](#) for more details). Higher values are linked with higher chances of a tumor being malignant, and lower values are more likely from benign tissue.

```
require(clustvsel)

# Download the Wisconsin Breast Cancer data
breast_cancer = read.table("http://archive.ics.uci.edu/ml/
  machine-learning-databases/breast-cancer-wisconsin/
  breast-cancer-wisconsin.data", sep=",", na.strings =
  c("NA", "?"), header=F, col.names=c("Sample",
  "Clump_Thickness", "Uniformity_Cell_Size",
  "Uniformity_Cell_Shape", "Marginal_Adhesion",
  "Single_Epithelial_Cell_Size", "Bare_Nuclei",
  "Bland_Chromatin", "Normal_Nucleoli", "Mitoses", "Class"))

# Label the outcomes
breast_cancer$Diagnosis = ifelse(breast_cancer$Class==2, "benign",
  ifelse(breast_cancer$Class==4, "malignant", "unk"))

# Remove incomplete rows, then create data-only object
bc_wisc = na.omit(breast_cancer[,c(2:10, 12)])

bc_wisc_data = bc_wisc[,1:8]

# Perform variable selection (with parallel processing)
varsel_fwd = clustvsel(bc_wisc_data, parallel = TRUE)

varsel_fwd
```

⁶⁶http://dollar.biz.uiowa.edu/~street/research/hu_path95/hp95.pdf

```
Stepwise (forward/backward) greedy search
```

	Variable proposed	Type of step	BIC difference	Decision
1	Bare_Nuclei	Add	3577.27017	Accepted
2	Normal_Nucleoli	Add	-1189.74368	Accepted
3	Bland_Chromatin	Add	619.81296	Accepted
4	Bland_Chromatin	Remove	247.84644	Rejected
5	Uniformity_Cell_Size	Add	67.87812	Accepted
6	Bland_Chromatin	Remove	493.01606	Rejected
7	Single_Epithelial_Cell_Size	Add	1122.53709	Accepted
8	Uniformity_Cell_Size	Remove	-1393.18387	Accepted
9	Clump_Thickness	Add	-474.17028	Rejected
10	Bland_Chromatin	Remove	1978.99023	Rejected

```
Selected subset: Bare_Nuclei, Normal_Nucleoli, Bland_Chromatin,
Single_Epithelial_Cell_Size
```

Based on BIC values, the algorithm selected only four of the nine variables that should be used for clustering. We'll now take those variables and run model-based clustering using `Mclust` on that reduced data set.

```
# Create data frame of the data with only the chosen variables
bc_fwd = bc_wisc_data[, varsel_fwd$subset]

# Run Mclust on subsetted data frame
bc_mclust_fwd = Mclust(bc_fwd)

# Add clustering result to data
bc_wisc$mclust_fwd_cluster = bc_mclust_fwd$classification

# Review Mclust results
summary(bc_mclust_fwd)
```

```
-----  
Gaussian finite mixture model fitted by EM algorithm  
-----
```

Mclust VII (spherical, varying volume) model with 2 components:

```
log.likelihood    n df      BIC      ICL
-4492.643 683 11 -9057.077 -9068.989
```

Clustering table:

1	2
399	284

We will compare the results from this reduced clustering to a clustering using the full variable set in the next recipe.



Cluster analysis results are strongly dependent on the method used.

If you run all of the clustering algorithms in this chapter on the same dataset, you will find sometimes similar and sometimes very different outcomes. There are lots of ways you can cluster data, but no method can substitute for careful comparison of a variety of results against subject matter knowledge.

For example, if you run the variable selection algorithm from the full set of variables, removing variables one-by-one *backwards*, you'll get different (and poorer) results than seen above with the default "forward" algorithm.

Some sort of sensitivity analysis of cluster analysis results is *always* warranted.

Error-checking cluster results with known outcomes

When you have known outcomes in your data (e.g., benign, malignant), the `mclust` package has two functions that help you evaluate your results.

`classError` provides an overall error rate:

```
classError(bc_wisc$Diagnosis, bc_wisc$mclust_fwd_cluster)

$misclassified
[1]  2  4  7 37 49 71 74 79 82 109 114 117 128 134 141 145 191 227
[19] 229 237 242 245 252 286 287 298 306 307 333 339 349 390 401 405 417 420
[37] 427 447 480 493 539 541 606 610 612 635 642
```

```
$errorRate
[1] 0.06881406
```

And `adjustedRandIndex` provides the adjusted Rand Index value, where 1 = perfect classification and 0 = random classification:

```
adjustedRandIndex(bc_wisc$Diagnosis, bc_wisc$mclust_fwd_cluster)
0.7426946
```

You can compare different clustering algorithms to see which performs better.

```
# Run Mclust on the full variable set of the Wisconsin Breast Cancer data
bc_mclust = Mclust(bc_wisc_data)
bc_wisc$mclust_all = bc_mclust$classification

classError(bc_wisc$Diagnosis, bc_wisc$mclust_all)

$misclassified
 [1]  2   3   4   7  14  24  37  44  46  60  63  71  74  76  77  79  82  89
[19]  92  94 107 109 113 114 115 117 118 120 124 128 134 141 144 145 150 159
[37] 160 166 167 188 191 192 197 199 207 211 212 220 227 229 234 237 238 239
[55] 242 245 252 258 262 269 270 283 286 287 290 292 295 296 298 306 307 311
[73] 312 314 319 324 328 333 334 339 341 349 350 357 363 365 366 369 372 374
[91] 376 389 390 391 392 394 395 397 401 403 404 405 406 408 412 414 416 417
[109] 420 427 428 429 430 440 447 467 471 480 491 493 526 528 537 538 539 541
[127] 548 553 559 563 564 565 570 585 599 606 610 612 622 635 636 642 645 658
[145] 659 672 679

$errorRate
[1] 0.2152269

adjustedRandIndex(bc_wisc$Diagnosis, bc_wisc$mclust_all)

[1] 0.3827008
```

It seems that the use of all of the variables contributes some noise to the classification as compared with the subset chosen by forward variable selection.

Exploring outliers

On the flip side of the coin of similarity is the exploration of outliers. If anyone has ever asked you for an “automatic outlier detection” data product, you immediately realized that they probably don’t understand analytics. What counts as an outlier is completely dependent on context. That said, sometimes you need to explore what might count as an outlier across enough variables that visualization becomes difficult, if not impossible. Other times, you might want to get a head-start on investigating a potential outlier or trend before it wrecks things. Or perhaps you need to understand the probabilities associated with unusual but possible events. Multivariate outlier exploration, extreme value analysis, and anomaly detection are useful exploratory tools for these purposes.

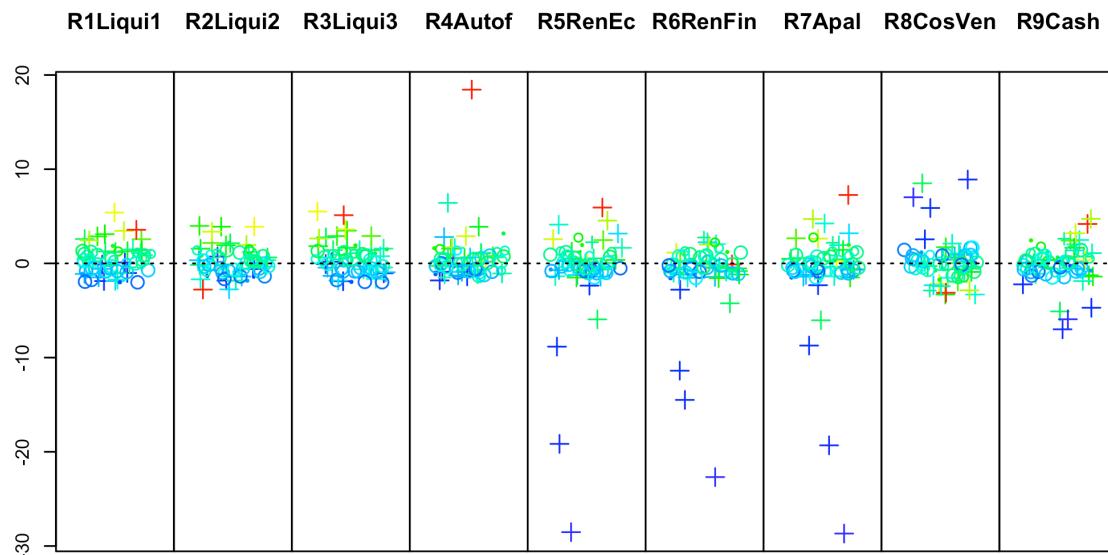
Outliers are easy enough to explore with standard EDA methods when you have three or fewer dimensions, but after that you need ways to detect and explore multivariate outliers. `mvoutlier` and `lof` are two packages that make it fairly simple. But don’t ever forget that what constitutes a *real* outlier is completely contextual—think of these tools as a way to explore, not as a way to define.

Identifying outliers with distance functions

`mvoutlier` is one package that makes multivariate outlier identification straightforward. We’ll first use the Spanish banks data to illustrate some of it, and later will use the bike sharing data to illustrate other components.

```
require(mvoutlier)

quiebra_outliers = uni.plot(banks[,3:11], symb = T)
```



The `uni.plot` function provides the numeric and visual results automatically, so if you assign it to an object, you can also attach the outlier designation and distance measures to your data frame:

```
# Assign outlier grouping to main data
banks$mv_outlier = quiebra_outliers$outliers

# Add Mahalanobis distance results to main data
banks$mv_md = quiebra_outliers$md

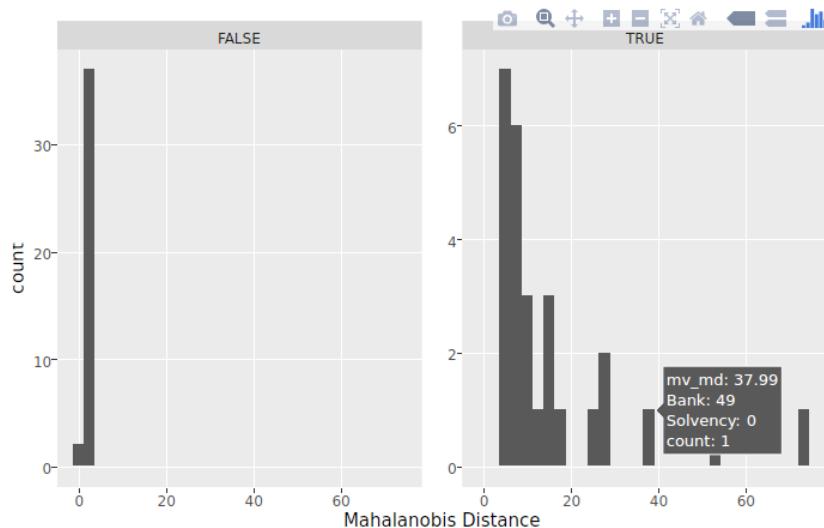
# Add Euclidean distance results to main data
banks$mv_euclidean = quiebra_outliers$euclidean
```

Interactive plots are useful for multivariate outlier exploration—static plots like the one above show us that there are multivariate outliers, but they don't tell us which observations they are. Tables can, of course, but at the expense of time. We can use the (newly open-sourced) `plotly` library's ability to make a ggplot of the Mahalanobis distance measure interactive, via the `ggplotly` function:

```
require(ggplot2)
require(plotly)

p1 = ggplot(banks, aes(mv_md, text = paste("Bank: ", 
  banks$Numero, "<br>Solvency: ", Output))) +
  xlab("Mahalanobis Distance") +
  geom_histogram() +
  facet_wrap(~mv_outlier, ncol=1, scales="free_y")

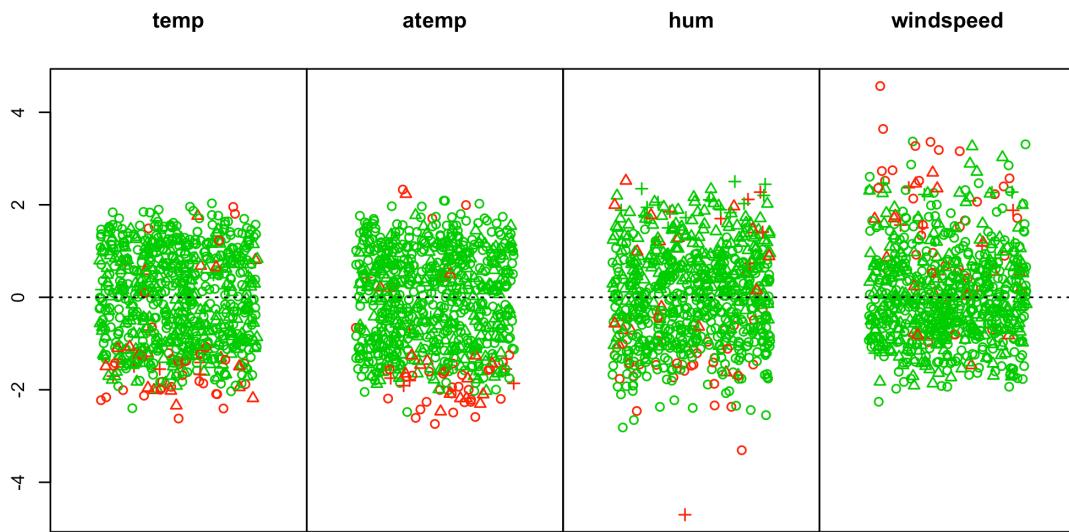
ggplotly(p1)
```



The `uni.plot` function allows you to pass through plotting options, but not always easily—for example, you’ll have to modify the function itself to change point colors. You can use point assignment via `pch`, however, to show a factor variable.

For our second `mvoutlier` example, we’ll use the bike sharing data from previous chapters. We can plot the weather-based variables and use the point type to show the weather situation relative to potential outlier observation. The values of the weather situation are mapped to the matching `pch` values, so `pch=1`, a circle, represents observations under clear weather, and so on, while the outlier status is mapped to red (outlier) or green (not an outlier).

```
uni.plot(bike_share_daily[,10:13], pch=as.numeric(bike_share_daily$weathersit))
```



Identifying outliers with the local outlier factor

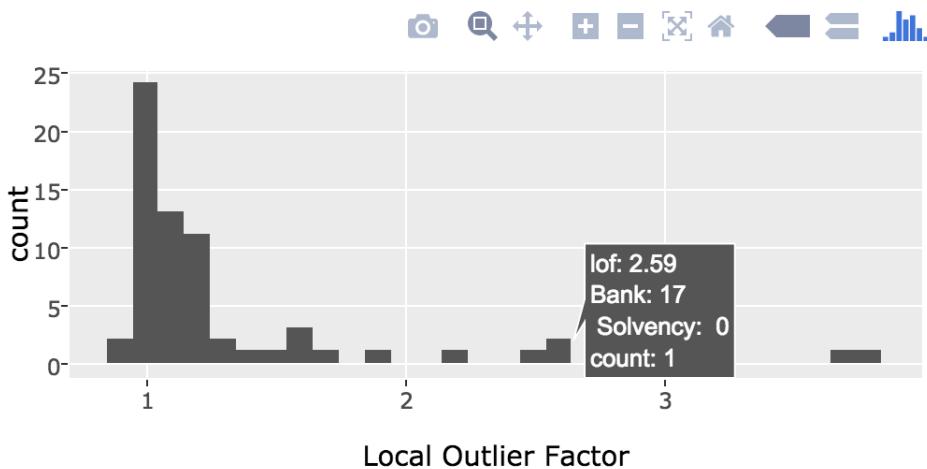
A more recently-developed outlier algorithm is the local outlier factor (lof), with an implementation in the `DMwR` package. You need to specify `k`, which here refers to the number of neighbors to use when calculating the lof value.

```
require(DMwR)

# Calculate the local outlier factor values
banks$lof = lofactor(banks[,3:11], k=4)
```

```
# Plot the lof results interactively
p_lof = ggplot(banks, aes(lof, text = paste("Bank: ", 
    banks$Numero, "<br>Solvency: ", Output))) +
  xlab("Local Outlier Factor") +
  geom_histogram()

ggplotly(p_lof)
```



Anomaly detection

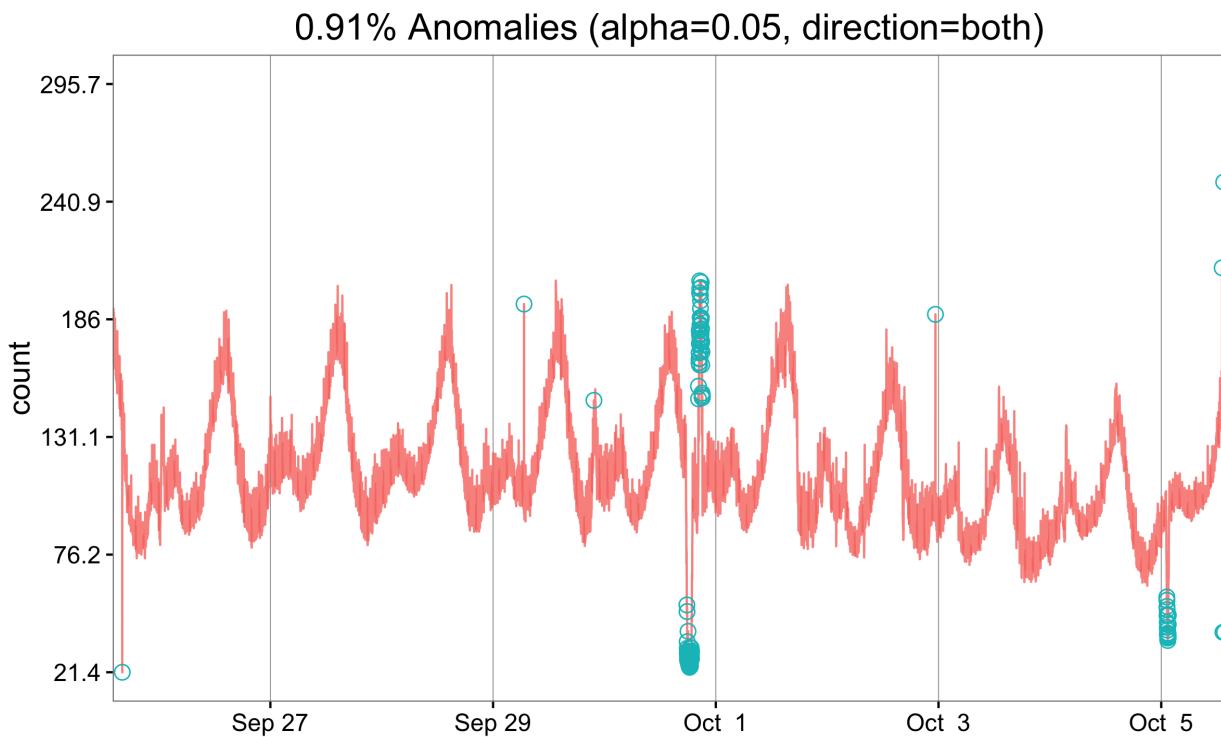
Twitter has come up with an anomaly detection algorithm for vectors, or for even things like time series that have cyclical patterns.

The vignette has a great built-in demo dataset—presumably tweets per minute—so we'll use that here.

```
#devtools::install_github("twitter/AnomalyDetection"
require(AnomalyDetection)
data(raw_data)

# Run the algorithm, include expected values and plotting features
raw_data_anomalies = AnomalyDetectionTs(raw_data, direction='both',
  e_value=TRUE, plot=TRUE)

# Plot the time series and the anomalies
raw_data_anomalies$plot
```



The algorithm defaults to determining no more than 10% of the data will be classified as anomalous. Use `max_anoms=0.01`, for example, if you wish to change that to 1% (or any other percent you deem important).

You can use the `threshold` option to identify values *above* the median, 95th, or 99th percentiles of daily maximums. There's no equivalent option for values *below* a threshold, unfortunately—using it on this data misses the low end anomalies.

Finally, you can extract the data with the expected values by pulling it from the list, e.g.,

```
raw_data_anomalies_df = raw_data_anomalies$anoms
```

Extreme value analysis

Extreme value analysis (EVA) is a different way to explore outliers; by acknowledging that outliers can be real data and can be incredibly useful for analysis, we can better assess the likelihood of “improbable” events.

There are a lot of packages in R that can do EVA, though some remain pretty technical in their documentation, when they have documentation at all. A great entry point for EVA in R is through the `extremeStat` package, which wraps around other R packages to allow for semi-automated distribution fitting and return-interval analysis.

We'll use data on economic damages in the United States caused by hurricanes between 1926 and

1995. The data are for each major hurricane, so we'll use `dplyr` to aggregate those to yearly sums and to insert 0 for years in which there was no major damage.

```
require(dplyr)

data("damage", package="extRemes")

hurricane_cost = damage %>% group_by(Year) %>% summarize(total_damage = sum(Dam))

yearz = data.frame(Year = seq(1926, 1995, 1))

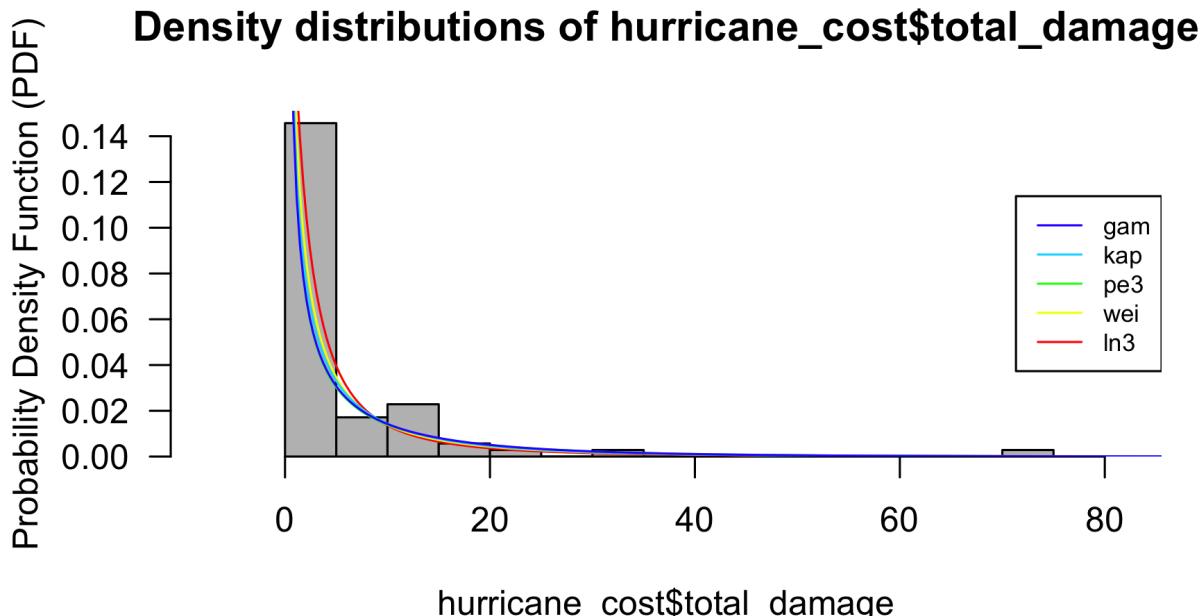
hurricane_cost = full_join(yearz, hurricane_cost)

hurricane_cost$total_damage[is.na(hurricane_cost$total_damage)] = 0
```

Now that the data is complete, we can use the `distLfit` function from `extremeStat` to fit a variety of extreme-value-oriented probability models to our data, and view the results:

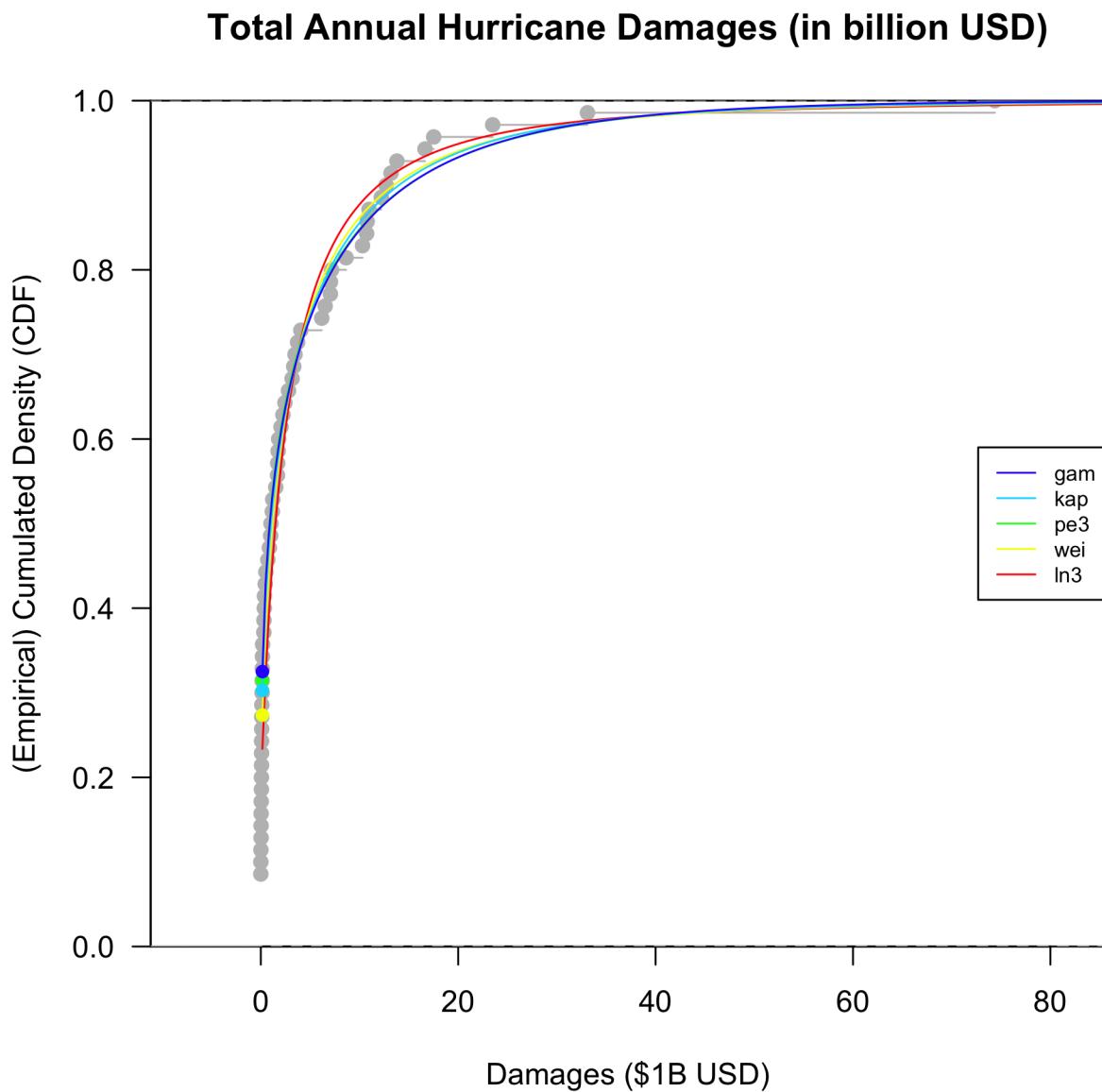
```
require(extremeStat)

hurr_fit = distLfit(hurricane_cost$total_damage)
```



A standard histogram is returned from this function call, which isn't as helpful as a CDF plot to compare against the actual values. Use `distLplot` with `cdf=TRUE` to return a CDF plot:

```
distLplot(hurr_fit, cdf=TRUE, main="Total Annual Hurricane Damages  
(in billion USD)", xlab="Damages ($1B USD)")
```



Unlike many R packages, `summary` and other universal functions don't work with `extremeStat` result objects because they return lists. So, to see a summary of the result, use `distLprint`.

```

distLprint(hurr_fit)

-----
Dataset 'hurricane_cost$total_damage' with 70 values.
min/median/max: 0/1.1/74.4 nNA: 0
truncate: 0 threshold: 0. dat_full with 70 values: 0/1.1/74.4 nNA: 0
dlf with 16 distributions. In descending order of fit quality:
gam, kap, pe3, wei, ln3, gno, wak, gpa, gev, glo, exp, gum, ray, nor,
revgum, rice
gofProp: 1, RMSE min/median/max: 0.037/0.076/0.3 nNA: 0
5 distribution colors: #3300FFFF, #00D9FFFF, #00FF19FF, #F2FF00FF, #FF0000FF

```

`distLfit` ranks the distribution models by goodness-of-fit, with the best fitting models at the top. You can evaluate the model list and their goodness of fit statistics with a call to the `$gof` portion of the list:

```
hurr_fit$gof
```

	RMSE	R2	weight1	weight2	weight3	weightc
gam	0.03687682	0.9831812	0.080260879	0.08360687	0.1513246	NaN
kap	0.03690034	0.9875113	0.080254559	0.08359936	0.1513110	NaN
pe3	0.04638276	0.9851915	0.077706869	0.08057171	0.1458311	NaN
wei	0.04791014	0.9728583	0.077296502	0.08008403	0.1449485	NaN
ln3	0.05935699	0.9641990	0.074221019	0.07642915	0.1383333	NaN
gno	0.06361586	0.9527421	0.073076767	0.07506933	0.1358721	NaN
wak	0.06965965	0.9435495	0.071452952	0.07313961	0.1323794	NaN
gpa	0.06965965	0.9435495	0.071452952	0.07313961	0.0000000	NaN
gev	0.08185683	0.9337443	0.068175873	0.06924516	0.0000000	NaN
glo	0.08242552	0.9309928	0.068023081	0.06906358	0.0000000	NaN
exp	0.11283198	0.8597058	0.059853627	0.05935507	0.0000000	NaN
gum	0.13746709	0.8141283	0.053234789	0.05148931	0.0000000	NaN
ray	0.14435908	0.7933789	0.051383083	0.04928876	0.0000000	NaN
nor	0.16469704	0.7654365	0.045918783	0.04279503	0.0000000	NaN
revgum	0.19498790	0.7189435	0.037780386	0.03312344	0.0000000	NaN
rice	0.29872851	0.7589475	0.009907879	0.0000000	0.0000000	NaN

As you probably noticed with the graphs, only the top five models are plotted, but this list will allow you to determine the extent to which other models may fit almost as well. The list of the actual model names that go with those abbreviations is below (also see `?prettydist` in the `lmomco` package).

R abbreviation	Distribution
aep4	4-p Asymmetric Exponential Power
cau	Cauchy
emu	Eta minus Mu
exp	Exponential
texp	Truncated Exponential
gam	Gamma
gep	Generalized Exponential Poisson
gev	Generalized Extreme Value
gld	Generalized Lambda
glo	Generalized Logistic
gno	Generalized Normal
gov	Govindarajulu
gpa	Generalized Pareto
gum	Gumbel
kap	Kappa
kmu	Kappa minus Mu
kur	Kumaraswamy
lap	Laplace
lmrq	Linear Mean Residual Quantile Function
ln3	log-Normal3
nor	Normal
pe3	Pearson Type III
ray	Rayleigh
revgum	Reverse Gumbel
rice	Rice
sla	Slash
st3	Student t (3-parameter)
tri	Triangular
wak	Wakeby
wei	Weibull

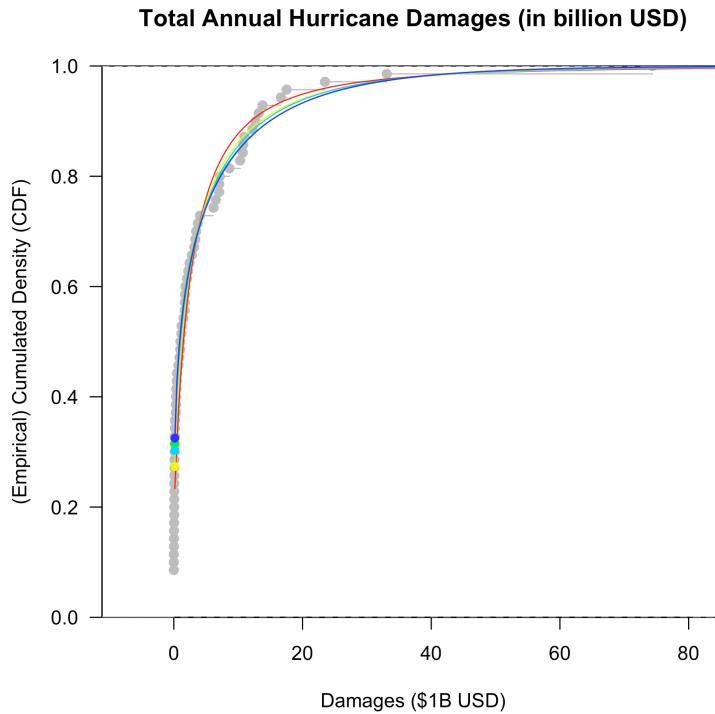
You can evaluate the quantiles of these models to help determine what “counts” as an outlier with the `distLquantile` function. Use the `probs` option to see the quantiles you want returned, whether low or high. You also need to use the `returnlist=TRUE` option to ensure you can work with the results after the call.

```
# Calculate quantiles for different models
hurr_quant = distLquantile(hurricane_cost$total_damage,
  probs=c(0.80, 0.90, 0.95, 0.99), returnlist=TRUE)

# Look at the top five models only
hurr_quant$quant[1:5,]

  80%      90%      95%      99%
gam 7.269448 14.90234 23.92810 47.89042
kap 6.949505 13.98193 22.79060 50.00316
pe3 7.159698 14.86357 24.03902 48.52204
wei 6.639476 13.45379 22.53004 52.15676
ln3 6.165669 11.94520 20.29897 53.86717

# Plot the cdf of top five models with their quantiles
distLplot(hurr_quant, cdf=T, main="Total Annual Hurricane Damages
(in billion USD)", xlab="Damages ($1B USD)", qlines=TRUE)
```



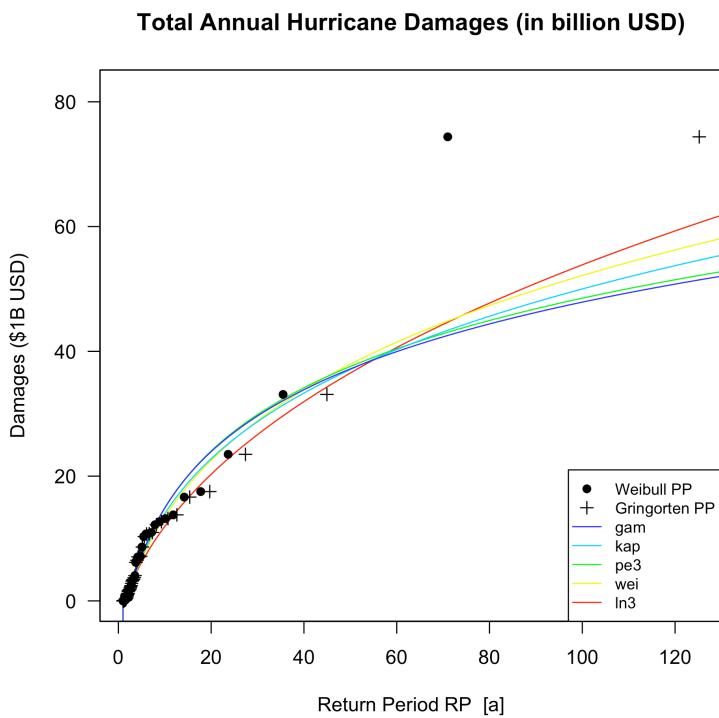
A great feature of this package is the easy of calculating and plotting return/recurrence interval estimates. Use the `distLextreme` function with your desired return levels set with the `RP` option.

```
# Calculate return intervals.
hurr_returns = distLextreme(hurricane_cost$total_damage, RPs = c(2,10,50,100))

# View top five model results
hurr_returns$returnlev[1:5,]

      RP.2     RP.10    RP.50    RP.100
gam 0.9028442 14.90234 37.18940 47.89042
kap 1.0322659 13.98193 37.07409 50.00316
pe3 0.8707481 14.86357 37.57498 48.52204
wei 1.1854233 13.45379 37.94427 52.15676
ln3 1.4838727 11.94520 36.52829 53.86717

# Plot return intervals
distLextremePlot(hurr_returns, main="Total Annual Hurricane Damages
(in billion USD)", ylab="Damages ($1B USD)")
```



Finding associations in shopping carts

Finding things that are (really) similar or (really) different is the common theme to the tools in this chapter. Sometimes, you just want to be purely empirical about it, and let the data itself give you information to explore based simply on common co-occurrences. Association analysis—more commonly called “shopping cart analysis”—is a great way to do this.

We haven’t worked with a data set that’s really amenable to association rules, so why not explore shopping cart analysis with a groceries dataset?

The `arules` package has a special function that allows you to read a data set directly in to R as a transaction object, for example, where you have the transaction identifier in one column and a particular item from that transaction in a second column.

```
require(arules)
require(arulesViz)

# Download data and convert to a transactions object
basket_url = "http://dmg.org/pmmml_examples/baskets1ntrans.csv"

basket_trans = read.transactions(basket_url, format = "single", sep = ",",
                                cols = c("cardid", "Product"), rm.duplicates=TRUE)
```

As usual, `summary` provides details of the resulting object:

```
summary(basket_trans)

transactions as itemMatrix in sparse format with
939 rows (elements/itemsets/transactions) and
11 columns (items) and a density of 0.2708878

most frequent items:
cannedveg frozenmeal fruitveg beer fish (Other)
303     302      299   293   292    1309

element (itemset/transaction) length distribution:
sizes
 1   2   3   4   5   6   7   8
174 227 218 176  81  38  21   4

Min. 1st Qu. Median Mean 3rd Qu. Max.
```

```
1.00    2.00    3.00    2.98    4.00    8.00
```

includes extended item information - examples:

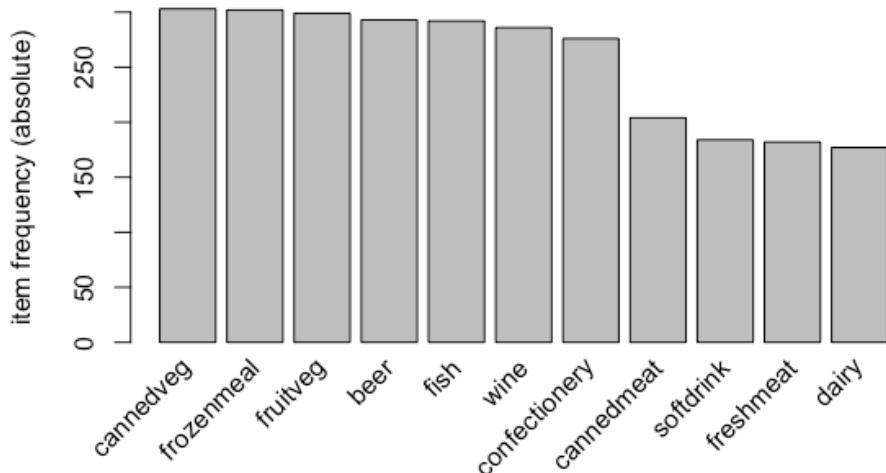
```
labels
1      beer
2 cannedmeat
3 cannedveg
```

includes extended transaction information - examples:

```
transactionID
1      100047
2      100255
3      100256
```

We can see that there are only 11 distinct items in this data set (must be a very small store!) from 939 distinct transactions, so we can plot all items by frequency using the `itemFrequencyPlot` function. Larger item sets can use the `topN` value to cut off the plot to show just the 15 or 20 or so most common items.

```
itemFrequencyPlot(basket_trans, topN=11, type="absolute")
```



The `apriori` function takes a transaction object and creates a rule set using default confidence and support levels of 0.8 and 0.1, respectively, with no more than 10 items on any one side of a rule. Accepting the defaults makes the call as simple as `basket_rules = apriori(basket_trans)`, but you will likely want to modify those to expand the rule set. To do so requires setting the parameter function with the details in a list. The `minlen` default is set at 1, which means that items purchased by themselves will show up—if you want to exclude single-item purchases, set `minlen=2` here. Note that

since there are a variety of possible target types (see `?ASparameter-classes`), you need to specify that you are looking for rules.

```
basket_rules_custom = apriori(basket_trans, parameter = list(support = 0.01,
  confidence = 0.01, target="rules"))
```

Apriori

Parameter specification:

confidence	minval	smax	arem	aval	originalSupport	support	minlen	maxlen	target	
0.01	0.1	1	none	FALSE		TRUE	0.01	1	10	rules
ext										
FALSE										

Algorithmic control:

filter	tree	heap	memopt	load	sort	verbose
0.1	TRUE	TRUE	FALSE	TRUE	2	TRUE

Absolute minimum support count: 9

```
set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[11 item(s), 939 transaction(s)] done [0.00s].
sorting and recoding items ... [11 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 3 4 5 done [0.00s].
writing ... [845 rule(s)] done [0.00s].
creating S4 object ... done [0.00s].
```

If you had run the `apriori` function with the defaults, you would have seen it created only 3 rules. By opening up the support and confidence levels, we now have 845 rules, which provides us with a much richer exploration space.



Confidence is the probability that a transaction including the items on one side of the rule also contains the items on the other side.

Support is the proportion of transactions that contain that item or item set. Higher support means the rule is relevant to larger proportion of transactions.

Now we can look at the rules. A key concept for rule usefulness is *lift*. A lift value at or close to one suggests that the items on either side of the rule are independent, so are usually of little further interest. Lift values above 1 suggest that the presence of the items on one side of the rule set increase

the chances of seeing items on the other side. Where items are all purchased or occur at once (e.g., as in a supermarket), the direction of the rule set is irrelevant. Where items are purchased or occur in a series (e.g., as in a web store, where selections of specific items might be placed in a meaningful order), the rule set is ordered from the first purchased (the {lhs}, left hand side), to those placed in the cart later (the {rhs}, right hand side).

We'll look first at the top 10 rule sets by greatest lift with the `inspect` function:

```
inspect(head(sort(basket_rules_custom, by="lift"), 10))
```

lhs	rhs	support	confidence	lift
838 {cannedveg, frozenmeal, fruitveg, wine}	=> {beer}	0.01171459	1.0000000	3.204778
802 {beer, cannedmeat, cannedveg, freshmeat}	=> {frozenmeal}	0.01171459	1.0000000	3.109272
807 {beer, cannedveg, fish, freshmeat}	=> {frozenmeal}	0.01064963	1.0000000	3.109272
817 {beer, cannedmeat, cannedveg, fish}	=> {frozenmeal}	0.01277955	1.0000000	3.109272
801 {beer, cannedmeat, freshmeat, frozenmeal}	=> {cannedveg}	0.01171459	1.0000000	3.099010
806 {beer, fish, freshmeat, frozenmeal}	=> {cannedveg}	0.01064963	1.0000000	3.099010
816 {beer, cannedmeat, fish, frozenmeal}	=> {cannedveg}	0.01277955	1.0000000	3.099010
821 {beer, cannedmeat, frozenmeal, fruitveg}	=> {cannedveg}	0.01064963	1.0000000	3.099010
609 {beer, freshmeat, frozenmeal}	=> {cannedveg}	0.03088392	0.9666667	2.995710
833 {cannedveg, fish, frozenmeal, wine}	=> {beer}	0.01490948	0.9333333	2.991126

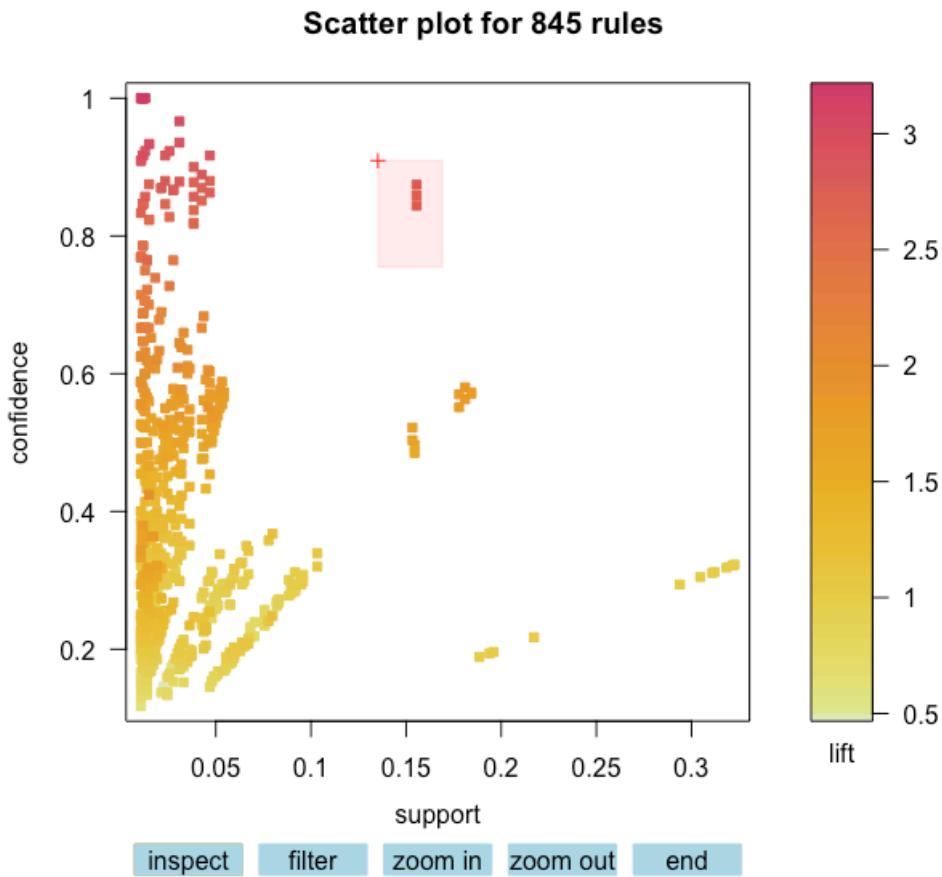
We can also plot the rule set using a variety of graph types; accepting the default gives you a scatterplot of confidence, support, and lift, and setting `interactive` to TRUE allows you to explore particular subsets based on the combination of those three values. For example, we can select the rules with relatively high values of all three within the graph, and see those rules in the console once we've clicked `inspect` in the graph:

```
plot(basket_rules_custom, interactive = T)
```

Interactive mode.
Select a region with two clicks!

Number of rules selected: 3

lhs	rhs	support	confidence	lift	order
555 {beer, cannedveg}	=> {frozenmeal}	0.1554846	0.8742515	2.718285	3
556 {cannedveg, frozenmeal}	=> {beer}	0.1554846	0.8439306	2.704610	3
554 {beer, frozenmeal}	=> {cannedveg}	0.1554846	0.8588235	2.661503	3



If you wanted to find items that were purchased with, say, beer, you can create a rule set specifically for that item:

```
beer_rules = apriori(data=basket_trans, parameter=list(supp=0.01,
conf = 0.01, minlen=2), appearance = list(default="rhs", lhs="beer"))
```

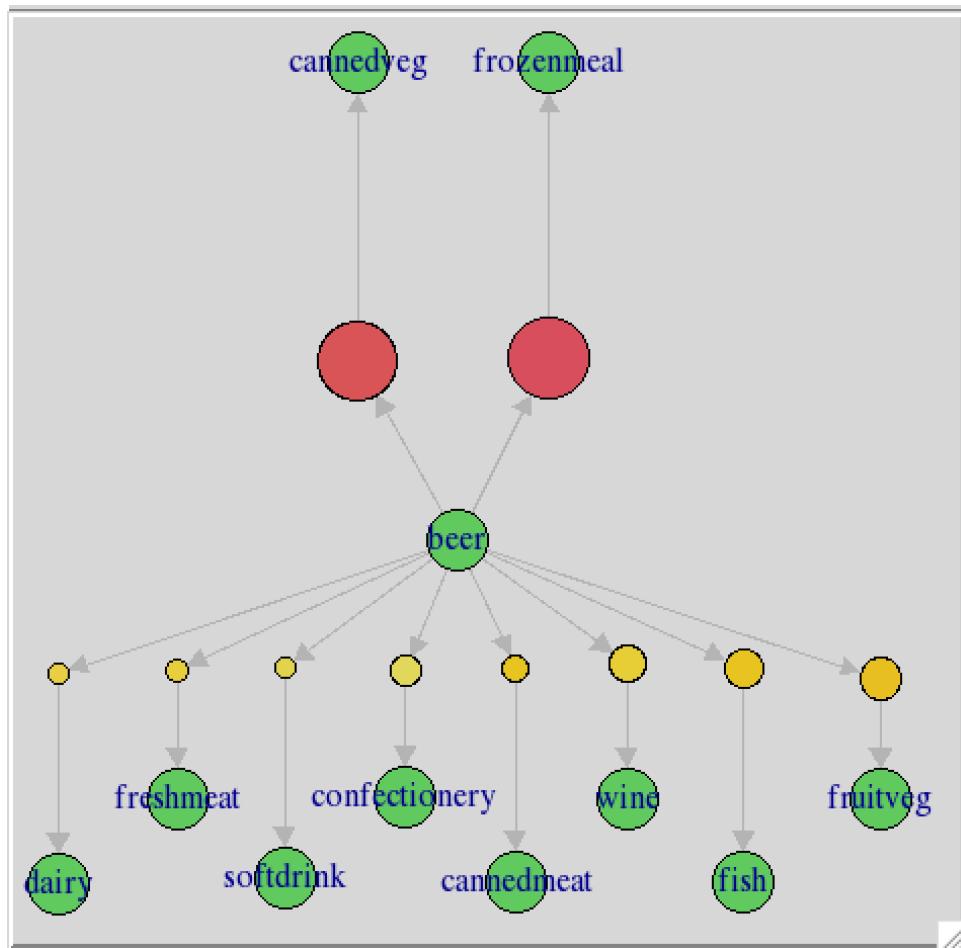
```
inspect(head(sort(beer_rules, by="lift"), 10))
```

lhs	rhs	support	confidence	lift
9 {beer} => {frozenmeal}		0.18104366	0.5802048	1.8040142
10 {beer} => {cannedveg}		0.17784878	0.5699659	1.7663299
8 {beer} => {fruitveg}		0.09478168	0.3037543	0.9539306
4 {beer} => {cannedmeat}		0.06389776	0.2047782	0.9425818
7 {beer} => {fish}		0.09052183	0.2901024	0.9328978
6 {beer} => {wine}		0.08200213	0.2627986	0.8628249
2 {beer} => {freshmeat}		0.05005325	0.1604096	0.8276075
1 {beer} => {dairy}		0.04792332	0.1535836	0.8147741

```
3 {beer} => {softdrink} 0.04792332 0.1535836 0.7837773
5 {beer} => {confectionery} 0.06815761 0.2184300 0.7431370
```

You can also create graph diagrams of a rule set, and move the nodes around to show the rule meanings. In this case, only two rules give positive lift, while the other eight show inverse associations. We can manually adjust the initial graph layout to show this relationship:

```
plot(beer_rules, method="graph", interactive=T)
```



To explore the rules in greater depth, you can write them to a text file to view outside of R using the `write` function.

```
write(basket_rules_custom, file = "basket_rules_custom.csv",
sep = ",", row.names=FALSE)
```

Chapter 9: Reporting and Dashboarding

- Reports and technical memos
- Dashboards
- Slide decks
- purl: scraping the raw code from your .Rmd files
- Shiny apps
- Tweaking the YAML headers

It's fine to create reports in word processing or slide show software, but it can be smarter to create data products entirely inside R. There are a number of benefits to doing this: you have the code embedded in your data product, so that final product automatically updates when you change the data or code the next time you 'knit' it; you can share documents that others can use for both the product and the code, which is great for reproducibility as well as documentation; and you can include interactive content, something impossible to do in static documents.

If you haven't created data products using R Markdown yet, you're in for a treat. To get started, just open a new R Markdown document from within RStudio , choose Document (keeping the HTML output default), and hit OK. A default template will open in the Source window, from which you can modify it as needed. Just click **Knit HTML** and provide a file name and your product is saved to your working directory as a stand-alone .html file. It also opens in a pop-up for immediate viewing. It is that simple to create data products that really impress decision makers used to PDFs and static slide decks.

Code occurs in "chunks", sections in which you place the code you want R to execute. Free text outside those chunks is converted into HTML or other output types via R Markdown, a flavor of the Markdown language that provides simple shortcuts to convert raw text into formatted HTML. You can also use regular HTML as well, such as
 for line breaks, and so on. See *Appendix 5* for a cheatsheet of R Markdown symbols and usage, and RStudio's [R Markdown page⁶⁷](#) for more detailed information.

The single biggest "oops" in using R Markdown—that you will immediately easily fix the first time it happens, but you will *also* inevitably forget to do this later—is ensuring you have put *two spaces* at the end of each line where you want a line break after text. I've used R Markdown for years, but even creating one of the examples below, I forgot to do this on a few lines and got a jumbled mess as a result. So, yeah, it's gonna happen.

⁶⁷<http://rmarkdown.rstudio.com/>

Ultimately, there is rapidly increasing emphasis on reproducible research in research and academia, the idea that *any* analytic product can be reproduced by an independent analyst. Industry is adopting the same philosophy, where ideally any analytic product created in one division of a company can be completely reproduced (and often subsequently modified) by another division, without having to spend time and resources reverse-engineering it. “The code *is* the documentation” is longer a viable excuse (or career choice!), so using R Markdown to create your data products from the start can save a lot of time later. Having it become an enterprise best practice/standard process is even better.

The examples in this chapter are just simple examples; there are many, many possibilities with all of these output options, and exploring the tutorials, galleries, and example pages are well worth the effort to find ready-to-go and ready-to-customize templates and apps you can use and modify for your own ends.

Output formats for .Rmd documents

Web (HTML) reports are ready to create in RStudio out-of-the-box. To create PDF or Word documents, you need to install programs on your machine that do not come with R. For PDF, you need a TeX program, and for Word docs you need either Word or Libre/Open Office:

OS	PDF	Word
Windows	MiKTeX ⁶⁸	MS Word
Mac OS X	MacTeX ⁶⁹	MS Word for Mac
Linux	Tex Live ⁷⁰	Libre Office ⁷¹ or Open Office ⁷²

Dashboards

There are a lot of dashboards in business intelligence that don’t really deserve the name. Just because you have graphs and interactivity doesn’t mean you have a real dashboard; much like driving a car, if you’re getting information about how fast you were going 10 minutes ago, it’s not of much use. Dashboards are usually about real-time analytics or for analytics that help decision makers plan their day’s activities after no more than a quick viewing. Forecasting fits into this group as well, much as a GPS or driving app provides you information about where you’re going. “Dashboards” meant for “what has happened”-type needs or for more methodical exploration and drill-downs are more accurately termed “reports.”

Still, several commercial products have allowed BI analysts to create interactive, dataviz-based reports. By habit, if nothing else, these have come to be known generally as dashboards anyway. So call them what you want, just make sure you’re clear on what type of tool you really need when you’re asked to create a “dashboard.”

⁶⁸<http://miktex.org/>

⁶⁹<https://tug.org/mactex/>

⁷⁰<https://www.tug.org/texlive/>

⁷¹<https://www.libreoffice.org/download/libreoffice-fresh/>

⁷²<https://www.openoffice.org/>

Simple dashboarding with R Markdown

Since .Rmd files can be exported as HTML, you can create simple dashboards easily in an R Markdown script. The resulting stand-alone file can be placed into a website directly or via an <iframe>; at my company, we can't upload javascript-based files into our Intranet, but we can upload them to Sharepoint and then call that file from its Sharepoint URL into an <iframe> on our Intranet.

Using data from the [University of Queensland Vital Signs Dataset⁷³](#), and the htmlwidgets-based package sparkline, we can create a simple example dashboard in an .Rmd file that shows vital signs for a surgery case (imagine it as streaming, rather than static data!).

Because the sparkline package is not yet fully developed, you need to add a small <style> hack to the CSS to get the tooltips to work properly. That hack is included at the top of the R Markdown file, just after the YAML header.

To explore this dashboard, open a new R Markdown file and replace the template text with the text below, then press the **Knit HTML** button.

```
---
```

```
output: html_document
```

```
---
```

```
<style>
.jqstooltip {
  -webkit-box-sizing: content-box;
  -moz-box-sizing: content-box;
  box-sizing: content-box;
}
</style>
```

```
### Vital Signs Dashboard
```

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = FALSE, message = FALSE, warning = FALSE)
```

```
devtools::install_github('htmlwidgets/sparkline')
see documentation at https://github.com/htmlwidgets/sparkline
require(sparkline)
require(lubridate)
```

---

<sup>73</sup><http://dx.doi.org/10.100.100/6914>

```
10 minute section of vital signs data from University of Queensland
vs = read.csv("https://outbox.eait.uq.edu.au/uqdl1u3/uqvitalsignsdataset/
_uqvitalsignsdata/case20/fulldata/uq_vsd_case20_fulldata_10.csv", header=T)

convert _ to . in case time and make into date time class
Date just defaults to current day since there's no date field in data
vs$Case_Time = gsub("_", ".", as.character(vs$Time), fixed = TRUE)
vs$Case_Time_ct = as.POSIXct(vs$Case_Time, format = "%H:%M:%OS")

Use start time from data set, create object for latest time
vs$Date_Time_ct = vs$Case_Time_ct + hours(11) + minutes(28)
curr_time = substr(as.character(vs$Date_Time_ct[length(vs$Date_Time_ct)]),
12, 21)
```

##### Patient MRN: **123456789**  

<br>
Case Start Time: `r vs$Date_Time_ct[1]`  

| *Vital* | *Trend up to current time:* `r curr_time` | *Distribution* |  

| ----- | ----- | ----- |  

| Heart Rate | `r sparkline(vs$HR, width=250, height=60)` | `r sparkline(vs$HR,
width=100, height=60, type="box")` |  

| Pulse | `r sparkline(vs$Pulse, width=250, height=60)` | `r sparkline(vs$Pulse,
width=100, height=40, type="box")` |  

| Minute Volume | `r sparkline(vs$Minute.Volume, width=250, height=60,
normalRangeMin=5,normalRangeMax=8, drawNormalOnTop=T,
normalRangeColor="lightgreen")` | `r sparkline(vs$Minute.Volume, width=100,
height=40, type="box")` |  

| Tidal Volume | `r sparkline(vs$Tidal.Volume, width=250, height=60,
normalRangeMin=280, normalRangeMax=400, drawNormalOnTop=T,
normalRangeColor="lightgreen", fillColor="transparent")` | `r
sparkline(vs$Tidal.Volume, width=100, height=60, type="box")` |  

<small>*Green bands show normal ranges.*</small>
```

Vital Signs Dashboard

Patient MRN: 123456789

Case Start Time: 2016-06-03 12:58:00



Green bands show normal ranges.

Note that the green bands are *not* the proper normal ranges for those vital signs, they're just there as an example. (Anyway, why would you take medical advice from a coding book?)

Dashboarding and reporting with flexdashboard

Yet another brilliant product from the RStudio folks is an easy-to-use dashboard template package called `flexdashboard`⁷⁴. It's built on top of the not-nearly-as-easy-to-use `Shiny dashboard`⁷⁵ approach, and it hides the complexity of the Shiny system for users who just want to make highly customizable dashboards using (mostly) R Markdown instead. That said, you can still use Shiny inside a flexdashboard, making it the best of both worlds.

View the [layouts](#)⁷⁶ and [examples](#)⁷⁷ before diving in; there's probably a combination of those already out there that will meet a good proportion of your initial design needs.

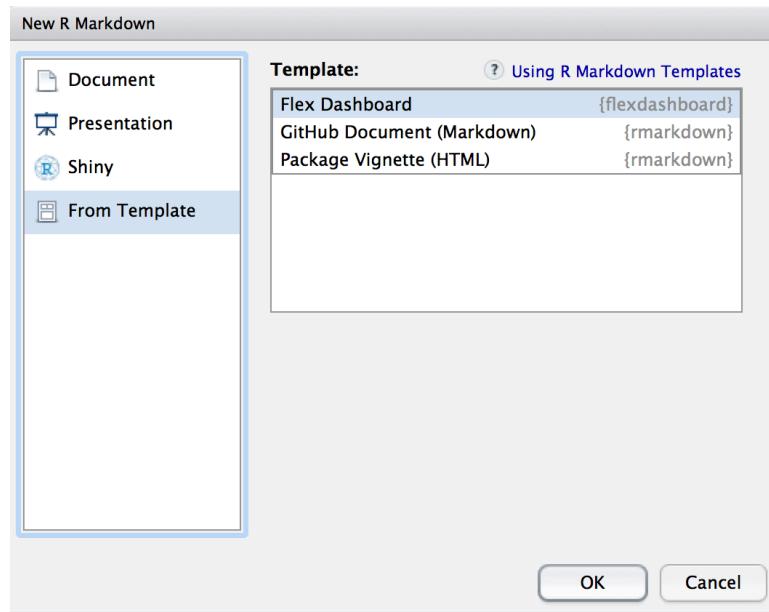
After you've installed `flexdashboard`, the creating a **New R Markdown** document dialog will have added the appropriate template to your options:

⁷⁴<http://rmarkdown.rstudio.com/flexdashboard/>

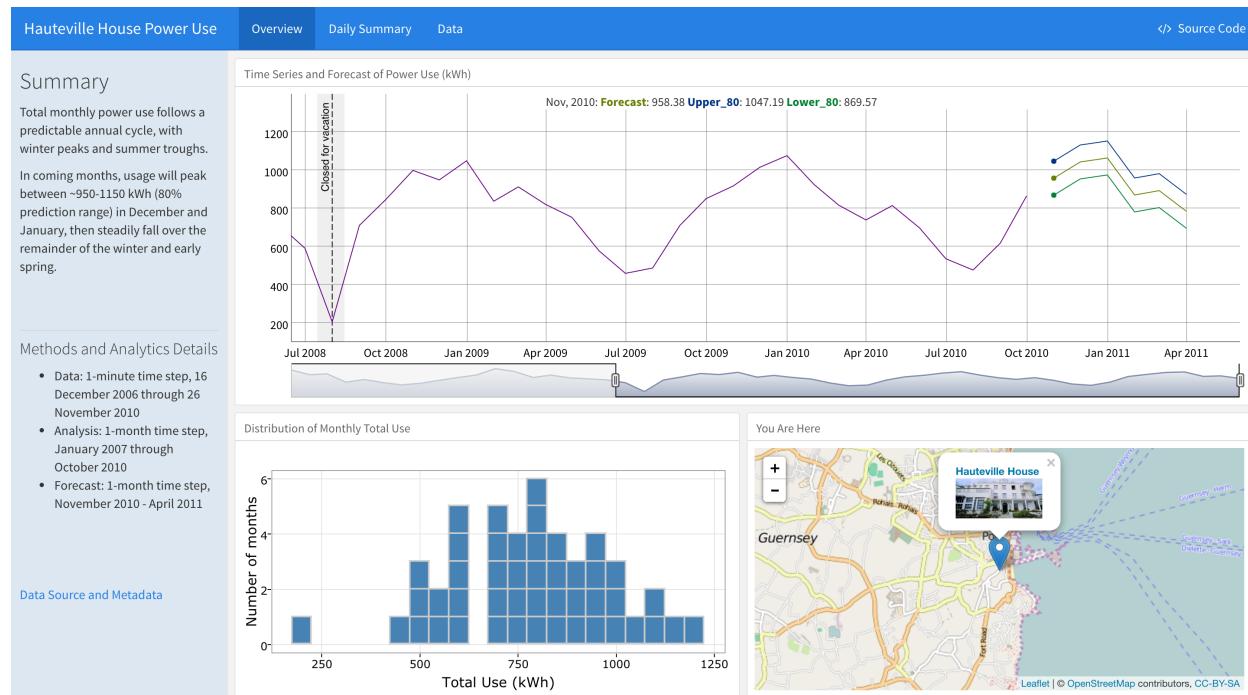
⁷⁵<https://rstudio.github.io/shinydashboard/>

⁷⁶<http://rmarkdown.rstudio.com/flexdashboard/layouts.html>

⁷⁷<http://rmarkdown.rstudio.com/flexdashboard/examples.html>



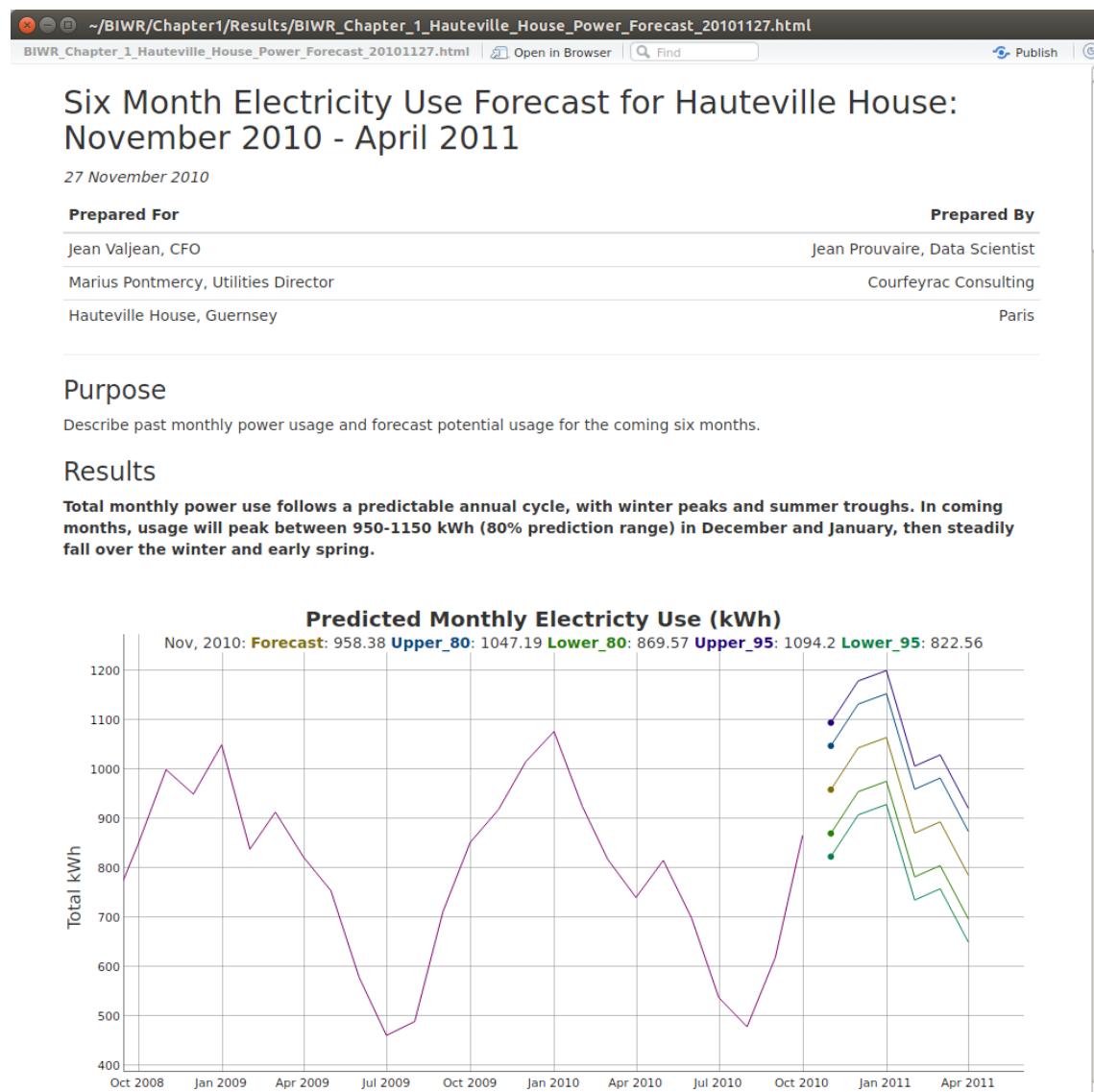
To show a simple example, we can use the same data and approach we used in *Chapter 1*, exploring and forecasting the power usage at the Hauteville House, only putting it into a dashboard-like format instead of a report. The code that creates this dashboard is contained in *Appendix 4*.



Reports and technical memos

The vast majority of analytics in modern BI tend to be report-oriented. Reports are typically look-back affairs, and can range from a simple dataset or table, to a set of KPI graphs on a single webpage, to a full-blown, multi-tab overview of a group of activities. Technical memos are usually one-off analytic products that answer a specific question, but can also provide guidance on next steps.

We saw an example of a data product that could be considered either a report or a memo in *Chapter 1*, in the exploration and forecasting of the Hauteville House's power consumption (full code in *Appendix 2*):



Slide decks

At some point, every analyst needs to report their results in a meeting or briefing. Traditional static slide decks are fine, but one of the powerful features of R Markdown is the ability to create slide decks or apps with interactive features, allowing the analyst to help decision makers explore the results and their implications interactively, in real time. Even if you don't create interactive content, having a slide deck that includes the code that generated the results is really useful for documentation, code review, or—most often—the inevitable revision when a decision maker asks another “what if you tried...?” question.

RStudio has two slide deck styles built in, `ioslides` and `slidy`. Which is better is probably a matter of taste, though I tend to like the aesthetics of `ioslides`.

```
---
```

```
title: Faster Than Thought
subtitle: A Symposium on Digital Computing Machines
author: A.A.L
date: December 10, 1815
output: ioslides_presentation
---
```

Faster Than Thought

A Symposium on Digital Computing Machines

A.A.L.
December 10, 1815

`ioslides` has a great default style, but I've found that some rooms (especially those with a lot of natural light) aren't amenable to the gray text. As an example of how to change this, get the latest `ioslides` style sheet (as of this writing it is [here⁷⁸](#)), and save it as a `.css` file into the same directory where you are saving your `.Rmd` file. I change the grey to black for these types of rooms by making these changes in the stylesheet:

⁷⁸[rmarkdown/inst/rmd/ioslides/ioslides-13.5.1/theme/css/default.css](#)

| Line | Style block | Original setting | Change to: |
|------|------------------------|------------------|---------------|
| 245 | slides > slide | color: #797979; | color: black; |
| 454 | h2 | color: #515151; | color: black; |
| 463 | h3 | color: #797979; | color: black; |
| 1339 | .title-slide hgroup h1 | color: #515151; | color: black; |
| 1344 | .title-slide hgroup h2 | color: darkgray; | color: black; |
| 1339 | .title-slide hgroup p | color: #797979; | color: black; |

I save this new stylesheet as `default_black.css`. Then, in the YAML header of your `.Rmd` file, change `output: ioslides_presentation` to (line breaks and spacing matters!):

```
---
title: Faster Than Thought
subtitle: A Symposium on Digital Computing Machines
author: A.A.L
date: December 10, 1815
output:
  ioslides_presentation:
    css: default_black.css
---
```

Faster Than Thought

A Symposium on Digital Computing Machines

A.A.L.
December 10, 1815



As long as you have both the `.css` and `.Rmd` files in the same folder, it will make the conversion for you. Obviously, any style element can be changed in the same way, so even if you don't know CSS, you can probably web search what you want to change to find out what that particular element is named in HTML, and go from there.

purl: scraping the raw code from your `.Rmd` files

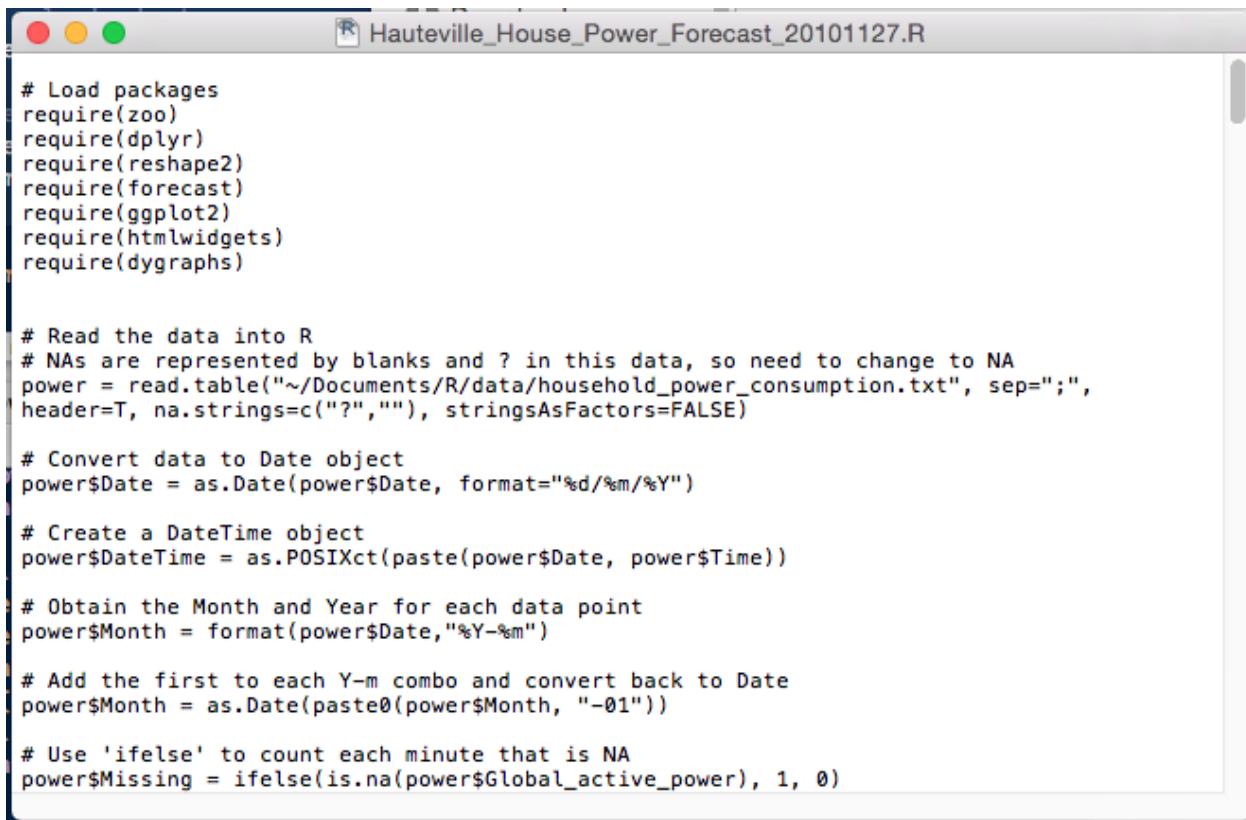
The `purl` function in `knitr` allows you to “scrape” your R Markdown files for the R code. So if you’ve created a data product in R Markdown and you want just the code from it copied into a separate

file, you can use `purl`. Using the Chapter 1 final data product as an example:

```
purl("Hauteville_House_Power_Forecast_20101127.Rmd", documentation = 0)
```

The `documentation = 0` option removes the `knitr` code chunk options from your output, so you just get the raw R code. Use `documentation = 2` to keep those options in the resulting file, which will appear as commented lines (#).

A new .R file will appear in your working directory with the raw R code.



```
# Load packages
require(zoo)
require(dplyr)
require(reshape2)
require(forecast)
require(ggplot2)
require(htmlwidgets)
require(dygraphs)

# Read the data into R
# NAs are represented by blanks and ? in this data, so need to change to NA
power = read.table("~/Documents/R/data/household_power_consumption.txt", sep=";",
header=T, na.strings=c("?", ""), stringsAsFactors=FALSE)

# Convert data to Date object
power$date = as.Date(power$date, format="%d/%m/%Y")

# Create a DateTime object
power$dateTime = as.POSIXct(paste(power$date, power$time))

# Obtain the Month and Year for each data point
power$month = format(power$date, "%Y-%m")

# Add the first to each Y-m combo and convert back to Date
power$month = as.Date(paste0(power$month, "-01"))

# Use 'ifelse' to count each minute that is NA
power$missing = ifelse(is.na(power$Global_active_power), 1, 0)
```

Shiny apps

Shiny is an incredible tool—being able to update parameters and inputs on the fly is a tremendous improvement for decision makers in exploring “what if?” scenarios alongside the analyst or even flying solo. Basically, if you can do it in R, you can do it in Shiny. RStudio’s [Shiny page⁷⁹](#) is a wealth of knowledge⁸⁰, [demos⁸¹](#), and [examples⁸²](#) you can pull from to create your own app.

⁷⁹<http://shiny.rstudio.com/>

⁸⁰<http://shiny.rstudio.com/tutorial/>

⁸¹<https://www.rstudio.com/products/shiny/shiny-user-showcase/>

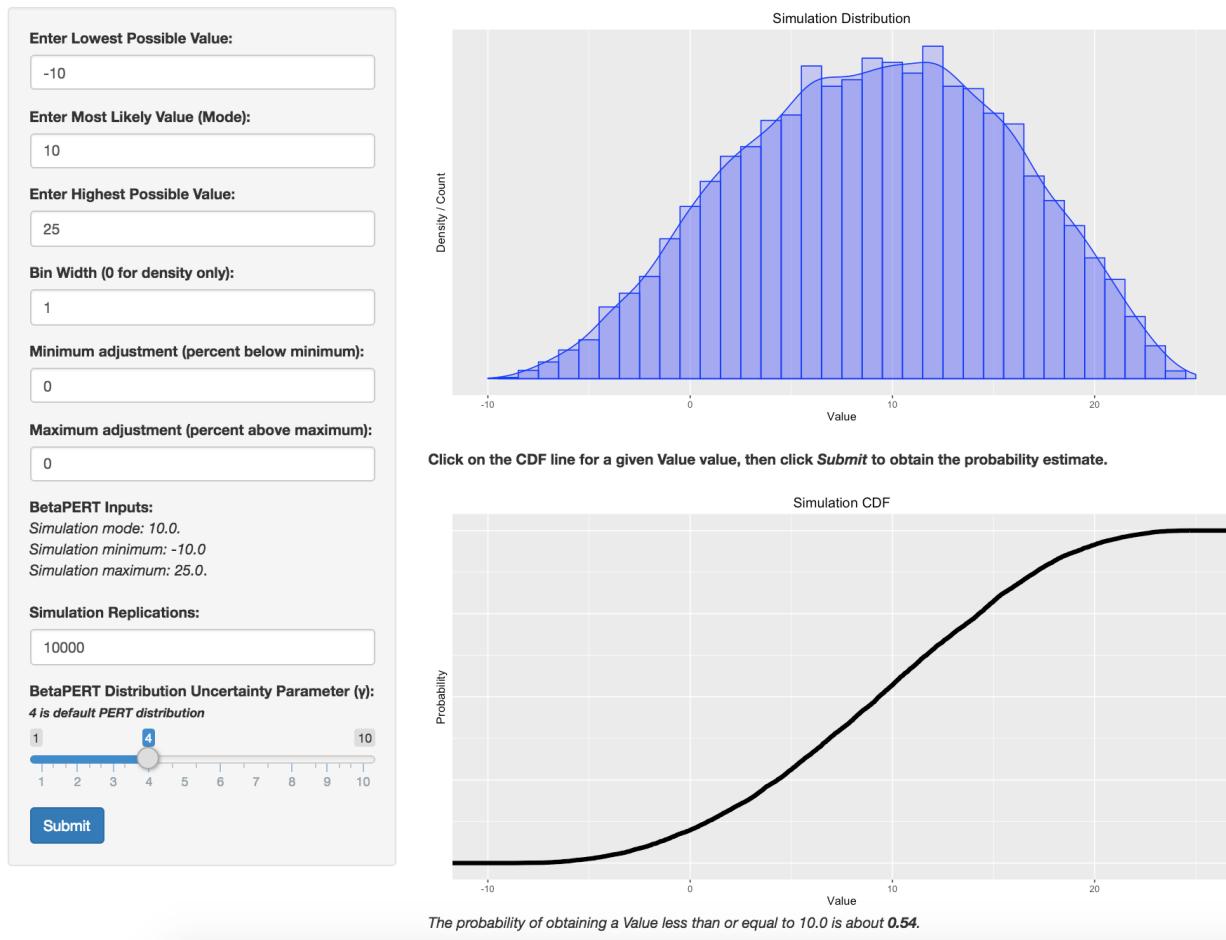
⁸²<http://shiny.rstudio.com/gallery/>

A great use-case for Shiny is in risk assessment, where the decision maker can see the effects of assumptions on the probability of a value or range of values, exploring them in real time. (This can be dangerous as well, if they change their assumptions to obtain the outcome they want to see!)

Below is a screenshot of an app that uses the basic risk assessment tool of the [betaPERT distribution⁸³](#) to create a probability model of expert opinion, where the subject matter expert provides the value of minimum and maximum possible values as well as the most likely value. An uncertainty parameter *gamma* provides the user with the ability to modify the kurtosis of the distribution; a value of 4 is the [standard betaPERT⁸⁴](#), while lower values show increased uncertainty and larger values show increased certainty in the estimate of the most likely value (termed the [modified betaPERT⁸⁵](#)).

The code that created this app is contained in *Appendix 3*.

Expert Opinion Risk Explorer



⁸³<https://www.riskamp.com/beta-pert>

⁸⁴http://vosesoftware.com/ModelRiskHelp/index.htm#Distributions/Continuous_distributions/PERT_distribution.htm

⁸⁵http://vosesoftware.com/ModelRiskHelp/index.htm#Distributions/Continuous_distributions/Modified_PERT_distribution.htm

Tweaking the YAML headers

The YAML header provides you with a way to customize the output of R Markdown files. The defaults are generally great for most needs, but occasionally you want to make some modifications, and I've found that the web doesn't have a lot of information readily available on this specific to R Markdown. There is some information [here⁸⁶](#), and a few tips and tricks I use often are below.

The header can include R code and HTML. For example, if you have an `ioslides` file and want to have the date automatically show up when run, use single quotes with back ticks to contain the R Markdown text:

```
---
```

```
title: This title doesn't have a short title
subtitle: Dude, where's my banana?
author: Dr. Nefario, Gru, and Kevin
date: `r format(Sys.Date(), "%B %d, %Y")`'
output: ioslides_presentation
---
```

This title doesn't have a short title

Dude, where's my banana?

Dr. Nefario, Gru, and Kevin
June 06, 2016



You can use HTML to customize the text. For example:

```
---
```

```
title: This title <br>doesn't have<br> a short title
subtitle: Dude, <em>where's</em> <del>my</del> <b>&beta;anana</b>?
author: Dr. Nefario, Gru, and Kevin
date: `r format(Sys.Date(), "%B %d, %Y")`'
```

⁸⁶http://rmarkdown.rstudio.com/authoring_pandoc_markdown.html#extension-yaml_metadata_block

```
output: ioslides_presentation  
---
```

This title doesn't have a short title

Dude, where's my banana?

Dr. Nefario, Gru, and Kevin
June 06, 2016

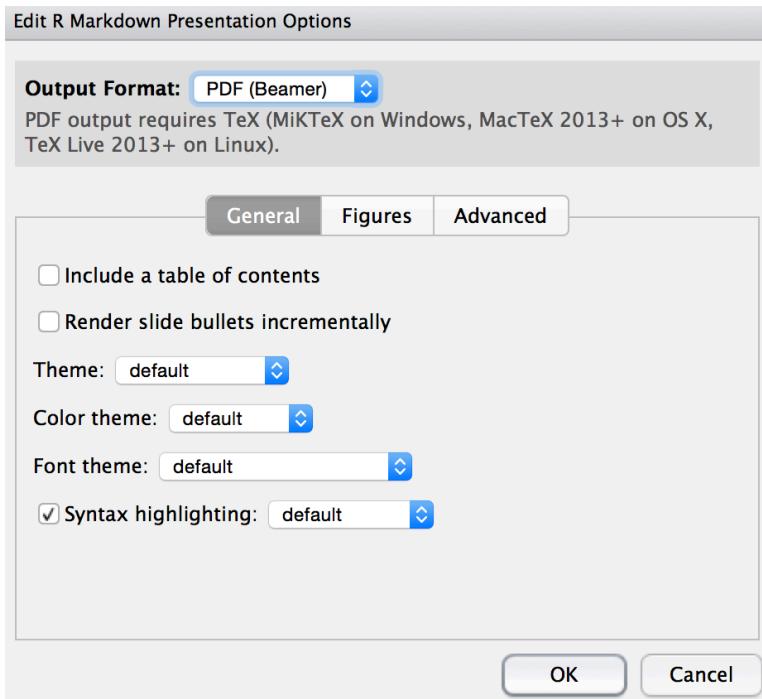
Some characters need to be escaped in YAML for it to work with text you wish to display—for example, to use a colon (:) you need to escape the whole string with double quotes (or use single quotes and double them up for apostrophes):

```
---  
title: "This title <br>doesn't have<br> a short title:  
      <br>Dude, <em>where's</em> <del>my</del> <b>&beta;anana</b>?"  
author: Dr. Nefario, Gru, and Kevin  
date: `r format(Sys.Date(), "%B %d, %Y")`'  
output: ioslides_presentation  
---
```

This title doesn't have a short title: **Dude, where's my Banana?**

Dr. Nefario, Gru, and Kevin
June 06, 2016

The gear button at the top of the Source window can be used to modify the YAML via a GUI for many of the major options for each type of R Markdown document. For example, if you wanted a PDF output of your slide deck, the gear dialog would show this:



If you poke around the [R Markdown site⁸⁷](#) and/or [Stack Overflow⁸⁸](#), you can probably find answers to most R/YAML-associated questions you may have.

⁸⁷<http://rmarkdown.rstudio.com/>

⁸⁸<http://stackoverflow.com/search?q=R+YAML+header>

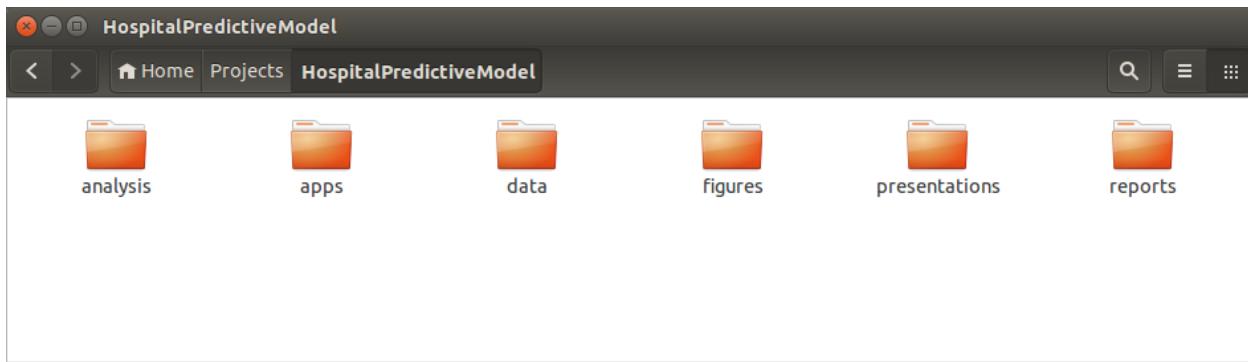
Appendix 1: Setting up projects and using “Make” files

Setting up a project with RStudio and Git

Setting up a project before you begin any analysis takes a little time, but is an essential task for professional data science. Having a standard set of processes makes redoing, recreating, or reproducing any analysis a breeze.

First, if you don’t already have Git or Subversion installed, you’ll need to install at least one of them, depending on your preference. Unless you have a really good reason not to, you should go with [Git⁸⁹](#) (Subversion is a “legacy” version control system). While it can provide all a solitary coder needs locally, Git is especially useful as it allows for concurrent and asynchronous collaboration and development via [GitHub⁹⁰](#), which serves as an online version control/backup/collaboration platform.

Outside of R, create a separate directory for every project, and within that directory, set up folders that allow you to contain code, data, and output in logical groupings. At a minimum, it’s best to have separate folders for data, analysis, and products. For example, a project for which you expect to produce a static report, a presentation, and an interactive app might be set up as follows:



The particular folder names and organization are not important. What’s important is having a system that is general enough to use for any project and straightforward enough to share with others with minimal explanation.

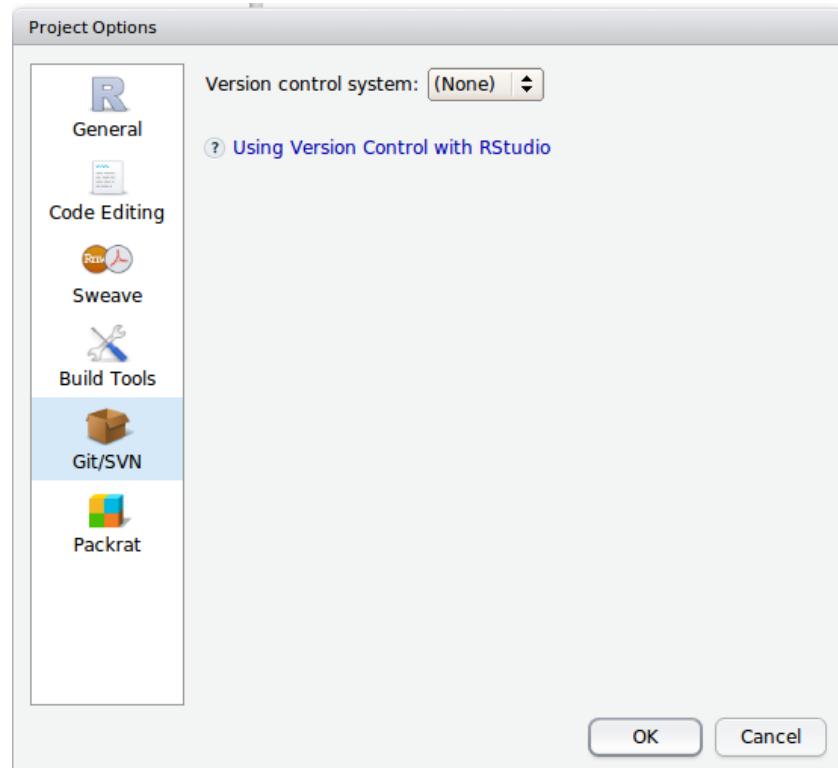
After completing the directory setup, open RStudio.

The best way to organize a project within R and put it under version control is through RStudio. Under the **File** menu, click *New Project*. Then click *Create project from: Existing directory*. A blank

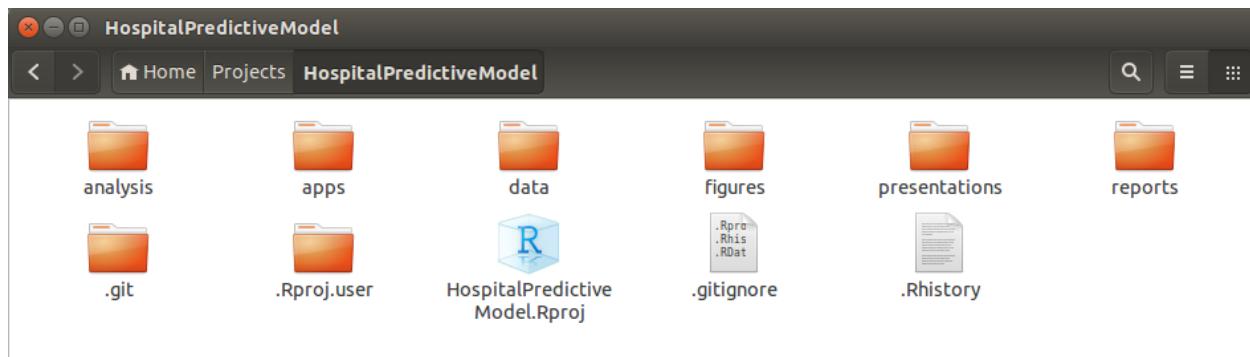
⁸⁹<https://git-scm.com/>

⁹⁰<https://github.com/>

RStudio environment will open. Then select the **Tools** menu, click *Version Control*, and then *Project Setup*. A window will open:



Select Git from the **Version control system:** button, then click yes, and yes again. RStudio will reboot, and your project is now under local version control. The directory will now look like this (including hidden files):



Making a habit of setting up folders like this in a systematic way makes it easy for others to understand and replicate your work. Adding version control gives you the ability review every change you made since the start of the project, to roll back to a previous state if a path of coding turns out to be a dead end or if something is accidentally deleted, and also allows for more efficient and safer collaboration with others.

Git works by storing the project as a set of “snapshots” that keep unmodified files alongside links to changes in modified files that will allow a rollback to a previous changepoint.

It’s important to remember to not use spaces in any of your folder or file names, as Git (as well as many other data science tools) often may not deal with them appropriately.

Committing your changes

Once your project is under version control with Git, a new tab will be available in your RStudio session titled “Git.” Each file appears in the list, and whenever changes are made and saved in your project files, the modified files in this list will have a set of question marks appear in the Status field. Clicking the checkbox alongside those files and then clicking the **Commit** button saves the project state at that point.

A dialogue box will then pop up that will allow you to provide notes on what changes were made. The level of detail to include is up to you, but it’s good practice to use enough detail that someone unfamiliar with your project could understand it enough to recreate or roll it back safely.

Rolling back to a previous state

Rolling back to a previous state is as simple as clicking the **Revert** button in the Git tab.

Packaging a project

By setting up a project this way, the only additional thing you need to do is use relative paths in all scripts and R markdown files—you’ll be able to tar/zip up the entire project and ship it to others, knowing it is already set up to run as-is.

RStudio has a good overview of its integration with version control for both [Git and Subversion](#)⁹¹.

“Make” files in R

The “makefile” is so named with a hat-tip to Make, a program that builds programs from source code. An R “makefile” runs a set of independent R scripts, which allows you to create modular code (which is far easier to debug than one long script), as well as mix and match scripts as needed across a variety of projects. For any given project, this R script could be run once and it would (re)create the entire analysis and output. An example of an R “makefile” is below, which also highlights the “clear text commenting” approach:

```
<<(code/Makefile_example.R)
```

Alongside the “makefile,” it’s useful to have an R Markdown file open in which you can document methods and results as you develop the project. This makes final product development considerably

⁹¹<https://support.rstudio.com/hc/en-us/articles/200532077-Version-Control-with-Git-and-SVN>

easier, but it also serves as a reminder to code with the audience in mind. If you habitually write out what you did and what it means alongside the code that generates the results, you’ll save time later—for you, as well as for any downstream or future users.

Because of system and platform nuances with PDF or Word export and slide creation styles, it’s usually best to use the HTML document option for the main markdown file’s format. It’s slightly easier to modify and convert that to LaTeX/PDF or Word, for example, than the other way around, especially if you haven’t yet set up RStudio to create PDFs (see *Chapter 9*). Once you’ve selected **R Markdown** in the **New File** menu, you’ll get a dialogue box that provides you an opportunity to add a title and your name. The choice defaults to HTML, so simply clicking on **OK** opens a default example.

The `knitr` package allows you to include R code within chunks, marked by three backticks at the start and end of each chunk, as well inline with text, as marked with a single backtick at the start and end of the portion of the R code to be analyzed. The code so designated is run each time you “knit” the file, so if the data changes, the analysis will repeat as before on the new data. The benefit of this for writing reports or presentations where revisions of the data are inevitable is clear.

Click the `?` or `MD` button in the `.Rmd` script window of RStudio for a quick overview of Rmarkdown syntax, and see *Appendix 2* for an example based on the *Chapter 1* analysis.

Appendix 2: .Rmd File for the *Chapter 1* final report example

To run this file manually, just load it into RStudio and click the **Knit HTML** button in the Source window. A preview window will pop up. In some systems, the HTML file is concurrently saved to the working directory, while in others you may need to **Open in browser** and then **Save as** from there.

To run an `.Rmd` file programatically, the `rmarkdown` package allows you to run it from the console, a command line, or a “make” file (see *Appendix 1*) via the `render` function, e.g.:

```
render("~/BIWR/Chapter1/Code/Hauteville_House_Power_Forecast_20101127.Rmd", "html_document")
```



This code *will fail* if you've already created the directories; comment those sections out if you're running it locally for testing.

```
---
```

```
title: ""
output: html_document
```

```
---
```

```
## Six Month Electricity Use Forecast for Hauteville House: November 2010 - April 1 2011
*27 November 2010*
```

```
Prepared For	Prepared By
Jean Valjean, CFO | Jean Prouvaire, Data Scientist
Marius Pontmercy, Utilities Director | Courfeyrac Consulting
Hauteville House, Guernsey | Paris
```

```
<hr>
```

```
### Purpose
```

```
Describe past monthly power usage and forecast potential usage for the coming si\
```

```
x months.
```

```
### Results
```

```
**Total monthly power use follows a predictable annual cycle, with winter peaks \
and summer troughs. In coming months, usage will peak between 950-1150 kWh (80% \
prediction range) in December and January, then steadily fall over the winter an\
d early spring.**
```

```
<br>
```

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo=FALSE, warning=FALSE, message=FALSE)
````
```

```
```{r everything}
```

```
Create project directories
Commented out here to avoid error
dir.create("~/BIWR/Chapter1/Code", recursive=T)
dir.create("~/BIWR/Chapter1/Data")
dir.create("~/BIWR/Chapter1/Results")
```

```
Set the working directory
setwd("~/BIWR/Chapter1")
```

```
Load packages
require(zoo)
require(dplyr)
require(reshape2)
require(forecast)
require(ggplot2)
require(htmlwidgets)
require(dygraphs)
```

```
Download the zip file into the data folder
#download.file("http://archive.ics.uci.edu/ml/machine-learning-databases/00235/h\
ousehold_power_consumption.zip", destfile="Data/household_power_consumption.zip"\\
)
```

```
Unzip the data from the zip file into the Data folder
#unzip("Data/household_power_consumption.zip", exdir="Data")
```

```
Read the data into R
NAs are represented by blanks and ? in this data, so need to change to NA
power = read.table("Data/household_power_consumption.txt", sep=";", header=T,
na.strings=c("?", ""), stringsAsFactors=FALSE)

Convert date to an ISO date
power$Date = as.Date(power$Date, format="%d/%m/%Y")

Create a DateTime object
power$DateTime = as.POSIXct(paste(power$Date, power$Time))

Obtain the Month and Year for each data point
power$Month = format(power$Date, "%Y-%m")

Add the first to each Y-m combo and convert back to ISO date
power$Month = as.Date(paste0(power$Month, "-01"))

Use ifelse to count each minute that is NA
power$Missing = ifelse(is.na(power$Global_active_power), 1, 0)

Use dplyr's group_by function to group the data by Date
power_group_day = group_by(power, Date)

Use dplyr to summarize by our NA indicator
(where 1 = 1 minute with NA)
power_day_missing = summarize(power_group_day, Count_Missing = sum(Missing))

Use zoo to perform interpolation for missing time series values
power$Global_active_power_locf = na.locf(power$Global_active_power)

Use dplyr to group by month
power_group = group_by(power, Month)

Use dplyr to get monthly max demand and total use results
power_monthly = summarize(power_group,
 Max_Demand_kw = max(Global_active_power_locf),
 Total_Use_kWh = sum(Global_active_power_locf)/60)

Remove partial months from data frame
power_monthly = power_monthly[2:47,]
```

```
Convert Month to Date
power_monthly$Month = as.Date(paste0(power_monthly$Month, "-01"))

Create a time series object of monthly total use
total_use_ts = ts(power_monthly$Total_Use_kWh, start=c(2007,1), frequency=12)

Automatically obtain the forecast for the next 6 months
total_use_fc = forecast(total_use_ts, h=6)

Create a data frame with the original data
and placeholders for the forecast details
use_df = data.frame(Total_Use = power_monthly$Total_Use_kWh,
Forecast = NA, Upper_80 = NA,
Lower_80 = NA, Upper_95 = NA, Lower_95 = NA)

Create a data frame for the forecast details
with a placeholder column for the original data
use_fc = data.frame(Total_Use = NA, Forecast = total_use_fc$mean, Upper_80 =
total_use_fc$upper[,1], Lower_80 = total_use_fc$lower[,1],
Upper_95 = total_use_fc$upper[,2], Lower_95 = total_use_fc$lower[,2])

"Union" the two data frames into one
use_ts_fc = rbind(use_df, use_fc)

Create a time series of the data and forecast results
total_use_forecast = ts(use_ts_fc, start=c(2007, 1), freq=12)

Create the widget
energy_use_prediction_widget = dygraph(total_use_forecast,
main = "Predicted Monthly Electricity Use (kWh)",
ylab = "Total kWh", width=900, height=500) %>%
dySeries(c("Total_Use"), label = "Actual kWh Usage") %>%
dyEvent(x = "2008-08-01", "Closed for vacation", labelLoc = "top") %>%
dyRangeSelector(dateWindow = c("2008-09-15", "2011-06-01")) %>%
dyLegend(width = 800)

Display the widget in the Viewer window
Hit the Zoom button for a pop-out
energy_use_prediction_widget
```

<br>
```

Using the upper 80% prediction interval is a reasonable way to plan conservatively for power costs.

```
```{r forecasttable}
require(htmlTable)
total_use_fc_df = data.frame(total_use_fc)
colnames(total_use_fc_df) = c("Point Forecast", "Low 80%", "High 80%",
 "Low 95%", "High 95%")

fc_table = htmlTable(txtRound(total_use_fc_df, 0),
 col.columns = c(rep("none", 2), rep("#E6E6F0", 1), rep("none", 2)),
 css.cell = "padding-left: 3.5em; padding-right: 1em;",
 align.header="rrrrrr",
 align="rrrrrr")
```

fc\_table

<br>  
<hr>

#### #### Methods and Analytics Details

##### \*\*Measurement Period\*\*

Data: 1-minute time step, `r format(min(power\$Date), "%d %B %Y")` through `r format(max(power\$Date), "%d %B %Y")`

Analysis: 1-month time step, `r format(min(power\_monthly\$Month), "%B %Y")` through `r format(max(power\_monthly\$Month), "%B %Y")`

Forecast: 1-month time step, November 2010 - April 2011

##### \*\*Data Source\*\*

<https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption>

##### \*\*Data Dictionary\*\*

\*Original\*

Dimensions: `r dim(power)[1]` observations of `r dim(power)[2]` variables

1. \*\*`date`\*\*: date in format dd/mm/yyyy
2. \*\*`time`\*\*: time in format hh:mm:ss
3. \*\*`global\_active\_power`\*\*: household global minute-averaged active power (in \ kilowatts)
4. \*\*`global\_reactive\_power`\*\*: household global minute-averaged reactive power \ (in kilowatts)
5. \*\*`voltage`\*\*: minute-averaged voltage (in volts)
6. \*\*`global\_intensity`\*\*: household global minute-averaged current intensity (in amperes)
7. \*\*`sub\_metering\_1`\*\*: energy sub-metering No. 1 (in watt-hours of active energy). It corresponds to the kitchen, containing mainly a dishwasher, an oven, and\ a microwave (hot plates are not electric but gas powered).
8. \*\*`sub\_metering\_2`\*\*: energy sub-metering No. 2 (in watt-hours of active energy). It corresponds to the laundry room, containing a washing-machine, a tumble-\ drier, a refrigerator, and a light.
9. \*\*`sub\_metering\_3`\*\*: energy sub-metering No. 3 (in watt-hours of active energy). It corresponds to an electric water-heater and an air-conditioner.

\*\*`global\_active\_power`\*\* is the variable of interest.

Missing values pattern:

```
```{r missingplot}
# Download the 'calendarHeat' function from revolutionanalytics.com
source("http://blog.revolutionanalytics.com/downloads/calendarHeat.R")

# Plot the calendar graph to view the missing data pattern
calendarHeat(power_day_missing$date, power_day_missing$count_missing,
             varname="Missing Data", color="w2b")
```

```

\*Transformations\*

Missing values were interpolated using the 'carry last value forward' approach. \ Comparison of distributions:

```
```{r missingvalues, fig.height=2.5}
# Compare the original and interpolated distributions
# Reshape the two variables into long form for ggplot
power_long = melt(power, id.vars= "DateTime", measure.vars=
  c("Global_active_power", "Global_active_power_locf"))
```

```
# Create density plot
density_plot = ggplot(power_long, aes(value, fill=variable,
                                         color=variable)) +
  geom_density(alpha=0.75) +
  facet_wrap(~variable)

# Display plot
density_plot
````
```

Values were aggregated to a monthly time step.

```
```{r monthlyuse, fig.height=2.5}
# Create plot of total use by month
total_use_plot = ggplot(power_monthly, aes(Month, Total_Use_kWh)) +
  geom_line(col="blue", lwd=1)

# Display plot
total_use_plot
````
```

\*Final\*

Dimensions: `r dim(power\_monthly)[1]` observations of `r dim(power\_monthly)[2]` \ variables

R time series object

1. \*\*`Month`\*\*: Observation or forecast month (`ts` identifier, not a variable in the object)
2. \*\*`Total\_Use`\*\*: Total monthly active power use for period of record (in kilo\\ watt-hours)
3. \*\*`Forecast`\*\*: Point forecast of total monthly active power use (in kilowatt\\ -hours)
4. \*\*`Upper\_80`\*\*: Upper 80% prediction interval **for** the forecast of total month\\ ly active power use (**in** kilowatt-hours)
5. \*\*`Lower\_80`\*\*: Lower 80% prediction interval for the forecast of total month\\ ly active power use (in kilowatt-hours)
6. \*\*`Upper\_95`\*\*: Upper 95% prediction interval **for** the forecast of total month\\ ly active power use (**in** kilowatt-hours)
7. \*\*`Lower\_95`\*\*: Lower 95% prediction interval **for** the forecast of total month\\ ly active power use (**in** kilowatt-hours)

```
```{r basic_fc_plot, fig.height=3}
# View model summary
summary(total_use_fc)

# View the forecast plot
plot(total_use_fc)
```

Session Info

File compiled:
```{r sessioninfo, echo=FALSE, warning=FALSE, message=FALSE}
Sys.time()
sessionInfo()
```

End of File
```

# Appendix 3: .R file for *Chapter 9 Shiny app example*

Below is the full code used to build the risk assessment Shiny app in *Chapter 9*. When I create R code that will be used by many analysts, I use a *lot* more whitespace. Again, use whatever standards you think are appropriate; I find that sharing code that many eyes will see is much nicer on those eyes when it's easy to see what each piece is doing.

If you download this file directly, it helps to put it into its own subdirectory (say, underneath a Shiny\_Apps folder), and to rename it app.R. That way, RStudio recognizes that it is a Shiny app, and will provide the appropriate changes to the Source window options.

```
#####
Risk Explorer Shiny App
Dwight Barry, PhD | Enterprise Analytics
November 2015 - Version 0.9 - Proof of concept
May 2016 - Version 1.2 - expert opinion only
Changes from 1.0: added data input and mix/max adjustment
Changes from 1.1: substituted function for mc2d rpert,
removed min values from inputs
#####

To have RStudio recognize this as a Shiny app,
create a (sub)directory for this app, and rename
it as app.R

Global
Load Packages
require(shiny)
require(ggplot2)
require(htmlTable)
require(mc2d)

#####
```

```
User Interface
ui = shinyUI(fluidPage(
 # Title
 titlePanel("Expert Opinion Risk Explorer"),

 sidebarLayout(
 # Sidebar Panel

 sidebarPanel(
 # Allow user to input betaPERT parameters

 # Enter lowest possible value
 numericInput(
 "low",
 label = "Enter Lowest Possible Value:",
 value = 0,
 step = 0.1
),

 # Enter most likely value (mode/maximum density)
 numericInput(
 "mode",
 label = "Enter Most Likely Value (Mode):",
 value = 10,
 step = 0.1
),

 # Enter highest possible value
 numericInput(
 "high",
 label = "Enter Highest Possible Value:",
 value = 25,
 step = 0.1
),

 # Produce empirical histogram/density plot
 numericInput(
 "bins",
 label = "Bin Width (0 for density only):",
 min = 0,
```

```
 value = 0.1,
 step = 0.1
),

User inputs for min/max percent adjustment
numericInput(
 "minperc",
 label = "Minimum adjustment (percent below minimum):",
 min = 0,
 max = 1,
 value = 0
),

numericInput(
 "maxperc",
 label = "Maximum adjustment (percent above maximum):",
 min = 0,
 max = 1,
 value = 0
),

Show betaPERT values
htmlOutput("maxdens"),

br(),

User inputs for simulation
numericInput(
 "reps",
 label = "Simulation Replications:",
 min = 1,
 value = 10000
),

sliderInput(
 "gamma",
 label = HTML(
 "BetaPERT Distribution Uncertainty Parameter (γ):

 <small><i>4 is default PERT</i></small>"
),
 min = 1,
```

```
max = 10,
value = 4,
step = 1
),

submitButton("Submit")

),

Main panel
mainPanel(

Simulation histogram/density plot
plotOutput(outputId = "dist_plot"),

Output of simulation cdf plot click

HTML(
 "
Click on the CDF line for a given x-axis value,
 then click <i>Submit</i> to obtain the probability estimate.
"
),

br(),

Simulation cdf plot
plotOutput(outputId = "cdf_plot", click = "plot_click"),

Click output results
htmlOutput("info")

)

)

#####
Server
server = shinyServer(function(input, output) {
```

```
Simulation distribution plot
output$dist_plot = renderPlot({
 Value = c(input$low, input$high, input$mode)
 df = data.frame(Value)

 sim_df = data.frame(Value = rpert(
 input$reps,
 (min(df$Value) - min(df$Value) * input$minperc),
 input$mode,
 (max(df$Value) + max(df$Value) * input$maxperc),
 input$gamma))

 ggplot(sim_df, aes(Value)) +
 ggtitle("Simulation Distribution") +
 ylab("Density / Count") +
 geom_histogram(
 aes(y = ..density..),
 binwidth = input$bins,
 col = "blue",
 fill = "blue",
 alpha = 0.2,
 na.rm = T) +
 xlim((min(df$Value) - min(df$Value) * input$minperc),
 (max(df$Value) + max(df$Value) * input$maxperc)) +
 geom_density(
 col = "blue",
 fill = "blue",
 alpha = 0.2,
 na.rm = T) +
 theme(
 axis.ticks.y = element_blank(),
 axis.text.y = element_blank(),
 panel.grid.major.y = element_blank(),
 panel.grid.minor.y = element_blank())
})

Simulation CDF plot
output$cdf_plot = renderPlot({
 Value = c(input$low, input$high, input$mode)
 df = data.frame(Value)
```

```
sim_df = data.frame(Value = rpert(
 input$reps,
 (min(df$Value) - min(df$Value) * input$minperc),
 input$mode,
 (max(df$Value) + max(df$Value) * input$maxperc),
 input$gamma
))

ggplot(sim_df, aes(Value)) +
 ggtitle("Simulation CDF") +
 ylab("Probability") +
 xlim((min(df$Value) - min(df$Value) * input$minperc),
 (max(df$Value) + max(df$Value) * input$maxperc)) +
 stat_ecdf(lwd = 2, na.rm = T) +
 theme(axis.ticks.y = element_blank(), axis.text.y = element_blank())

})

Plot-click result for simulation
output$info = renderText({
 paste0(
 "<i>The probability of obtaining a Value less than
 or equal to ",
 txtRound(input$plot_click$x, 1),
 " is about ",
 txtRound(input$plot_click$y, 2),
 ".</i>"
)
})

Details on betaPERT parameters
output$maxdens = renderText({
 Value = c(input$low, input$high, input$mode)
 df = data.frame(Value)

 sim_df = data.frame(Value = rpert(
 input$reps,
 (min(df$Value) - min(df$Value) * input$minperc),
 input$mode,
 (max(df$Value) + max(df$Value) * input$maxperc),
 input$gamma
```

```
)
```

```
paste0(
 "BetaPERT Inputs:
<i>Simulation mode: " ,
 txtRound(input$mode, 1),
 ".
Simulation minimum: " ,
 txtRound((min(df$value) - min(df$value) * input$minperc), 1),
 "
Simulation maximum: " ,
 txtRound(max(df$value) + max(df$value) * input$maxperc, 1),
 "</i>."
)
```

```
})
```

```
App
shinyApp(ui = ui, server = server)
```

# Appendix 4: .Rmd file for the *Chapter 9 flexdashboard example*

This code creates the flexdashboard shown in *Chapter 9*.

```

```

```
title: "Hauteville House Power Use"
output:
 flexdashboard::flex_dashboard:
 orientation: rows
 vertical_layout: fill
 source_code: embed

```

```
```{r setup, include=FALSE, cache=TRUE}
require(flexdashboard)
require(plotly)
require(zoo)
require(dplyr)
require(reshape2)
require(forecast)
require(ggplot2)
require(leaflet)
require(dygraphs)
require(taucharts)
require(DT)
```

```
# Read the data into R
#download.file("https://archive.ics.uci.edu/ml/machine-learning-databases/00235/\`household_power_consumption.zip", "household_power_consumption.zip")
# NAs are represented by blanks and ? in this data, so need to change to NA
power = read.table(unz("household_power_consumption.zip", "household_power_consumption.txt"), sep=";", header=T, na.strings=c("?", ""), stringsAsFactors=FALSE)

# Convert data to Date object
power$Date = as.Date(power$Date, format="%d/%m/%Y")

# Obtain the Month and Year for each data point
```

```
power$Month = format(power$Date, "%Y-%m")

# Add the first to each Y-m combo and convert back to Date
power$Month = as.Date(paste0(power$Month, "-01"))

# Use 'zoo' to perform interpolation for missing time series values
power$Global_active_power_locf = na.locf(power$Global_active_power)

# Use 'dplyr' to group by month
power_monthly = power %>%
  group_by(Month) %>%
  summarise(Max_Demand_kW = max(Global_active_power_locf),
    Total_Use_kWh = sum(Global_active_power_locf)/60)

# Remove partial months from data frame
power_monthly = power_monthly[2:47,]
````
```

## Overview

---

### Results { .sidebar}

---

#### #### Summary

Total monthly power use follows a predictable annual cycle, with winter peaks and summer troughs.

In coming months, usage will peak between ~950-1150 kWh (80% prediction **range**) in December and January, then steadily fall over the remainder of the winter and early spring.

<br>  
<hr>

#### #### Methods and Analytics Details

- Data: 1-minute time step, 16 December 2006 through 26 November 2010
- Analysis: 1-month time step, January 2007 through October 2010
- Forecast: 1-month time step, November 2010 - April 2011

```


Data Source and Metadata

Row {data-height=550}

Time Series and Forecast of Power Use (kWh)

```{r forecastplot}
# Create a time series object of Total (Monthly) Usage
total_use_ts = ts(power_monthly$Total_Use_kWh, start=c(2007,1), frequency=12)

# Automatically obtain the forecast for the next 6 months
total_use_fc = forecast(total_use_ts, h=6)

# Create a data frame with the original data
# and space for the forecast details
use_df = data.frame(Total_Use = power_monthly$Total_Use_kWh,
Forecast = NA, Upper_80 = NA, Lower_80 = NA)

# Create a data frame for the forecast details
# with a column for the original data
use_fc = data.frame(Total_Use = NA, Forecast = total_use_fc$mean, Upper_80 =
total_use_fc$upper[,1], Lower_80 = total_use_fc$lower[,1])

# "Union" the two data frames into one
use_ts_fc = rbind(use_df, use_fc)

# Create a time series of the data and forecast results
total_use_forecast = ts(use_ts_fc, start=c(2007, 1), freq=12)

# Create the forecasting widget
dygraph(total_use_forecast) %>%
  dyAxis("y", valueRange = c(100, 1400)) %>%
  dySeries(c("Total_Use"), label="Actual kWh Use") %>%
  dyEvent(x = "2008-08-01", "Closed for vacation", labelLoc = "top") %>%
  dyShading(from = "2008-07-15", to = "2008-08-15") %>%
  dyRangeSelector(dateWindow = c("2008-06-15", "2011-06-01")) %>%
  dyLegend(width = 800)
```

```

```
Row {data-height=450}
```

```
=====
```

```
Distribution of Monthly Total Use
```

```
```{r useplot}
use_histo = ggplot(power_monthly, aes(Total_Use_kWh, text=paste("Month: ",
  format(Month, "%B %Y"), "<br>Total kWh Use: ", Total_Use_kWh))) +
  geom_histogram(color="gray80", fill="steelblue", binwidth=50) +
  ylab("Number of months") +
  xlab("Total Use (kWh)") +
  theme_bw()
```

```
ggplotly(use_histo)
````
```

```
You Are Here
```

```
```{r map}
popup_content = paste(sep = '<br>',
  '<b><a href="http://www.visitguernsey.com/victor-hugo-house">
  Hauteville House</a></b>',
  ''
)

leaflet() %>%
  addTiles() %>%
  setView(-2.5376379, 49.4513514, zoom=13) %>%
  addMarkers(-2.5376379, 49.4513514, popup = popup_content)
````
```

```
Daily Summary
```

```
=====
```

```
```{r calendarplot}
source("http://blog.revolutionanalytics.com/downloads/calendarHeat.R")

# Use 'dplyr' to group by day, excluding 2006
power_daily = power %>%
  group_by(Date) %>%
```

```
summarise(Max_Demand_kW = max(Global_active_power_locf),
           Total_Use_kWh = sum(Global_active_power_locf)/60) %>%
filter(Date >= "2007-01-01")

# Plot the calendar graph to view the kWh use pattern
calendarHeat(power_daily$date, power_daily$Total_Use_kWh, color="r2b",
             varname="Total Daily Power Use (kWh)")
````
```

## Data

---

```
Results {.sidebar data-width=450}
```

---

### ### Forecast Details

```
Forecasting Model
`r total_use_fc$method`
```

```
Forecast Results
```

```
```{r forecasting}
round(data.frame(total_use_fc), 0)
````
```

## Row

---

```
```{r datatable}
power_monthly[,2:3] = round(power_monthly[,2:3], 0)
datatable(power_monthly, filter = "top")
````
```

# Appendix 5: R Markdown Quick Reference

## Text

|                                |                        |
|--------------------------------|------------------------|
| plain text                     | plain text             |
| <i>italicized text</i>         | *italicized text*      |
| <b>bold text</b>               | **bold text**          |
| <b><i>bold italic text</i></b> | ***bold italic text*** |
| superscript <sup>2</sup>       | <sup>2</sup>           |
| subscript <sub>2</sub>         | <sub>2</sub>           |
| en dash –                      | --                     |
| em dash —                      | ---                    |
| equation $E = mc^2$            | \$E = mc^2\$           |
| block quote (indented)         | > text for quote       |

## Links and Images

|             |                               |
|-------------|-------------------------------|
| weblink/url | [display text](completeURL)   |
| image       | ![caption](path/to/image.png) |

## Code

inline code

``r some_code_here``

block code

```
```{r}
some
code
here
```
```

## Lists

```
`* bullet point 1`
`* bullet point 2`
` + sub-bullet`
```

- bullet point 1
  - bullet point 2
    - sub-bullet
  - 1. number point 1
  - 2. number point 2
    - ‘ + sub-number’
1. number point 1
  2. number point 2
    - sub-number

## Tables

| Header | Header | Header |
|--------|--------|--------|
| Header | Header | Header |
| Cell   | Cell   | Cell   |
| Cell   | Cell   | Cell   |

| Header | Header | Header |
|--------|--------|--------|
| Cell   | Cell   | Cell   |
| Cell   | Cell   | Cell   |

## Section Formatting

|                     |                 |
|---------------------|-----------------|
| Top level header    | # Header 1      |
| Second level header | ## Header 2     |
| Third level header  | ### Header 3    |
| Fourth level header | #### Header 4   |
| Fifth level header  | ##### Header 5  |
| Sixth level header  | ###### Header 6 |

Create a horizontal rule: \*\*\* or <hr>

---

## ***Useful knitr Chunk Options***

Use `?opts_chunk` or see the [knitr options page<sup>92</sup>](#) for more options and details.

| Option                  | Default Value          | Action                                                                           | Recommended Use                                                                                                            |
|-------------------------|------------------------|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>background</code> | <code>"#F7F7F7"</code> | Background color for code portions output                                        | Leave as is.                                                                                                               |
| <code>cache</code>      | <code>FALSE</code>     | Cache results for future renders                                                 | Change to <code>T</code> for slow code and/or testing.                                                                     |
| <code>collapse</code>   | <code>FALSE</code>     | Show all output from code chunk together                                         | If you want to show code, modify as needed.                                                                                |
| <code>comment</code>    | <code>"##"</code>      | Comment character for results                                                    | Change to <code>" "</code> (i.e., nothing) to make output results look cleaner.                                            |
| <code>dpi</code>        | <code>72</code>        | Plot output resolution (dots per inch)                                           | Leave as is for html output; modify as needed for print output.                                                            |
| <code>echo</code>       | <code>TRUE</code>      | Show code in output                                                              | Leave as is for training, change to <code>F</code> for production.                                                         |
| <code>error</code>      | <code>FALSE</code>     | Show error messages in output                                                    | Leave as is.                                                                                                               |
| <code>eval</code>       | <code>TRUE</code>      | Run the code                                                                     | Leave as is, except when including placeholder code for future development.                                                |
| <code>fig.height</code> | <code>7</code>         | Output plot height in inches                                                     | Modify as needed.                                                                                                          |
| <code>fig.width</code>  | <code>7</code>         | Output plot width in inches                                                      | Modify as needed. Note: use <code>6.5</code> for output for US Letter paper with 1" margins.                               |
| <code>message</code>    | <code>TRUE</code>      | Show messages from code run in output                                            | Leave as is for training, change to <code>F</code> for production.                                                         |
| <code>out.extra</code>  | <code>NULL</code>      | Extra options for figures                                                        | Use as necessary. Example: <code>rotate 90</code><br><code>out.extra='angle=90'</code> .                                   |
| <code>out.width</code>  | <code>NULL</code>      | Plot width in final output file                                                  | Can use, e.g.,<br><code>0.8\\linewidth, 6.5in</code> , or<br><code>16.5cm</code> (LaTeX/pdf) or <code>656px</code> (html). |
| <code>out.height</code> | <code>NULL</code>      | Plot height in final output file                                                 | Can use, e.g.,<br><code>0.8\\linewidth, 6.5in</code> , or<br><code>16.5cm</code> (LaTeX/pdf) or <code>656px</code> (html). |
| <code>results</code>    | <code>"markup"</code>  | Use <code>"asis"</code> for html or table output, leave as is for other results. |                                                                                                                            |
| <code>split</code>      | <code>FALSE</code>     | Whether to split output into separate files.                                     | Modify if needed.                                                                                                          |

---

<sup>92</sup><http://yihui.name/knitr/options/>

| Option  | Default Value | Action                           | Recommended Use                                                      |
|---------|---------------|----------------------------------|----------------------------------------------------------------------|
| tidy    | FALSE         | Show tidied-up code in output    | Leave as is for production, change to F for training.                |
| warning | TRUE          | Show warning messages in output. | Leave as is for training or development, change to F for production. |