Practical Recipes for Visualizing Data

R Graphics Cookbook

O'REILLY®

Winston Chang

R Graphics Cookbook

The *R Graphics Cookbook* provides more than 150 recipes to help you generate high-quality graphs quickly, without having to comb through all the details of R's graphing systems. Each recipe tackles a specific problem with a solution you can apply to your own project, and includes a discussion of how and why the recipe works.

Most of the recipes use the ggplot2 package, a powerful and flexible way to make graphs in R. If you have a basic understanding of the R language, you're ready to get started with the wealth of options found in R for visualizing data.

- Use R's default graphics for quick exploration of data
- Create a variety of bar graphs, line graphs, and scatter plots
- Summarize data distributions with histograms, density curves, box plots, and other examples
- Provide annotations to help viewers interpret data
- Control the overall appearance of graphics
- Render data groups alongside each other for easy comparison
- Use colors in plots
- Create network graphs, heat maps, and 3D scatter plots
- Structure data for graphing

Winston Chang is a software engineer at RStudio, where he works on data visualization and software development tools for R. His "Cookbook for R" website contains recipes for common tasks in R.

"The R Graphics Cookbook shows you how to create the most common types of graphics, and then shows you how to tweak them to meet your needs. Not only is it a readable introduction to visualization in R, but the breadth of its examples also make it a rich source of inspiration."

—Hadley Wickham
Assistant Professor of Statistics at Rice University

US $39.99 CAN $41.99
ISBN: 978-1-449-31695-2

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

R Graphics Cookbook

Winston Chang

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

R Graphics Cookbook

by Winston Chang

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Courtney Nash
Production Editor: Holly Bauer
Copyeditor: Rachel Head

Proofreader: Jilly Gagnon
Indexer: Lucie Haskins
Cover Designer: Randall Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest and Robert Romano

December 2012: First Edition

Revision History for the First Edition:

2012-12-04 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449316952* for release details.

ISBN: 978-1-449-31695-2

[CK]

Table of Contents

Preface

I started using R several years ago to analyze data I had collected for my research in graduate school. My motivation at first was to escape from the restrictive environments and canned analyses offered by statistical programs like SPSS. And even better, because it's freely available, I didn't need to convince someone to buy me a copy of the software—very important for a poor graduate student! As I delved deeper into R, I discovered that it could also create excellent data graphics.

Each recipe in this book lists a problem and a solution. In most cases, the solutions I offer aren't the only way to do things in R, but they are, in my opinion, the best way. One of the reasons for R's popularity is that there are many available add-on packages, each of which provides some functionality for R. There are many packages for visualizing data in R, but this book primarily uses ggplot2. (Disclaimer: it's now part of my job to do development on ggplot2. However, I wrote much of this book before I had any idea that I would start a job related to ggplot2.)

This book isn't meant to be a comprehensive manual of all the different ways of creating data visualizations in R, but hopefully it will help you figure out how to make the graphics you have in mind. Or, if you're not sure what you want to make, browsing its pages may give you some ideas about what's possible.

Recipes

This book is intended for readers who have at least a basic understanding of R. The recipes in this book will show you how to do specific tasks. I've tried to use examples that are simple, so that you can understand how they work and transfer the solutions over to your own problems.

Software and Platform Notes

Most of the recipes here use the ggplot2 graphing package. Some of the recipes require the most recent version of ggplot2, 0.9.3, and this in turn requires a relatively recent version of R. You can always get the latest version of R from the main R project site.

If you are not familiar with ggplot2, see Appendix A for a brief introduction to the package.

Once you've installed R, you can install the necessary packages. In addition to ggplot2, you'll also want to install the gcookbook package, which contains data sets for many of the examples in this book. To install them both, run:

```
install.packages("ggplot2")
install.packages("gcookbook")
```

You may be asked to choose a mirror site for CRAN, the Comprehensive R Archive Network. Any of the sites should work, but it's a good idea to choose one close to you because it will likely be faster than one far away. Once you've installed the packages, run this in each R session in which you want to use ggplot2:

```
library(ggplot2)
```

The recipes in this book will assume that you've already loaded ggplot2, so they won't show this line.

If you see an error like this, it means that you forgot to load ggplot2:

```
Error: could not find function "ggplot"
```

The major platforms for R are Mac OS X, Linux, and Windows, and all the recipes in this book should work on all of these platforms. There are some platform-specific differences when it comes to creating bitmap output files, and these differences are covered in Chapter 14.

Conventions Used in This Book

The following typographical conventions are used in this book:

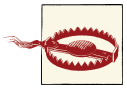
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*R Graphics Cookbook* by Winston Chang (O'Reilly). Copyright 2013 Winston Chang, 978-1-449-31695-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreil.ly/R_Graphics_Cookbook

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

No book is the product of a single person. There are many people who helped make this book possible, directly and indirectly. I'd like to thank the R community for creating R

and for fostering a dynamic ecosystem around it. Thanks to Hadley Wickham for creating the software that this book revolves around, for pointing O'Reilly in my direction when they were considering a book about R graphics, and for opening up many opportunities for me to deepen my knowledge of R.

Thanks to the technical reviewers for this book: Paul Teetor, Hadley Wickham, Dennis Murphy, and Erik Iverson. Their depth of knowledge and attention to detail has greatly improved this book. I'd like to thank the editors at O'Reilly who have shepherded this book along: Mike Loukides, for guiding me through the early stages, and Courtney Nash, for pulling me through to the end. I also owe a big thanks to Holly Bauer and the rest of the production team at O'Reilly, for putting up with many last-minute edits, and for handling the unusual features of this book.

Finally, I would like to thank my wife, Sylia, for her support and understanding—and not just with regard to the book.

CHAPTER 1
R Basics

This chapter covers the basics: installing and using packages and loading data.

If you want to get started quickly, most of the recipes in this book require the ggplot2 and gcookbook packages to be installed on your computer. To do this, run:

```
install.packages(c("ggplot2", "gcookbook"))
```

Then, in each R session, before running the examples in this book, you can load them with:

```
library(ggplot2)
library(gcookbook)
```

Appendix A provides an introduction to the ggplot2 graphing package, for readers who are not already familiar with its use.

Packages in R are collections of functions and/or data that are bundled up for easy distribution, and installing a package will extend the functionality of R on your computer. If an R user creates a package and thinks that it might be useful for others, that user can distribute it through a package repository. The primary repository for distributing R packages is called CRAN (the Comprehensive R Archive Network), but there are others, such as Bioconductor and Omegahat.

1.1. Installing a Package

Problem

You want to install a package from CRAN.

Solution

Use `install.packages()` and give it the name of the package you want to install. To install ggplot2, run:

```
install.packages("ggplot2")
```

At this point you may be prompted to select a download mirror. You can either choose the one nearest to you, or, if you want to make sure you have the most up-to-date version of your package, choose the Austria site, which is the primary CRAN server.

Discussion

When you tell R to install a package, it will automatically install any other packages that the first package depends on.

CRAN is a repository of packages for R, and it is mirrored on servers around the globe. It's the default repository system used by R. There are other package repositories; Bioconductor, for example, is a repository of packages related to analyzing genomic data.

1.2. Loading a Package

Problem

You want to load an installed package.

Solution

Use `library()` and give it the name of the package you want to install. To load ggplot2, run:

```
library(ggplot2)
```

The package must already be installed on the computer.

Discussion

Most of the recipes in this book require loading a package before running the code, either for the graphing capabilities (as in the ggplot2 package) or for example data sets (as in the MASS and gcookbook packages).

One of R's quirks is the package/library terminology. Although you use the `library()` function to load a package, a package is not a library, and some longtime R users will get irate if you call it that.

A *library* is a directory that contains a set of packages. You might, for example, have a system-wide library as well as a library for each user.

1.3. Loading a Delimited Text Data File

Problem

You want to load data from a delimited text file.

Solution

The most common way to read in a file is to use comma-separated values (CSV) data:

```
data <- read.csv("datafile.csv")
```

Discussion

Since data files have many different formats, there are many options for loading them. For example, if the data file does *not* have headers in the first row:

```
data <- read.csv("datafile.csv", header=FALSE)
```

The resulting data frame will have columns named `V1`, `V2`, and so on, and you will probably want to rename them manually:

```
# Manually assign the header names
names(data) <- c("Column1","Column2","Column3")
```

You can set the delimiter with `sep`. If it is space-delimited, use `sep=" "`. If it is tab-delimited, use `\t`, as in:

```
data <- read.csv("datafile.csv", sep="\t")
```

By default, strings in the data are treated as factors. Suppose this is your data file, and you read it in using `read.csv()`:

```
"First","Last","Sex","Number"
"Currer","Bell","F",2
"Dr.","Seuss","M",49
"","Student",NA,21
```

The resulting data frame will store `First` and `Last` as *factors*, though it makes more sense in this case to treat them as strings (or *characters* in R terminology). To differentiate this, set `stringsAsFactors=FALSE`. If there are any columns that should be treated as factors, you can then convert them individually:

```
data <- read.csv("datafile.csv", stringsAsFactors=FALSE)

# Convert to factor
data$Sex <- factor(data$Sex)

str(data)

'data.frame':   3 obs. of  4 variables:
```

```
$ First : chr  "Currer" "Dr." ""
$ Last  : chr  "Bell" "Seuss" "Student"
$ Sex   : Factor w/ 2 levels "F","M": 1 2 NA
$ Number: int  2 49 21
```

Alternatively, you could load the file with strings as factors, and then convert individual columns from factors to characters.

See Also

`read.csv()` is a convenience wrapper function around `read.table()`. If you need more control over the input, see `?read.table`.

1.4. Loading Data from an Excel File

Problem

You want to load data from an Excel file.

Solution

The xlsx package has the function `read.xlsx()` for reading Excel files. This will read the first sheet of an Excel spreadsheet:

```
# Only need to install once
install.packages("xlsx")

library(xslx)
data <- read.xlsx("datafile.xlsx", 1)
```

For reading older Excel files in the *.xls* format, the gdata package has the function `read.xls()`:

```
# Only need to install once
install.packages("gdata")

library(gdata)
# Read first sheet
data <- read.xls("datafile.xls")
```

Discussion

With `read.xlsx()`, you can load from other sheets by specifying a number for `sheetIndex` or a name for `sheetName`:

```
data <- read.xlsx("datafile.xls", sheetIndex=2)

data <- read.xlsx("datafile.xls", sheetName="Revenues")
```

With `read.xls()`, you can load from other sheets by specifying a number for `sheet`:

```
data <- read.xls("datafile.xls", sheet=2)
```

Both the xlsx and gdata packages require other software to be installed on your computer. For xlsx, you need to install Java on your machine. For gdata, you need Perl, which comes as standard on Linux and Mac OS X, but not Windows. On Windows, you'll need ActiveState Perl. The Community Edition can be obtained for free.

If you don't want to mess with installing this stuff, a simpler alternative is to open the file in Excel and save it as a standard format, such as CSV.

See Also

See `?read.xls` and `?read.xlsx` for more options controlling the reading of these files.

1.5. Loading Data from an SPSS File

Problem

You want to load data from an SPSS file.

Solution

The foreign package has the function `read.spss()` for reading SPSS files. To load data from the first sheet of an SPSS file:

```
# Only need to install the first time
install.packages("foreign")

library(foreign)
data <- read.spss("datafile.sav")
```

Discussion

The foreign package also includes functions to load from other formats, including:

- `read.octave()`: Octave and MATLAB
- `read.systat()`: SYSTAT
- `read.xport()`: SAS XPORT
- `read.dta()`: Stata

See Also

See `ls("package:foreign")` for a full list of functions in the package.

CHAPTER 2

Quickly Exploring Data

Although I've used the ggplot2 package for most of the graphics in this book, it is not the only way to make graphs. For very quick exploration of data, it's sometimes useful to use the plotting functions in base R. These are installed by default with R and do not require any additional packages to be installed. They're quick to type, are straightforward to use in simple cases, and run very quickly.

If you want to do anything beyond very simple graphs, though, it's generally better to switch to ggplot2. This is in part because ggplot2 provides a unified interface and set of options, instead of the grab bag of modifiers and special cases required in base graphics. Once you learn how ggplot2 works, you can use that knowledge for everything from scatter plots and histograms to violin plots and maps.

Each recipe in this section shows how to make a graph with base graphics. Each recipe also shows how to make a similar graph with the `qplot()` function in ggplot2, which has a syntax similar to the base graphics functions. For each `qplot()` graph, there is also an equivalent using the more powerful `ggplot()` function.

If you already know how to use base graphics, having these examples side by side will help you transition to using ggplot2 for when you want to make more sophisticated graphics.

2.1. Creating a Scatter Plot

Problem

You want to create a scatter plot.

Solution

To make a scatter plot (Figure 2-1), use `plot()` and pass it a vector of *x* values followed by a vector of *y* values:

```
plot(mtcars$wt, mtcars$mpg)
```

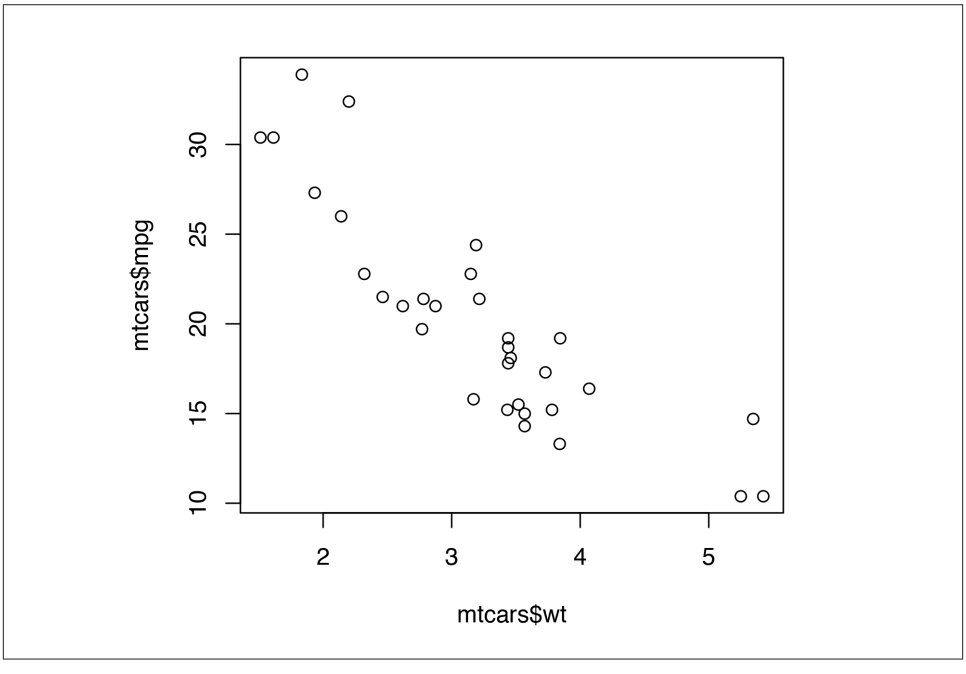


Figure 2-1. Scatter plot with base graphics

With the ggplot2 package, you can get a similar result using `qplot()` (Figure 2-2):

```
library(ggplot2)
qplot(mtcars$wt, mtcars$mpg)
```

If the two vectors are already in the same data frame, you can use the following syntax:

```
qplot(wt, mpg, data=mtcars)
# This is equivalent to:
ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point()
```

See Also

See Chapter 5 for more in-depth information about creating scatter plots.

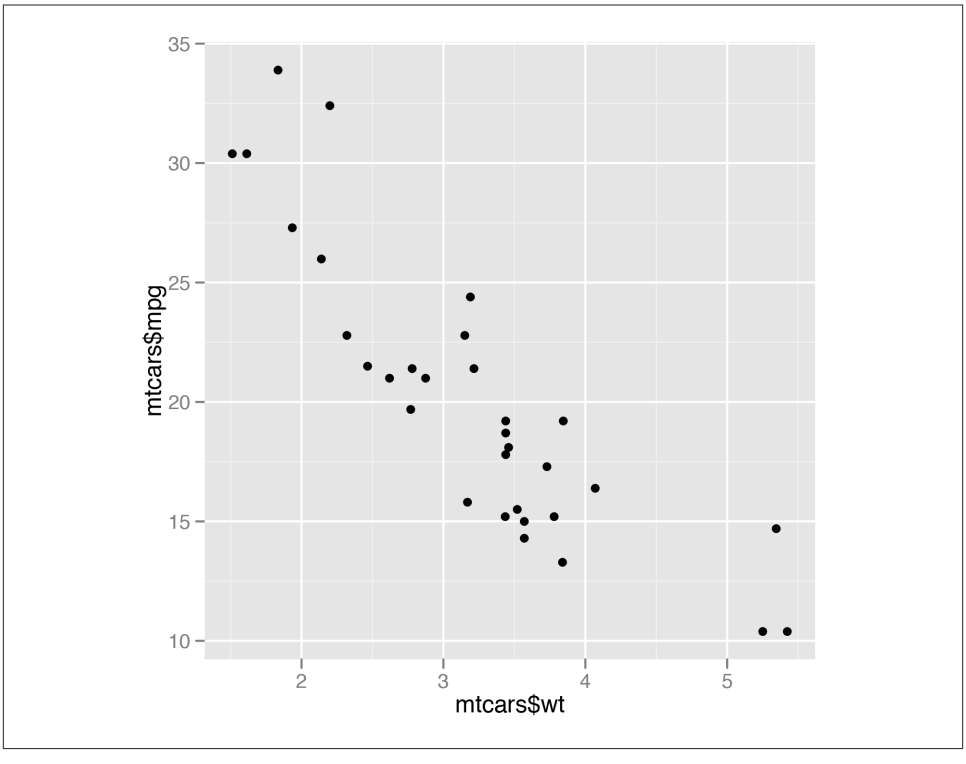


Figure 2-2. Scatter plot with qplot() from ggplot2

2.2. Creating a Line Graph

Problem

You want to create a line graph.

Solution

To make a line graph using `plot()` (Figure 2-3, left), pass it a vector of *x* values and a vector of *y* values, and use `type="l"`:

```
plot(pressure$temperature, pressure$pressure, type="l")
```

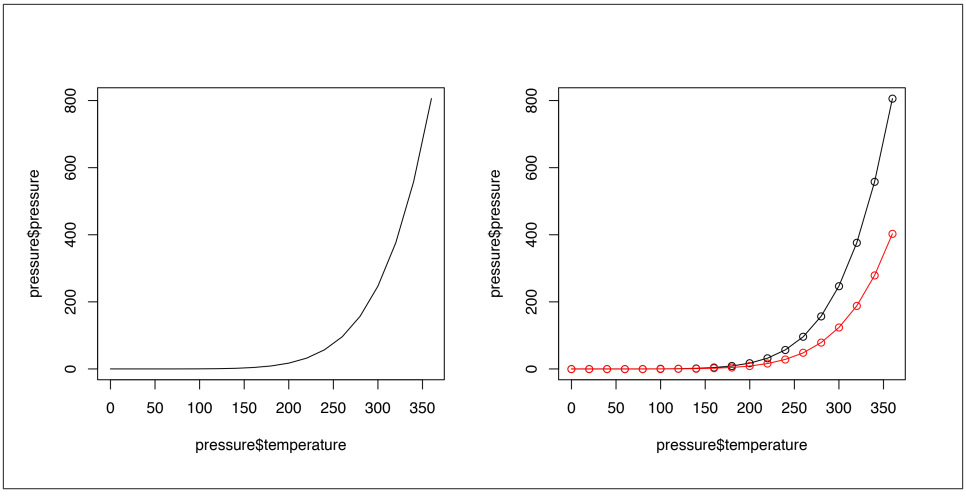


Figure 2-3. Left: line graph with base graphics; right: with points and another line

To add points and/or multiple lines (Figure 2-3, right), first call `plot()` for the first line, then add points with `points()` and additional lines with `lines()`:

```
plot(pressure$temperature, pressure$pressure, type="l")
points(pressure$temperature, pressure$pressure)

lines(pressure$temperature, pressure$pressure/2, col="red")
points(pressure$temperature, pressure$pressure/2, col="red")
```

With ggplot2, you can get a similar result using `qplot()` with `geom="line"` (Figure 2-4):

```
library(ggplot2)
qplot(pressure$temperature, pressure$pressure, geom="line")
```

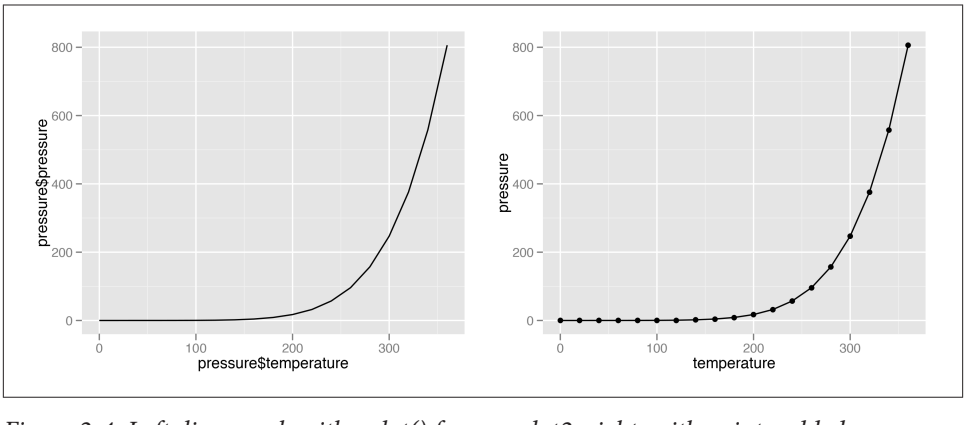


Figure 2-4. Left: line graph with qplot() from ggplot2; right: with points added

If the two vectors are already in the same data frame, you can use the following syntax:

```
qplot(temperature, pressure, data=pressure, geom="line")
# This is equivalent to:
ggplot(pressure, aes(x=temperature, y=pressure)) + geom_line()


# Lines and points together
qplot(temperature, pressure, data=pressure, geom=c("line", "point"))
# Equivalent to:
ggplot(pressure, aes(x=temperature, y=pressure)) + geom_line() + geom_point()
```

See Also

See Chapter 4 for more in-depth information about creating line graphs.

2.3. Creating a Bar Graph

Problem

You want to make a bar graph.

Solution

To make a bar graph of values (Figure 2-5), use `barplot()` and pass it a vector of values for the height of each bar and (optionally) a vector of labels for each bar. If the vector has names for the elements, the names will automatically be used as labels:

```
barplot(BOD$demand, names.arg=BOD$Time)
```

Sometimes "bar graph" refers to a graph where the bars represent the *count* of cases in each category. This is similar to a histogram, but with a discrete instead of continuous x-axis. To generate the count of each unique value in a vector, use the `table()` function:

```
table(mtcars$cyl)
```

```
 4  6  8
11  7 14
# There are 11 cases of the value 4, 7 cases of 6, and 14 cases of 8
```

Simply pass the table to `barplot()` to generate the graph of counts:

```
# Generate a table of counts
barplot(table(mtcars$cyl))
```

With the ggplot2 package, you can get a similar result using `qplot()` (Figure 2-6). To plot a bar graph of *values*, use `geom="bar"` and `stat="identity"`. Notice the difference in the output when the *x* variable is continuous and when it is discrete:

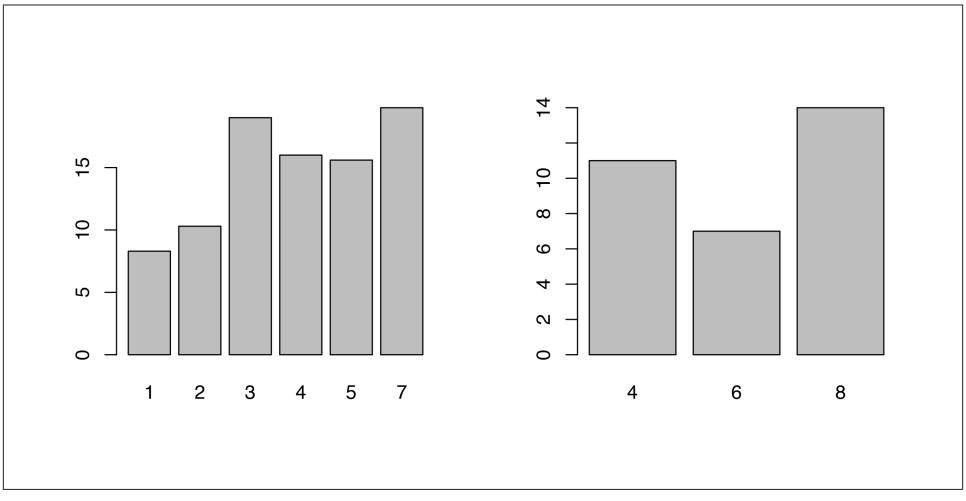


Figure 2-5. Left: bar graph of values with base graphics; right: bar graph of counts

```
library(ggplot2)
qplot(BOD$Time, BOD$demand, geom="bar", stat="identity")

# Convert the x variable to a factor, so that it is treated as discrete
qplot(factor(BOD$Time), BOD$demand, geom="bar", stat="identity")
```

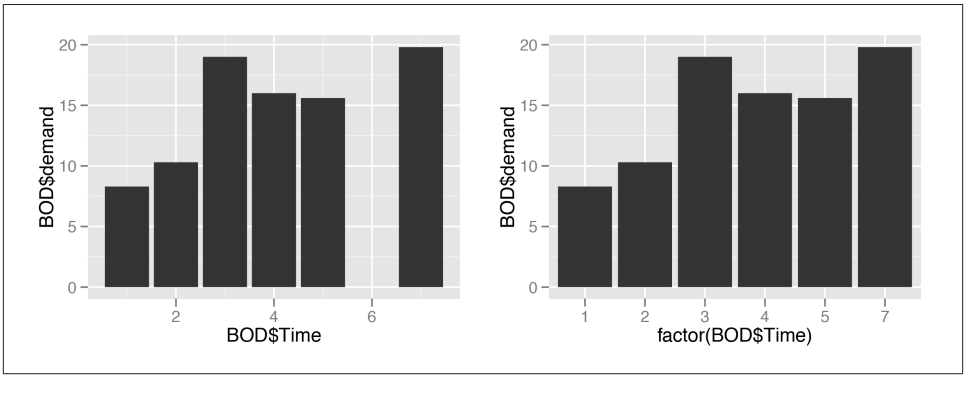


Figure 2-6. Left: bar graph of values with qplot() with continuous x variable; right: with x variable converted to a factor (notice that there is no entry for 6)

`qplot()` can also be used to graph the *counts* in each category (Figure 2-7). This is in fact the default way that ggplot2 creates bar graphs, and requires less typing than a bar graph of values. Once again, notice the difference between a continuous x-axis and a discrete one.

```
# cyl is continuous here
qplot(mtcars$cyl)

# Treat cyl as discrete
qplot(factor(mtcars$cyl))
```

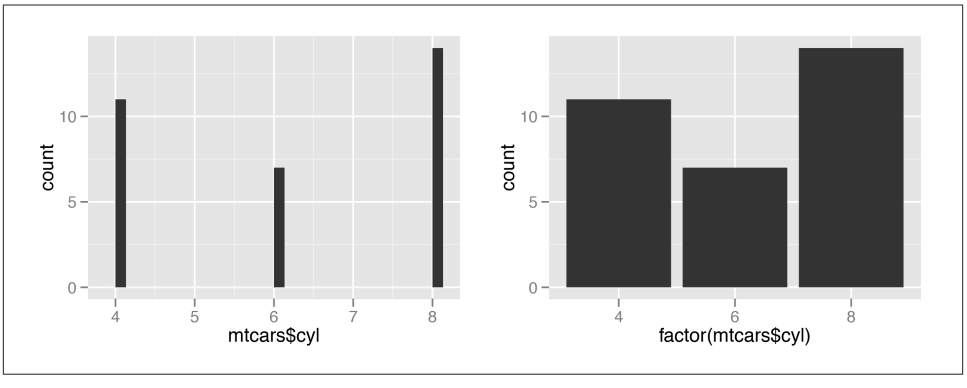


Figure 2-7. Left: bar graph of counts with qplot() with continuous x variable; right: with x variable converted to a factor

If the vector is in a data frame, you can use the following syntax:

```
# Bar graph of values. This uses the BOD data frame, with the
#"Time" column for x values and the "demand" column for y values.
qplot(Time, demand, data=BOD, geom="bar", stat="identity")
# This is equivalent to:
ggplot(BOD, aes(x=Time, y=demand)) + geom_bar(stat="identity")


# Bar graph of counts
qplot(factor(cyl), data=mtcars)
# This is equivalent to:
ggplot(mtcars, aes(x=factor(cyl))) + geom_bar()
```

See Also

See Chapter 3 for more in-depth information about creating bar graphs.

2.4. Creating a Histogram

Problem

You want to view the distribution of one-dimensional data with a histogram.

Solution

To make a histogram (Figure 2-8), use `hist()` and pass it a vector of values:

```
hist(mtcars$mpg)

# Specify approximate number of bins with breaks
hist(mtcars$mpg, breaks=10)
```

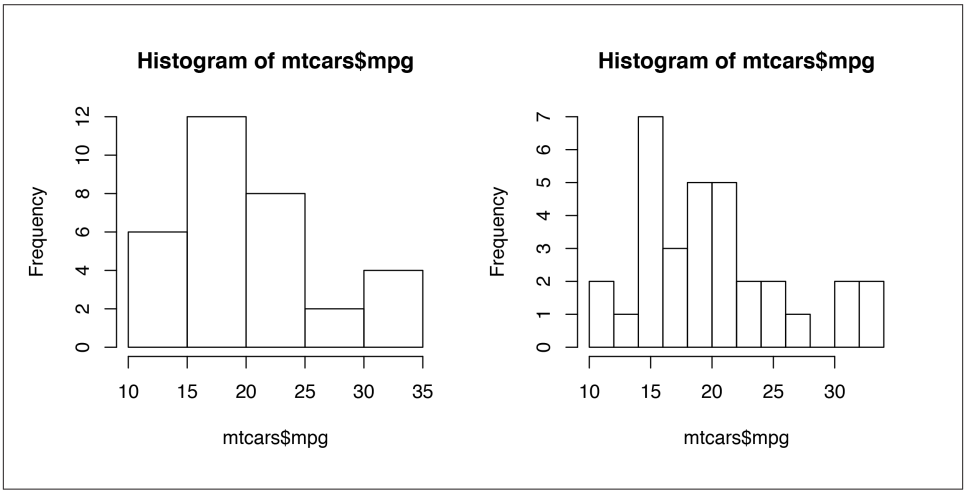


Figure 2-8. Left: histogram with base graphics; right: with more bins. Notice that because the bins are narrower, there are fewer items in each bin.

With the ggplot2 package, you can get a similar result using `qplot()` (Figure 2-9):

```
qplot(mtcars$mpg)
```

If the vector is in a data frame, you can use the following syntax:

```
library(ggplot2)
qplot(mpg, data=mtcars, binwidth=4)
# This is equivalent to:
ggplot(mtcars, aes(x=mpg)) + geom_histogram(binwidth=4)
```

See Also

For more in-depth information about creating histograms, see Recipes 6.1 and 6.2.

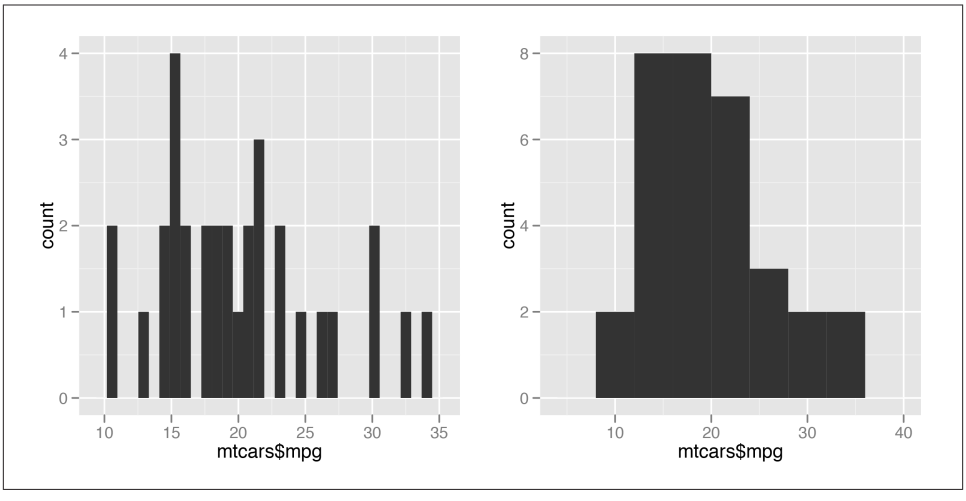


Figure 2-9. Left: histogram with qplot() from ggplot2, with default bin width; right: with wider bins

2.5. Creating a Box Plot

Problem

You want to create a box plot for comparing distributions.

Solution

To make a box plot (Figure 2-10), use `plot()` and pass it a factor of *x* values and a vector of *y* values. When *x* is a factor (as opposed to a numeric vector), it will automatically create a box plot:

```
plot(ToothGrowth$supp, ToothGrowth$len)
```

If the two vectors are in the same data frame, you can also use formula syntax. With this syntax, you can combine two variables on the x-axis, as in Figure 2-10:

```
# Formula syntax
boxplot(len ~ supp, data = ToothGrowth)

# Put interaction of two variables on x-axis
boxplot(len ~ supp + dose, data = ToothGrowth)
```

With the ggplot2 package, you can get a similar result using `qplot()` (Figure 2-11), with `geom="boxplot"`:

```
library(ggplot2)
qplot(ToothGrowth$supp, ToothGrowth$len, geom="boxplot")
```

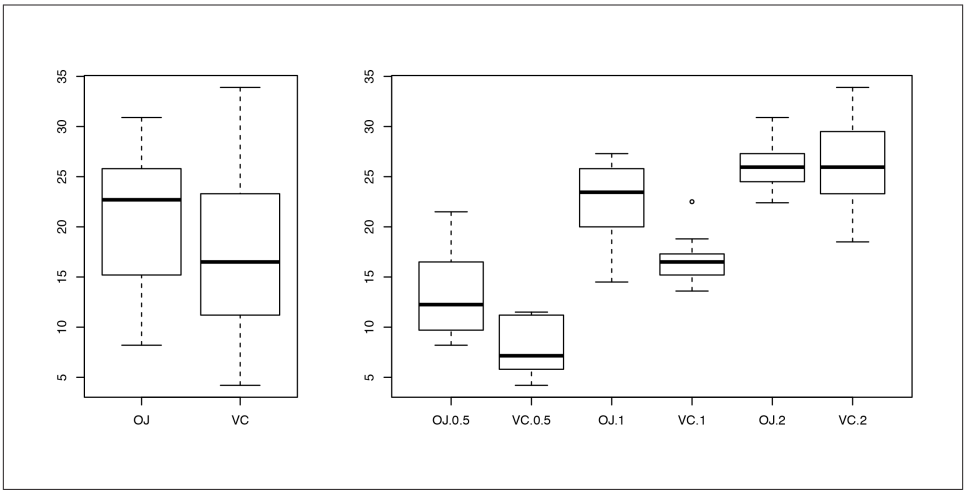


Figure 2-10. Left: box plot with base graphics; right: with multiple grouping variables

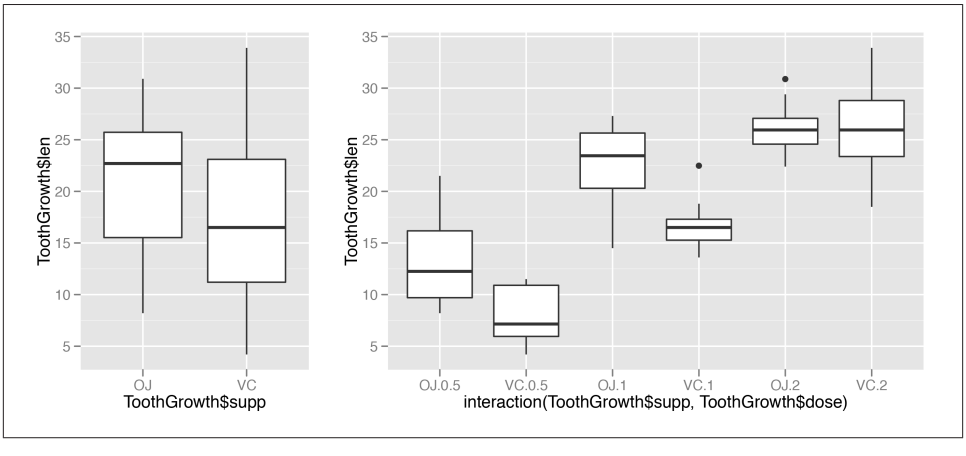


Figure 2-11. Left: box plot with qplot(); right: with multiple grouping variables

If the two vectors are already in the same data frame, you can use the following syntax:

```
qplot(supp, len, data=ToothGrowth, geom="boxplot")
# This is equivalent to:
ggplot(ToothGrowth, aes(x=supp, y=len)) + geom_boxplot()
```

It's also possible to make box plots for multiple variables, by combining the variables with `interaction()`, as in Figure 2-11. In this case, the `dose` variable is numeric, so we must convert it to a factor to use it as a grouping variable:

```
# Using three separate vectors
qplot(interaction(ToothGrowth$supp, ToothGrowth$dose), ToothGrowth$len,
      geom="boxplot")

# Alternatively, get the columns from the data frame
qplot(interaction(supp, dose), len, data=ToothGrowth, geom="boxplot")
# This is equivalent to:
ggplot(ToothGrowth, aes(x=interaction(supp, dose), y=len)) + geom_boxplot()
```

You may have noticed that the box plots from base graphics are ever-so-slightly different from those from ggplot2. This is because they use slightly different methods for calculating quantiles. See `?geom_boxplot` and `?boxplot.stats` for more information on how they differ.

See Also

For more on making basic box plots, see Recipe 6.6.

2.6. Plotting a Function Curve

Problem

You want to plot a function curve.

Solution

To plot a function curve, as in Figure 2-12, use `curve()` and pass it an expression with the variable `x`:

```
curve(x^3 - 5*x, from=-4, to=4)
```

You can plot any function that takes a numeric vector as input and returns a numeric vector, including functions that you define yourself. Using `add=TRUE` will add a curve to the previously created plot:

```
# Plot a user-defined function
myfun <- function(xvar) {
    1/(1 + exp(-xvar + 10))
}
curve(myfun(x), from=0, to=20)
# Add a line:
curve(1-myfun(x), add = TRUE, col = "red")
```

With the ggplot2 package, you can get a similar result using `qplot()` (Figure 2-13), by using `stat="function"` and `geom="line"` and passing it a function that takes a numeric vector as input and returns a numeric vector:

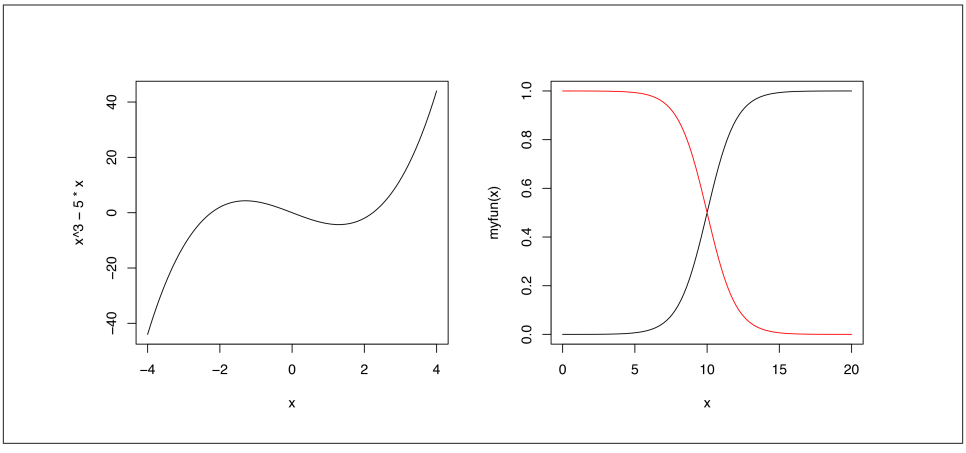


Figure 2-12. Left: function curve with base graphics; right: with user-defined function

```
library(ggplot2)
# This sets the x range from 0 to 20
qplot(c(0,20), fun=myfun, stat="function", geom="line")
# This is equivalent to:
ggplot(data.frame(x=c(0, 20)), aes(x=x)) + stat_function(fun=myfun, geom="line")
```

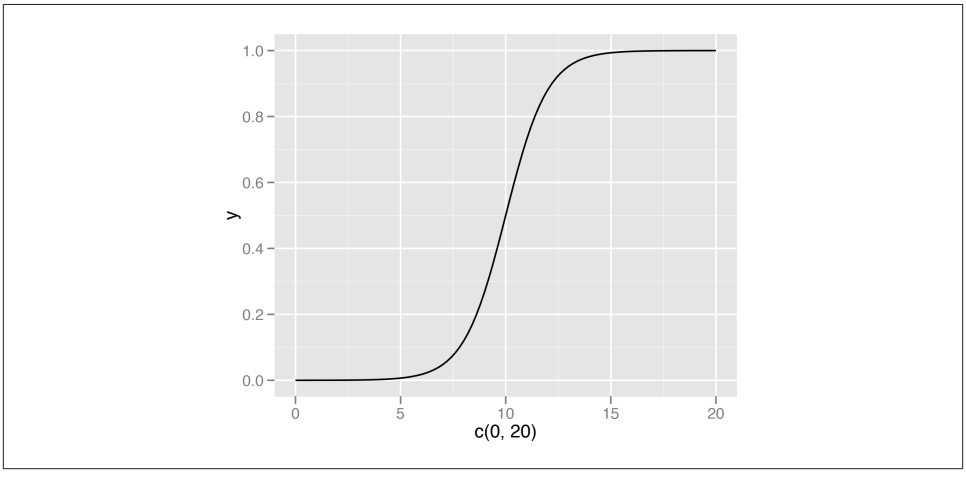


Figure 2-13. A function curve with qplot()

See Also

See Recipe 13.2 for more in-depth information about plotting function curves.

CHAPTER 3

Bar Graphs

Bar graphs are perhaps the most commonly used kind of data visualization. They're typically used to display numeric values (on the y-axis), for different categories (on the x-axis). For example, a bar graph would be good for showing the prices of four different kinds of items. A bar graph generally wouldn't be as good for showing prices over time, where time is a continuous variable—though it can be done, as we'll see in this chapter.

There's an important distinction you should be aware of when making bar graphs: sometimes the bar heights represent *counts* of cases in the data set, and sometimes they represent *values* in the data set. Keep this distinction in mind—it can be a source of confusion since they have very different relationships to the data, but the same term is used for both of them. In this chapter I'll discuss this more, and present recipes for both types of bar graphs.

3.1. Making a Basic Bar Graph

Problem

You have a data frame where one column represents the *x* position of each bar, and another column represents the vertical (y) height of each bar.

Solution

Use `ggplot()` with `geom_bar(stat="identity")` and specify what variables you want on the x- and y-axes (Figure 3-1):

```
library(gcookbook) # For the data set
ggplot(pg_mean, aes(x=group, y=weight)) + geom_bar(stat="identity")
```

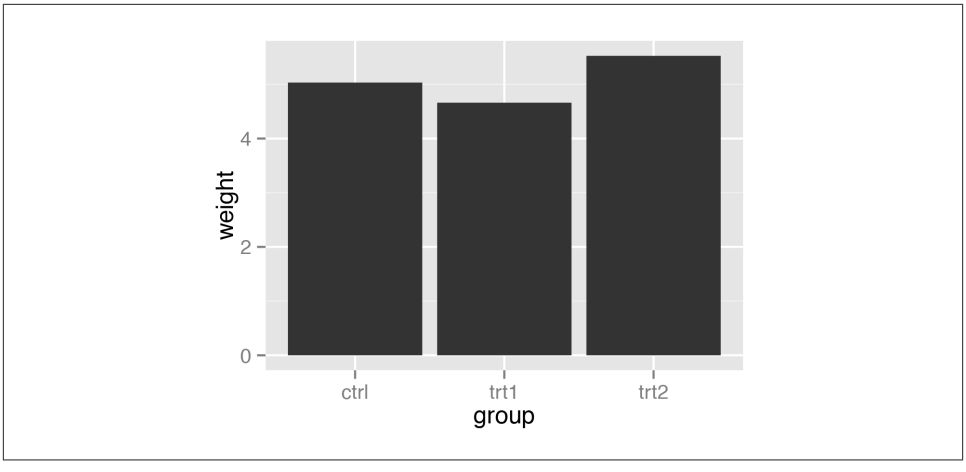


Figure 3-1. Bar graph of values (with stat="identity") with a discrete x-axis

Discussion

When x is a continuous (or numeric) variable, the bars behave a little differently. Instead of having one bar at each actual x value, there is one bar at each possible x value between the minimum and the maximum, as in Figure 3-2. You can convert the continuous variable to a discrete variable by using `factor()`:

```
# There's no entry for Time == 6
BOD
```

```
 Time demand
    1    8.3
    2   10.3
    3   19.0
    4   16.0
    5   15.6
    7   19.8
```

```
# Time is numeric (continuous)
str(BOD)
```

```
'data.frame':   6 obs. of  2 variables:
 $ Time  : num  1 2 3 4 5 7
 $ demand: num  8.3 10.3 19 16 15.6 19.8
 - attr(*, "reference")= chr "A1.4, p. 270"
```

```
ggplot(BOD, aes(x=Time, y=demand)) + geom_bar(stat="identity")

# Convert Time to a discrete (categorical) variable with factor()
ggplot(BOD, aes(x=factor(Time), y=demand)) + geom_bar(stat="identity")
```

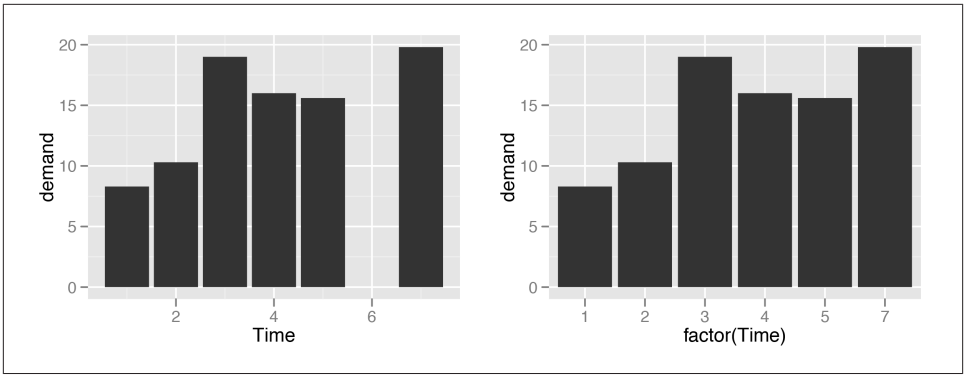


Figure 3-2. Left: bar graph of values (with stat="identity") with a continuous x-axis; right: with x variable converted to a factor (notice that the space for 6 is gone)

In these examples, the data has a column for `x` values and another for `y` values. If you instead want the height of the bars to represent the *count* of cases in each group, see Recipe 3.3.

By default, bar graphs use a very dark grey for the bars. To use a color fill, use `fill`. Also, by default, there is no outline around the fill. To add an outline, use `colour`. For Figure 3-3, we use a light blue fill and a black outline:

```
ggplot(pg_mean, aes(x=group, y=weight)) +
    geom_bar(stat="identity", fill="lightblue", colour="black")
```

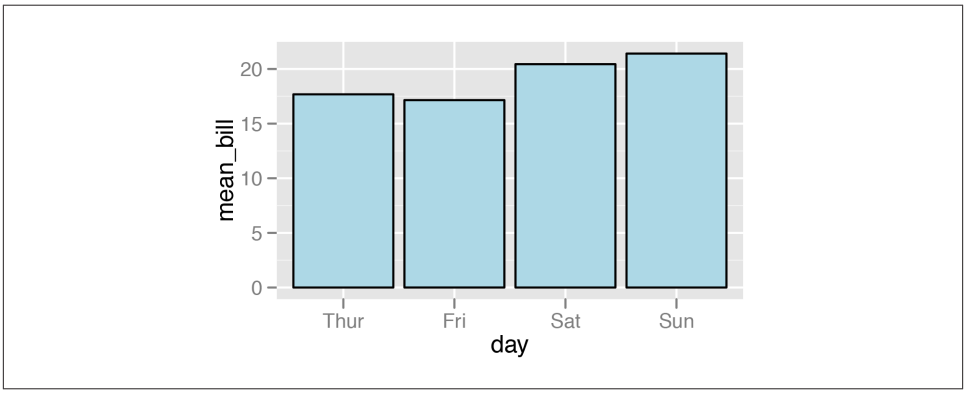


Figure 3-3. A single fill and outline color for all bars

In ggplot2, the default is to use the British spelling, `colour`, instead of the American spelling, `color`. Internally, American spellings are remapped to the British ones, so if you use the American spelling it will still work.

See Also

If you want the height of the bars to represent the count of cases in each group, see Recipe 3.3.

To reorder the levels of a factor based on the values of another variable, see Recipe 15.9. To manually change the order of factor levels, see Recipe 15.8.

For more information about using colors, see Chapter 12.

3.2. Grouping Bars Together

Problem

You want to group bars together by a second variable.

Solution

Map a variable to `fill`, and use `geom_bar(position="dodge")`.

In this example we'll use the `cabbage_exp` data set, which has two categorical variables, `Cultivar` and `Date`, and one continuous variable, `Weight`:

```
library(gcookbook) # For the data set
cabbage_exp
```

```
 Cultivar Date Weight
      c39  d16   3.18
      c39  d20   2.80
      c39  d21   2.74
      c52  d16   2.26
      c52  d20   3.11
      c52  d21   1.47
```

We'll map `Date` to the *x* position and map `Cultivar` to the fill color (Figure 3-4):

```
ggplot(cabbage_exp, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(position="dodge")
```

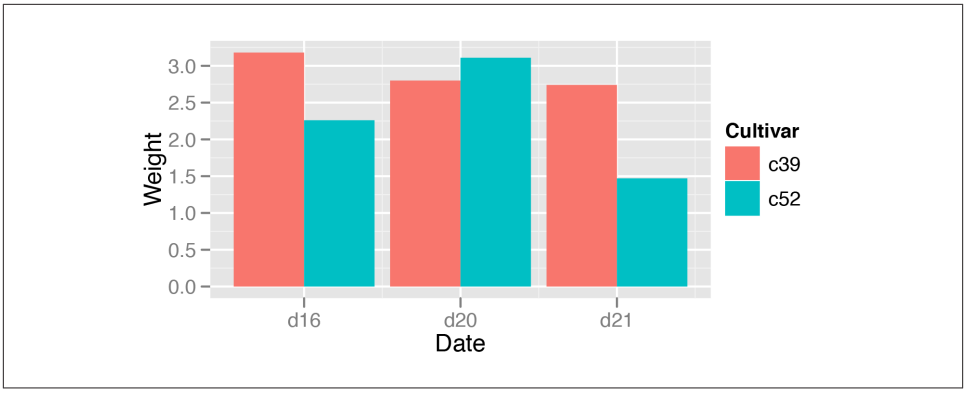


Figure 3-4. Graph with grouped bars

Discussion

The most basic bar graphs have one categorical variable on the x-axis and one continuous variable on the y-axis. Sometimes you'll want to use another categorical variable to divide up the data, in addition to the variable on the x-axis. You can produce a grouped bar plot by mapping that variable to `fill`, which represents the fill color of the bars. You must also use `position="dodge"`, which tells the bars to "dodge" each other horizontally; if you don't, you'll end up with a stacked bar plot (Recipe 3.7).

As with variables mapped to the x-axis of a bar graph, variables that are mapped to the fill color of bars must be categorical rather than continuous variables.

To add a black outline, use `colour="black"` inside `geom_bar()`. To set the colors, you can use `scale_fill_brewer()` or `scale_fill_manual()`. In Figure 3-5 we'll use the `Pastel1` palette from `RColorBrewer`:

```
ggplot(cabbage_exp, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(position="dodge", colour="black") +
    scale_fill_brewer(palette="Pastel1")
```

Other aesthetics, such as `colour` (the color of the outlines of the bars) or `linestyle`, can also be used for grouping variables, but `fill` is probably what you'll want to use.

Note that if there are any missing combinations of the categorical variables, that bar will be missing, and the neighboring bars will expand to fill that space. If we remove the last row from our example data frame, we get Figure 3-6:

```
ce <- cabbage_exp[1:5, ]   # Copy the data without last row
ce
```

```
 Cultivar Date Weight
      c39  d16   3.18
      c39  d20   2.80
```

```
c39  d21   2.74
c52  d16   2.26
c52  d20   3.11
```

```
ggplot(ce, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(position="dodge", colour="black") +
    scale_fill_brewer(palette="Pastel1")
```

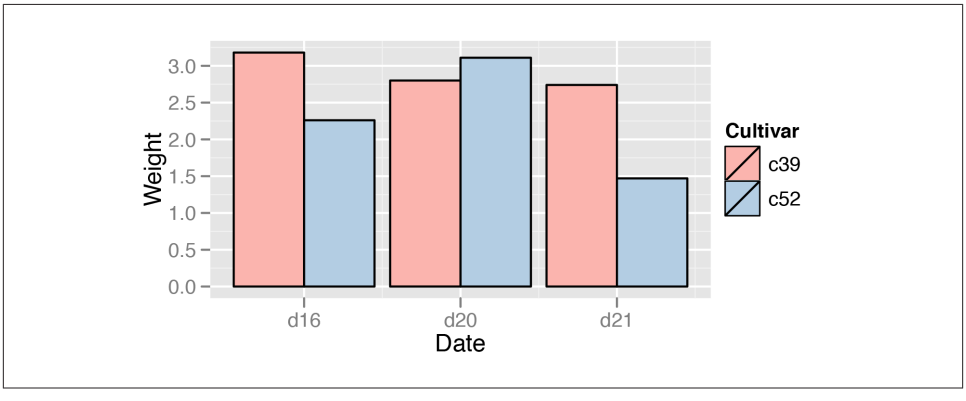


Figure 3-5. Grouped bars with black outline and a different color palette

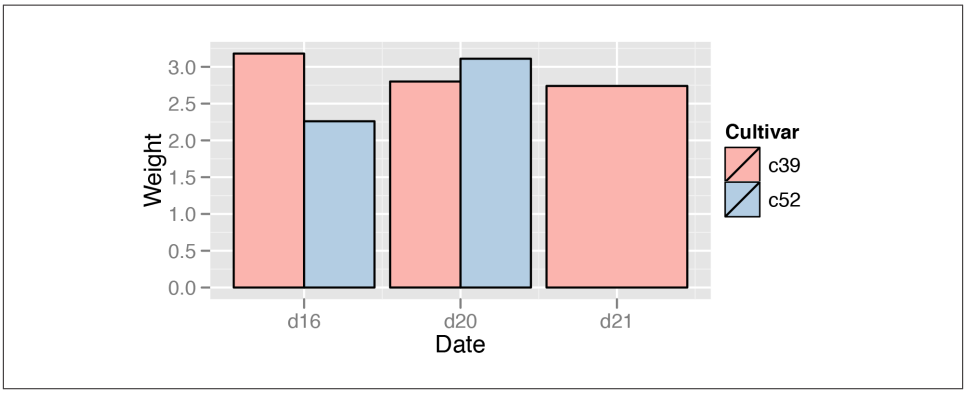


Figure 3-6. Graph with a missing bar—the other bar fills the space

If your data has this issue, you can manually make an entry for the missing factor level combination with an NA for the *y* variable.

See Also

For more on using colors in bar graphs, see Recipe 3.4.

To reorder the levels of a factor based on the values of another variable, see Recipe 15.9.

3.3. Making a Bar Graph of Counts

Problem

Your data has one row representing each case, and you want plot counts of the cases.

Solution

Use `geom_bar()` without mapping anything to `y` (Figure 3-7):

```
ggplot(diamonds, aes(x=cut)) + geom_bar()
# Equivalent to using geom_bar(stat="bin")
```

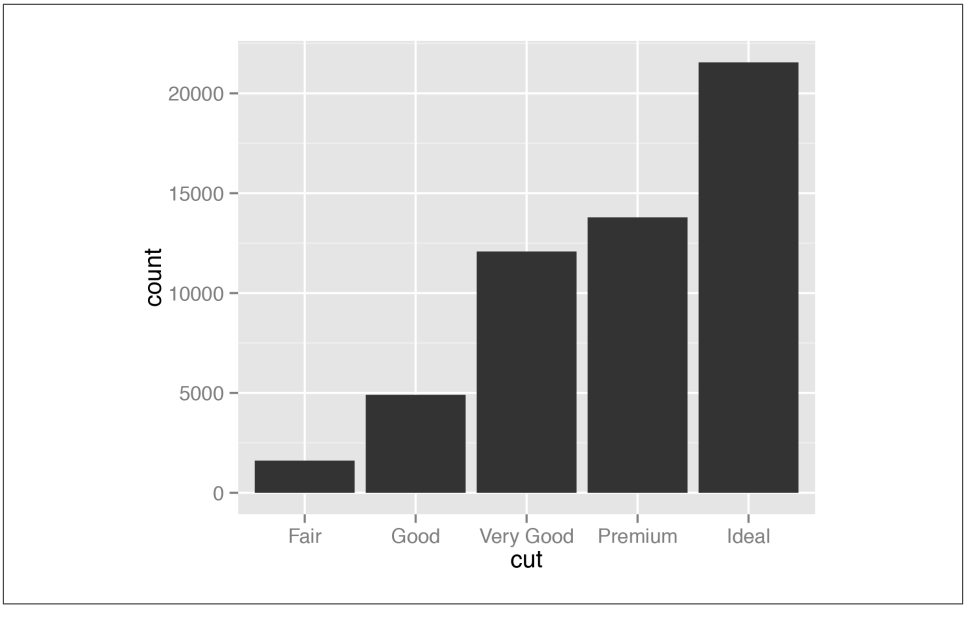


Figure 3-7. Bar graph of counts

Discussion

The `diamonds` data set has 53,940 rows, each of which represents information about one diamond:

```
diamonds
```

```
      carat       cut color clarity depth table price    x    y    z
1      0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
```

```
2      0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
3      0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
 ...
53939  0.86   Premium     H     SI2  61.0    58  2757 6.15 6.12 3.74
53940  0.75     Ideal     D     SI2  62.2    55  2757 5.83 5.87 3.64
```

With `geom_bar()`, the default behavior is to use `stat="bin"`, which counts up the number of cases for each group (each *x* position, in this example). In the graph we can see that there are about 23,000 cases with an ideal cut.

In this example, the variable on the x-axis is discrete. If we use a continuous variable on the x-axis, we'll get a histogram, as shown in Figure 3-8:

```
ggplot(diamonds, aes(x=carat)) + geom_bar()
```

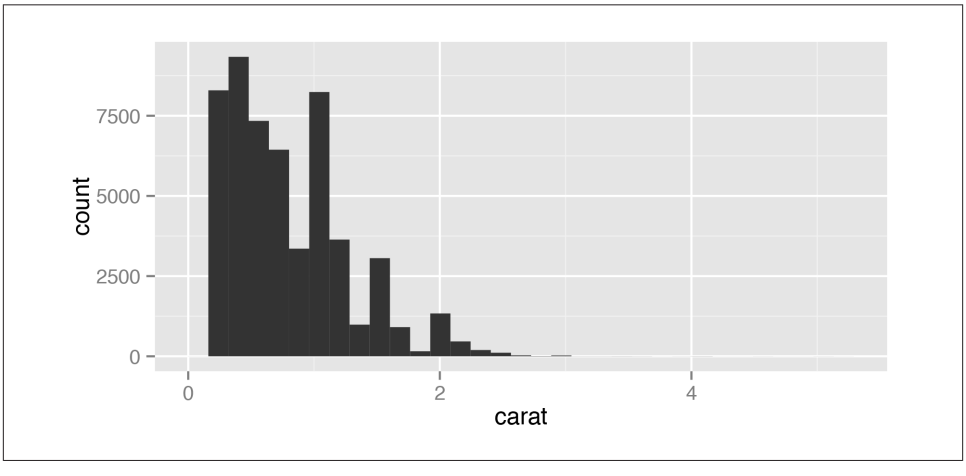


Figure 3-8. Bar graph of counts on a continuous axis, also known as a histogram

It turns out that in this case, the result is the same as if we had used `geom_histogram()` instead of `geom_bar()`.

See Also

If, instead of having `ggplot()` count up the number of rows in each group, you have a column in your data frame representing the `y` values, see Recipe 3.1.

You could also get the same graphical output by calculating the counts before sending the data to `ggplot()`. See Recipe 15.17 for more on summarizing data.

For more about histograms, see Recipe 6.1.

3.4. Using Colors in a Bar Graph

Problem

You want to use different colors for the bars in your graph.

Solution

Map the appropriate variable to the `fill` aesthetic.

We'll use the `uspopchange` data set for this example. It contains the percentage change in population for the US states from 2000 to 2010. We'll take the top 10 fastest-growing states and graph their percentage change. We'll also color the bars by region (Northeast, South, North Central, or West).

First, we'll take the top 10 states:

```
library(gcookbook) # For the data set
upc <- subset(uspopchange, rank(Change)>40)
upc
```

```
          State Abb Region Change
        Arizona  AZ   West   24.6
       Colorado  CO   West   16.9
        Florida  FL  South   17.6
        Georgia  GA  South   18.3
          Idaho  ID   West   21.1
         Nevada  NV   West   35.1
 North Carolina  NC  South   18.5
 South Carolina  SC  South   15.3
          Texas  TX  South   20.6
           Utah  UT   West   23.8
```

Now we can make the graph, mapping `Region` to `fill` (Figure 3-9):

```
ggplot(upc, aes(x=Abb, y=Change, fill=Region)) + geom_bar(stat="identity")
```

Discussion

The default colors aren't very appealing, so you may want to set them, using `scale_fill_brewer()` or `scale_fill_manual()`. With this example, we'll use the latter, and we'll set the outline color of the bars to black, with `colour="black"` (Figure 3-10). Note that *setting* occurs outside of `aes()`, while *mapping* occurs within `aes()`:

```
ggplot(upc, aes(x=reorder(Abb, Change), y=Change, fill=Region)) +
    geom_bar(stat="identity", colour="black") +
    scale_fill_manual(values=c("#669933", "#FFCC66")) +
    xlab("State")
```

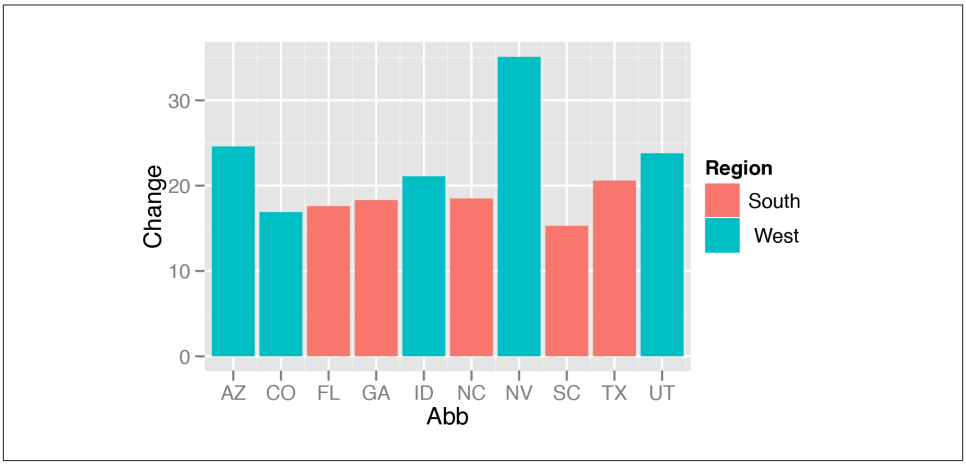


Figure 3-9. A variable mapped to fill

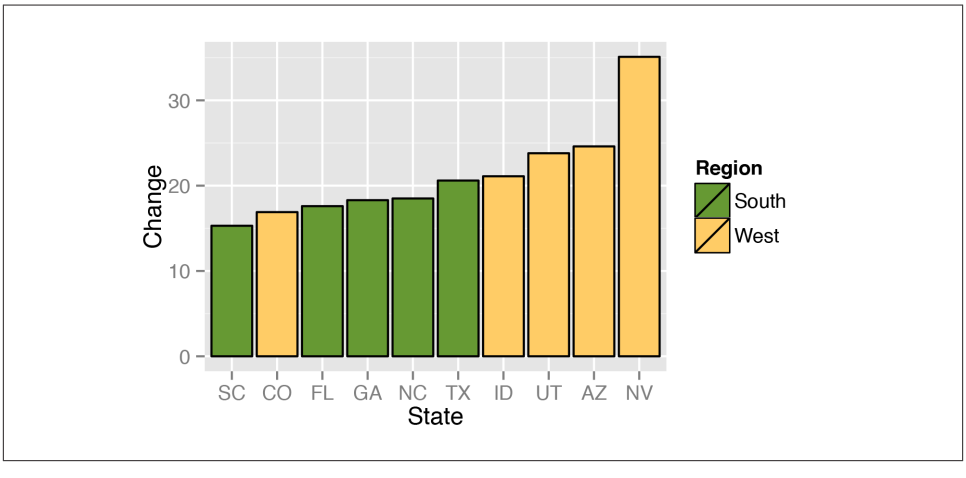


Figure 3-10. Graph with different colors, black outlines, and sorted by percentage change

This example also uses the `reorder()` function, as in this particular case it makes sense to sort the bars by their height, instead of in alphabetical order.

See Also

For more about using the `reorder()` function to reorder the levels of a factor based on the values of another variable, see Recipe 15.9.

For more information about using colors, see Chapter 12.

3.5. Coloring Negative and Positive Bars Differently

Problem

You want to use different colors for negative and positive-valued bars.

Solution

We'll use a subset of the `climate` data and create a new column called `pos`, which indicates whether the value is positive or negative:

```
library(gcookbook) # For the data set
csub <- subset(climate, Source=="Berkeley" & Year >= 1900)
csub$pos <- csub$Anomaly10y >= 0

csub
```

```
   Source Year Anomaly1y Anomaly5y Anomaly10y Unc10y
 Berkeley 1900        NA        NA     -0.171  0.108 FALSE
 Berkeley 1901        NA        NA     -0.162  0.109 FALSE
 Berkeley 1902        NA        NA     -0.177  0.108 FALSE
 ...
 Berkeley 2002        NA        NA      0.856  0.028  TRUE
 Berkeley 2003        NA        NA      0.869  0.028  TRUE
 Berkeley 2004        NA        NA      0.884  0.029  TRUE
```

Once we have the data, we can make the graph and map `pos` to the fill color, as in Figure 3-11. Notice that we use `position="identity"` with the bars. This will prevent a warning message about stacking not being well defined for negative numbers:

```
ggplot(csub, aes(x=Year, y=Anomaly10y, fill=pos)) +
  geom_bar(stat="identity", position="identity")
```

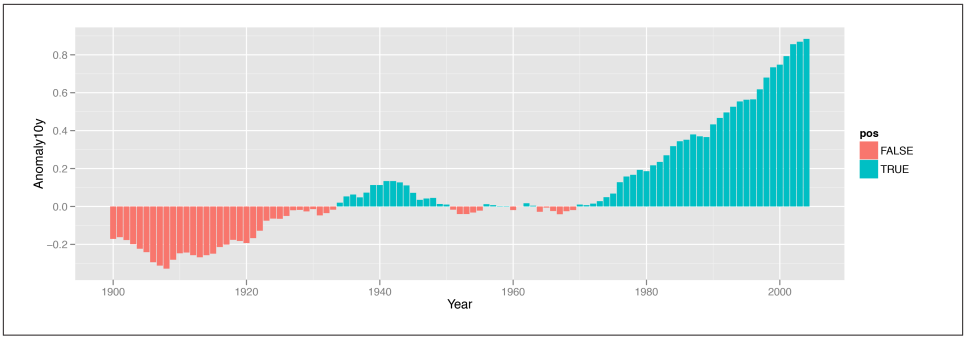


Figure 3-11. Different colors for positive and negative values

Discussion

There are a few problems with the first attempt. First, the colors are probably the reverse of what we want: usually, blue means cold and red means hot. Second, the legend is redundant and distracting.

We can change the colors with `scale_fill_manual()` and remove the legend with `guide=FALSE`, as shown in Figure 3-12. We'll also add a thin black outline around each of the bars by setting `colour` and specifying `size`, which is the thickness of the outline, in millimeters:

```
ggplot(csub, aes(x=Year, y=Anomaly10y, fill=pos)) +
    geom_bar(stat="identity", position="identity", colour="black", size=0.25) +
    scale_fill_manual(values=c("#CCEEFF", "#FFDDDD"), guide=FALSE)
```

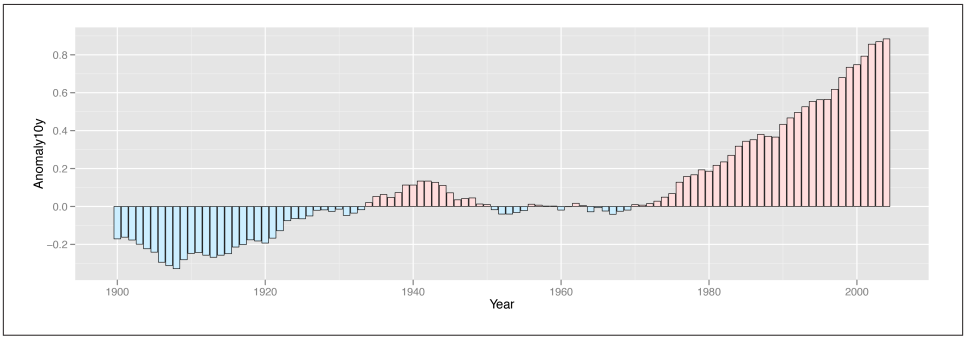


Figure 3-12. Graph with customized colors and no legend

See Also

To change the colors used, see Recipes 12.3 and 12.4.

To hide the legend, see Recipe 10.1.

3.6. Adjusting Bar Width and Spacing

Problem

You want to adjust the width of bars and the spacing between them.

Solution

To make the bars narrower or wider, set `width` in `geom_bar()`. The default value is 0.9; larger values make the bars wider, and smaller values make the bars narrower (Figure 3-13).

For example, for standard-width bars:

```
library(gcookbook) # For the data set

ggplot(pg_mean, aes(x=group, y=weight)) + geom_bar(stat="identity")
```

For narrower bars:

```
ggplot(pg_mean, aes(x=group, y=weight)) + geom_bar(stat="identity", width=0.5)
```

And for wider bars (these have the maximum width of 1):

```
ggplot(pg_mean, aes(x=group, y=weight)) + geom_bar(stat="identity", width=1)
```

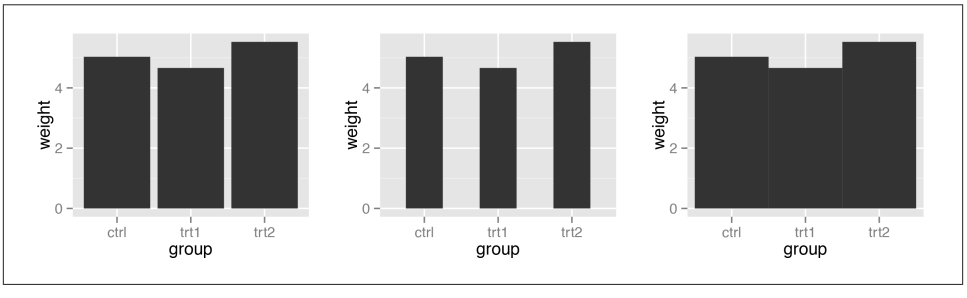


Figure 3-13. Different bar widths

For grouped bars, the default is to have no space between bars within each group. To add space between bars within a group, make `width` smaller and set the value for `position_dodge` to be larger than `width` (Figure 3-14).

For a grouped bar graph with narrow bars:

```
ggplot(cabbage_exp, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(stat="identity", width=0.5, position="dodge")
```

And with some space between the bars:

```
ggplot(cabbage_exp, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(stat="identity", width=0.5, position=position_dodge(0.7))
```

The first graph used `position="dodge"`, and the second graph used `position=position_dodge()`. This is because `position="dodge"` is simply shorthand for `position=position_dodge()` with the default value of 0.9, but when we want to set a specific value, we need to use the more verbose command.

Discussion

The default value of `width` is 0.9, and the default value used for `position_dodge()` is the same. To be more precise, the value of `width` in `position_dodge()` is the same as `width` in `geom_bar()`.

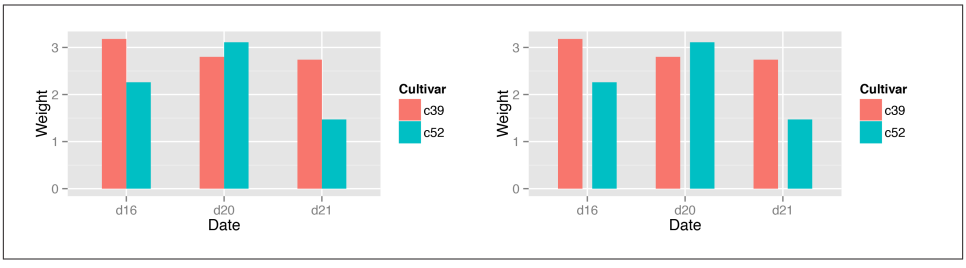


Figure 3-14. Left: bar graph with narrow grouped bars; right: with space between the bars

All of these will have the same result:

```
geom_bar(position="dodge")
geom_bar(width=0.9, position=position_dodge())
geom_bar(position=position_dodge(0.9))
geom_bar(width=0.9, position=position_dodge(width=0.9))
```

The items on the x-axis have `x` values of 1, 2, 3, and so on, though you typically don't refer to them by these numerical values. When you use `geom_bar(width=0.9)`, it makes each group take up a total width of 0.9 on the x-axis. When you use `position_dodge(width=0.9)`, it spaces the bars so that the *middle* of each bar is right where it would be if the bar width were 0.9 and the bars were touching. This is illustrated in Figure 3-15. The two graphs both have the same dodge width of 0.9, but while the top has a bar width of 0.9, the bottom has a bar width of 0.2. Despite the different bar widths, the middles of the bars stay aligned.

If you make the entire graph wider or narrower, the bar dimensions will scale proportionally. To see how this works, you can just resize the window in which the graphs appear. For information about controlling this when writing to a file, see Chapter 14.

3.7. Making a Stacked Bar Graph

Problem

You want to make a stacked bar graph.

Solution

Use `geom_bar()` and map a variable `fill`. This will put `Date` on the x-axis and use `Cultivar` for the fill color, as shown in Figure 3-16:

```
library(gcookbook) # For the data set
ggplot(cabbage_exp, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(stat="identity")
```

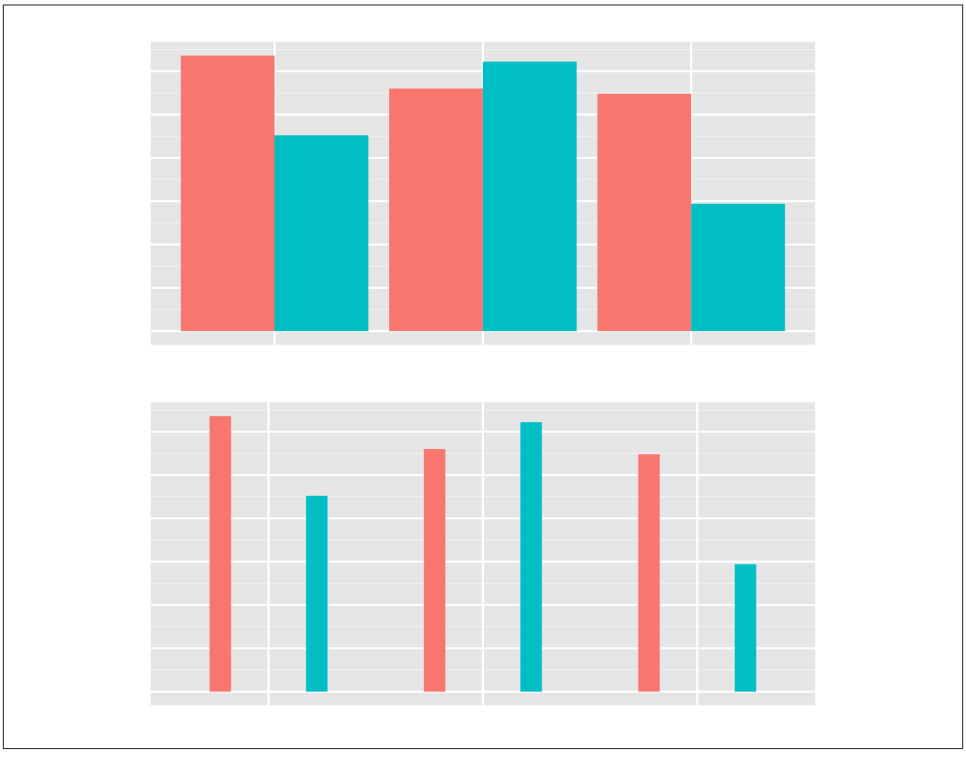


Figure 3-15. Same dodge width of 0.9, but different bar widths of 0.9 (top) and 0.2 (bottom)

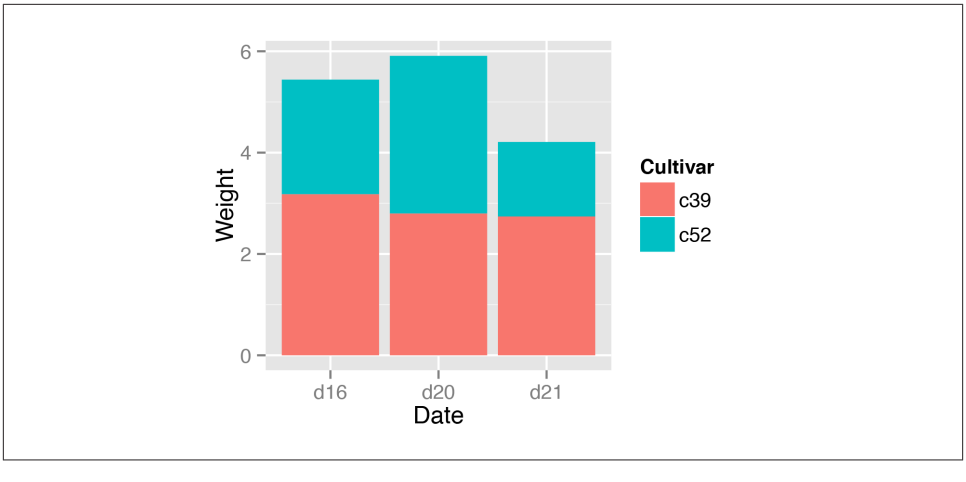


Figure 3-16. Stacked bar graph

Discussion

To understand how the graph is made, it's useful to see how the data is structured. There are three levels of `Date` and two levels of `Cultivar`, and for each combination there is a value for `Weight`:

```
cabbage_exp
```

```
 Cultivar Date Weight       sd  n         se
      c39  d16   3.18 0.9566144 10 0.30250803
      c39  d20   2.80 0.2788867 10 0.08819171
      c39  d21   2.74 0.9834181 10 0.31098410
      c52  d16   2.26 0.4452215 10 0.14079141
      c52  d20   3.11 0.7908505 10 0.25008887
      c52  d21   1.47 0.2110819 10 0.06674995
```

One problem with the default output is that the stacking order is the opposite of the order of items in the legend. As shown in Figure 3-17, you can reverse the order of items in the legend by using `guides()` and specifying the aesthetic for which the legend should be reversed. In this case, it's the `fill` aesthetic:

```
ggplot(cabbage_exp, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(stat="identity") +
    guides(fill=guide_legend(reverse=TRUE))
```

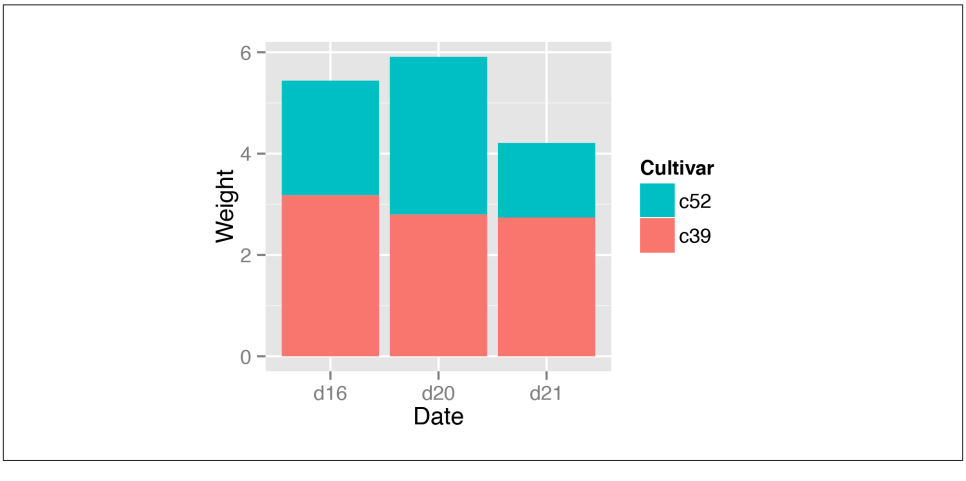


Figure 3-17. Stacked bar graph with reversed legend order

If you'd like to reverse the stacking order, as in Figure 3-18, specify `order=desc()` in the aesthetic mapping:

```
libary(plyr) # Needed for desc()
ggplot(cabbage_exp, aes(x=Date, y=Weight, fill=Cultivar, order=desc(Cultivar))) +
    geom_bar(stat="identity")
```

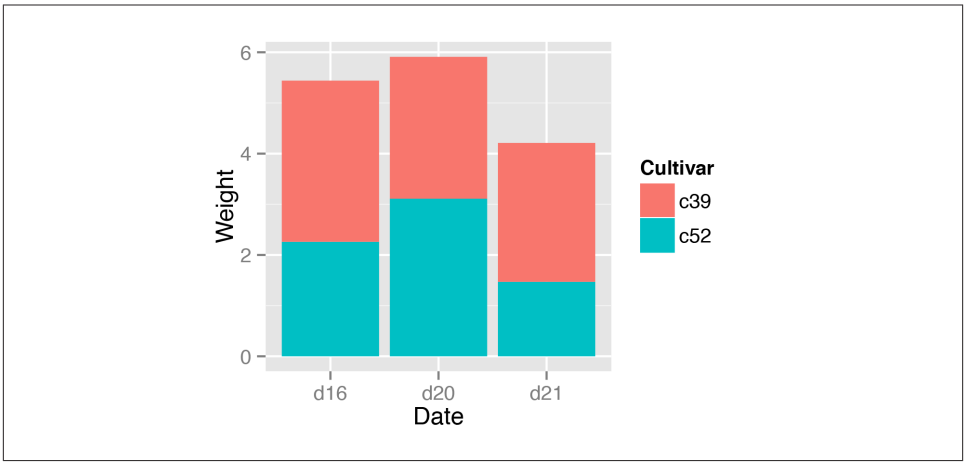


Figure 3-18. Stacked bar graph with reversed stacking order

It's also possible to modify the column of the data frame so that the factor levels are in a different order (see Recipe 15.8). Do this with care, since the modified data could change the results of other analyses.

For a more polished graph, we'll keep the reversed legend order, use `scale_fill_brewer()` to get a different color palette, and use `colour="black"` to get a black outline (Figure 3-19):

```
ggplot(cabbage_exp, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(stat="identity", colour="black") +
    guides(fill=guide_legend(reverse=TRUE)) +
    scale_fill_brewer(palette="Pastel1")
```

See Also

For more on using colors in bar graphs, see Recipe 3.4.

To reorder the levels of a factor based on the values of another variable, see Recipe 15.9. To manually change the order of factor levels, see Recipe 15.8.

3.8. Making a Proportional Stacked Bar Graph

Problem

You want to make a stacked bar graph that shows proportions (also called a 100% stacked bar graph).

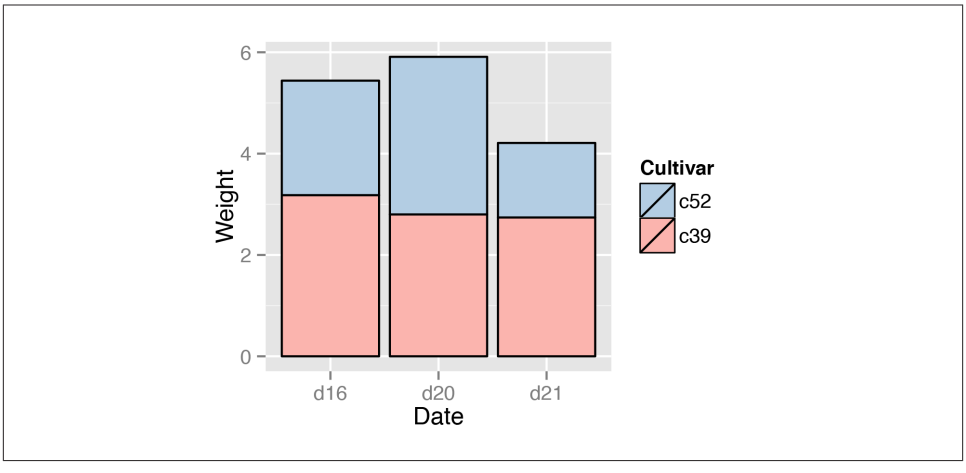


Figure 3-19. Stacked bar graph with reversed legend, new palette, and black outline

Solution

First, scale the data to 100% within each stack. This can be done by using `ddply()` from the plyr package, with `transform()`. Then plot the resulting data, as shown in Figure 3-20:

```
library(gcookbook) # For the data set
library(plyr)
# Do a group-wise transform(), splitting on "Date"
ce <- ddply(cabbage_exp, "Date", transform,
            percent_weight = Weight / sum(Weight) * 100)

ggplot(ce, aes(x=Date, y=percent_weight, fill=Cultivar)) +
    geom_bar(stat="identity")
```

Discussion

To calculate the percentages within each `Weight` group, we used the `ddply()` function. In the example here, the `ddply()` function splits the input data frame, `cabbage_exp`, by the specified variable, `Weight`, and applies a function, `transform()`, to each piece. (Any remaining arguments in the `ddply()` call are passed along to the function.)

This is what `cabbage_exp` looks like, and what the `ddply()` call does to it:

```
cabbage_exp
```

```
 Cultivar Date Weight        sd  n         se
      c39  d16   3.18 0.9566144 10 0.30250803
      c39  d20   2.80 0.2788867 10 0.08819171
      c39  d21   2.74 0.9834181 10 0.31098410
```

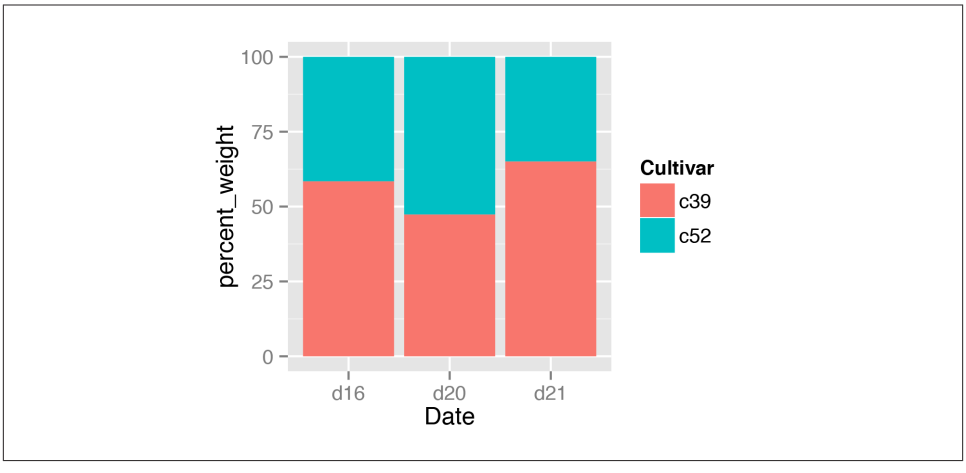


Figure 3-20. Proportional stacked bar graph

```
      c52  d16   2.26 0.4452215 10 0.14079141
      c52  d20   3.11 0.7908505 10 0.25008887
      c52  d21   1.47 0.2110819 10 0.06674995
```

```
ddply(cabbage_exp, "Date", transform,
      percent_weight = Weight / sum(Weight) * 100)
```

```
 Cultivar Date Weight        sd  n         se percent_weight
      c39  d16   3.18 0.9566144 10 0.30250803       58.45588
      c52  d16   2.26 0.4452215 10 0.14079141       41.54412
      c39  d20   2.80 0.2788867 10 0.08819171       47.37733
      c52  d20   3.11 0.7908505 10 0.25008887       52.62267
      c39  d21   2.74 0.9834181 10 0.31098410       65.08314
      c52  d21   1.47 0.2110819 10 0.06674995       34.91686
```

Once the percentages are computed, making the graph is the same as with a regular stacked bar graph.

As with regular stacked bar graphs, it makes sense to reverse the legend order, change the color palette, and add an outline. This is shown in (Figure 3-21):

```
ggplot(ce, aes(x=Date, y=percent_weight, fill=Cultivar)) +
    geom_bar(stat="identity", colour="black") +
    guides(fill=guide_legend(reverse=TRUE)) +
    scale_fill_brewer(palette="Pastel1")
```

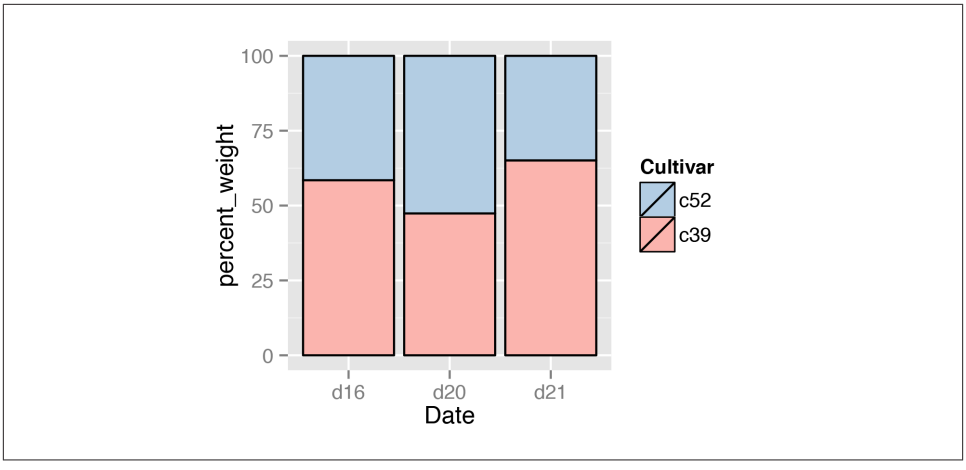


Figure 3-21. Proportional stacked bar graph with reversed legend, new palette, and black outline

See Also

For more on transforming data by groups, see Recipe 15.16.

3.9. Adding Labels to a Bar Graph

Problem

You want to add labels to the bars in a bar graph.

Solution

Add `geom_text()` to your graph. It requires a mapping for `x`, `y`, and the text itself. By setting `vjust` (the vertical justification), it is possible to move the text above or below the tops of the bars, as shown in Figure 3-22:

```
library(gcookbook) # For the data set

# Below the top
ggplot(cabbage_exp, aes(x=interaction(Date, Cultivar), y=Weight)) +
    geom_bar(stat="identity") +
    geom_text(aes(label=Weight), vjust=1.5, colour="white")

# Above the top
ggplot(cabbage_exp, aes(x=interaction(Date, Cultivar), y=Weight)) +
    geom_bar(stat="identity") +
    geom_text(aes(label=Weight), vjust=-0.2)
```

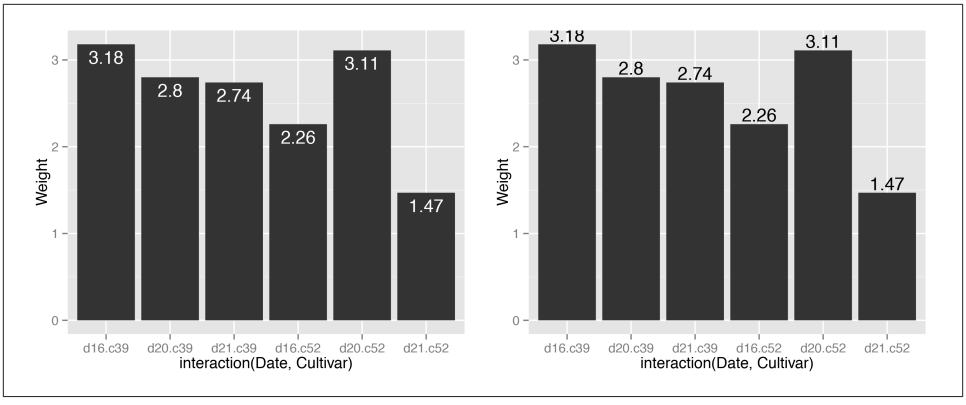


Figure 3-22. Left: labels under the tops of bars; right: labels above bars

Notice that when the labels are placed atop the bars, they may be clipped. To remedy this, see Recipe 8.4.

Discussion

In Figure 3-22, the *y* coordinates of the labels are centered at the top of each bar; by setting the vertical justification (`vjust`), they appear below or above the bar tops. One drawback of this is that when the label is above the top of the bar, it can go off the top of the plotting area. To fix this, you can manually set the *y* limits, or you can set the *y* positions of the text *above* the bars and not change the vertical justification. One drawback to changing the text's *y* position is that if you want to place the text fully above or below the bar top, the value to add will depend on the *y* range of the data; in contrast, changing `vjust` to a different value will always move the text the same distance relative to the height of the bar:

```
# Adjust y limits to be a little higher
ggplot(cabbage_exp, aes(x=interaction(Date, Cultivar), y=Weight)) +
    geom_bar(stat="identity") +
    geom_text(aes(label=Weight), vjust=-0.2) +
    ylim(0, max(cabbage_exp$Weight) * 1.05)

# Map y positions slightly above bar top - y range of plot will auto-adjust
ggplot(cabbage_exp, aes(x=interaction(Date, Cultivar), y=Weight)) +
    geom_bar(stat="identity") +
    geom_text(aes(y=Weight+0.1, label=Weight))
```

For grouped bar graphs, you also need to specify `position=position_dodge()` and give it a value for the dodging width. The default dodge width is 0.9. Because the bars are narrower, you might need to use `size` to specify a smaller font to make the labels fit. The default value of `size` is 5, so we'll make it smaller by using 3 (Figure 3-23):

```
ggplot(cabbage_exp, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(stat="identity", position="dodge") +
    geom_text(aes(label=Weight), vjust=1.5, colour="white",
              position=position_dodge(.9), size=3)
```

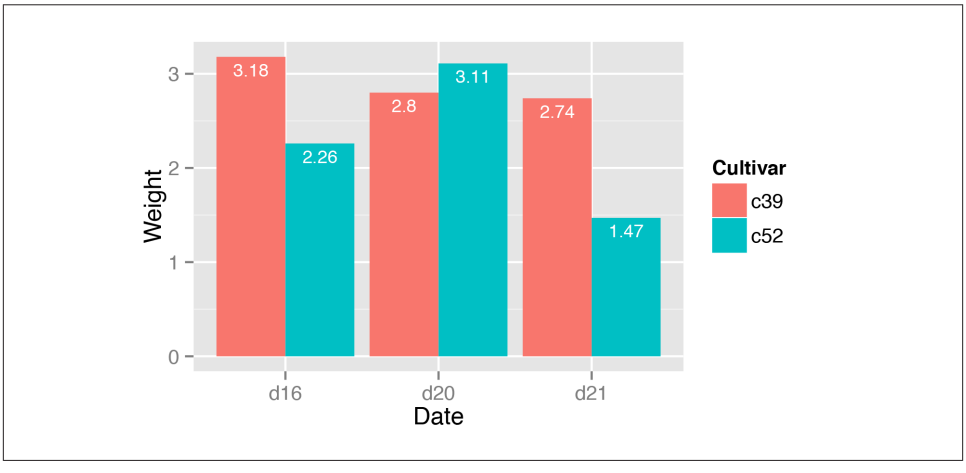


Figure 3-23. Labels on grouped bars

Putting labels on stacked bar graphs requires finding the cumulative sum for each stack. To do this, first make sure the data is sorted properly—if it isn't, the cumulative sum might be calculated in the wrong order. We'll use the `arrange()` function from the plyr package, which automatically gets loaded with ggplot2:

```
library(plyr)
# Sort by the day and sex columns
ce <- arrange(cabbage_exp, Date, Cultivar)
```

Once we make sure the data is sorted properly, we'll use `ddply()` to chunk it into groups by `Date`, then calculate a cumulative sum of `Weight` within each chunk:

```
# Get the cumulative sum
ce <- ddply(ce, "Date", transform, label_y=cumsum(Weight))
ce
```

```
 Cultivar Date Weight        sd  n         se label_y
      c39  d16   3.18 0.9566144 10 0.30250803    3.18
      c52  d16   2.26 0.4452215 10 0.14079141    5.44
      c39  d20   2.80 0.2788867 10 0.08819171    2.80
      c52  d20   3.11 0.7908505 10 0.25008887    5.91
      c39  d21   2.74 0.9834181 10 0.31098410    2.74
      c52  d21   1.47 0.2110819 10 0.06674995    4.21
```

```
ggplot(ce, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(stat="identity") +
    geom_text(aes(y=label_y, label=Weight), vjust=1.5, colour="white")
```

The result is shown in Figure 3-24.

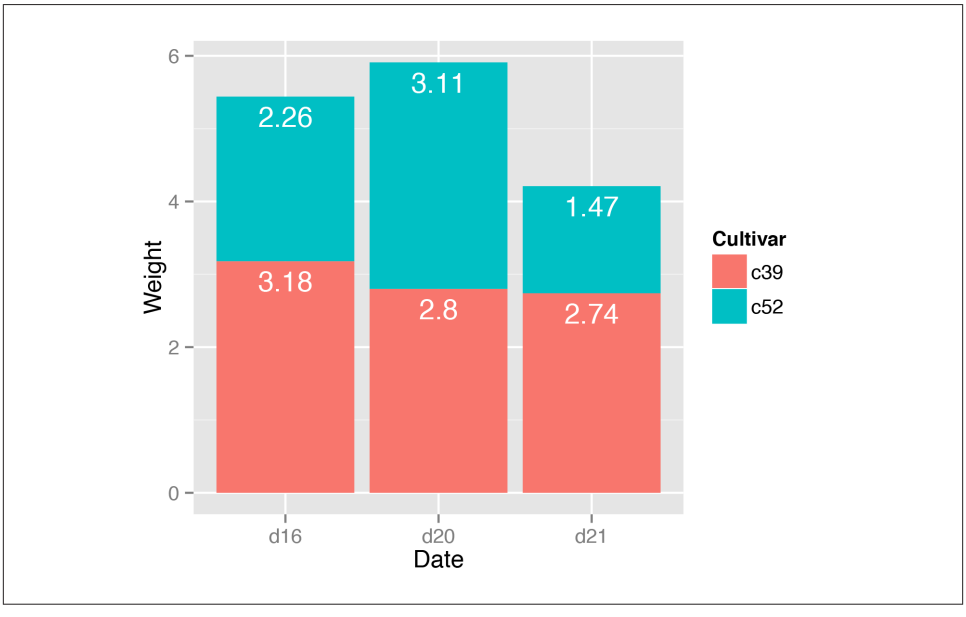


Figure 3-24. Labels on stacked bars

When using labels, changes to the stacking order are best done by modifying the order of levels in the factor (see Recipe 15.8) before taking the cumulative sum. The other method of changing stacking order, by specifying `breaks` in a scale, won't work properly, because the order of the cumulative sum won't be the same as the stacking order.

To put the labels in the middle of each bar (Figure 3-25), there must be an adjustment to the cumulative sum, and the *y* offset in `geom_bar()` can be removed:

```
ce <- arrange(cabbage_exp, Date, Cultivar)

# Calculate y position, placing it in the middle
ce <- ddply(ce, "Date", transform, label_y=cumsum(Weight)-0.5*Weight)

ggplot(ce, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(stat="identity") +
    geom_text(aes(y=label_y, label=Weight), colour="white")
```

For a more polished graph (Figure 3-26), we'll change the legend order and colors, add labels in the middle with a smaller font using `size`, add a "kg" using `paste`, and make sure there are always two digits after the decimal point by using `format`:

```
ggplot(ce, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(stat="identity", colour="black") +
```

Figure 3-25. Labels in the middle of stacked bars

```
geom_text(aes(y=label_y, label=paste(format(Weight, nsmall=2), "kg")),
          size=4) +
guides(fill=guide_legend(reverse=TRUE)) +
scale_fill_brewer(palette="Pastel1")
```

See Also

To control the appearance of the text, see Recipe 9.2.

For more on transforming data by groups, see Recipe 15.16.

3.10. Making a Cleveland Dot Plot

Problem

You want to make a Cleveland dot plot.

Solution

Cleveland dot plots are sometimes used instead of bar graphs because they reduce visual clutter and are easier to read.

The simplest way to create a dot plot (as shown in Figure 3-27) is to use `geom_point()`:

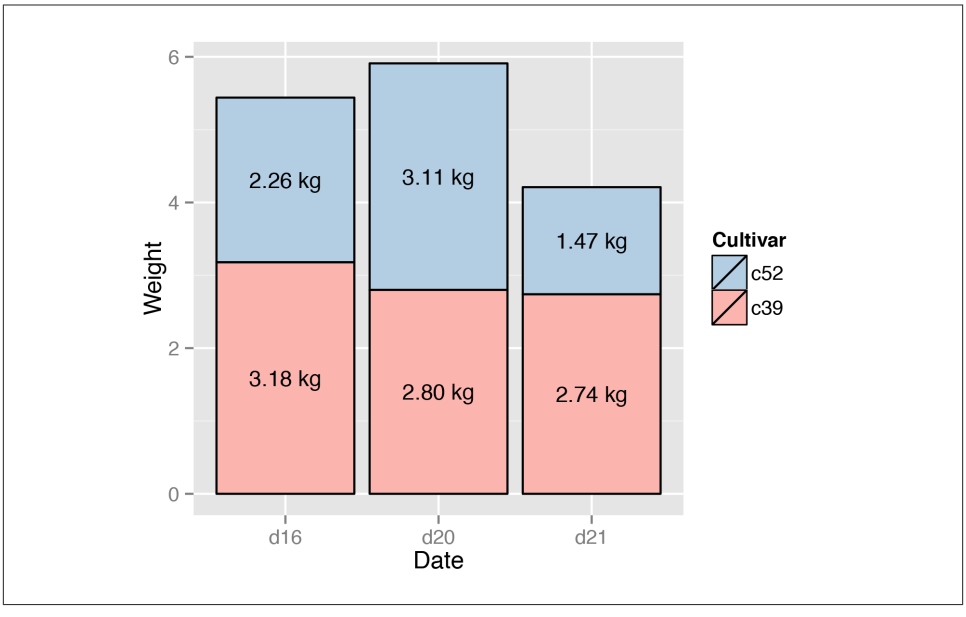


Figure 3-26. Customized stacked bar graph with labels

```
library(gcookbook) # For the data set
tophit <- tophitters2001[1:25, ] # Take the top 25 from the tophitters data set

ggplot(tophit, aes(x=avg, y=name)) + geom_point()
```

Discussion

The `tophitters2001` data set contains many columns, but we'll focus on just three of them for this example:

```
tophit[, c("name", "lg", "avg")]
```

```
             name lg    avg
   Larry Walker NL 0.3501
  Ichiro Suzuki AL 0.3497
   Jason Giambi AL 0.3423
...
     Jeff Conine AL 0.3111
     Derek Jeter AL 0.3111
```

In Figure 3-27 the names are sorted alphabetically, which isn't very useful in this graph. Dot plots are often sorted by the value of the continuous variable on the horizontal axis.

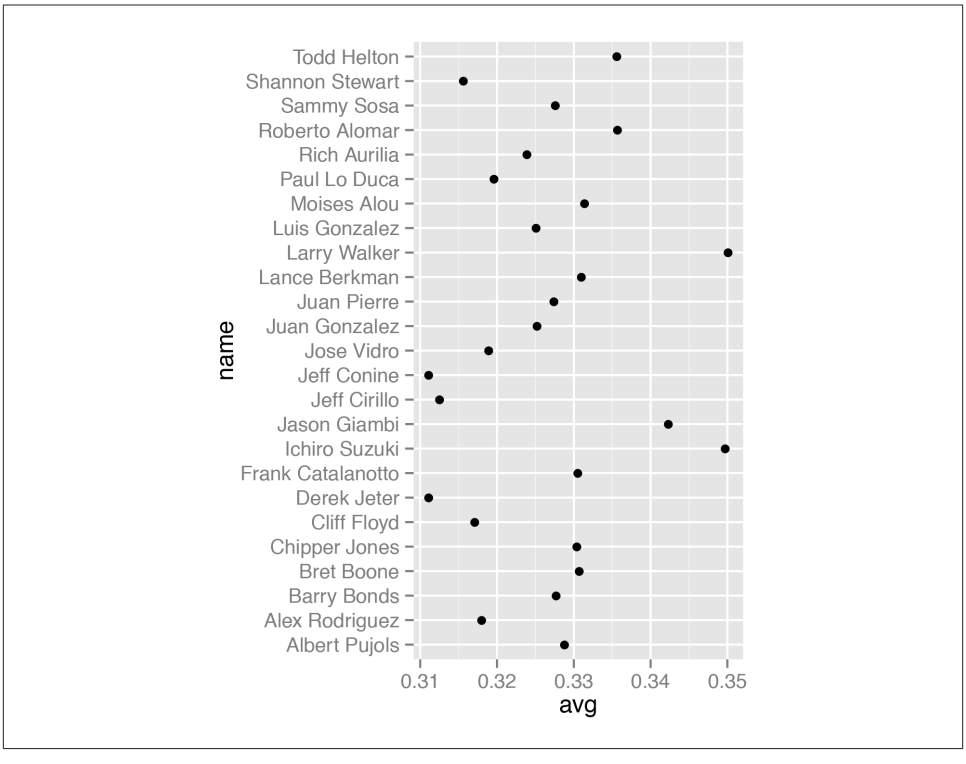


Figure 3-27. Basic dot plot

Although the rows of `tophit` happen to be sorted by `avg`, that doesn't mean that the items will be ordered that way in the graph. By default, the items on the given axis will be ordered however is appropriate for the data type. `name` is a character vector, so it's ordered alphabetically. If it were a factor, it would use the order defined in the factor levels. In this case, we want `name` to be sorted by a different variable, `avg`.

To do this, we can use `reorder(name, avg)`, which takes the `name` column, turns it into a factor, and sorts the factor levels by `avg`. To further improve the appearance, we'll make the vertical grid lines go away by using the theming system, and turn the horizontal grid lines into dashed lines (Figure 3-28):

```
ggplot(tophit, aes(x=avg, y=reorder(name, avg))) +
    geom_point(size=3) +                        # Use a larger dot
    theme_bw() +
    theme(panel.grid.major.x = element_blank(),
          panel.grid.minor.x = element_blank(),
          panel.grid.major.y = element_line(colour="grey60", linetype="dashed"))
```

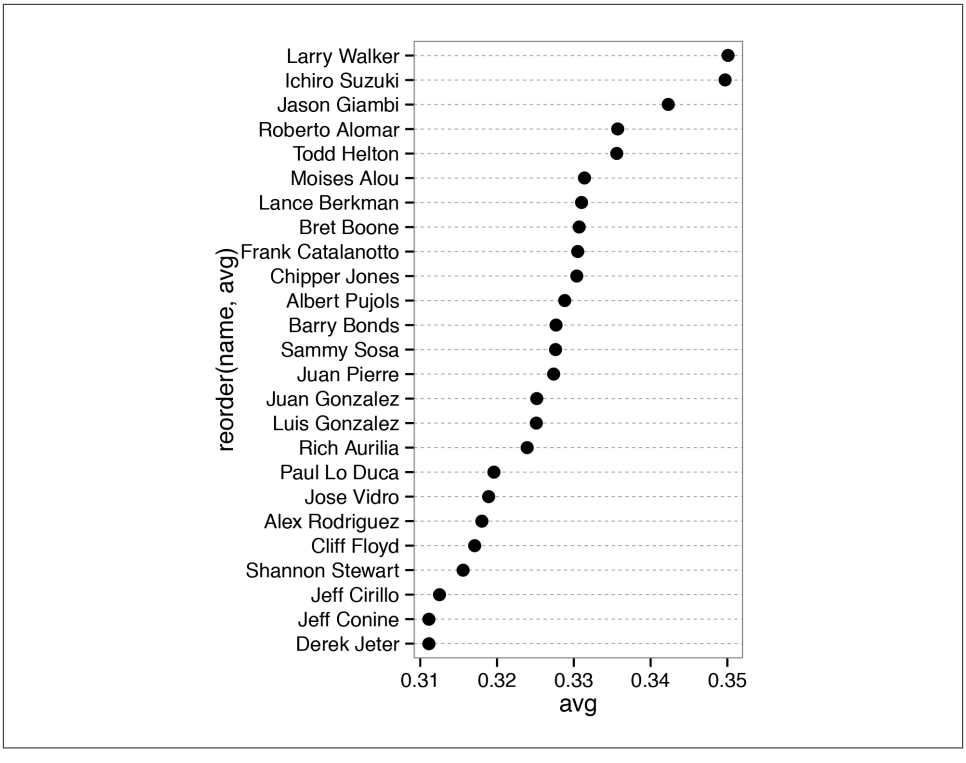


Figure 3-28. Dot plot, ordered by batting average

It's also possible to swap the axes so that the names go along the x-axis and the values go along the y-axis, as shown in Figure 3-29. We'll also rotate the text labels by 60 degrees:

```
ggplot(tophit, aes(x=reorder(name, avg), y=avg)) +
    geom_point(size=3) +                          # Use a larger dot
    theme_bw() +
    theme(axis.text.x = element_text(angle=60, hjust=1),
          panel.grid.major.y = element_blank(),
          panel.grid.minor.y = element_blank(),
          panel.grid.major.x = element_line(colour="grey60", linetype="dashed"))
```

It's also sometimes desirable to group the items by another variable. In this case we'll use the factor `lg`, which has the levels `NL` and `AL`, representing the National League and the American League. This time we want to sort first by `lg` and then by `avg`. Unfortunately, the `reorder()` function will only order factor levels by one other variable; to order the factor levels by two variables, we must do it manually:

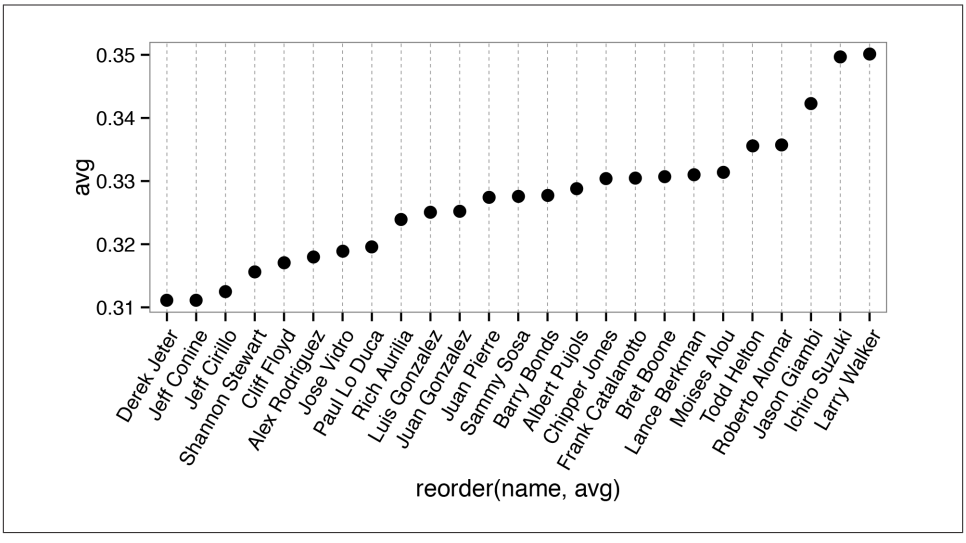


Figure 3-29. Dot plot with names on x-axis and values on y-axis

```
# Get the names, sorted first by lg, then by avg
nameorder <- tophit$name[order(tophit$lg, tophit$avg)]

# Turn name into a factor, with levels in the order of nameorder
tophit$name <- factor(tophit$name, levels=nameorder)
```

To make the graph (Figure 3-30), we'll also add a mapping of `lg` to the color of the points. Instead of using grid lines that run all the way across, this time we'll make the lines go only up to the points, by using `geom_segment()`. Note that `geom_segment()` needs values for `x`, `y`, `xend`, and `yend`:

```
ggplot(tophit, aes(x=avg, y=name)) +
    geom_segment(aes(yend=name), xend=0, colour="grey50") +
    geom_point(size=3, aes(colour=lg)) +
    scale_colour_brewer(palette="Set1", limits=c("NL","AL")) +
    theme_bw() +
    theme(panel.grid.major.y = element_blank(),   # No horizontal grid lines
          legend.position=c(1, 0.55),             # Put legend inside plot area
          legend.justification=c(1, 0.5))
```

Another way to separate the two groups is to use facets, as shown in Figure 3-31. The order in which the facets are displayed is different from the sorting order in Figure 3-30; to change the display order, you must change the order of factor levels in the `lg` variable:

```
ggplot(tophit, aes(x=avg, y=name)) +
    geom_segment(aes(yend=name), xend=0, colour="grey50") +
    geom_point(size=3, aes(colour=lg)) +
```

```
scale_colour_brewer(palette="Set1", limits=c("NL","AL"), guide=FALSE) +
theme_bw() +
theme(panel.grid.major.y = element_blank()) +
facet_grid(lg ~ ., scales="free_y", space="free_y")
```

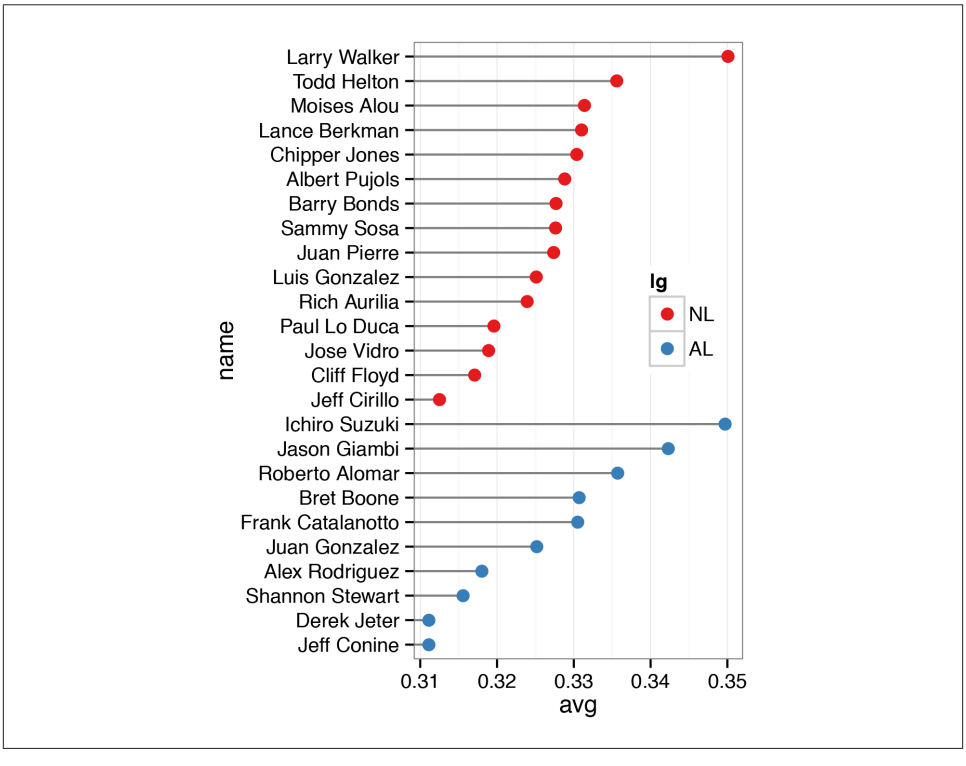


Figure 3-30. Grouped by league, with lines that stop at the point

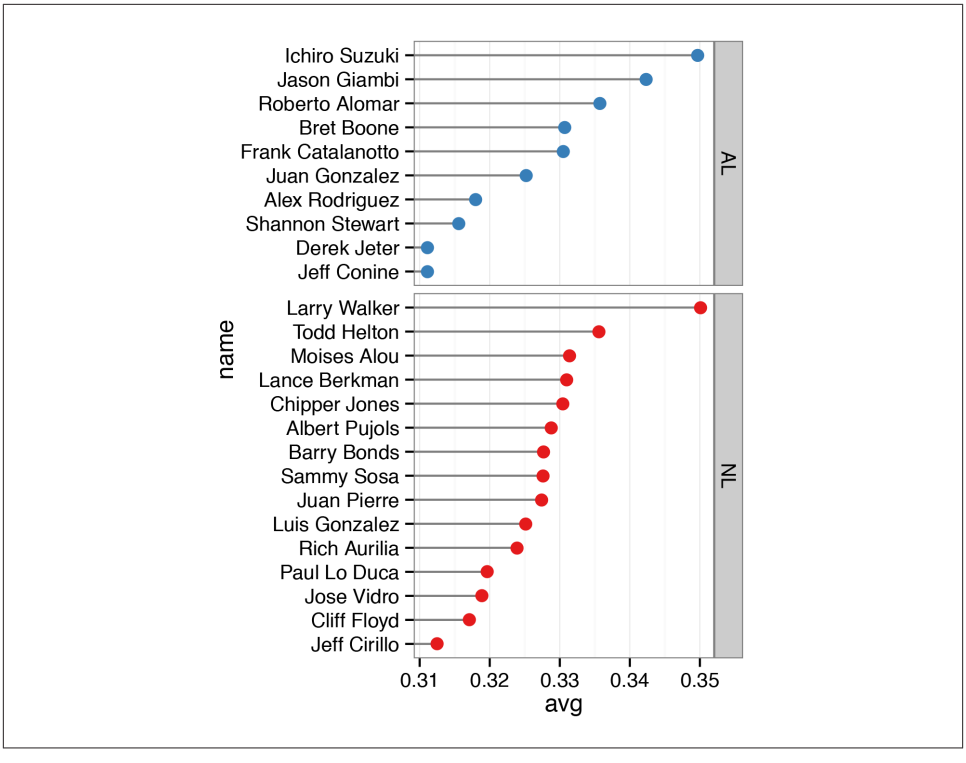


Figure 3-31. Faceted by league

See Also

For more on changing the order of factor levels, see Recipe 15.8. Also see Recipe 15.9 for details on changing the order of factor levels based on some other values.

For more on moving the legend, see Recipe 10.2. To hide grid lines, see Recipe 9.6.

CHAPTER 4
Line Graphs

Line graphs are typically used for visualizing how one continuous variable, on the y-axis, changes in relation to another continuous variable, on the x-axis. Often the *x* variable represents time, but it may also represent some other continuous quantity, like the amount of a drug administered to experimental subjects.

As with bar graphs, there are exceptions. Line graphs can also be used with a discrete variable on the x-axis. This is appropriate when the variable is ordered (e.g., "small", "medium", "large"), but not when the variable is unordered (e.g., "cow", "goose", "pig"). Most of the examples in this chapter use a continuous *x* variable, but we'll see one example where the variable is converted to a factor and thus treated as a discrete variable.

4.1. Making a Basic Line Graph

Problem

You want to make a basic line graph.

Solution

Use `ggplot()` with `geom_line()`, and specify what variables you mapped to `x` and `y` (Figure 4-1):

```
ggplot(BOD, aes(x=Time, y=demand)) + geom_line()
```

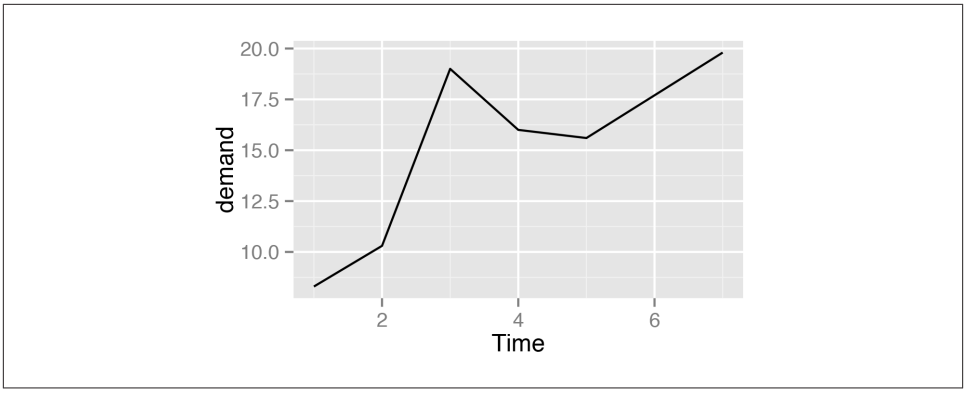


Figure 4-1. Basic line graph

Discussion

In this sample data set, the *x* variable, `Time`, is in one column and the *y* variable, `demand`, is in another:

```
BOD
```

```
  Time demand
1    1    8.3
2    2   10.3
3    3   19.0
4    4   16.0
5    5   15.6
7    7   19.8
```

Line graphs can be made with discrete (categorical) or continuous (numeric) variables on the x-axis. In the example here, the variable `demand` is numeric, but it could be treated as a categorical variable by converting it to a factor with `factor()` (Figure 4-2). When the *x* variable is a factor, you must also use `aes(group=1)` to ensure that `ggplot()` knows that the data points belong together and should be connected with a line (see Recipe 4.3 for an explanation of why `group` is needed with factors):

```
BOD1 <- BOD # Make a copy of the data
BOD1$Time <- factor(BOD1$Time)
ggplot(BOD1, aes(x=Time, y=demand, group=1)) + geom_line()
```

In the `BOD` data set there is no entry for Time=6, so there is no level 6 when `Time` is converted to a factor. Factors hold categorical values, and in that context, 6 is just another value. It happens to not be in the data set, so there's no space for it on the x-axis.

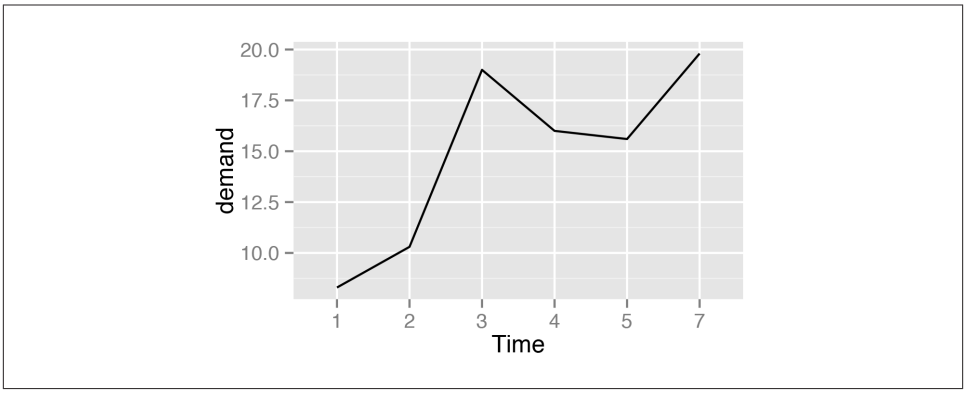


Figure 4-2. Basic line graph with a factor on the x-axis (notice that no space is allocated on the x-axis for 6)

With ggplot2, the default *y* range of a line graph is just enough to include the *y* values in the data. For some kinds of data, it's better to have the *y* range start from zero. You can use `ylim()` to set the range, or you can use `expand_limits()` to expand the range to include a value. This will set the range from zero to the maximum value of the `demand` column in `BOD` (Figure 4-3):

```
# These have the same result
ggplot(BOD, aes(x=Time, y=demand)) + geom_line() + ylim(0, max(BOD$demand))
ggplot(BOD, aes(x=Time, y=demand)) + geom_line() + expand_limits(y=0)
```

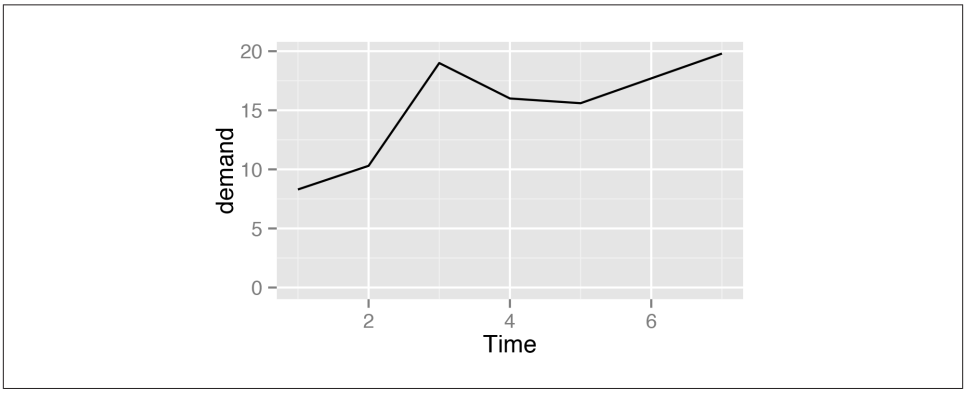


Figure 4-3. Line graph with manually set y range

See Also

See Recipe 8.2 for more on controlling the range of the axes.

4.2. Adding Points to a Line Graph

Problem

You want to add points to a line graph.

Solution

Add `geom_point()` (Figure 4-4):

```
ggplot(BOD, aes(x=Time, y=demand)) + geom_line() + geom_point()
```

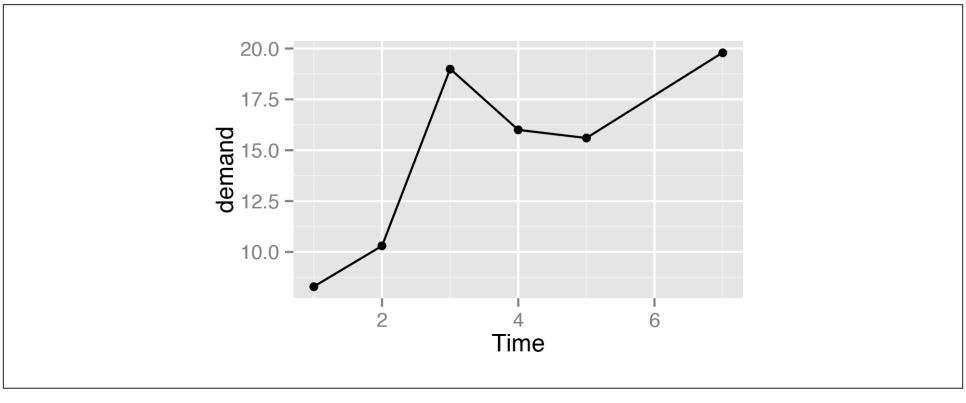


Figure 4-4. Line graph with points

Discussion

Sometimes it is useful to indicate each data point on a line graph. This is helpful when the density of observations is low, or when the observations do not happen at regular intervals. For example, in the `BOD` data set there is no entry for `Time`=6, but this is not apparent from just a bare line graph (compare Figure 4-3 with Figure 4-4).

In the `worldpop` data set, the intervals between each data point are not consistent. In the far past, the estimates were not as frequent as they are in the more recent past. Displaying points on the graph illustrates when each estimate was made (Figure 4-5):

```
library(gcookbook) # For the data set

ggplot(worldpop, aes(x=Year, y=Population)) + geom_line() + geom_point()

# Same with a log y-axis
ggplot(worldpop, aes(x=Year, y=Population)) + geom_line() + geom_point() +
    scale_y_log10()
```

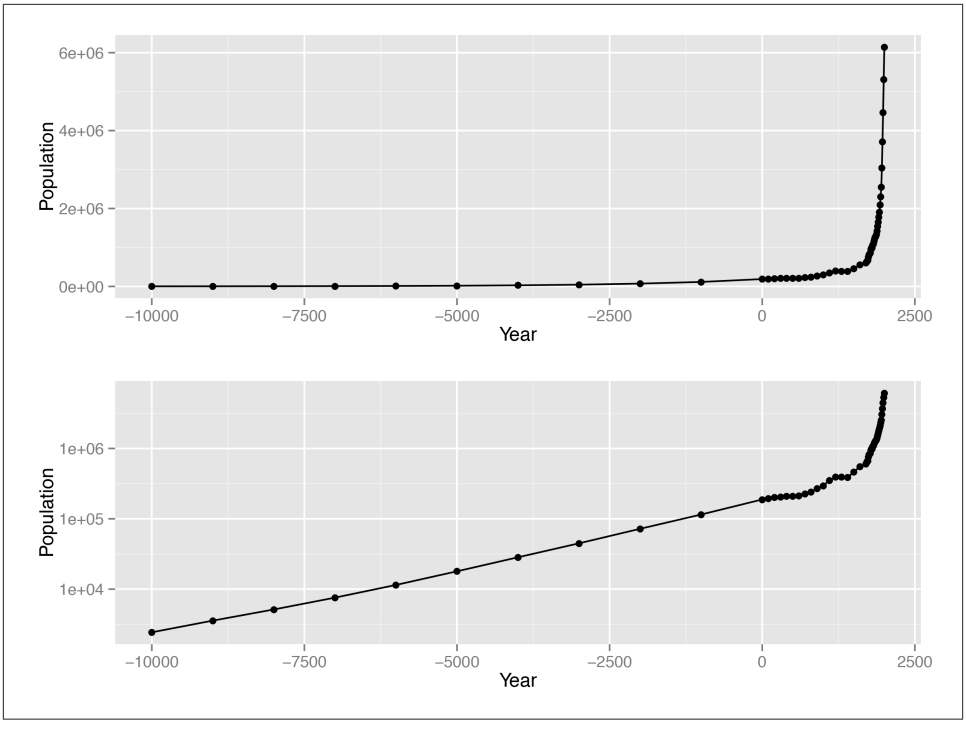


Figure 4-5. Top: points indicate where each data point is; bottom: the same data with a log y-axis

With the log y-axis, you can see that the rate of proportional change has increased in the last thousand years. The estimates for the years before 0 have a roughly constant rate of change of 10 times per 5,000 years. In the most recent 1,000 years, the population has increased at a much faster rate. We can also see that the population estimates are much more frequent in recent times—and probably more accurate!

See Also

To change the appearance of the points, see Recipe 4.5.

4.3. Making a Line Graph with Multiple Lines

Problem

You want to make a line graph with more than one line.

Solution

In addition to the variables mapped to the x- and y-axes, map another (discrete) variable to `colour` or `linetype`, as shown in Figure 4-6:

```
# Load plyr so we can use ddply() to create the example data set
library(plyr)
# Summarize the ToothGrowth data
tg <- ddply(ToothGrowth, c("supp", "dose"), summarise, length=mean(len))

# Map supp to colour
ggplot(tg, aes(x=dose, y=length, colour=supp)) + geom_line()

# Map supp to linetype
ggplot(tg, aes(x=dose, y=length, linetype=supp)) + geom_line()
```

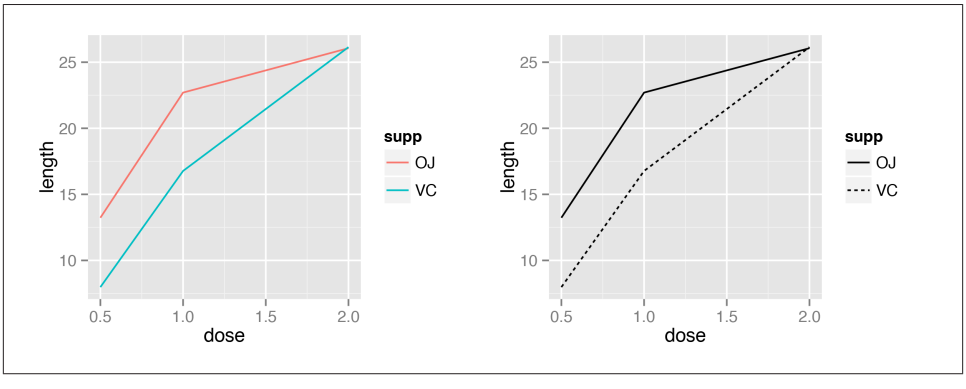


Figure 4-6. Left: a variable mapped to colour; right: a variable mapped to linetype

Discussion

The `tg` data has three columns, including the factor `supp`, which we mapped to `colour` and `linetype`:

```
tg
```

```
 supp dose length
   OJ  0.5  13.23
   OJ  1.0  22.70
   OJ  2.0  26.06
   VC  0.5   7.98
   VC  1.0  16.77
   VC  2.0  26.14
```

```
str(tg)
```

```
'data.frame': 6 obs. of  3 variables:
 $ supp  : Factor w/ 2 levels "OJ","VC": 1 1 1 2 2 2
 $ dose  : num  0.5 1 2 0.5 1 2
 $ length: num  13.23 22.7 26.06 7.98 16.77 ...
```

If the *x* variable is a factor, you must also tell `ggplot()` to `group` by that same variable, as described momentarily.

Line graphs can be used with a continuous or categorical variable on the x-axis. Sometimes the variable mapped to the x-axis is *conceived* of as being categorical, even when it's stored as a number. In the example here, there are three values of `dose`: 0.5, 1.0, and 2.0. You may want to treat these as categories rather than values on a continuous scale. To do this, convert `dose` to a factor (Figure 4-7):

```
ggplot(tg, aes(x=factor(dose), y=length, colour=supp, group=supp)) + geom_line()
```

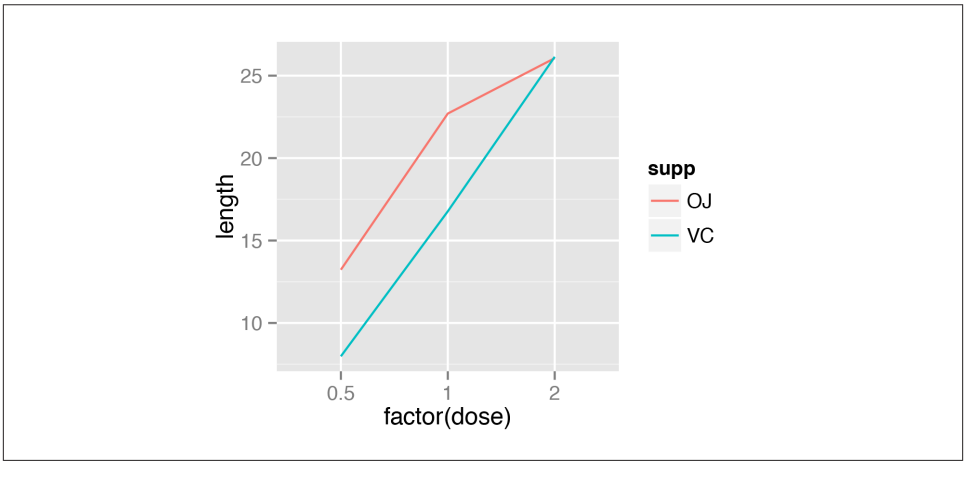


Figure 4-7. Line graph with continuous x variable converted to a factor

Notice the use of `group=supp`. Without this statement, `ggplot()` won't know how to group the data together to draw the lines, and it will give an error:

```
ggplot(tg, aes(x=factor(dose), y=length, colour=supp)) + geom_line()
```

```
  geom_path: Each group consists of only one observation. Do you need to adjust the
group aesthetic?
```

Another common problem when the incorrect grouping is used is that you will see a jagged sawtooth pattern, as in Figure 4-8:

```
ggplot(tg, aes(x=dose, y=length)) + geom_line()
```

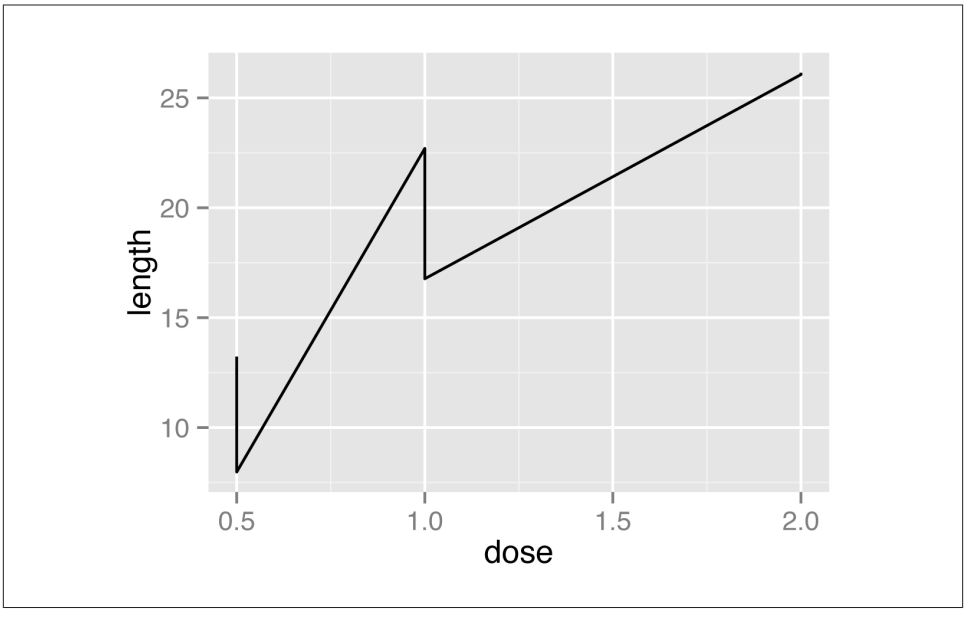


Figure 4-8. A sawtooth pattern indicates improper grouping

This happens because there are multiple data points at each *y* location, and `ggplot()` thinks they're all in one group. The data points for each group are connected with a single line, leading to the sawtooth pattern. If any *discrete* variables are mapped to aesthetics like `colour` or `linetype`, they are automatically used as grouping variables. But if you want to use other variables for grouping (that aren't mapped to an aesthetic), they should be used with `group`.

When in doubt, if your line graph looks wrong, try explicitly specifying the grouping variable with `group`. It's common for problems to occur with line graphs because `ggplot()` is unsure of how the variables should be grouped.

If your plot has points along with the lines, you can also map variables to properties of the points, such as `shape` and `fill` (Figure 4-9):

```
ggplot(tg, aes(x=dose, y=length, shape=supp)) + geom_line() +
    geom_point(size=4)           # Make the points a little larger

ggplot(tg, aes(x=dose, y=length, fill=supp)) + geom_line() +
    geom_point(size=4, shape=21) # Also use a point with a color fill
```

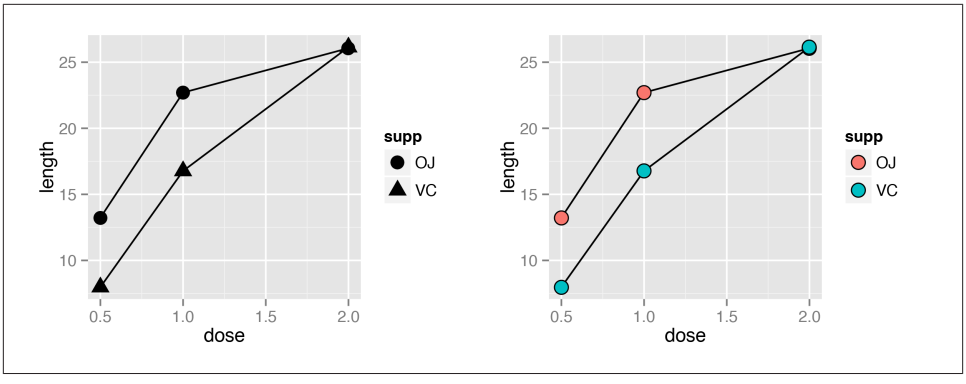


Figure 4-9. Left: line graph with different shapes; right: with different colors

Sometimes points will overlap. In these cases, you may want to *dodge* them, which means their positions will be adjusted left and right (Figure 4-10). When doing so, you must also dodge the lines, or else only the points will move and they will be misaligned. You must also specify how far they should move when dodged:

```
ggplot(tg, aes(x=dose, y=length, shape=supp)) +
    geom_line(position=position_dodge(0.2)) +          # Dodge lines by 0.2
    geom_point(position=position_dodge(0.2), size=4)  # Dodge points by 0.2
```

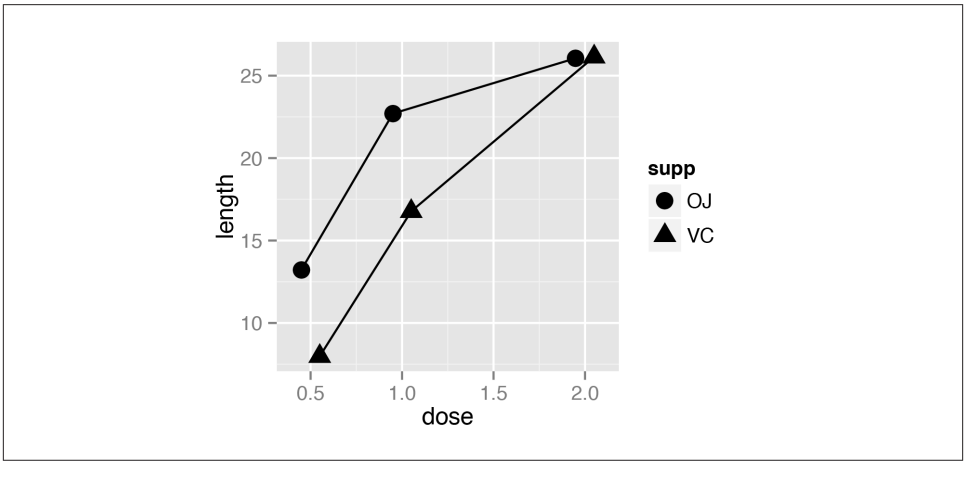


Figure 4-10. Dodging to avoid overlapping points

4.4. Changing the Appearance of Lines

Problem

You want to change the appearance of the lines in a line graph.

Solution

The type of line (solid, dashed, dotted, etc.) is set with `linetype`, the thickness (in mm) with `size`, and the color of the line with `colour`.

These properties can be set (as shown in Figure 4-11) by passing them values in the call to `geom_line()`:

```
ggplot(BOD, aes(x=Time, y=demand)) +
    geom_line(linetype="dashed", size=1, colour="blue")
```

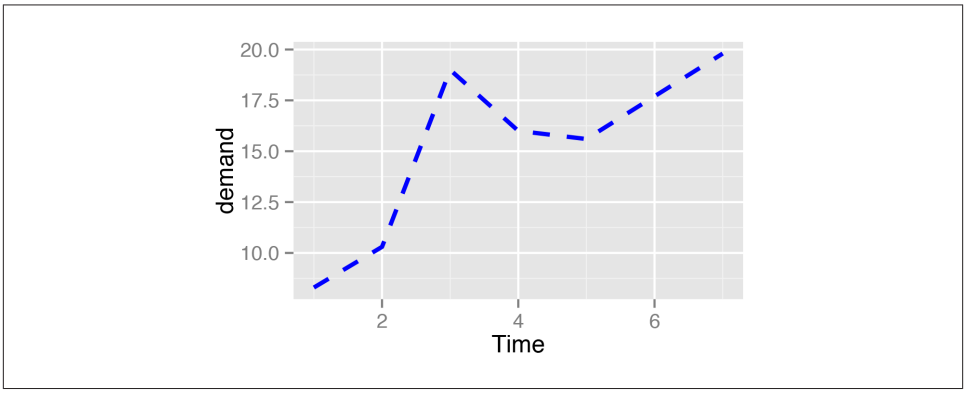


Figure 4-11. Line graph with custom linetype, size, and colour

If there is more than one line, setting the aesthetic properties will affect all of the lines. On the other hand, *mapping* variables to the properties, as we saw in Recipe 4.3, will result in each line looking different. The default colors aren't the most appealing, so you may want to use a different palette, as shown in Figure 4-12, by using `scale_colour_brewer()` or `scale_colour_manual()`:

```
# Load plyr so we can use ddply() to create the example data set
library(plyr)
# Summarize the ToothGrowth data
tg <- ddply(ToothGrowth, c("supp", "dose"), summarise, length=mean(len))

ggplot(tg, aes(x=dose, y=length, colour=supp)) +
    geom_line() +
    scale_colour_brewer(palette="Set1")
```

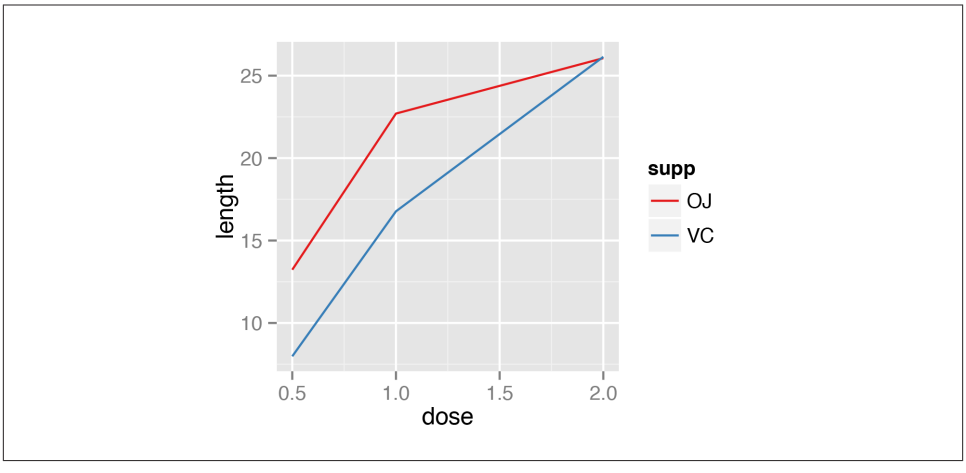


Figure 4-12. Using a palette from RColorBrewer

Discussion

To set a single constant color for all the lines, specify `colour` outside of `aes()`. The same works for `size`, `linetype`, and point `shape` (Figure 4-13). You may also have to specify the grouping variable:

```
# If both lines have the same properties, you need to specify a variable to
# use for grouping
ggplot(tg, aes(x=dose, y=length, group=supp)) +
    geom_line(colour="darkgreen", size=1.5)

# Since supp is mapped to colour, it will automatically be used for grouping
ggplot(tg, aes(x=dose, y=length, colour=supp)) +
    geom_line(linetype="dashed") +
    geom_point(shape=22, size=3, fill="white")
```

See Also

For more information about using colors, see Chapter 12.

4.5. Changing the Appearance of Points

Problem

You want to change the appearance of the points in a line graph.

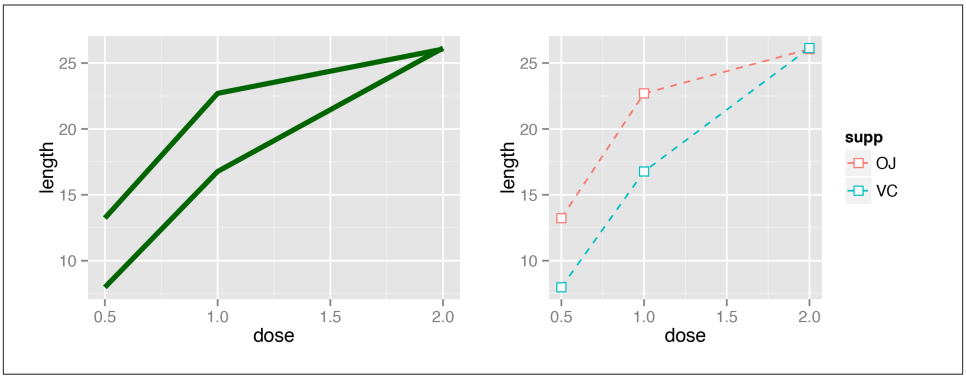


Figure 4-13. Left: line graph with constant size and color; right: with supp mapped to colour, and with points added

Solution

In `geom_point()`, set the `size`, `shape`, `colour`, and/or `fill` outside of `aes()` (the result is shown in Figure 4-14):

```
ggplot(BOD, aes(x=Time, y=demand)) +
    geom_line() +
    geom_point(size=4, shape=22, colour="darkred", fill="pink")
```

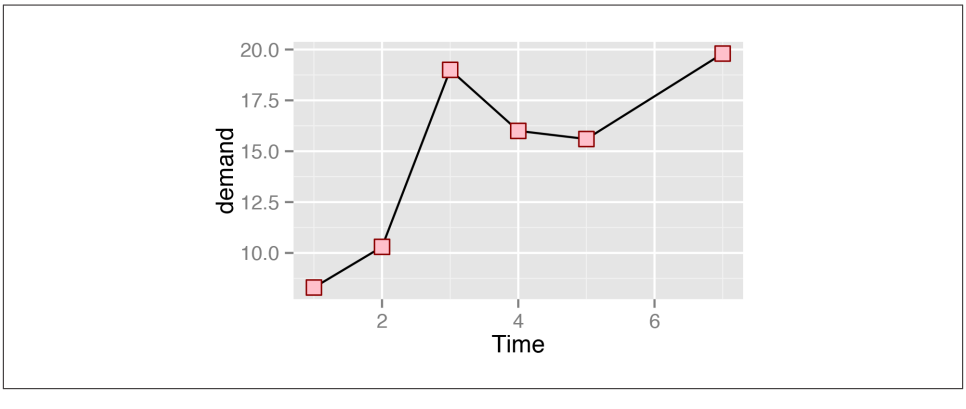


Figure 4-14. Points with custom size, shape, color, and fill

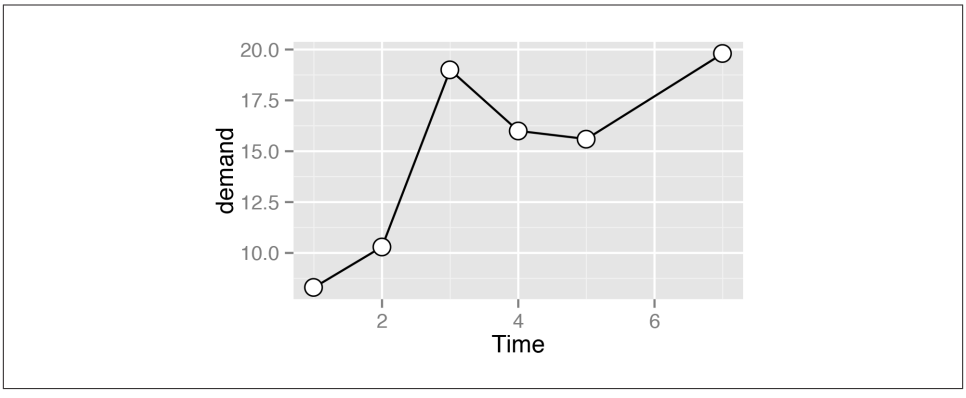


Figure 4-15. Points with a white fill

Discussion

The default `shape` for points is a solid circle, the default `size` is 2, and the default `colour` is `"black"`. The `fill` color is relevant only for some point shapes (numbered 21–25), which have separate outline and fill colors (see Recipe 5.3 for a chart of shapes). The fill color is typically `NA`, or empty; you can fill it with white to get hollow-looking circles, as shown in Figure 4-15:

```
ggplot(BOD, aes(x=Time, y=demand)) +
    geom_line() +
    geom_point(size=4, shape=21, fill="white")
```

If the points and lines have different colors, you should specify the points after the lines, so that they are drawn on top. Otherwise, the lines will be drawn on top of the points.

For multiple lines, we saw in Recipe 4.3 how to draw differently colored points for each group by mapping variables to aesthetic properties of points, inside of `aes()`. The default colors are not very appealing, so you may want to use a different palette, using `scale_colour_brewer()` or `scale_colour_manual()`. To set a single constant shape or size for all the points, as in Figure 4-16, specify `shape` or `size` outside of `aes()`:

```
# Load plyr so we can use ddply() to create the example data set
library(plyr)
# Summarize the ToothGrowth data
tg <- ddply(ToothGrowth, c("supp", "dose"), summarise, length=mean(len))

# Save the position_dodge specification because we'll use it multiple times
pd <- position_dodge(0.2)

ggplot(tg, aes(x=dose, y=length, fill=supp)) +
    geom_line(position=pd) +
    geom_point(shape=21, size=3, position=pd) +
    scale_fill_manual(values=c("black","white"))
```

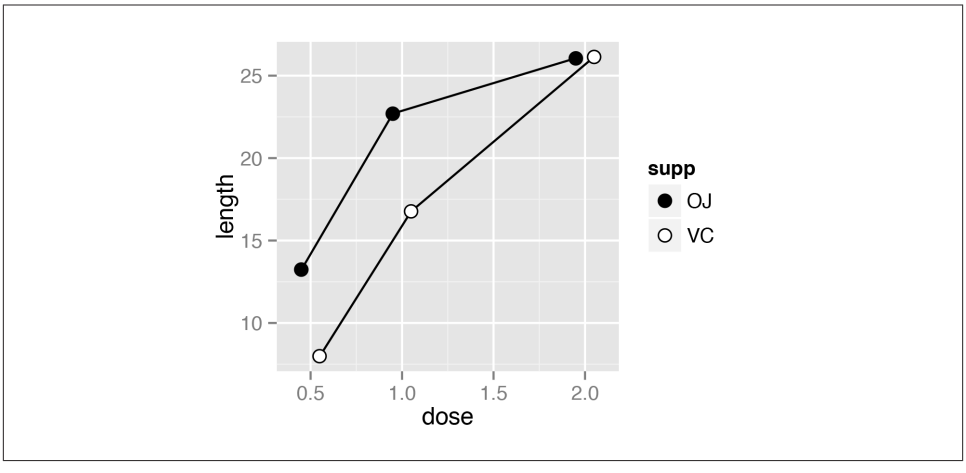


Figure 4-16. Line graph with manually specified fills of black and white, and a slight dodge

See Also

See Recipe 5.3 for more on using different shapes, and Chapter 12 for more about colors.

4.6. Making a Graph with a Shaded Area

Problem

You want to make a graph with a shaded area.

Solution

Use `geom_area()` to get a shaded area, as in Figure 4-17:

```
# Convert the sunspot.year data set into a data frame for this example
sunspotyear <- data.frame(
    Year     = as.numeric(time(sunspot.year)),
    Sunspots = as.numeric(sunspot.year)
)

ggplot(sunspotyear, aes(x=Year, y=Sunspots)) + geom_area()
```

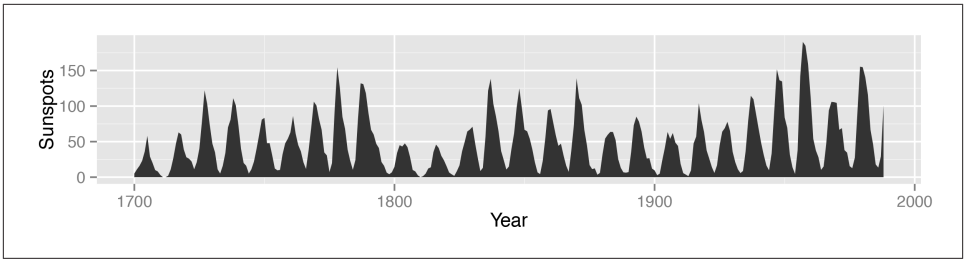


Figure 4-17. Graph with a shaded area

Discussion

By default, the area will be filled with a very dark grey and will have no outline. The color can be changed by setting `fill`. In the following example, we'll set it to `"blue"`, and we'll also make it 80% transparent by setting `alpha` to 0.2. This makes it possible to see the grid lines through the area, as shown in Figure 4-18. We'll also add an outline, by setting `colour`:

```
ggplot(sunspotyear, aes(x=Year, y=Sunspots)) +
    geom_area(colour="black", fill="blue", alpha=.2)
```

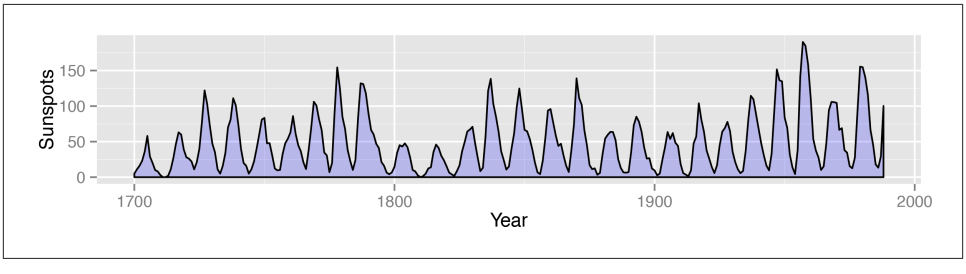


Figure 4-18. Graph with a semitransparent shaded area and an outline

Having an outline around the entire area might not be desirable, because it puts a vertical line at the beginning and end of the shaded area, as well as one along the bottom. To avoid this issue, we can draw the area without an outline (by not specifying `colour`), and then layer a `geom_line()` on top, as shown in Figure 4-19:

```
ggplot(sunspotyear, aes(x=Year, y=Sunspots)) +
    geom_area(fill="blue", alpha=.2) +
    geom_line()
```

See Also

See Chapter 12 for more on choosing colors.

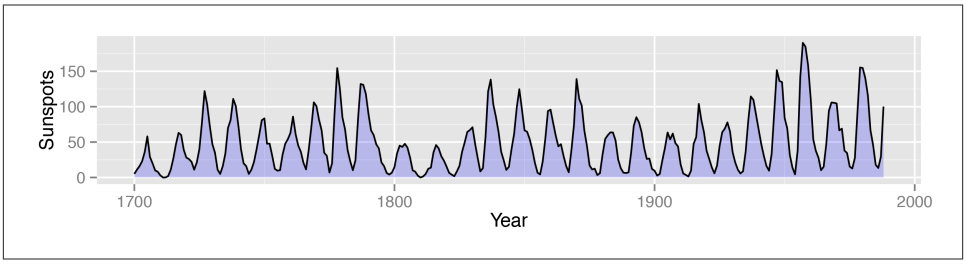


Figure 4-19. Line graph with a line just on top, using geom_line()

4.7. Making a Stacked Area Graph

Problem

You want to make a stacked area graph.

Solution

Use `geom_area()` and map a factor to `fill` (Figure 4-20):

```
library(gcookbook) # For the data set

ggplot(uspopage, aes(x=Year, y=Thousands, fill=AgeGroup)) + geom_area()
```

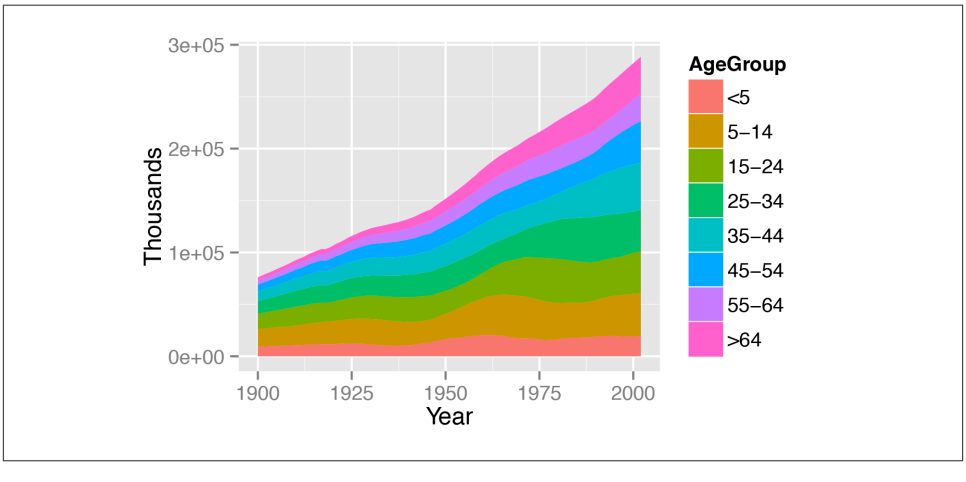


Figure 4-20. Stacked area graph

Discussion

The sort of data that is plotted with a stacked area chart is often provided in a wide format, but `ggplot2()` requires data to be in long format. To convert it, see Recipe 15.19.

In the example here, we used the `uspopage` data set:

```
uspopage
```

```
 Year AgeGroup Thousands
 1900       <5       9181
 1900     5-14      16966
 1900    15-24      14951
 1900    25-34      12161
 1900    35-44       9273
 1900    45-54       6437
 1900    55-64       4026
 1900      >64       3099
 1901       <5       9336
 1901     5-14      17158
...
```

The default order of legend items is the opposite of the stacking order. The legend can be reversed by setting the breaks in the scale. This version of the chart (Figure 4-21) reverses the legend order, changes the palette to a range of blues, and adds thin (`size=.2`) lines between each area. It also makes the filled areas semitransparent (`alpha=.4`), so that it is possible to see the grid lines through them:

```
ggplot(uspopage, aes(x=Year, y=Thousands, fill=AgeGroup)) +
    geom_area(colour="black", size=.2, alpha=.4) +
    scale_fill_brewer(palette="Blues", breaks=rev(levels(uspopage$AgeGroup)))
```

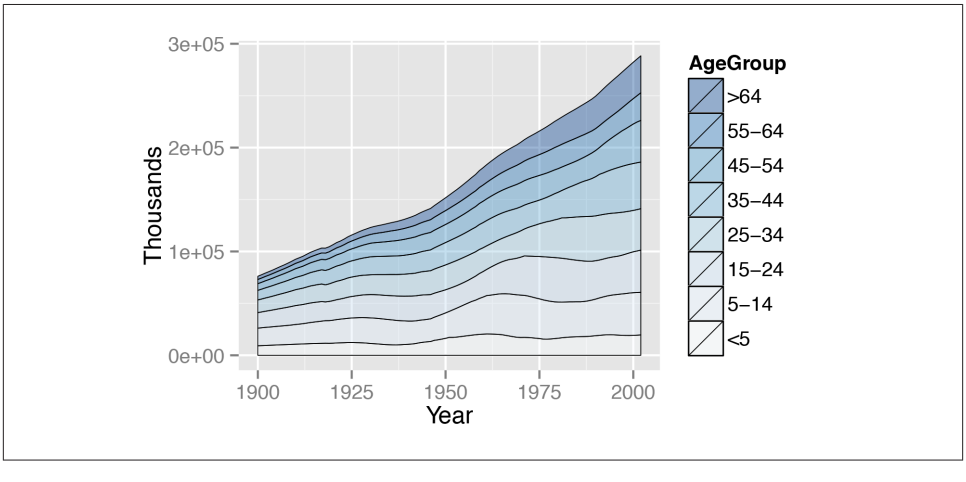


Figure 4-21. Reversed legend order, lines, and a different palette

To reverse the stacking order, we'll put `order=desc(AgeGroup)` inside of `aes()` (Figure 4-22):

```
library(plyr) # For the desc() function
ggplot(uspopage, aes(x=Year, y=Thousands, fill=AgeGroup, order=desc(AgeGroup))) +
    geom_area(colour="black", size=.2, alpha=.4) +
    scale_fill_brewer(palette="Blues")
```

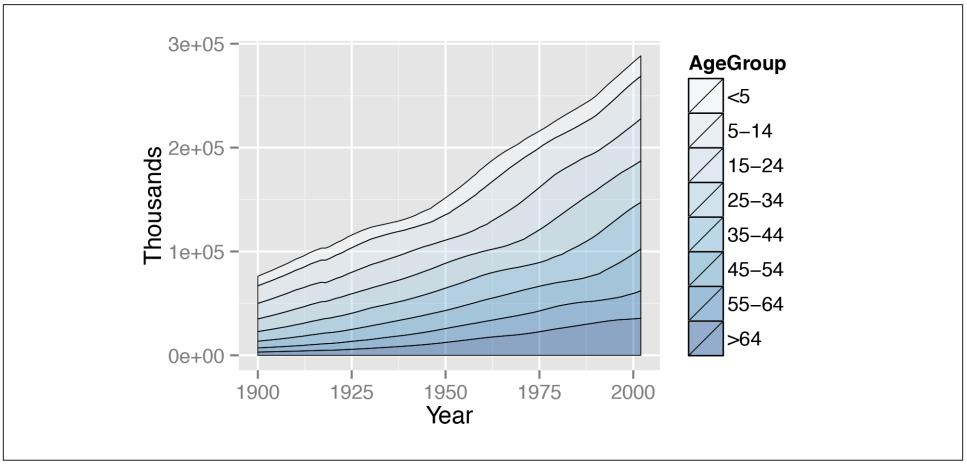


Figure 4-22. Reversed stacking order

Since each filled area is drawn with a polygon, the outline includes the left and right sides. This might be distracting or misleading. To get rid of it (Figure 4-23), first draw the stacked areas *without* an outline (by leaving `colour` as the default `NA` value), and then add a `geom_line()` on top:

```
ggplot(uspopage, aes(x=Year, y=Thousands, fill=AgeGroup, order=desc(AgeGroup))) +
    geom_area(colour=NA, alpha=.4) +
    scale_fill_brewer(palette="Blues") +
    geom_line(position="stack", size=.2)
```

See Also

See Recipe 15.19 for more on converting data from wide to long format.

For more on reordering factor levels, see Recipe 15.8.

See Chapter 12 for more on choosing colors.

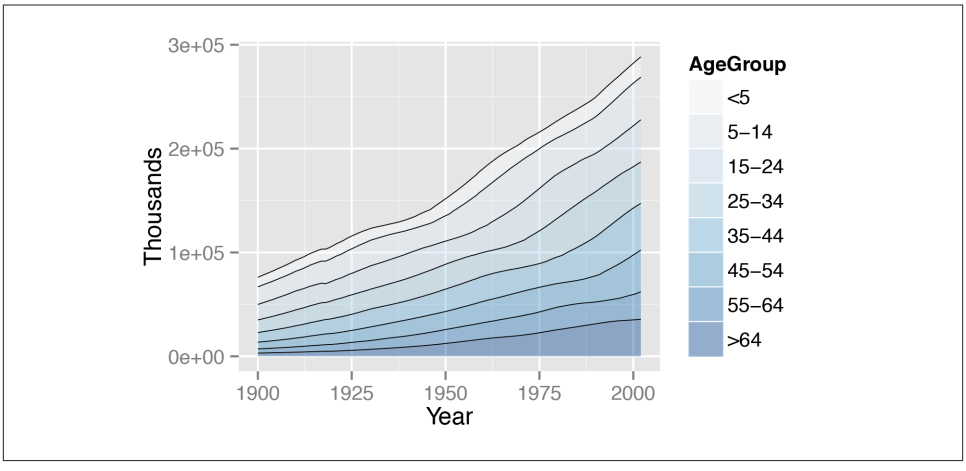


Figure 4-23. No lines on the left and right of the graph

4.8. Making a Proportional Stacked Area Graph

Problem

You want to make a stacked area graph with the overall height scaled to a constant value.

Solution

First, calculate the proportions. In this example, we'll use `ddply()` to break `uspopage` into groups by `Year`, then calculate a new column, `Percent`. This value is the `Thousands` for each row, divided by the sum of `Thousands` for each `Year` group, multiplied by 100 to get a percent value:

```
library(gcookbook) # For the data set
library(plyr)      # For the ddply() function

# Convert Thousands to Percent
uspopage_prop <- ddply(uspopage, "Year", transform,
                       Percent = Thousands / sum(Thousands) * 100)
```

Once we've calculated the proportions, plotting is the same as with a regular stacked area graph (Figure 4-24):

```
ggplot(uspopage_prop, aes(x=Year, y=Percent, fill=AgeGroup)) +
    geom_area(colour="black", size=.2, alpha=.4) +
    scale_fill_brewer(palette="Blues", breaks=rev(levels(uspopage$AgeGroup)))
```

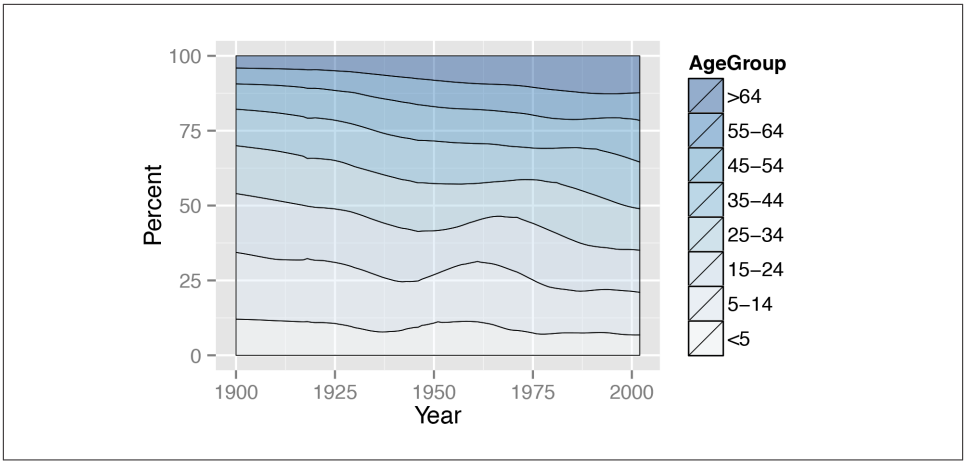


Figure 4-24. A proportional stacked area graph

Discussion

Let's take a closer look at the data and how it was summarized:

```
uspopage
```

```
 Year AgeGroup Thousands
 1900       <5      9181
 1900     5-14     16966
 1900    15-24     14951
 1900    25-34     12161
 1900    35-44      9273
 1900    45-54      6437
 1900    55-64      4026
 1900      >64      3099
 1901       <5      9336
 1901     5-14     17158
...
```

We'll use `ddply()` to split it into separate data frames for each value of `Year`, then apply the `transform()` function to each piece and calculate the `Percent` for each piece. Then `ddply()` puts all the data frames back together:

```
uspopage_prop <- ddply(uspopage, "Year", transform,
                       Percent = Thousands / sum(Thousands) * 100)
```

```
 Year AgeGroup Thousands   Percent
 1900       <5      9181 12.065340
 1900     5-14     16966 22.296107
 1900    15-24     14951 19.648067
 1900    25-34     12161 15.981549
 1900    35-44      9273 12.186243
```

```
 1900    45-54      6437  8.459274
 1900    55-64      4026  5.290825
 1900      >64      3099  4.072594
 1901       <5      9336 12.033409
 1901     5-14     17158 22.115385
...
```

See Also

For more on summarizing data by groups, see Recipe 15.17.

4.9. Adding a Confidence Region

Problem

You want to add a confidence region to a graph.

Solution

Use `geom_ribbon()` and map values to `ymin` and `ymax`.

In the `climate` data set, `Anomaly10y` is a 10-year running average of the deviation (in Celsius) from the average 1950–1980 temperature, and `Unc10y` is the 95% confidence interval. We'll set `ymax` and `ymin` to `Anomaly10y` plus or minus `Unc10y` (Figure 4-25):

```
library(gcookbook) # For the data set

# Grab a subset of the climate data
clim <- subset(climate, Source == "Berkeley",
               select=c("Year", "Anomaly10y", "Unc10y"))

clim

Year Anomaly10y Unc10y
 1800     -0.435  0.505
 1801     -0.453  0.493
 1802     -0.460  0.486
...
 2003      0.869  0.028
 2004      0.884  0.029

# Shaded region
ggplot(clim, aes(x=Year, y=Anomaly10y)) +
    geom_ribbon(aes(ymin=Anomaly10y-Unc10y, ymax=Anomaly10y+Unc10y),
                alpha=0.2) +
    geom_line()
```

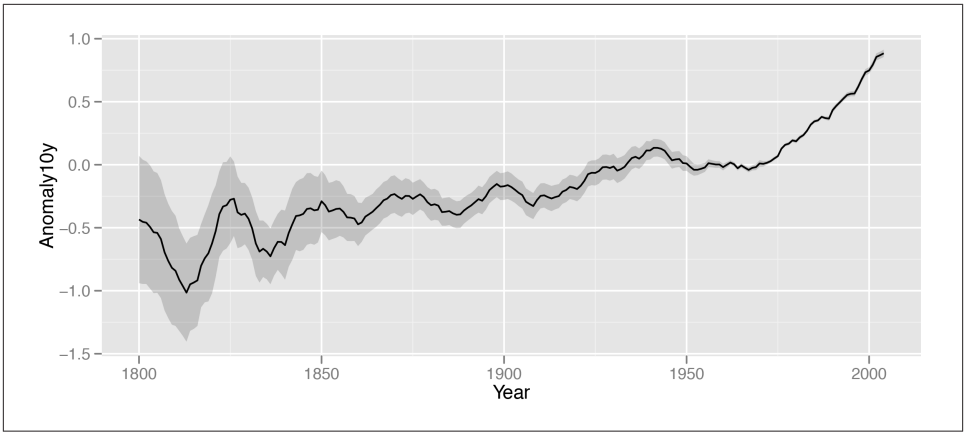


Figure 4-25. A line graph with a shaded confidence region

The shaded region is actually a very dark grey, but it is mostly transparent. The transparency is set with `alpha=0.2`, which makes it 80% transparent.

Discussion

Notice that the `geom_ribbon()` is before `geom_line()`, so that the line is drawn on top of the shaded region. If the reverse order were used, the shaded region could obscure the line. In this particular case that wouldn't be a problem since the shaded region is mostly transparent, but it would be a problem if the shaded region were opaque.

Instead of a shaded region, you can also use dotted lines to represent the upper and lower bounds (Figure 4-26):

```
# With a dotted line for upper and lower bounds
ggplot(clim, aes(x=Year, y=Anomaly10y)) +
    geom_line(aes(y=Anomaly10y-Unc10y), colour="grey50", linetype="dotted") +
    geom_line(aes(y=Anomaly10y+Unc10y), colour="grey50", linetype="dotted") +
    geom_line()
```

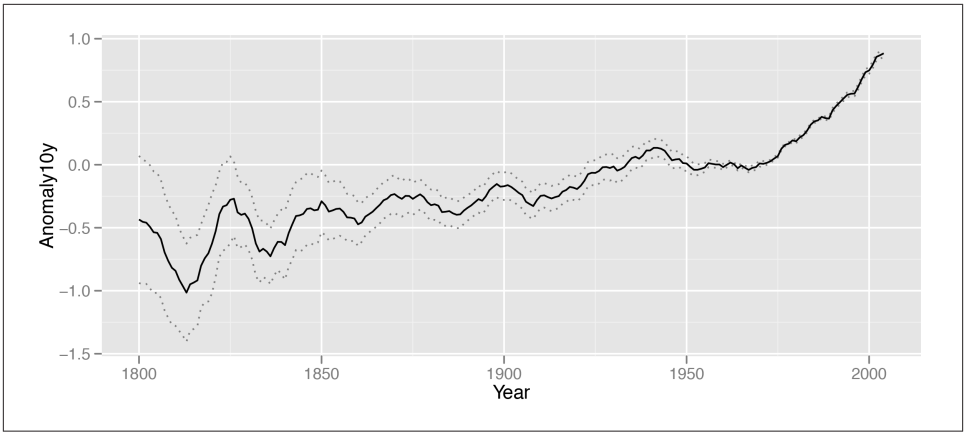


Figure 4-26. A line graph with dotted lines representing a confidence region

Shaded regions can represent things other than confidence regions, such as the difference between two values, for example.

In the area graphs in Recipe 4.7, the *y* range of the shaded area goes from `0` to `y`. Here, it goes from `ymin` to `ymax`.

CHAPTER 5

Scatter Plots

Scatter plots are used to display the relationship between two continuous variables. In a scatter plot, each observation in a data set is represented by a point. Often, a scatter plot will also have a line showing the predicted values based on some statistical model. This is easy to do with R and ggplot2, and can help to make sense of data when the trends aren't immediately obvious just by looking at it.

With large data sets, it can be problematic to plot every single observation because the points will be overplotted, obscuring one another. When this happens, you'll probably want to summarize the data before displaying it. We'll also see how to do that in this chapter.

5.1. Making a Basic Scatter Plot

Problem

You want to make a scatter plot.

Solution

Use `geom_point()`, and map one variable to `x` and one to `y`.

In the `heightweight` data set, there are a number of columns, but we'll only use two in this example (Figure 5-1):

```
library(gcookbook) # For the data set

# List the two columns we'll use
heightweight[, c("ageYear", "heightIn")]
```

```
 ageYear heightIn
   11.92     56.3
```

```
   12.92     62.3
   12.75     63.3
...
   13.92     62.0
   12.58     59.3
```

```
ggplot(heightweight, aes(x=ageYear, y=heightIn)) + geom_point()
```

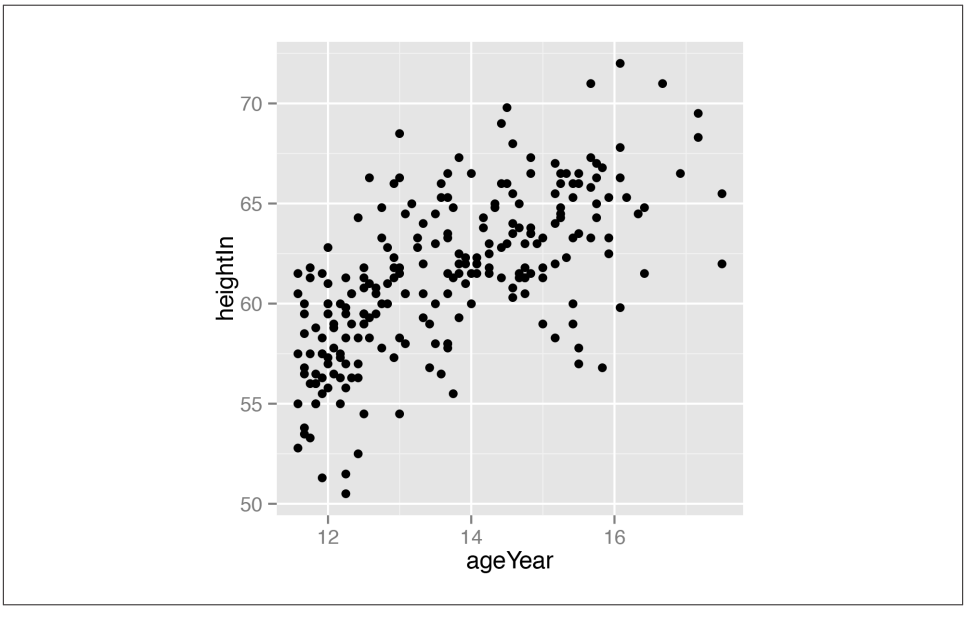


Figure 5-1. A basic scatter plot

Discussion

To use different shapes in a scatter plot, set `shape`. A common alternative to the default solid circles (shape #16) is hollow ones (#21), as seen in Figure 5-2 (left):

```
ggplot(heightweight, aes(x=ageYear, y=heightIn)) + geom_point(shape=21)
```

The size of the points can be controlled with `size`. The default value of `size` is 2. The following will set `size=1.5`, for smaller points (Figure 5-2, right):

```
ggplot(heightweight, aes(x=ageYear, y=heightIn)) + geom_point(size=1.5)
```

When displaying to screen or outputting to bitmap files like PNG, the default solid circle shape (#16) can result in aliased (jagged-looking) edges on some platforms. An alternative is to use shape 19, which is also a solid circle, but comes out smooth in more cases (see Figure 5-3). See Recipe 14.5 for more about anti-aliased output.

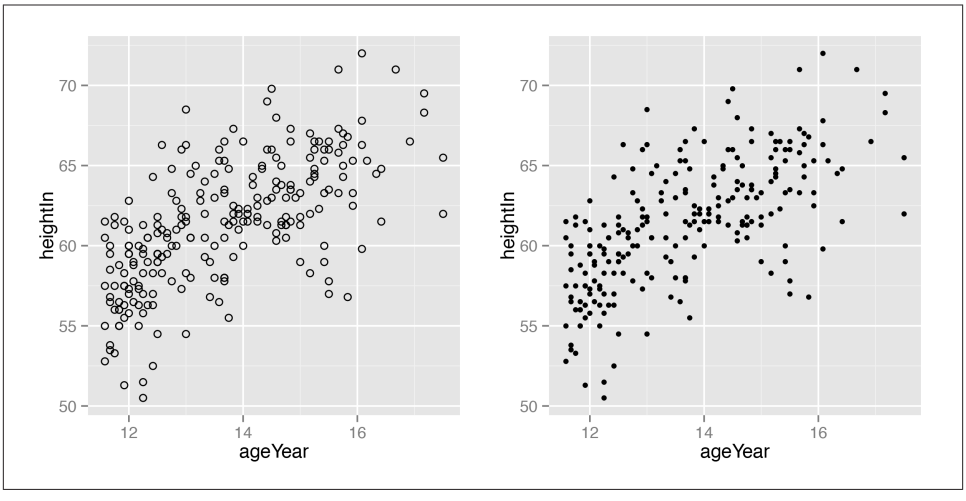


Figure 5-2. Left: scatter plot with hollow circles (shape 21); right: with smaller points

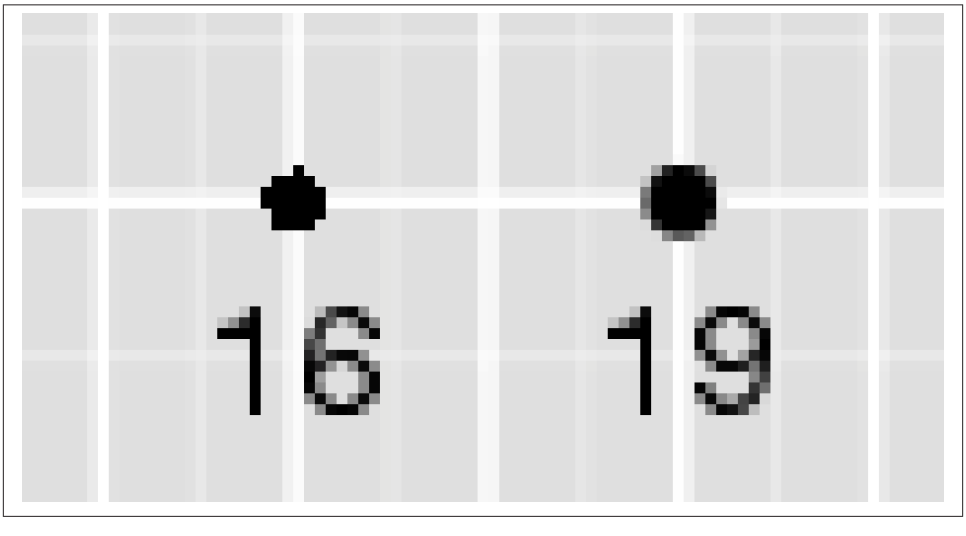


Figure 5-3. Point shapes 16 and 19, as they appear with some bitmap output devices

5.2. Grouping Data Points by a Variable Using Shape or Color

Problem

You want to group points by some variable, using shape or color.

Solution

Map the grouping variable to `shape` or `colour`. In the `heightweight` data set, there are many columns, but we'll only use three of them in this example:

```
library(gcookbook) # For the data set
# Show the three columns we'll use
heightweight[, c("sex", "ageYear", "heightIn")]
```

```
 sex ageYear heightIn
   f   11.92     56.3
   f   12.92     62.3
   f   12.75     63.3
...
   m   13.92     62.0
   m   12.58     59.3
```

We can group points on the variable `sex`, by mapping `sex` to one of the aesthetics `colour` or `shape` (Figure 5-4):

```
ggplot(heightweight, aes(x=ageYear, y=heightIn, colour=sex)) + geom_point()

ggplot(heightweight, aes(x=ageYear, y=heightIn, shape=sex)) + geom_point()
```

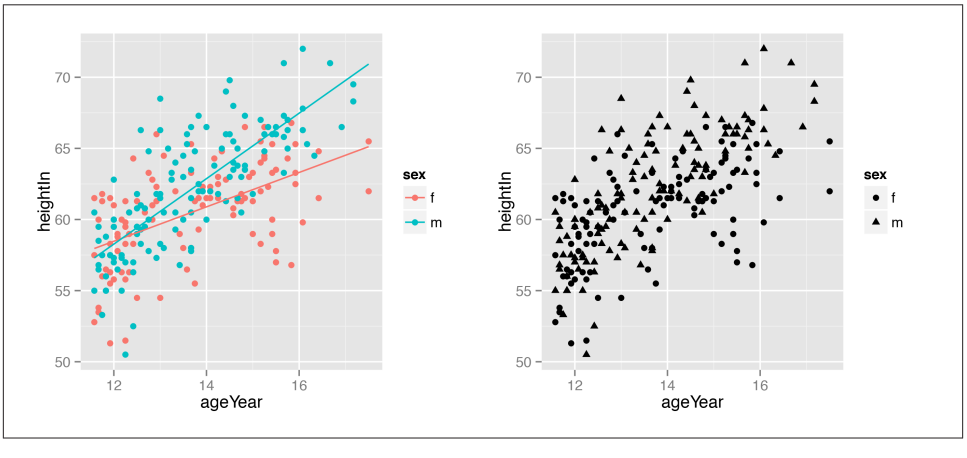


Figure 5-4. Grouping points by a variable mapped to colour (left), and to shape (right)

Discussion

The grouping variable must be categorical—in other words, a factor or character vector. If it is stored as a vector of numeric values, it should be converted to a factor before it is used as a grouping variable.

It is possible to map a variable to both `shape` and `colour`, or, if you have multiple grouping variables, to map different variables to them. Here, we'll map `sex` to `shape` and `colour` (Figure 5-5, left):

```
ggplot(heightweight, aes(x=ageYear, y=heightIn, shape=sex, colour=sex)) +
    geom_point()
```

The default shapes and colors may not be very appealing. Other shapes can be used with `scale_shape_manual()`, and other colors can be used with `scale_colour_brewer()` or `scale_colour_manual()`.

This will set different shapes and colors for the grouping variables (Figure 5-5, right):

```
ggplot(heightweight, aes(x=ageYear, y=heightIn, shape=sex, colour=sex)) +
    geom_point() +
    scale_shape_manual(values=c(1,2)) +
    scale_colour_brewer(palette="Set1")
```

Figure 5-5. Left: mapping to both shape and colour; right: with manually set shapes and colors

See Also

To use different shapes, see Recipe 5.3.

For more on using different colors, see Chapter 12.

5.3. Using Different Point Shapes

Problem

You want to use point shapes that are different from the defaults.

Solution

If you want to set the shape of all the points (Figure 5-6), specify the `shape` in `geom_point()`:

```
library(gcookbook) # For the data set

ggplot(heightweight, aes(x=ageYear, y=heightIn)) + geom_point(shape=3)
```

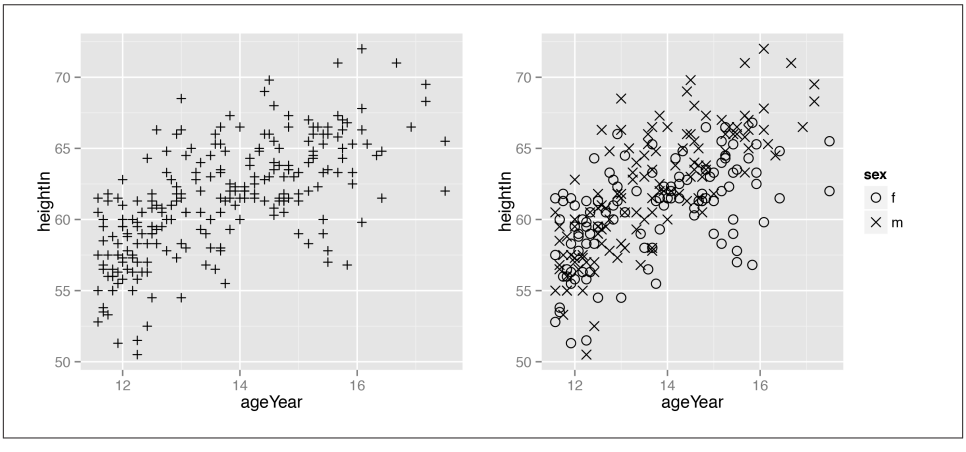


Figure 5-6. Left: scatter plot with the shape aesthetic set to a custom value; right: with a variable mapped to shape, using a custom shape palette

If you have mapped a variable to `shape`, use `scale_shape_manual()` to change the shapes:

```
# Use slightly larger points and use a shape scale with custom values
ggplot(heightweight, aes(x=ageYear, y=heightIn, shape=sex)) +
    geom_point(size=3) + scale_shape_manual(values=c(1, 4))
```

Discussion

Figure 5-7 shows the shapes that are available in R graphics. Some of the point shapes (1–14) have just an outline, some (15–20) are solid, and some (21–25) have an outline and fill that can be controlled separately. (You can also use characters for points.)

For shapes 1–20, the color of the entire point—even the points that are solid—is controlled by the `colour` aesthetic. For shapes 21–25, the outline is controlled by `colour` and the fill is controlled by `fill`.

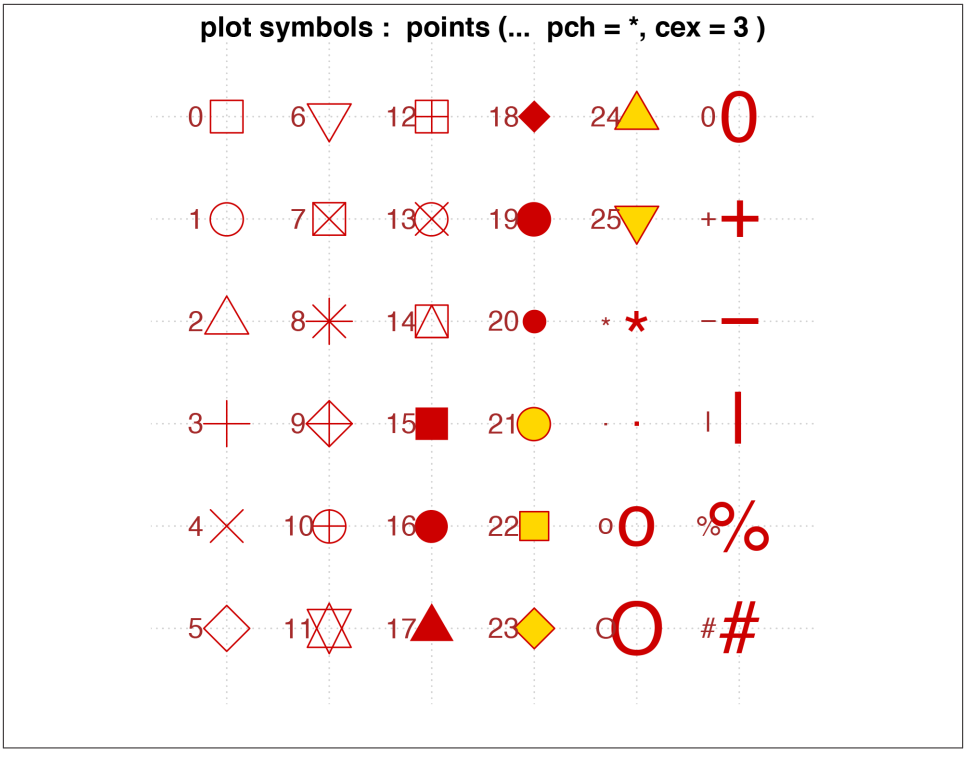


Figure 5-7. Shapes in R

It's possible to have the shape represent one variable and the fill (empty or solid) represent another variable. This is done a little indirectly, by choosing shapes that have both `colour` and `fill`, and a color palette that includes `NA` and another color (the `NA` will result in a hollow shape). For example, we'll take the `heightweight` data set and add another column that indicates whether the child weighed 100 pounds or more (Figure 5-8):

```
# Make a copy of the data
hw <- heightweight
# Categorize into <100 and >=100 groups
hw$weightGroup <- cut(hw$weightLb, breaks=c(-Inf, 100, Inf),
                      labels=c("< 100", ">= 100"))

# Use shapes with fill and color, and use colors that are empty (NA) and
# filled
ggplot(hw, aes(x=ageYear, y=heightIn, shape=sex, fill=weightGroup)) +
    geom_point(size=2.5) +
    scale_shape_manual(values=c(21, 24)) +
    scale_fill_manual(values=c(NA, "black"),
      guide=guide_legend(override.aes=list(shape=21)))
```

Figure 5-8. A variable mapped to shape and another mapped to fill

See Also

For more on using different colors, see Chapter 12.

For more information about recoding a continuous variable to a categorical one, see Recipe 15.14.

5.4. Mapping a Continuous Variable to Color or Size

Problem

You want to represent a third continuous variable using color or size.

Solution

Map the continuous variable to `size` or `colour`. In the `heightweight` data set, there are many columns, but we'll only use four of them in this example:

```
library(gcookbook) # For the data set

# List the four columns we'll use
heightweight[, c("sex", "ageYear", "heightIn", "weightLb")]
```

```
 sex ageYear heightIn weightLb
```

```
   f   11.92     56.3     85.0
   f   12.92     62.3    105.0
   f   12.75     63.3    108.0
...
   m   13.92     62.0    107.5
   m   12.58     59.3     87.0
```

The basic scatter plot in Recipe 5.1 shows the relationship between the continuous variables `ageYear` and `heightIn`. To represent a third continuous variable, `weightLb`, we must map it to another aesthetic property. We can map it to `colour` or `size`, as shown in Figure 5-9:

```
ggplot(heightweight, aes(x=ageYear, y=heightIn, colour=weightLb)) + geom_point()

ggplot(heightweight, aes(x=ageYear, y=heightIn, size=weightLb)) + geom_point()
```

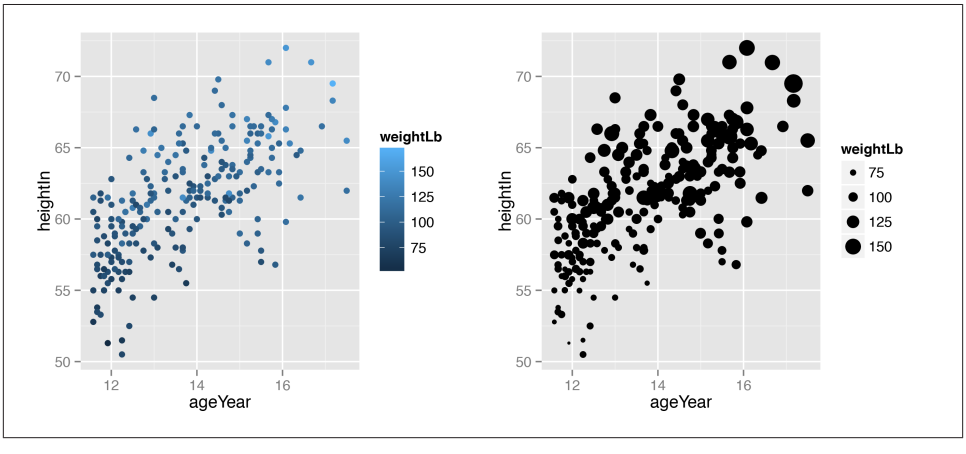


Figure 5-9. Left: a continuous variable mapped to colour; right: mapped to size

Discussion

A basic scatter plot shows the relationship between two continuous variables: one mapped to the x-axis, and one to the y-axis. When there are more than two continuous variables, they must be mapped to other aesthetics: size and/or color.

We can easily perceive small differences in spatial position, so we can interpret the variables mapped to *x* and *y* coordinates with high accuracy. We aren't very good at perceiving small differences in size and color, though, so we will interpret variables mapped to these aesthetic attributes with a much lower accuracy. When you map a variable to one of these properties, it should be one where accuracy is not very important for interpretation.

When a variable is mapped to `size`, the results can be perceptually misleading. The largest dots in Figure 5-9 have about 36 times the area of the smallest ones, but they

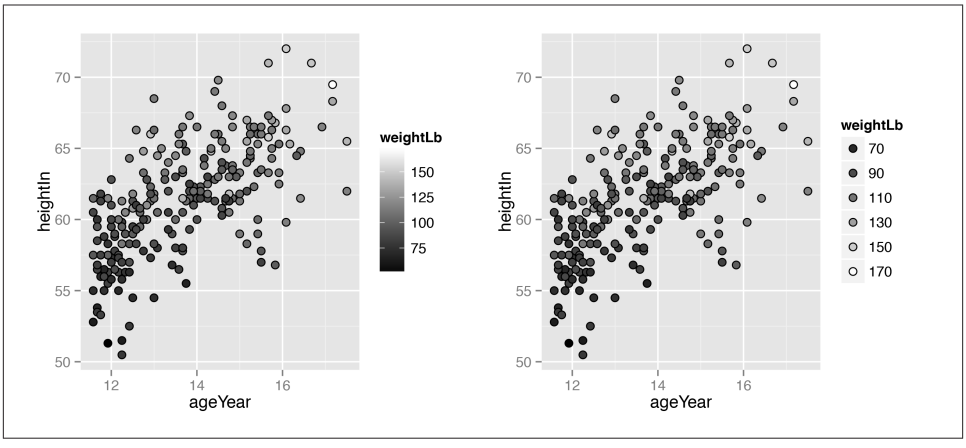


Figure 5-10. Left: outlined points with a continuous variable mapped to fill; right: with a discrete legend instead of continuous colorbar

represent only about 3.5 times the weight. If it is important for the sizes to proportionally represent the quantities, you can change the range of sizes. By default the sizes of points go from 1 to 6 mm. You could reduce the range to, say, 2 to 5 mm, with `scale_size_continuous(range=c(2, 5))`. However, the point size numbers don't map linearly to diameter or area, so this still won't give a very accurate representation of the values. (See Recipe 5.12 for details on making the area of dots proportional to the value.)

When it comes to color, there are actually two aesthetic attributes that can be used: `colour` and `fill`. For most point shapes, you use `colour`. However, shapes 21–25 have an outline with a solid region in the middle where the color is controlled by `fill`. These outlined shapes can be useful when using a color scale with light colors, as in Figure 5-10, because the outline sets them off from the background. In this example, we also set the fill gradient to go from black to white and make the points larger so that the fill is easier to see:

```
ggplot(heightweight, aes(x=weightLb, y=heightIn, fill=ageYear)) +
    geom_point(shape=21, size=2.5) +
    scale_fill_gradient(low="black", high="white")

# Using guide_legend() will result in a discrete legend instead of a colorbar
ggplot(heightweight, aes(x=weightLb, y=heightIn, fill=ageYear)) +
    geom_point(shape=21, size=2.5) +
    scale_fill_gradient(low="black", high="white", breaks=12:17,
                        guide=guide_legend())
```

When we map a continuous variable to an aesthetic, that doesn't prevent us from mapping a categorical variable to other aesthetics. In Figure 5-11, we'll map `weightLb` to

`size`, and also map `sex` to `colour`. Because there is a fair amount of overplotting, we'll make the points 50% transparent by setting `alpha=.5`. We'll also use `scale_size_area()` to make the area of the points proportional to the value (see Recipe 5.12), and change the color palette to one that is a little more appealing:

```
ggplot(heightweight, aes(x=ageYear, y=heightIn, size=weightLb, colour=sex)) +
    geom_point(alpha=.5) +
    scale_size_area() +     # Make area proportional to numeric value
    scale_colour_brewer(palette="Set1")
```

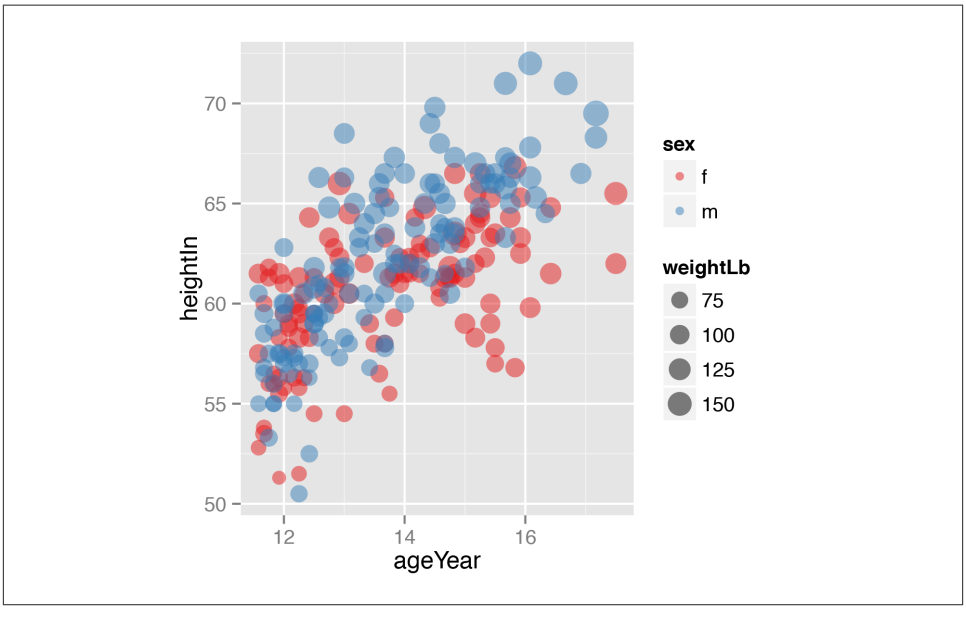


Figure 5-11. Continuous variable mapped to size and categorical variable mapped to colour

When a variable is mapped to `size`, it's a good idea to *not* map a variable to `shape`. This is because it is difficult to compare the sizes of different shapes; for example, a size 4 triangle could appear larger than a size 3.5 circle. Also, some of the shapes really are different sizes: shapes 16 and 19 are both circles, but at any given numeric size, shape 19 circles are visually larger than shape 16 circles.

See Also

To use different colors from the default, see Recipe 12.6.

See Recipe 5.12 for creating a balloon plot.

5.5. Dealing with Overplotting

Problem

You have many points and they obscure each other.

Solution

With large data sets, the points in a scatter plot may obscure each other and prevent the viewer from accurately assessing the distribution of the data. This is called *overplotting*. If the amount of overplotting is low, you may be able to alleviate it by using smaller points, or by using a different shape (like shape 1, a hollow circle) through which other points can be seen. Figure 5-2 in Recipe 5.1 demonstrates both of these solutions.

If there's a high degree of overplotting, there are a number of possible solutions:

- Make the points semitransparent
- Bin the data into rectangles (better for quantitative analysis)
- Bin the data into hexagons
- Use box plots

Discussion

The scatter plot in Figure 5-12 contains about 54,000 points. They are heavily overplotted, making it impossible to get a sense of the relative density of points in different areas of the graph:

```
sp <- ggplot(diamonds, aes(x=carat, y=price))

sp + geom_point()
```

We can make the points semitransparent using `alpha`, as in Figure 5-13. Here, we'll make them 90% transparent and then 99% transparent, by setting `alpha=.1` and `alpha=.01`:

```
sp + geom_point(alpha=.1)

sp + geom_point(alpha=.01)
```

Now we can see that there are vertical bands at nice round values of `carats`, indicating that diamonds tend to be cut to those sizes. Still, the data is so dense that even when the points are 99% transparent, much of the graph appears solid black, and the data distribution is still somewhat obscured.

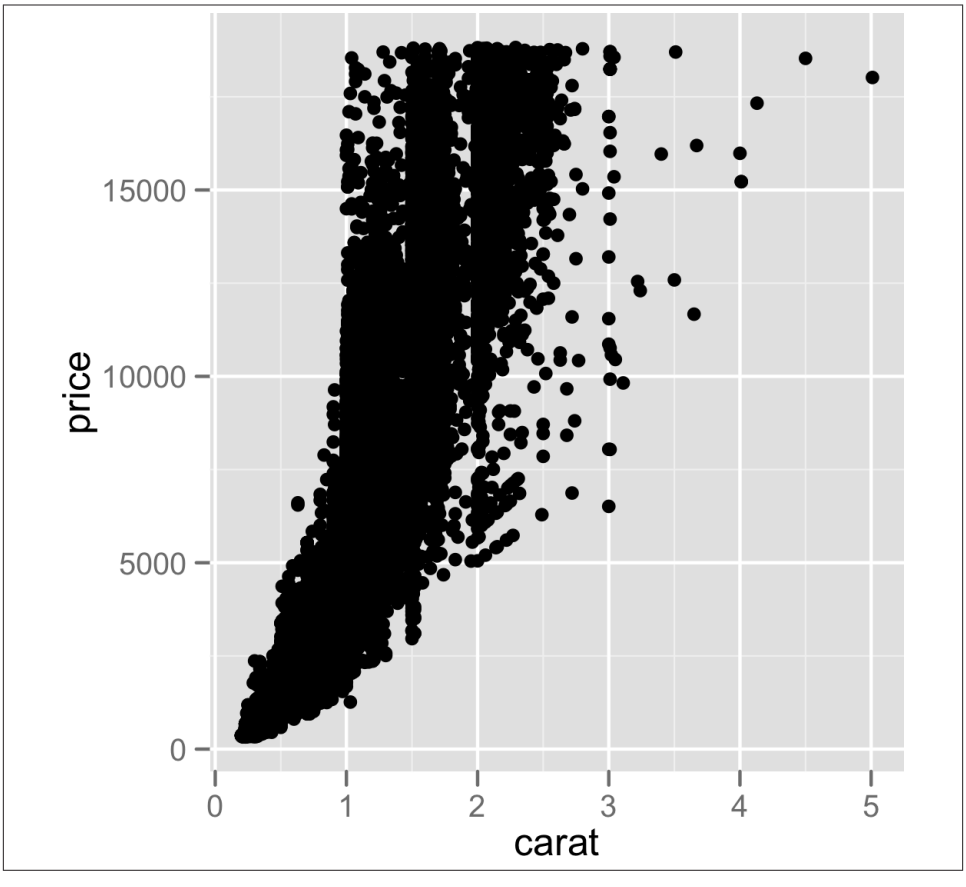


Figure 5-12. Overplotting, with about 54,000 points

For most graphs, vector formats (such as PDF, EPS, and SVG) result in smaller output files than bitmap formats (such as TIFF and PNG). But in cases where there are tens of thousands of points, vector output files can be very large and slow to render—the scatter plot here with 99% transparent points is 1.5 MB! In these cases, high-resolution bitmaps will be smaller and faster to display on computer screens. See Chapter 14 for more information.

Another solution is to *bin* the points into rectangles and map the density of the points to the fill color of the rectangles, as shown in Figure 5-14. With the binned visualization, the vertical bands are barely visible. The density of points in the lower-left corner is much greater, which tells us that the vast majority of diamonds are small and inexpensive.

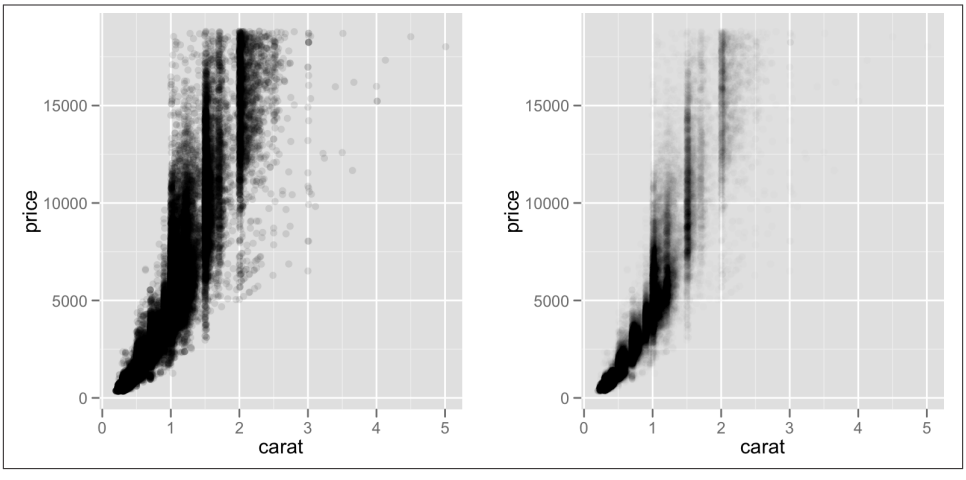


Figure 5-13. Left: semitransparent points with alpha=.1; right: with alpha=.01

By default, `stat_bin_2d()` divides the space into 30 groups in the *x* and *y* directions, for a total of 900 bins. In the second version, we increase the number of bins with `bins=50`.

The default colors are somewhat difficult to distinguish because they don't vary much in luminosity. In the second version we set the colors by using `scale_fill_gradient()` and specifying the `low` and `high` colors. By default, the legend doesn't show an entry for the lowest values. This is because the range of the color scale starts not from zero, but from the smallest nonzero quantity in a bin—probably 1, in this case. To make the legend show a zero (as in Figure 5-14, right), we can manually set the range from 0 to the maximum, 6000, using `limits` (Figure 5-14, left):

```
sp + stat_bin2d()

sp + stat_bin2d(bins=50) +
    scale_fill_gradient(low="lightblue", high="red", limits=c(0, 6000))
```

Another alternative is to bin the data into hexagons instead of rectangles, with `stat_binhex()` (Figure 5-15). It works just like `stat_bin2d()`. To use it, you must first install the hexbin package, with `install.packages("hexbin")`:

```
library(hexbin)

sp + stat_binhex() +
    scale_fill_gradient(low="lightblue", high="red",
                        limits=c(0, 8000))
```

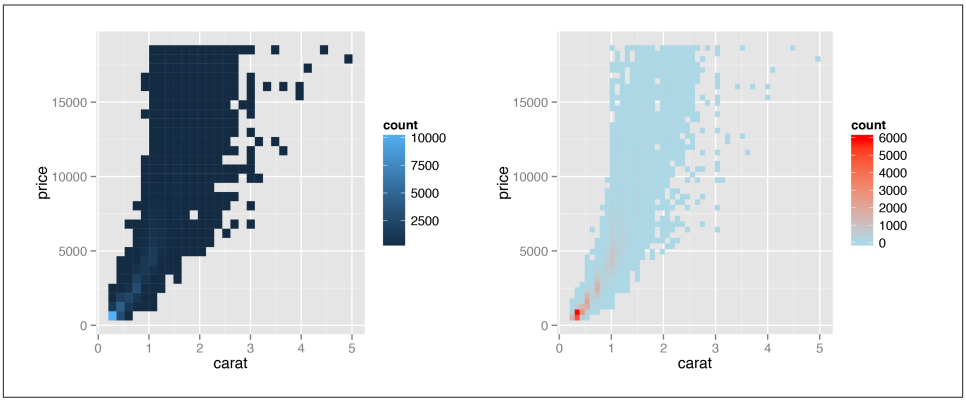


Figure 5-14. Left: binning data with stat_bin2d(); right: with more bins, manually specified colors, and legend breaks

```
sp + stat_binhex() +
    scale_fill_gradient(low="lightblue", high="red",
                        breaks=c(0, 250, 500, 1000, 2000, 4000, 6000),
                        limits=c(0, 6000))
```

For both of these methods, if you manually specify the range, and there is a bin that falls outside that range because it has too many or too few points, that bin will show up as grey rather than the color at the high or low end of the range, as seen in the graph on the right in Figure 5-15.

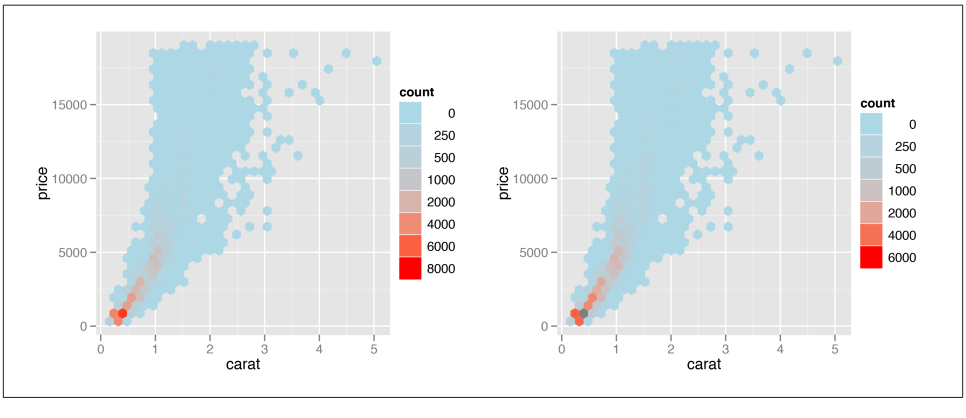


Figure 5-15. Left: binning data with stat_binhex(); right: cells outside of the range shows in grey

Overplotting can also occur when the data is *discrete* on one or both axes, as shown in Figure 5-16. In these cases, you can randomly *jitter* the points with `position_jitter()`. By default the amount of jitter is 40% of the resolution of the data in each direction, but these amounts can be controlled with `width` and `height`:

```
sp1 <- ggplot(ChickWeight, aes(x=Time, y=weight))

sp1 + geom_point()

sp1 + geom_point(position="jitter")
# Could also use geom_jitter(), which is equivalent

sp1 + geom_point(position=position_jitter(width=.5, height=0))
```

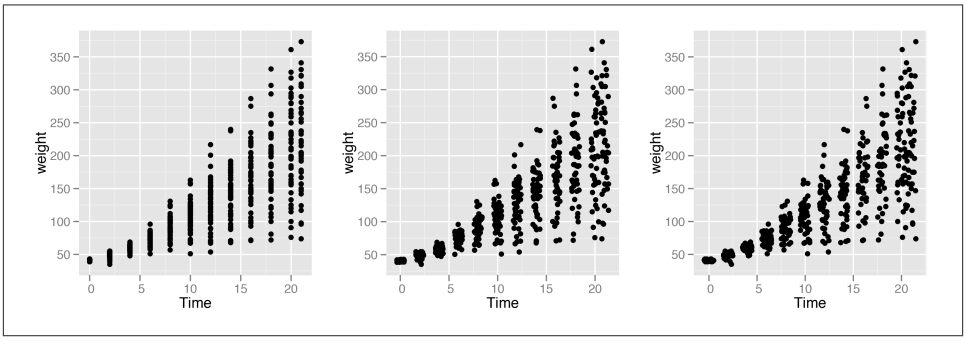


Figure 5-16. Left: data with a discrete x variable; middle: jittered; right: jittered horizontally only

When the data has one discrete axis and one continuous axis, it might make sense to use box plots, as shown in Figure 5-17. This will convey a different story than a standard scatter plot because it will obscure the *number* of data points at each location on the discrete axis. This may be problematic in some cases, but desirable in others.

With the `ChickWeights` data, the x-axis is conceptually discrete, but since it is stored numerically, `ggplot()` doesn't know how to group the data for each box. If you don't tell it how to group the data, you get a result like the graph on the right in Figure 5-17. To tell it how to group the data, use `aes(group=...)`. In this case, we'll group by each distinct value of `Time`:

```
sp1 + geom_boxplot(aes(group=Time))
```

See Also

Instead of binning the data, it may be useful to display a 2D density estimate. To do this, see Recipe 6.12.

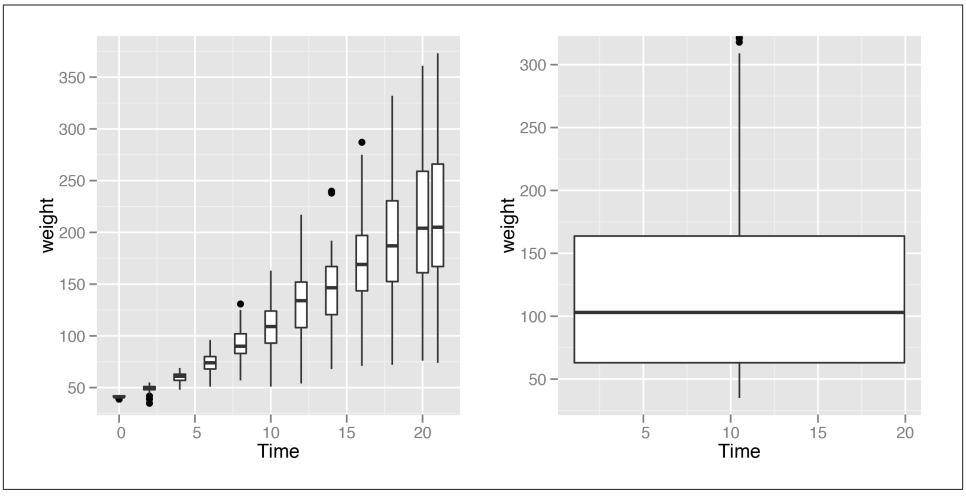


Figure 5-17. Left: grouping into box plots; right: what happens if you don't specify groups

5.6. Adding Fitted Regression Model Lines

Problem

You want to add lines from a fitted regression model to a scatter plot.

Solution

To add a linear regression line to a scatter plot, add `stat_smooth()` and tell it to use `method=lm`. This instructs it to fit the data with the `lm()` (linear model) function. First we'll save the base plot object in `sp`, then we'll add different components to it:

```
library(gcookbook) # For the data set

# The base plot
sp <- ggplot(heightweight, aes(x=ageYear, y=heightIn))

sp + geom_point() + stat_smooth(method=lm)
```

By default, `stat_smooth()` also adds a 95% confidence region for the regression fit. The confidence interval can be changed by setting `level`, or it can be disabled with `se=FALSE` (Figure 5-18):

```
# 99% confidence region
sp + geom_point() + stat_smooth(method=lm, level=0.99)

# No confidence region
sp + geom_point() + stat_smooth(method=lm, se=FALSE)
```

The default color of the fit line is blue. This can be change by setting `colour`. As with any other line, the attributes `linetype` and `size` can also be set. To emphasize the line, you can make the dots less prominent by setting `colour` (Figure 5-18, bottom right):

```
sp + geom_point(colour="grey60") +
    stat_smooth(method=lm, se=FALSE, colour="black")
```

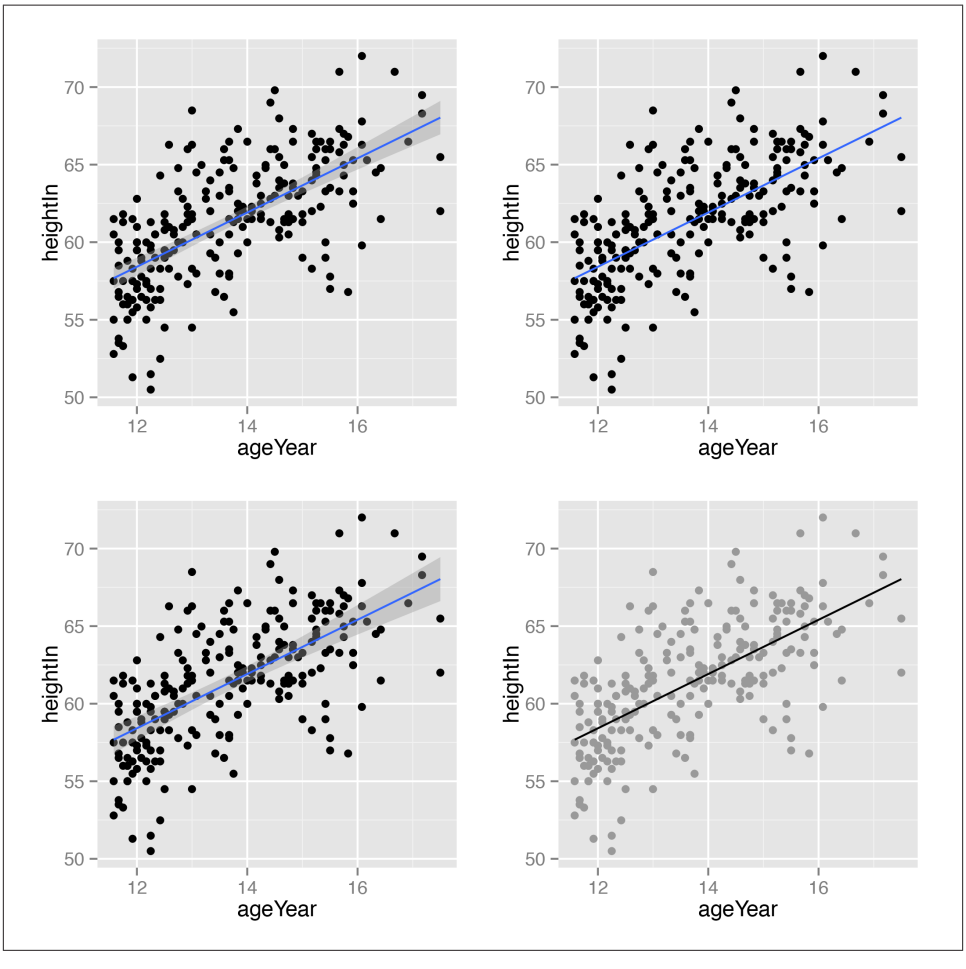


Figure 5-18. Top left: an lm fit with the default 95% confidence region; bottom left: a 99% confidence region; top right: no confidence region; bottom right: in black with grey points

Discussion

The linear regression line is not the only way of fitting a model to the data—in fact, it's not even the default. If you add `stat_smooth()` without specifying the method, it will use a `loess` (locally weighted polynomial) curve, as shown in Figure 5-19. Both of these will have the same result:

```
sp + geom_point(colour="grey60") + stat_smooth()
sp + geom_point(colour="grey60") + stat_smooth(method=loess)
```

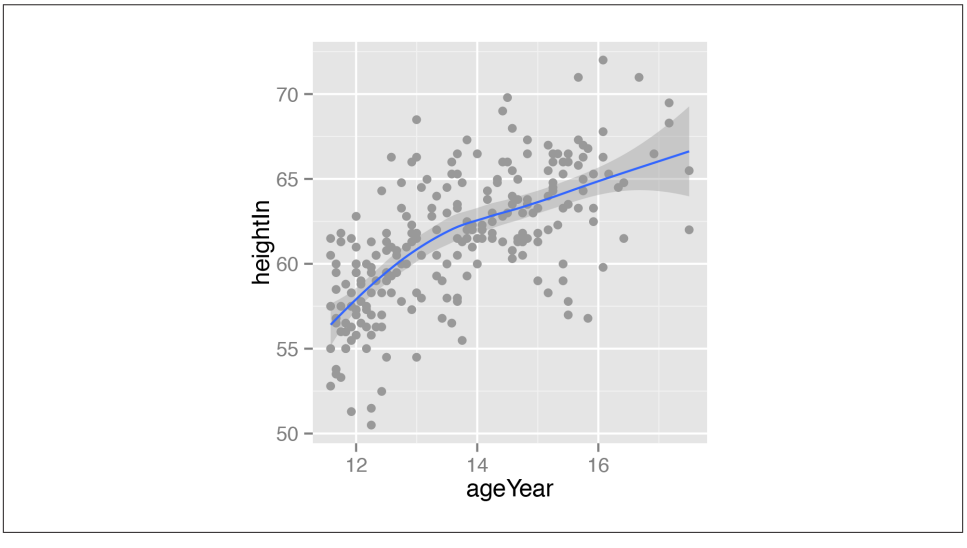


Figure 5-19. A LOESS fit

Additional parameters can be passed along to the `loess()` function by just passing them to `stat_smooth()`.

Another common type of model fit is a logistic regression. Logistic regression isn't appropriate for the `heightweight` data set, but it's perfect for the `biopsy` data set in the `MASS` library. In this data set, there are nine different measured attributes of breast cancer biopsies, as well as the class of the tumor, which is either `benign` or `malignant`. To prepare the data for logistic regression, we must convert the factor `class`, with the levels `benign` and `malignant`, to a vector with numeric values of 0 and 1. We'll make a copy of the `biopsy` data frame, then store the numeric coded class in a column called `classn`:

```
library(MASS) # For the data set

b <- biopsy

b$classn[b$class=="benign"]    <- 0
b$classn[b$class=="malignant"] <- 1
```

```
b
```

```
      ID V1 V2 V3 V4 V5 V6 V7 V8 V9     class classn
 1000025  5  1  1  1  2  1  3  1  1    benign      0
 1002945  5  4  4  5  7 10  3  2  1    benign      0
 1015425  3  1  1  1  2  2  3  1  1    benign      0
...
  897471  4  8  6  4  3  4 10  6  1 malignant      1
  897471  4  8  8  5  4  5 10  4  1 malignant      1
```

Although there are many attributes we could examine, for this example we'll just look at the relationship of `V1` (clump thickness) and the class of the tumor. Because there is a large degree of overplotting, we'll jitter the points and make them semitransparent (`alpha=0.4`), hollow (`shape=21`), and slightly smaller (`size=1.5`). Then we'll add a fitted logistic regression line (Figure 5-20) by telling `stat_smooth()` to use the `glm()` function with the option `family=binomial`:

```
ggplot(b, aes(x=V1, y=classn)) +
    geom_point(position=position_jitter(width=0.3, height=0.06), alpha=0.4,
               shape=21, size=1.5) +
    stat_smooth(method=glm, family=binomial)
```

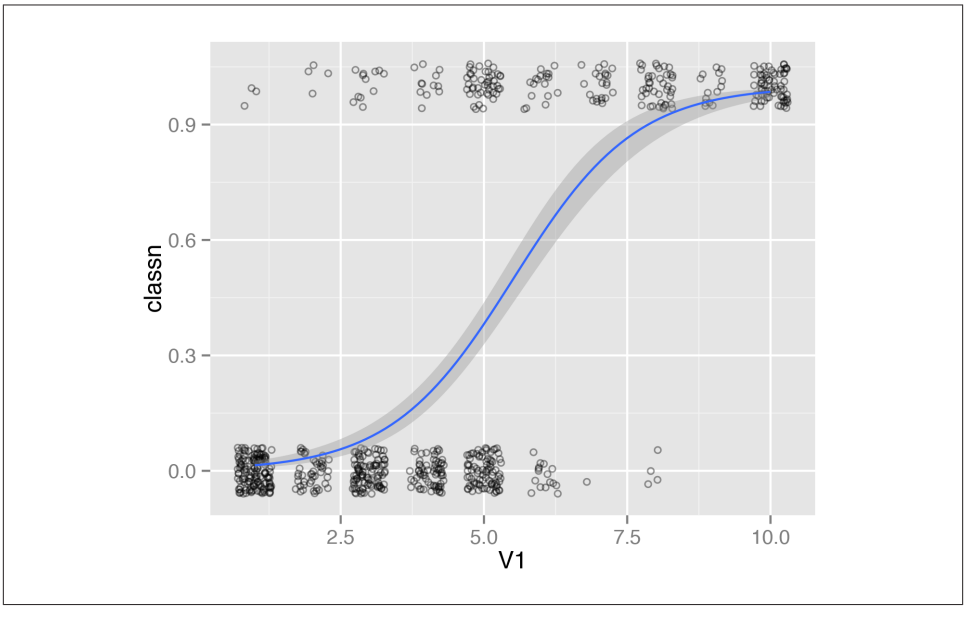


Figure 5-20. A logistic model

If your scatter plot has points grouped by a factor, using `colour` or `shape`, one fit line will be drawn for each group. First we'll make the base plot object `sps`, then we'll add the `loess` lines to it. We'll also make the points less prominent by making them semi-transparent, using `alpha=.4` (Figure 5-21):

```
sps <- ggplot(heightweight, aes(x=ageYear, y=heightIn, colour=sex)) +
       geom_point() +
       scale_colour_brewer(palette="Set1")

sps + geom_smooth()
```

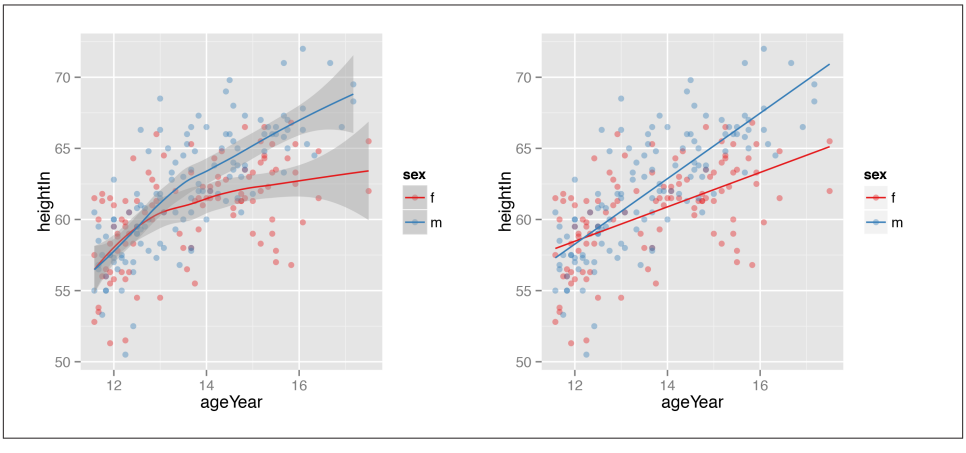


Figure 5-21. Left: LOESS fit lines for each group; right: extrapolated linear fit lines

Notice that the blue line, for males, doesn't run all the way to the right side of the graph. There are two reasons for this. The first is that, by default, `stat_smooth()` limits the prediction to within the range of the predictor data (on the x-axis). The second is that even if it extrapolates, the `loess()` function only offers prediction within the *x* range of the data.

If you want the lines to extrapolate from the data, as shown in the right-hand image of Figure 5-21, you must use a model method that allows extrapolation, like `lm()`, and pass `stat_smooth()` the option `fullrange=TRUE`:

```
sps + geom_smooth(method=lm, se=FALSE, fullrange=TRUE)
```

In this example with the `heightweight` data set, the default settings for `stat_smooth()` (with LOESS and no extrapolation) make more sense than the extrapolated linear predictions, because we don't grow linearly and we don't grow forever.

5.7. Adding Fitted Lines from an Existing Model

Problem

You have already created a fitted regression model object for a data set, and you want to plot the lines for that model.

Solution

Usually the easiest way to overlay a fitted model is to simply ask `stat_smooth()` to do it for you, as described in Recipe 5.6. Sometimes, however, you may want to create the model yourself and then add it to your graph. This allows you to be sure that the model you're using for other calculations is the same one that you see.

In this example, we'll build a quadratic model using `lm()` with `ageYear` as a predictor of `heightIn`. Then we'll use the `predict()` function and find the predicted values of `heightIn` across the range of values for the predictor, `ageYear`:

```
library(gcookbook) # For the data set

model <- lm(heightIn ~ ageYear + I(ageYear^2), heightweight)
model

Call:
lm(formula = heightIn ~ ageYear + I(ageYear^2), data = heightweight)

Coefficients:
 (Intercept)       ageYear  I(ageYear^2)
    -10.3136        8.6673       -0.2478


# Create a data frame with ageYear column, interpolating across range
xmin <- min(heightweight$ageYear)
xmax <- max(heightweight$ageYear)
predicted <- data.frame(ageYear=seq(xmin, xmax, length.out=100))

# Calculate predicted values of heightIn
predicted$heightIn <- predict(model, predicted)
predicted

  ageYear heightIn
 11.58000 56.82624
 11.63980 57.00047
 ...
 17.44020 65.47875
 17.50000 65.47933
```

We can now plot the data points along with the values predicted from the model (as you'll see in Figure 5-22):

```
sp <- ggplot(heightweight, aes(x=ageYear, y=heightIn)) +
      geom_point(colour="grey40")

sp + geom_line(data=predicted, size=1)
```

Discussion

Any model object can be used, so long as it has a corresponding `predict()` method. For example, `lm` has `predict.lm()`, `loess` has `predict.loess()`, and so on.

Adding lines from a model can be simplified by using the function `predictvals()`, defined next. If you simply pass in a model, it will do the work of finding the variable names and range of the predictor, and will return a data frame with predictor and predicted values. That data frame can then be passed to `geom_line()` to draw the fitted line, as we did earlier:

```
# Given a model, predict values of yvar from xvar
# This supports one predictor and one predicted variable
# xrange: If NULL, determine the x range from the model object. If a vector with
#   two numbers, use those as the min and max of the prediction range.
# samples: Number of samples across the x range.
# ...: Further arguments to be passed to predict()
predictvals <- function(model, xvar, yvar, xrange=NULL, samples=100, ...) {

  # If xrange isn't passed in, determine xrange from the models.
  # Different ways of extracting the x range, depending on model type
  if (is.null(xrange)) {
    if (any(class(model) %in% c("lm", "glm")))
      xrange <- range(model$model[[xvar]])
    else if (any(class(model) %in% "loess"))
      xrange <- range(model$x)
  }

  newdata <- data.frame(x = seq(xrange[1], xrange[2], length.out = samples))
  names(newdata) <- xvar
  newdata[[yvar]] <- predict(model, newdata = newdata, ...)
  newdata
}
```

With the `heightweight` data set, we'll make a linear model with `lm()` and a LOESS model with `loess()` (Figure 5-22):

```
modlinear <- lm(heightIn ~ ageYear, heightweight)

modloess  <- loess(heightIn ~ ageYear, heightweight)
```

Then we can call `predictvals()` on each model, and pass the resulting data frames to `geom_line()`:

```
lm_predicted    <- predictvals(modlinear, "ageYear", "heightIn")
loess_predicted <- predictvals(modloess, "ageYear", "heightIn")

sp + geom_line(data=lm_predicted, colour="red", size=.8) +
     geom_line(data=loess_predicted, colour="blue", size=.8)
```

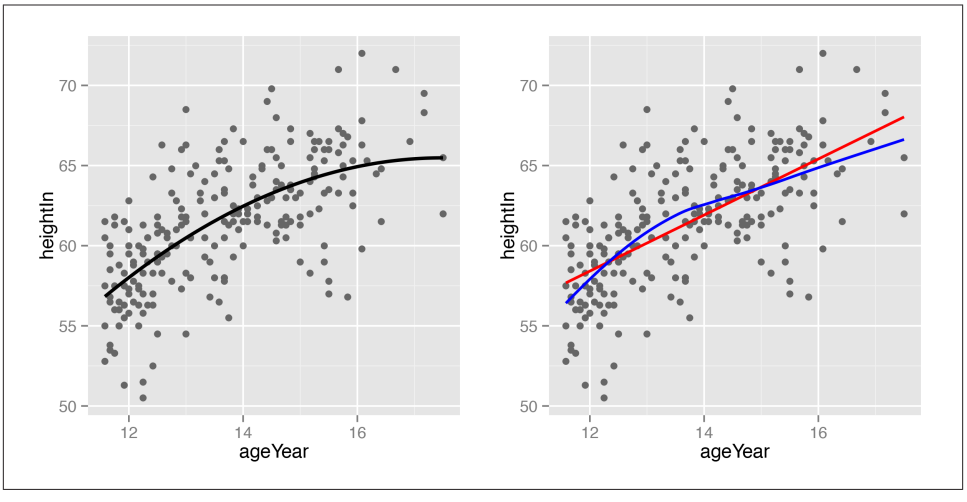


Figure 5-22. Left: a quadratic prediction line from an lm object; right: prediction lines from linear (red) and LOESS (blue) models

For `glm` models that use a nonlinear link function, you need to specify `type="response"` to the `predictvals()` function. This is because the default behavior is to return predicted values in the scale of the linear predictors, instead of in the scale of the response (*y*) variable.

To illustrate this, we'll use the `biopsy` data set from the `MASS` library. As we did in Recipe 5.6, we'll use `V1` to predict `class`. Since logistic regression uses values from 0 to 1, while `class` is a factor, we'll first have to convert `class` to 0s and 1s:

```
library(MASS) # For the data set
b <- biopsy

b$classn[b$class=="benign"]    <- 0
b$classn[b$class=="malignant"] <- 1
```

Next, we'll perform the logistic regression:

```
fitlogistic <- glm(classn ~ V1, b, family=binomial)
```

Finally, we'll make the graph with jittered points and the `fitlogistic` line. We'll make the line in a shade of blue by specifying a color in RGB values, and slightly thicker, with `size=1` (Figure 5-23):

```
# Get predicted values
glm_predicted <- predictvals(fitlogistic, "V1", "classn", type="response")

ggplot(b, aes(x=V1, y=classn)) +
    geom_point(position=position_jitter(width=.3, height=.08), alpha=0.4,
               shape=21, size=1.5) +
    geom_line(data=glm_predicted, colour="#1177FF", size=1)
```

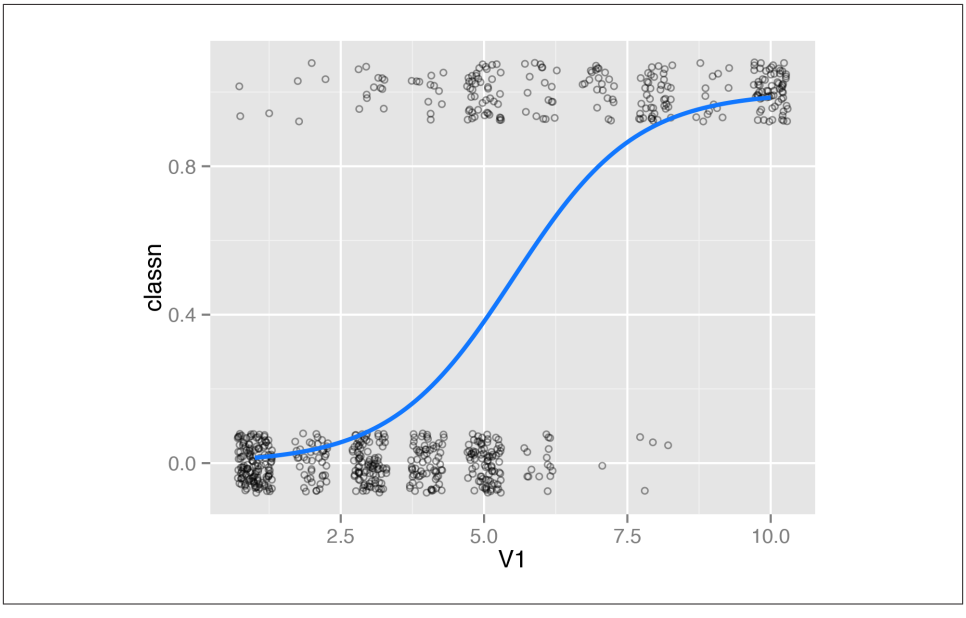


Figure 5-23. A fitted logistic model

5.8. Adding Fitted Lines from Multiple Existing Models

Problem

You have already created a fitted regression model object for a data set, and you want to plot the lines for that model.

Solution

Use the `predictvals()` function from the previous recipe along with `dlply()` and `ldply()` from the plyr package.

With the `heightweight` data set, we'll make a linear model with `lm()` for each of the levels of `sex`, and put those model objects in a list. The model building is done with a function, `make_model()`, defined here. If you pass it a data frame, it simply returns an `lm` object. The model can be customized for your data:

```
make_model <- function(data) {
    lm(heightIn ~ ageYear, data)
}
```

With this function, we can use the `dlply()` function to build a model for each subset of data. This will split the data frame into subsets by the grouping variable `sex`, and apply `make_model()` to each subset. In this case, the `heightweight` data will be split into two data frames, one for males and one for females, and `make_model()` will be run on each subset. With `dlply()`, the models are put into a list and the list is returned:

```
library(gcookbook) # For the data set
library(plyr)
models <- dlply(heightweight, "sex", .fun = make_model)

# Print out the list of two lm objects, f and m
models

$f

Call:
lm(formula = heightIn ~ ageYear, data = data)

Coefficients:
(Intercept)      ageYear
     43.963        1.209

$m

Call:
lm(formula = heightIn ~ ageYear, data = data)

Coefficients:
(Intercept)      ageYear
     30.658        2.301

attr(,"split_type")
[1] "data.frame"
attr(,"split_labels")
  sex
1   f
2   m
```

Now that we have the list of model objects, we can run `predictvals()` to get predicted values from each model, using the `ldply()` function:

```
predvals <- ldply(models, .fun=predictvals, xvar="ageYear", yvar="heightIn")
predvals
```

```
 sex  ageYear heightIn
   f 11.58000 57.96250
   f 11.63980 58.03478
   f 11.69960 58.10707
 ...
   m 17.38040 70.64912
   m 17.44020 70.78671
   m 17.50000 70.92430
```

Finally, we can plot the data with the predicted values (Figure 5-24):

```
ggplot(heightweight, aes(x=ageYear, y=heightIn, colour=sex)) +
    geom_point() + geom_line(data=predvals)
```

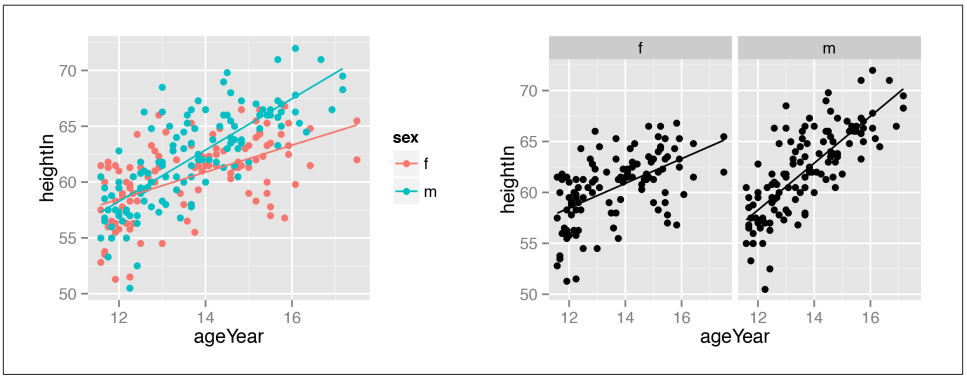


Figure 5-24. Left: predictions from two separate lm objects, one for each subset of data; right: with facets

Discussion

The `dlply()` and `ldply()` calls are used for splitting the data into parts, running functions on those parts, and then reassembling the output.

With the preceding code, the *x* range of the predicted values for each group spans the *x* range of each group, and no further; for the males, the prediction line stops at the oldest male, while for females, the prediction line continues further right, to the oldest female. To form prediction lines that have the same *x* range across all groups, we can simply pass in `xrange`, like this:

```
predvals <- ldply(models, .fun=predictvals, xvar="ageYear", yvar="heightIn",
                  xrange=range(heightweight$ageYear))
```

Then we can plot it, the same as we did before:

```
ggplot(heightweight, aes(x=ageYear, y=heightIn, colour=sex)) +
    geom_point() + geom_line(data=predvals)
```

As you can see in Figure 5-25, the line for males now extends as far to the right as the line for females. Keep in mind that extrapolating past the data isn't always appropriate, though; whether or not it's justified will depend on the nature of your data and the assumptions you bring to the table.

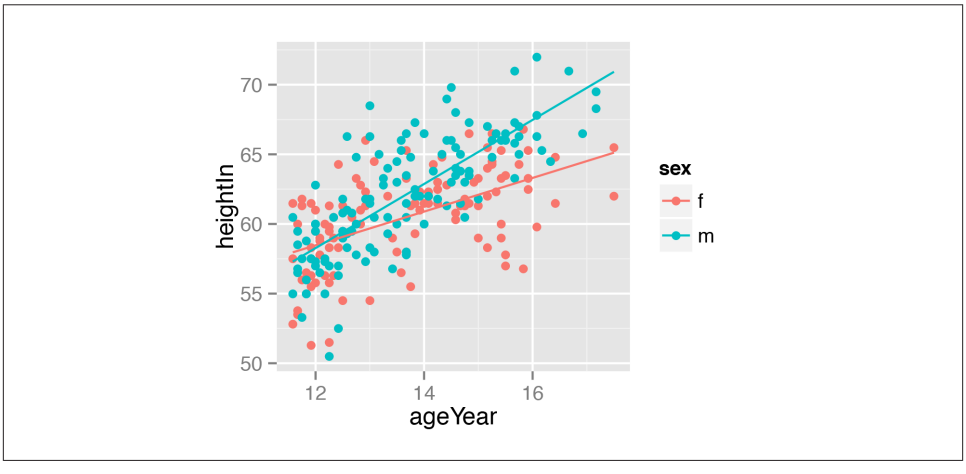


Figure 5-25. Predictions for each group extend to the full x range of all groups together

5.9. Adding Annotations with Model Coefficients

Problem

You want to add numerical information about a model to a plot.

Solution

To add simple text to a plot, simply add an annotation. In this example, we'll create a linear model and use the `predictvals()` function defined in Recipe 5.7 to create a prediction line from the model. Then we'll add an annotation:

```
library(gcookbook) # For the data set

model <- lm(heightIn ~ ageYear, heightweight)
summary(model)

Call:
lm(formula = heightIn ~ ageYear, data = heightweight)

Residuals:
```

```
    Min      1Q  Median      3Q     Max
-8.3517 -1.9006  0.1378  1.9071  8.3371

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  37.4356     1.8281   20.48   <2e-16 ***
ageYear       1.7483     0.1329   13.15   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.989 on 234 degrees of freedom
Multiple R-squared: 0.4249, Adjusted R-squared: 0.4225
F-statistic: 172.9 on 1 and 234 DF,  p-value: < 2.2e-16
```

This shows that the r^2 value is 0.4249. We'll create a graph and manually add the text using `annotate()` (Figure 5-26):

```
# First generate prediction data
pred <- predictvals(model, "ageYear", "heightIn")
sp <- ggplot(heightweight, aes(x=ageYear, y=heightIn)) + geom_point() +
    geom_line(data=pred)

sp + annotate("text", label="r^2=0.42", x=16.5, y=52)
```

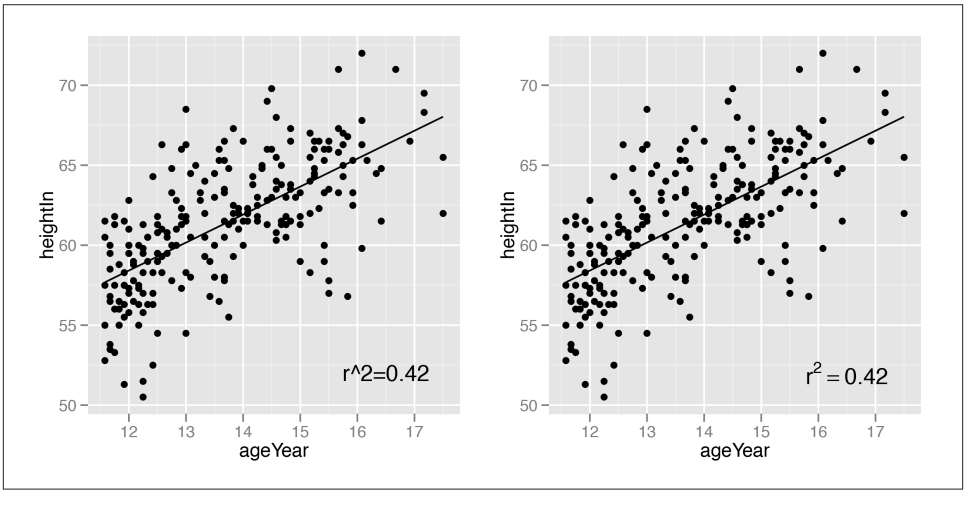


Figure 5-26. Left: plain text; right: math expression

Instead of using a plain text string, it's also possible to enter formulas using R's math expression syntax, by setting `parse=TRUE`:

```
sp + annotate("text", label="r^2 == 0.42", parse = TRUE, x=16.5, y=52)
```

Discussion

Text geoms in ggplot2 do not take expression objects directly; instead, they take character strings that are turned into expressions with `parse(text="a + b")`.

If you use a math expression, the syntax must be correct for it to be a valid R expression object. You can test validity by wrapping it in `expression()` and seeing if it throws an error (make sure *not* to use quotes around the expression). In the example here, == is a valid construct in an expression to express equality, but = is not:

```
expression(r^2 == 0.42)  # Valid
```

```
expression(r^2 == 0.42)
```

```
expression(r^2 = 0.42)   # Not valid
```

```
Error: unexpected '=' in "expression(r^2 ="
```

It's possible to automatically extract values from the model object and build an expression using those values. In this example, we'll create a string that, when parsed, returns a valid expression:

```
eqn <- as.character(as.expression(
  substitute(italic(y) == a + b * italic(x) * "," ~~ italic(r)^2 ~ "=" ~ r2,
    list(a = format(coef(model)[1], digits=3),
         b = format(coef(model)[2], digits=3),
         r2 = format(summary(model)$r.squared, digits=2)
        ))))
eqn
```

```
"italic(y) == \"37.4\" + \"1.75\" * italic(x) * \",\" ~ ~italic(r)^2 ~ \"=\" ~
\"0.42\""
```

```
parse(text=eqn)  # Parsing turns it into an expression
```

```
expression(italic(y) == "37.4" + "1.75" * italic(x) * "," ~ ~italic(r)^2 ~ "=" ~
"0.42")
```

Now that we have the expression string, we can add it to the plot. In this example we'll put the text in the bottom-right corner, by setting `x=Inf` and `y=-Inf` and using horizontal and vertical adjustments so that the text all fits inside the plotting area (Figure 5-27):

```
sp + annotate("text", label=eqn, parse=TRUE, x=Inf, y=-Inf, hjust=1.1, vjust=-.5)
```

See Also

The math expression syntax in R can be a bit tricky. See Recipe 7.2 for more information.

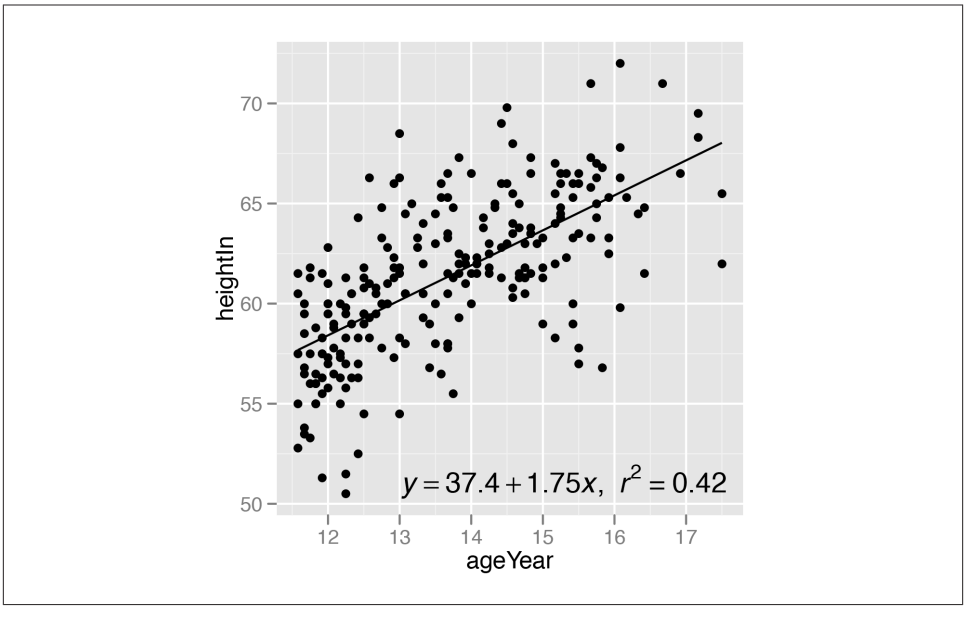


Figure 5-27. Scatter plot with automatically generated expression

5.10. Adding Marginal Rugs to a Scatter Plot

Problem

You want to add marginal rugs to a scatter plot.

Solution

Use `geom_rug()`. For this example (Figure 5-28), we'll use the `faithful` data set, which contains data about the Old Faithful geyser in two columns—`eruptions`, which is the length of each eruption, and `waiting`, which is the length of time to the next eruption:

```
ggplot(faithful, aes(x=eruptions, y=waiting)) + geom_point() + geom_rug()
```

Discussion

A marginal rug plot is essentially a one-dimensional scatter plot that can be used to visualize the distribution of data on each axis.

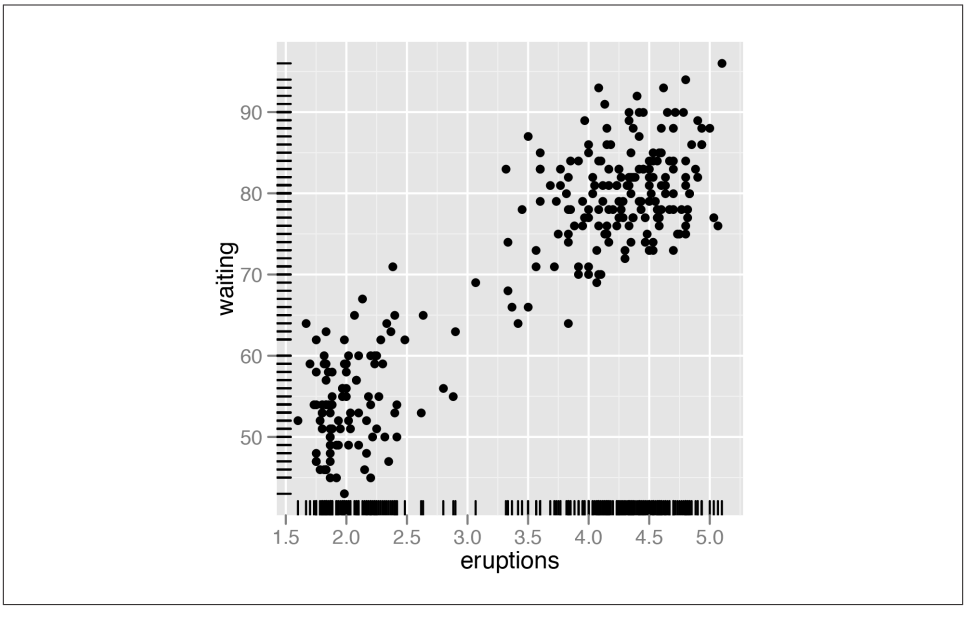


Figure 5-28. Marginal rug added to a scatter plot

In this particular data set, the marginal rug is not as informative as it could be. The resolution of the `waiting` variable is in whole minutes, and because of this, the rug lines have a lot of overplotting. To reduce the overplotting, we can jitter the line positions and make them slightly thinner by specifying `size` (Figure 5-29). This helps the viewer see the distribution more clearly:

```
ggplot(faithful, aes(x=eruptions, y=waiting)) + geom_point() +
    geom_rug(position="jitter", size=.2)
```

See Also

For more about overplotting, see Recipe 5.5.

5.11. Labeling Points in a Scatter Plot

Problem

You want to add labels to points in a scatter plot.

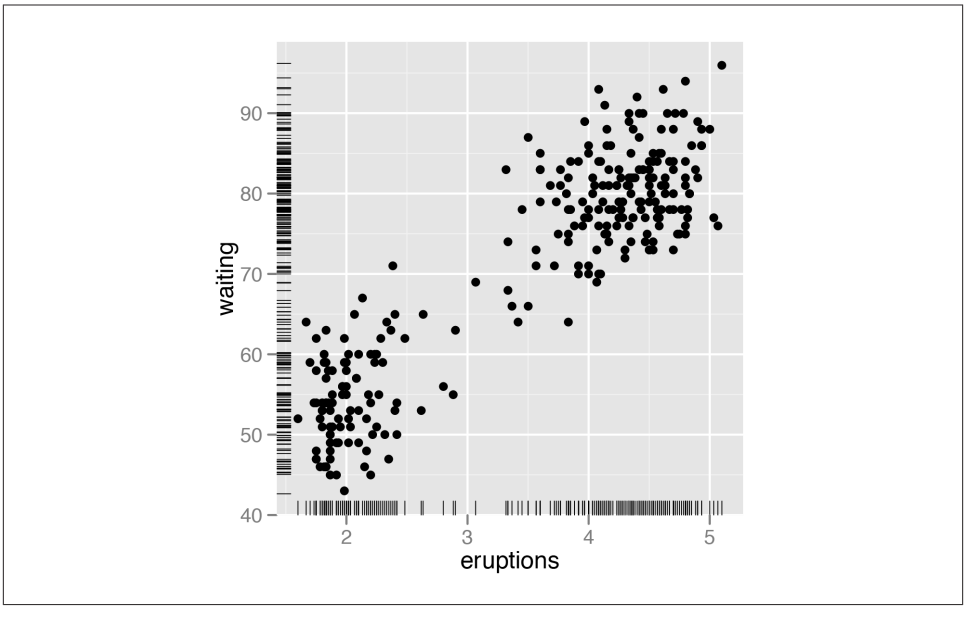


Figure 5-29. Marginal rug with thinner, jittered lines

Solution

For annotating just one or a few points, you can use `annotate()` or `geom_text()`. For this example, we'll use the `countries` data set and visualize the relationship between health expenditures and infant mortality rate per 1,000 live births. To keep things manageable, we'll just take the subset of countries that spent more than $2000 USD per capita:

```
library(gcookbook) # For the data set
subset(countries, Year==2009 & healthexp>2000)
```

```
           Name Code Year       GDP laborrate healthexp infmortality
        Andorra  AND 2009        NA        NA  3089.636          3.1
      Australia  AUS 2009  42130.82      65.2  3867.429          4.2
        Austria  AUT 2009  45555.43      60.4  5037.311          3.6
...
 United Kingdom  GBR 2009  35163.41      62.2  3285.050          4.7
  United States  USA 2009  45744.56      65.0  7410.163          6.6
```

We'll save the basic scatter plot object in `sp` and add then add things to it. To manually add annotations, use `annotate()`, and specify the coordinates and label (Figure 5-30, left). It may require some trial-and-error tweaking to get them positioned just right:

```
sp <- ggplot(subset(countries, Year==2009 & healthexp>2000),
             aes(x=healthexp, y=infmortality)) +
```

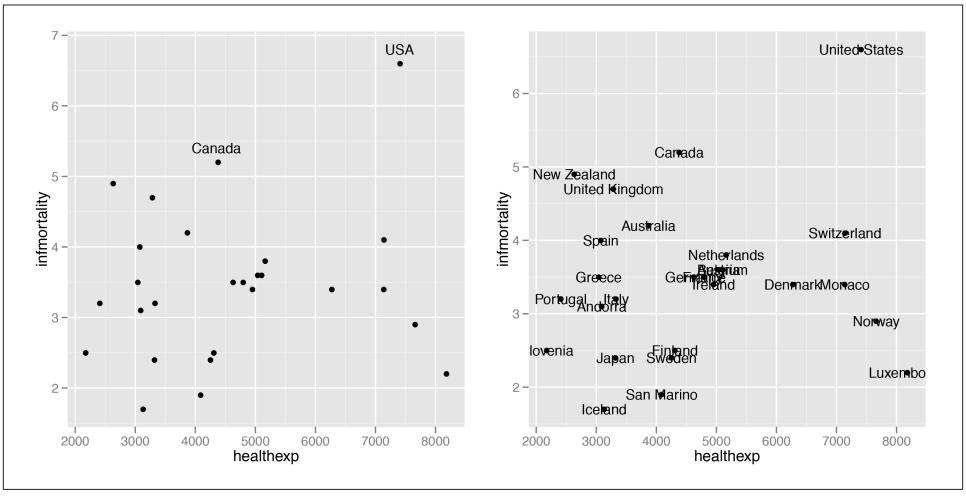


Figure 5-30. Left: a scatter plot with manually labeled points; right: with automatically labeled points and a smaller font

```
        geom_point()

sp + annotate("text", x=4350, y=5.4, label="Canada") +
     annotate("text", x=7400, y=6.8, label="USA")
```

To automatically add the labels from your data (Figure 5-30, right), use `geom_text()` and map a column that is a factor or character vector to the `label` aesthetic. In this case, we'll use `Name`, and we'll make the font slightly smaller to reduce crowding. The default value for `size` is 5, which doesn't correspond directly to a point size:

```
sp + geom_text(aes(label=Name), size=4)
```

Discussion

The automatic method for placing annotations centers each annotation on the *x* and *y* coordinates. You'll probably want to shift the text vertically, horizontally, or both.

Setting `vjust=0` will make the baseline of the text on the same level as the point (Figure 5-31, left), and setting `vjust=1` will make the top of the text level with the point. This usually isn't enough, though—you can either increase or decrease `vjust` to shift the labels higher or lower, or you can add or subtract a bit to or from the `y` mapping to get the same effect (Figure 5-31, right):

```
sp + geom_text(aes(label=Name), size=4, vjust=0)

# Add a little extra to y
sp + geom_text(aes(y=infmortality+.1, label=Name), size=4, vjust=0)
```

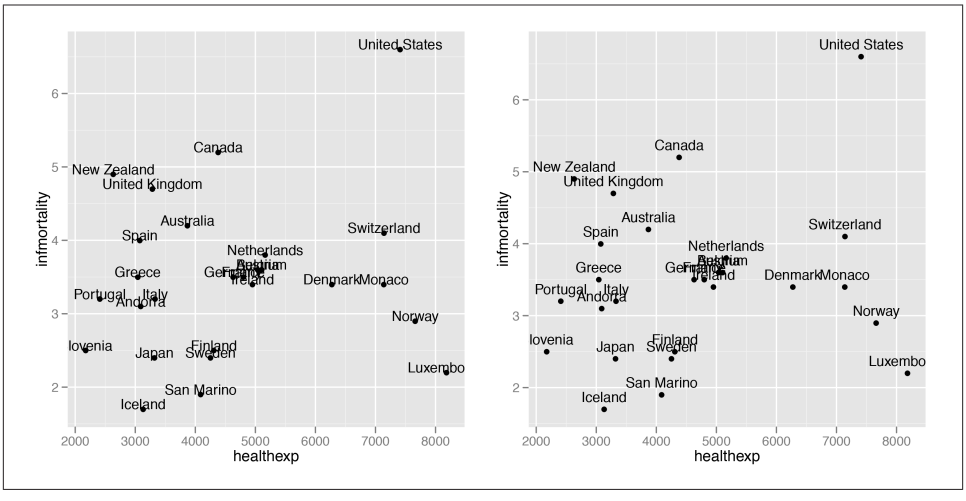


Figure 5-31. Left: a scatter plot with vjust=0; right: with a little extra added to y

It often makes sense to right- or left-justify the labels relative to the points. To left-justify, set `hjust=0` (Figure 5-32, left), and to right-justify, set `hjust=1`. As was the case with `vjust`, the labels will still slightly overlap with the points. This time, though, it's not a good idea to try to fix it by increasing or decreasing `hjust`. Doing so will shift the labels a distance proportional to the length of the label, making longer labels move further than shorter ones. It's better to just set `hjust` to 0 or 1, and then add or subtract a bit to or from `x` (Figure 5-32, right):

```
sp + geom_text(aes(label=Name), size=4, hjust=0)

sp + geom_text(aes(x=healthexp+100, label=Name), size=4, hjust=0)
```

If you are using a logarithmic axis, instead of adding to `x` or `y`, you'll need to *multiply* the `x` or `y` value by a number to shift the labels a consistent amount.

If you want to label just some of the points but want the placement to be handled automatically, you can add a new column to your data frame containing just the labels you want. Here's one way to do that: first we'll make a copy of the data we're using, then we'll duplicate the `Name` column into `Name1`:

```
cdat <- subset(countries, Year==2009 & healthexp>2000)

cdat$Name1 <- cdat$Name
```

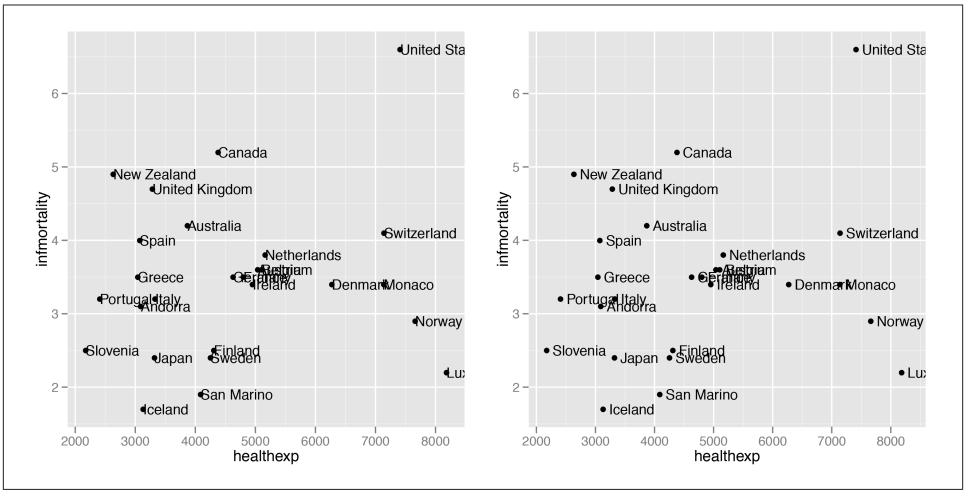


Figure 5-32. Left: a scatter plot with hjust=0; right: with a little extra added to x

Next, we'll use the `%in%` operator to find *where* each name that we want to keep is. This returns a logical vector indicating which entries in the first vector, `cdat$Name1`, are present in the second vector, in which we specify the names of the countries we want to show:

```
idx <- cdat$Name1 %in% c("Canada", "Ireland", "United Kingdom", "United States",
                         "New Zealand", "Iceland", "Japan", "Luxembourg",
                         "Netherlands", "Switzerland")
idx
```

```
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
[13] FALSE  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
[25]  TRUE  TRUE  TRUE
```

Then we'll use that Boolean vector to overwrite all the *other* entries in `Name1` with `NA`:

```
cdat$Name1[!idx] <- NA
```

This is what the result looks like:

```
cdat
```

```
          Name Code Year    GDP laborrate healthexp infmortality          Name1
       Andorra  AND 2009     NA        NA  3089.636          3.1          <NA>
     Australia  AUS 2009  42130      65.2  3867.429          4.2          <NA>
...
   Switzerland  CHE 2009  63524      66.9  7140.729          4.1   Switzerland
United Kingdom  GBR 2009  35163      62.2  3285.050          4.7 United Kingdom
 United States  USA 2009  45744      65.0  7410.163          6.6  United States
```

Now we can make the plot (Figure 5-33). This time, we'll also expand the *x* range so that the text will fit:

```
ggplot(cdat, aes(x=healthexp, y=infmortality)) +
    geom_point() +
    geom_text(aes(x=healthexp+100, label=Name1), size=4, hjust=0) +
    xlim(2000, 10000)
```

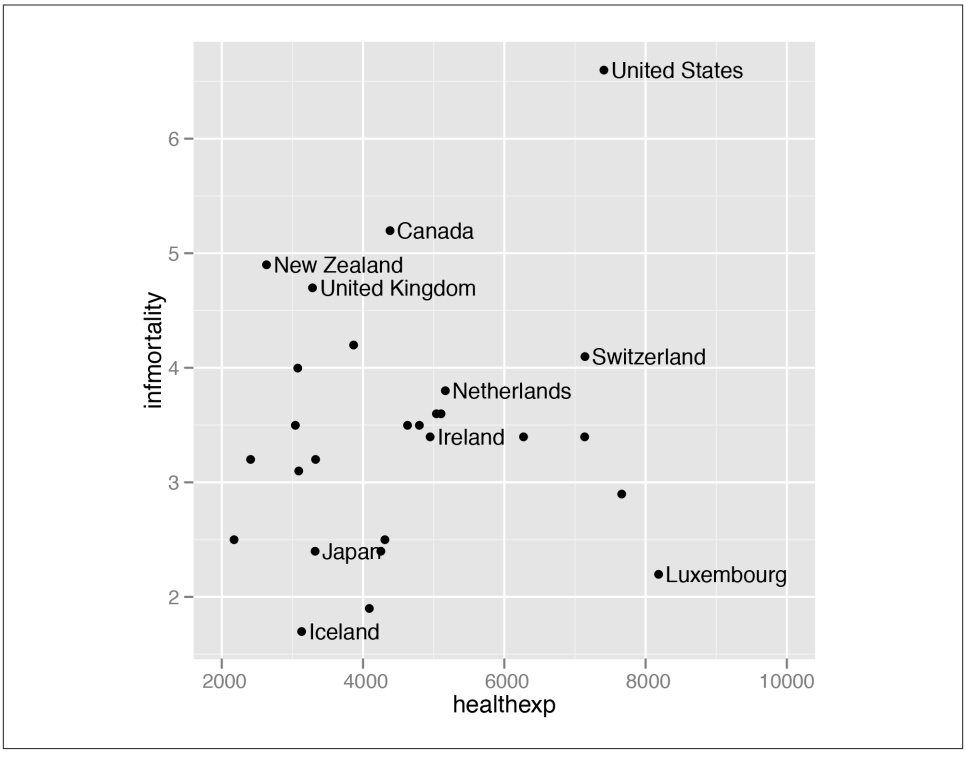


Figure 5-33. Scatter plot with selected labels and expanded x range

If any individual position adjustments are needed, you have a couple of options. One option is to copy the columns used for the *x* and *y* coordinates and modify the numbers for the individual items to move the text around. Make sure to use the original numbers for the coordinates of the points, of course! Another option is to save the output to a vector format such as PDF or SVG (see Recipes 14.1 and 14.2), then edit it in a program like Illustrator or Inkscape.

See Also

For more on controlling the appearance of the text, see Recipe 9.2.

If you want to manually edit a PDF or SVG file, see Recipe 14.4.

5.12. Creating a Balloon Plot

Problem

You want to make a balloon plot, where the area of the dots is proportional to their numerical value.

Solution

Use `geom_point()` with `scale_size_area()`. For this example, we'll use a subset of the `countries` data set:

```
library(gcookbook) # For the data set

cdat <- subset(countries, Year==2009 &
    Name %in% c("Canada", "Ireland", "United Kingdom", "United States",
                "New Zealand", "Iceland", "Japan", "Luxembourg",
                "Netherlands", "Switzerland"))

cdat
```

```
           Name Code Year       GDP laborrate healthexp infmortality
         Canada  CAN 2009  39599.04      67.8  4379.761          5.2
        Iceland  ISL 2009  37972.24      77.5  3130.391          1.7
        Ireland  IRL 2009  49737.93      63.6  4951.845          3.4
          Japan  JPN 2009  39456.44      59.5  3321.466          2.4
     Luxembourg  LUX 2009 106252.24      55.5  8182.855          2.2
    Netherlands  NLD 2009  48068.35      66.1  5163.740          3.8
    New Zealand  NZL 2009  29352.45      68.6  2633.625          4.9
    Switzerland  CHE 2009  63524.65      66.9  7140.729          4.1
 United Kingdom  GBR 2009  35163.41      62.2  3285.050          4.7
  United States  USA 2009  45744.56      65.0  7410.163          6.6
```

If we just map `GDP` to `size`, the value of `GDP` gets mapped to the *radius* of the dots (Figure 5-34, left), which is not what we want; a doubling of value results in a quadrupling of area, and this will distort the interpretation of the data. We instead want to map it to the *area*, and we can do this using `scale_size_area()` (Figure 5-34, right):

```
p <- ggplot(cdat, aes(x=healthexp, y=infmortality, size=GDP)) +
     geom_point(shape=21, colour="black", fill="cornsilk")

# GDP mapped to radius (default with scale_size_continuous)
p

# GDP mapped to area instead, and larger circles
p + scale_size_area(max_size=15)
```

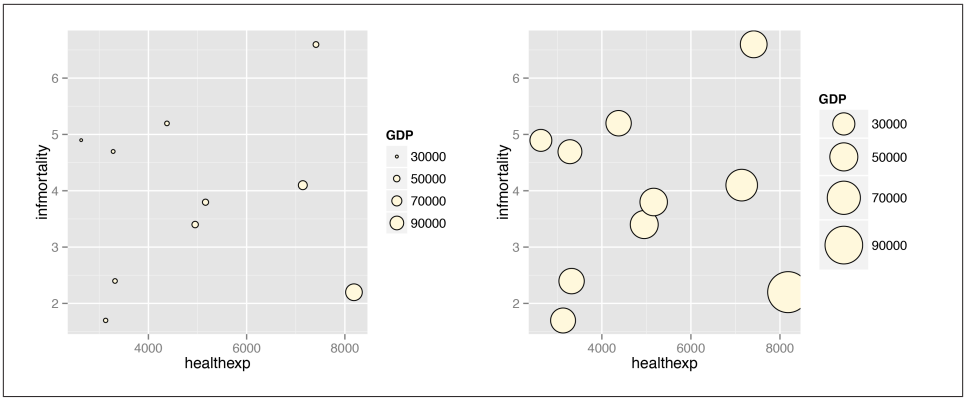


Figure 5-34. Left: balloon plot with value mapped to radius; right: with value mapped to area

Discussion

The example here is a scatter plot, but that is not the only way to use balloon plots. It may also be useful to use them to represent values on a grid, where the x- and y-axes are categorical, as in Figure 5-35:

```
# Add up counts for male and female
hec <- HairEyeColor[,,"Male"] + HairEyeColor[,,"Female"]

# Convert to long format
library(reshape2)
hec <- melt(hec, value.name="count")

ggplot(hec, aes(x=Eye, y=Hair)) +
    geom_point(aes(size=count), shape=21, colour="black", fill="cornsilk") +
    scale_size_area(max_size=20, guide=FALSE) +
    geom_text(aes(y=as.numeric(Hair)-sqrt(count)/22, label=count), vjust=1,
              colour="grey60", size=4)
```

In this example we've used a few tricks to add the text labels under the circles. First, we used `vjust=1` to top-justify the text to the *y* coordinate. Next, we wanted to set the *y* coordinate so that it is just underneath the bottom of each circle. This requires a little arithmetic: take the *numeric* value of `Hair` and subtract a small value from it, where the value depends in some way on `count`. This actually requires taking the square root of `count`, since the radius has a linear relationship with the square root of `count`. The number that this value divided by (22 in this case) is found by trial and error; it depends on the particular data values, radius, and text size.

The text under the circles is in a shade of grey. This is so that it doesn't jump out at the viewer and overwhelm the perceptual impact of the circles, but is still available if the viewer wants to know the exact values.

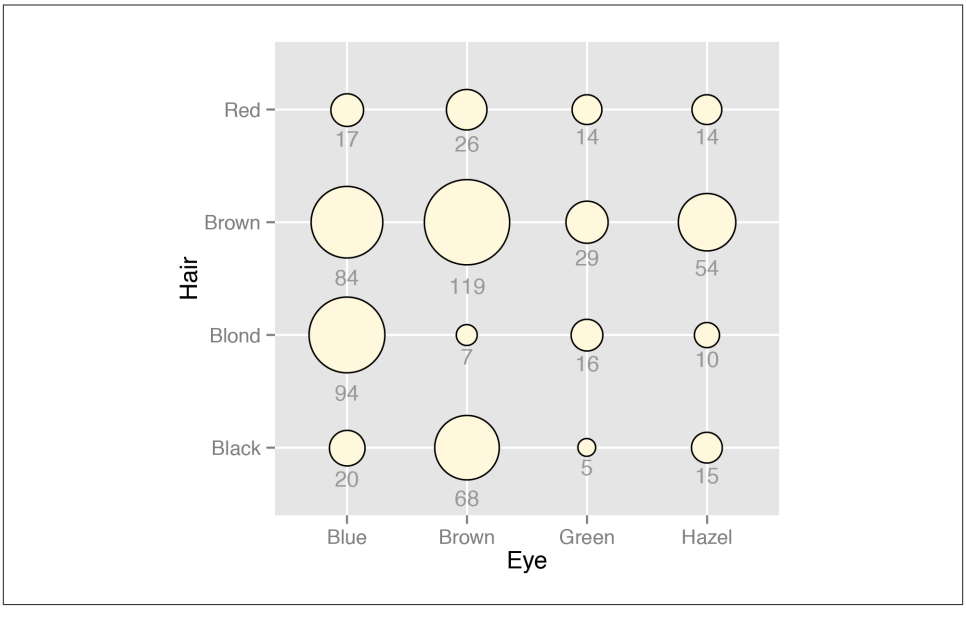


Figure 5-35. Balloon plot with categorical axes and text labels

See Also

To add labels to the circles, see Recipes 5.11 and 7.1.

See Recipe 5.4 for ways of mapping variables to other aesthetics in a scatter plot.

5.13. Making a Scatter Plot Matrix

Problem

You want to make a scatter plot matrix.

Solution

A scatter plot matrix is an excellent way of visualizing the pairwise relationships among several variables. To make one, use the `pairs()` function from R's base graphics.

For this example, we'll use a subset of the `countries` data set. We'll pull out the data for the year 2009, and keep only the columns that are relevant:

```
library(gcookbook) # For the data set
c2009 <- subset(countries, Year==2009,
                select=c(Name, GDP, laborrate, healthexp, infmortality))
```

```
c2009
```

```
        Name        GDP laborrate  healthexp infmortality
Afghanistan         NA       59.8   50.88597        103.2
    Albania  3772.6047       59.5  264.60406         17.2
    Algeria  4022.1989       58.5  267.94653         32.0
...
     Zambia  1006.3882       69.2   47.05637         71.5
   Zimbabwe   467.8534       66.8         NA         52.2
```

To make the scatter plot matrix (Figure 5-36), we'll use columns 2 through 5—using the `Name` column wouldn't make sense, and it would produce strange-looking results:

```
pairs(c2009[,2:5])
```

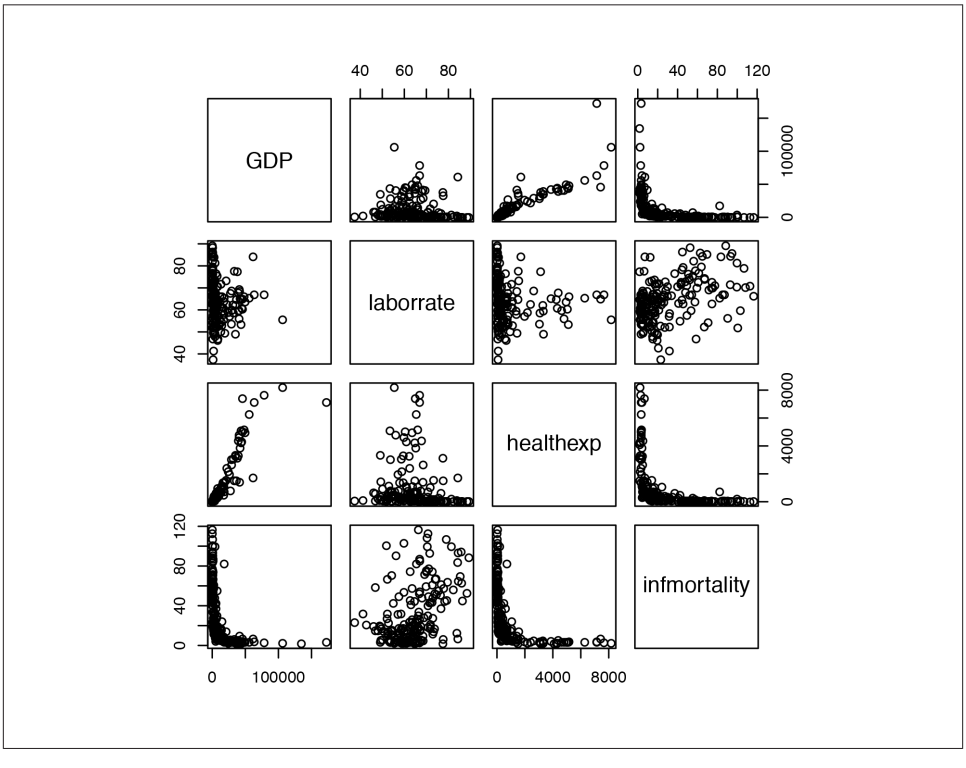


Figure 5-36. A scatter plot matrix

Discussion

We didn't use ggplot2 here because it doesn't make scatter plot matrices (at least, not well).

You can also use customized functions for the panels. To show the correlation coefficient of each pair of variables instead of a scatter plot, we'll define the function `panel.cor`. This will also show higher correlations in a larger font. Don't worry about the details for now—just paste this code into your R session or script:

```
panel.cor <- function(x, y, digits=2, prefix="", cex.cor, ...) {
    usr <- par("usr")
    on.exit(par(usr))
    par(usr = c(0, 1, 0, 1))
    r <- abs(cor(x, y, use="complete.obs"))
    txt <- format(c(r, 0.123456789), digits=digits)[1]
    txt <- paste(prefix, txt, sep="")
    if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
    text(0.5, 0.5, txt, cex =  cex.cor * (1 + r) / 2)
}
```

To show histograms of each variable along the diagonal, we'll define `panel.hist`:

```
panel.hist <- function(x, ...) {
    usr <- par("usr")
    on.exit(par(usr))
    par(usr = c(usr[1:2], 0, 1.5) )
    h <- hist(x, plot = FALSE)
    breaks <- h$breaks
    nB <- length(breaks)
    y <- h$counts
    y <- y/max(y)
    rect(breaks[-nB], 0, breaks[-1], y, col="white", ...)
}
```

Both of these panel functions are taken from the `pairs` help page, so if it's more convenient, you can simply open that help page, then copy and paste. The last line of this version of the `panel.cor` function is slightly modified, however, so that the changes in font size aren't as extreme as with the original.

Now that we've defined these functions we can use them for our scatter plot matrix, by telling `pairs()` to use `panel.cor` for the upper panels and `panel.hist` for the diagonal panels.

We'll also throw in one more thing: `panel.smooth` for the lower panels, which makes a scatter plot and adds a LOWESS smoothed line, as shown in Figure 5-37. (LOWESS is slightly different from LOESS, which we saw in Recipe 5.6, but the differences aren't important for this sort of rough exploratory visualization):

```
pairs(c2009[,2:5], upper.panel = panel.cor,
                   diag.panel  = panel.hist,
                   lower.panel = panel.smooth)
```

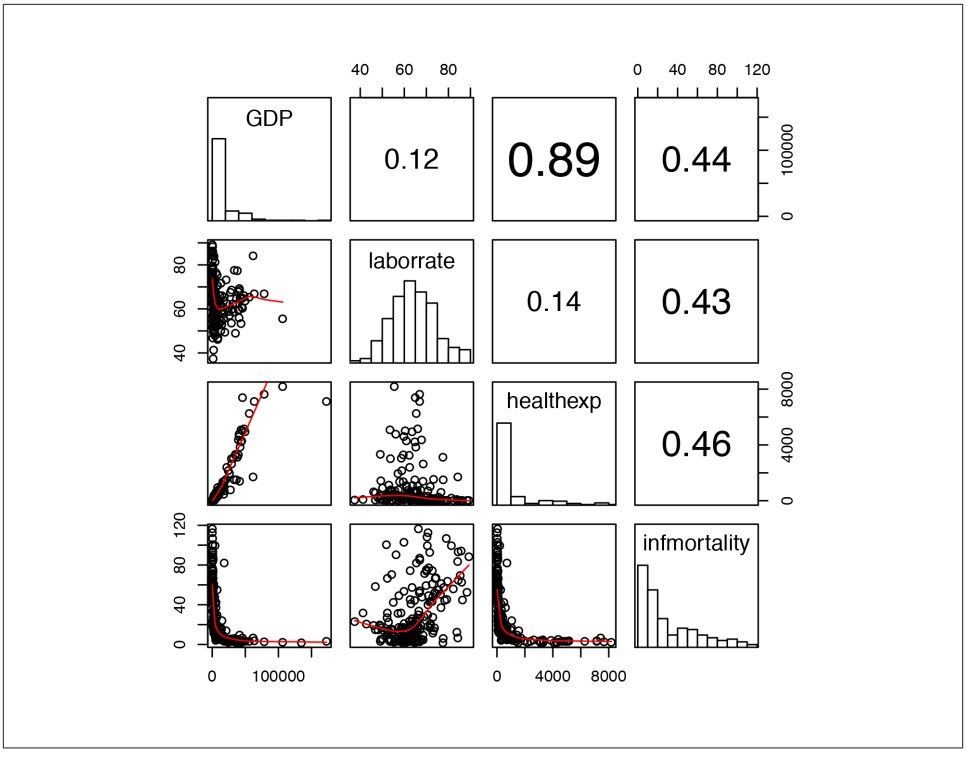


Figure 5-37. Scatter plot with correlations in the upper triangle, smoothing lines in the lower triangle, and histograms on the diagonal

It may be more desirable to use linear regression lines instead of LOWESS lines. The `panel.lm` function will do the trick (unlike the previous panel functions, this one isn't in the `pairs` help page):

```
panel.lm <- function (x, y, col = par("col"), bg = NA, pch = par("pch"),
                          cex = 1, col.smooth = "black", ...) {
    points(x, y, pch = pch, col = col, bg = bg, cex = cex)
    abline(stats::lm(y ~ x),  col = col.smooth, ...)
}
```

This time the default line color is black instead of red, though you can change it here (and with `panel.smooth`) by setting `col.smooth` when you call `pairs`().

We'll also use small points in the visualization, so that we can distinguish them a bit better (Figure 5-38). This is done by setting `pch="."`:

```
pairs(c2009[,2:5], pch=".",
                   upper.panel = panel.cor,
                   diag.panel  = panel.hist,
                   lower.panel = panel.lm)
```

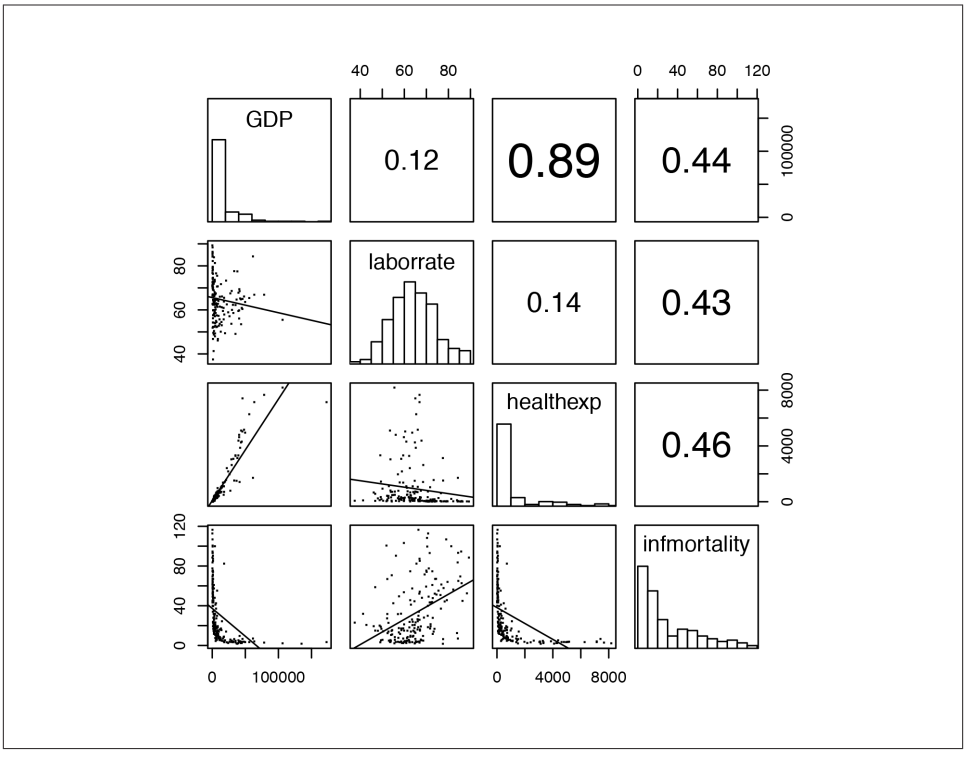


Figure 5-38. Scatter plot matrix with smaller points and linear fit lines

The size of the points can also be controlled using the `cex` parameter. The default value for `cex` is 1; make it smaller for smaller points and larger for larger points. Values below .5 might not render properly with PDF output.

See Also

To create a correlation matrix, see Recipe 13.1.

The `ggpairs()` function from the GGally package can also make scatter plot matrices.

CHAPTER 6

Summarized Data Distributions

This chapter explores how to visualize summarized distributions of data.

6.1. Making a Basic Histogram

Problem

You want to make a histogram.

Solution

Use `geom_histogram()` and map a continuous variable to `x` (Figure 6-1):

```
ggplot(faithful, aes(x=waiting)) + geom_histogram()
```

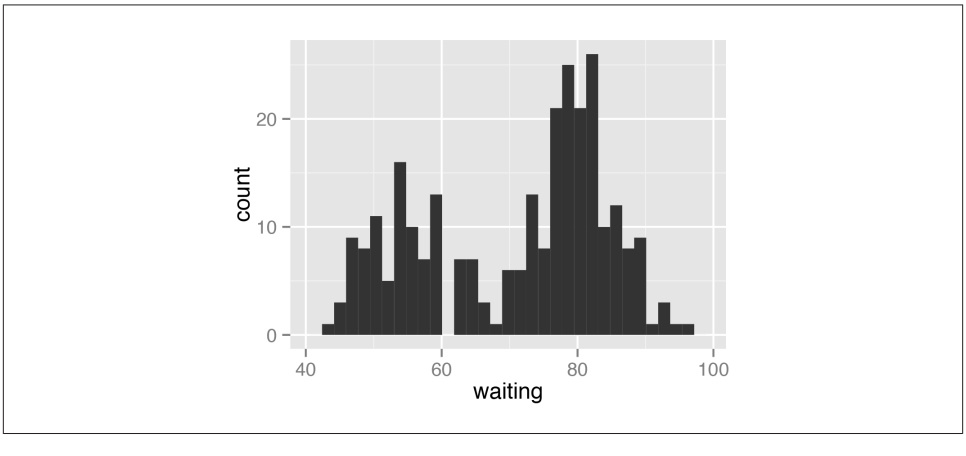


Figure 6-1. A basic histogram

Discussion

All `geom_histogram()` requires is one column from a data frame or a single vector of data. For this example we'll use the `faithful` data set, which contains data about the Old Faithful geyser in two columns: `eruptions`, which is the length of each eruption, and `waiting`, which is the length of time to the next eruption. We'll only use the `waiting` column in this example:

```
faithful

eruptions waiting
    3.600      79
    1.800      54
    3.333      74
...
```

If you just want to get a quick look at some data that isn't in a data frame, you can get the same result by passing in `NULL` for the data frame and giving `ggplot()` a vector of values. This would have the same result as the previous code:

```
# Store the values in a simple vector
w <- faithful$waiting

ggplot(NULL, aes(x=w)) + geom_histogram()
```

By default, the data is grouped into 30 bins. This may be too fine or too coarse for your data. You can change the size of the bins by using `binwidth`, or you can divide the range of the data into a specific number of bins. The default colors—a dark fill without an outline—can make it difficult to see which bar corresponds to which value, so we'll also change the colors, as shown in Figure 6-2.

```
# Set the width of each bin to 5
ggplot(faithful, aes(x=waiting)) +
    geom_histogram(binwidth=5, fill="white", colour="black")

# Divide the x range into 15 bins
binsize <- diff(range(faithful$waiting))/15
ggplot(faithful, aes(x=waiting)) +
    geom_histogram(binwidth=binsize, fill="white", colour="black")
```

Sometimes the appearance of the histogram will be very dependent on the width of the bins and where exactly the boundaries between bins are. In Figure 6-3, we'll use a bin width of 8. In the version on the left, we'll use the `origin` parameter to put boundaries at 31, 39, 47, etc., while in the version on the right, we'll shift it over by 4, putting boundaries at 35, 43, 51, etc.:

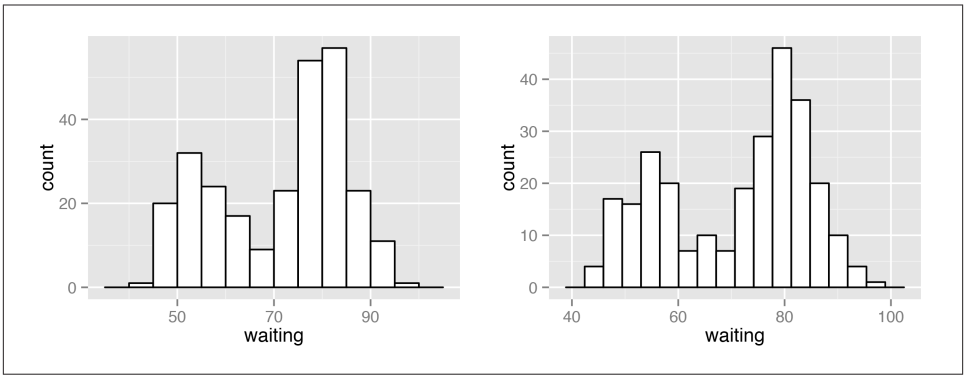


Figure 6-2. Left: histogram with binwidth=5, and with different colors; right: with 15 bins

```
h <- ggplot(faithful, aes(x=waiting))  # Save the base object for reuse

h + geom_histogram(binwidth=8, fill="white", colour="black", origin=31)

h + geom_histogram(binwidth=8, fill="white", colour="black", origin=35)
```

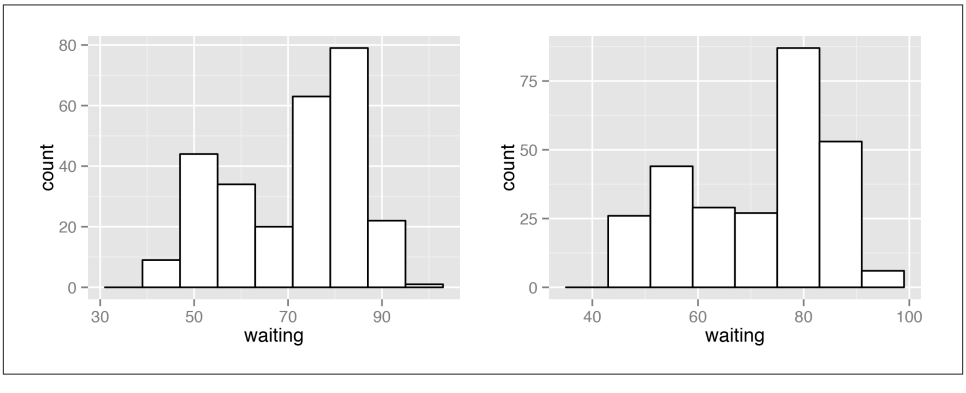


Figure 6-3. Different appearance of histograms with the origin at 31 and 35

The results look quite different, even though they have the same bin size. The `faithful` data set is not particularly small, with 272 observations; with smaller data sets, this is even more of an issue. When visualizing your data, it's a good idea to experiment with different bin sizes and boundary points.

Also, if your data has discrete values, it may matter that the histogram bins are asymmetrical. They are *closed* on the lower bound and *open* on the upper bound. If you have bin boundaries at 1, 2, 3, etc., then the bins will be [1, 2), [2, 3), and so on. In other words, the first bin contains 1 but not 2, and the second bin contains 2 but not 3.

It is also possible to use `geom_bar(stat="bin")` for the same effect, although I find it easier to interpret the code if it uses `geom_histogram()`.

See Also

Frequency polygons provide a better way of visualizing multiple distributions without the bars interfering with each other. See Recipe 6.5.

6.2. Making Multiple Histograms from Grouped Data

Problem

You want to make histograms of multiple groups of data.

Solution

Use `geom_histogram()` and use facets for each group, as shown in Figure 6-4:

```
library(MASS) # For the data set

# Use smoke as the faceting variable
ggplot(birthwt, aes(x=bwt)) + geom_histogram(fill="white", colour="black") +
    facet_grid(smoke ~ .)
```

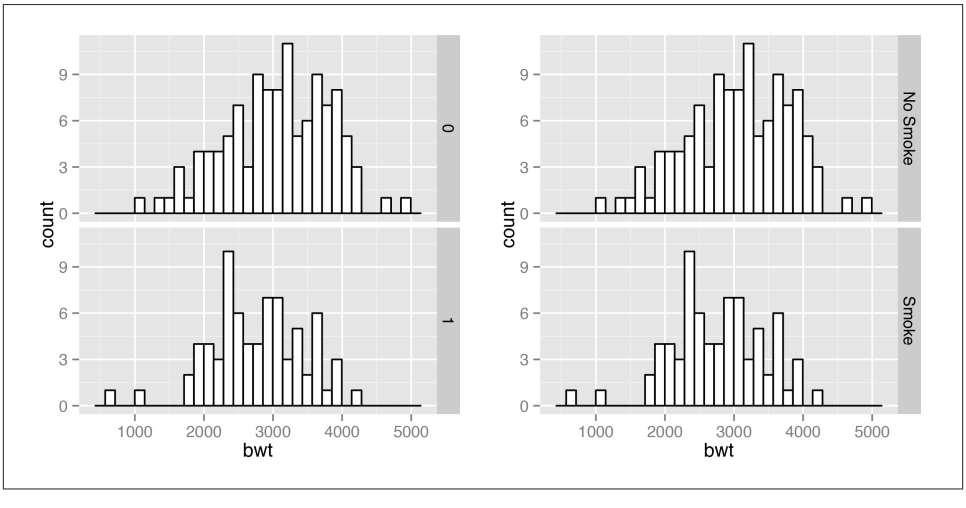


Figure 6-4. Left: two histograms with facets; right: with different facet labels

Discussion

To make these plots, the data must all be in one data frame, with one column containing a categorical variable used for grouping.

For this example, we used the `birthwt` data set. It contains data about birth weights and a number of risk factors for low birth weight:

```
birthwt

low age lwt race smoke ptl ht ui ftv  bwt
  0  19 182    2     0   0  0  1   0 2523
  0  33 155    3     0   0  0  0   3 2551
  0  20 105    1     1   0  0  0   1 2557
...
```

One problem with the faceted graph is that the facet labels are just 0 and 1, and there's no label indicating that those values are for `smoke`. To change the labels, we need to change the names of the factor levels. First we'll take a look at the factor levels, then we'll assign new factor level names, in the same order:

```
birthwt1 <- birthwt # Make a copy of the data

# Convert smoke to a factor
birthwt1$smoke <- factor(birthwt1$smoke)
levels(birthwt1$smoke)

 "0" "1"


library(plyr) # For the revalue() function
birthwt1$smoke <- revalue(birthwt1$smoke, c("0"="No Smoke", "1"="Smoke"))
```

Now when we plot it again, it shows the new labels (Figure 6-4, right).

```
ggplot(birthwt1, aes(x=bwt)) + geom_histogram(fill="white", colour="black") +
    facet_grid(smoke ~ .)
```

With facets, the axes have the same *y* scaling in each facet. If your groups have different sizes, it might be hard to compare the *shapes* of the distributions of each one. For example, see what happens when we facet the birth weights by `race` (Figure 6-5, left):

```
ggplot(birthwt, aes(x=bwt)) + geom_histogram(fill="white", colour="black") +
    facet_grid(race ~ .)
```

To allow the *y* scales to be resized independently (Figure 6-5, right), use `scales="free"`. Note that this will only allow the *y* scales to be free—the *x* scales will still be fixed because the histograms are aligned with respect to that axis:

```
ggplot(birthwt, aes(x=bwt)) + geom_histogram(fill="white", colour="black") +
    facet_grid(race ~ ., scales="free")
```

Another approach is to map the grouping variable to `fill`, as shown in Figure 6-6. The grouping variable must be a factor or character vector. In the `birthwt` data set, the desired grouping variable, `smoke`, is stored as a number, so we'll use the `birthwt1` data set we created above, in which smoke is a factor:

```
# Convert smoke to a factor
birthwt1$smoke <- factor(birthwt1$smoke)

# Map smoke to fill, make the bars NOT stacked, and make them semitransparent
ggplot(birthwt1, aes(x=bwt, fill=smoke)) +
    geom_histogram(position="identity", alpha=0.4)
```

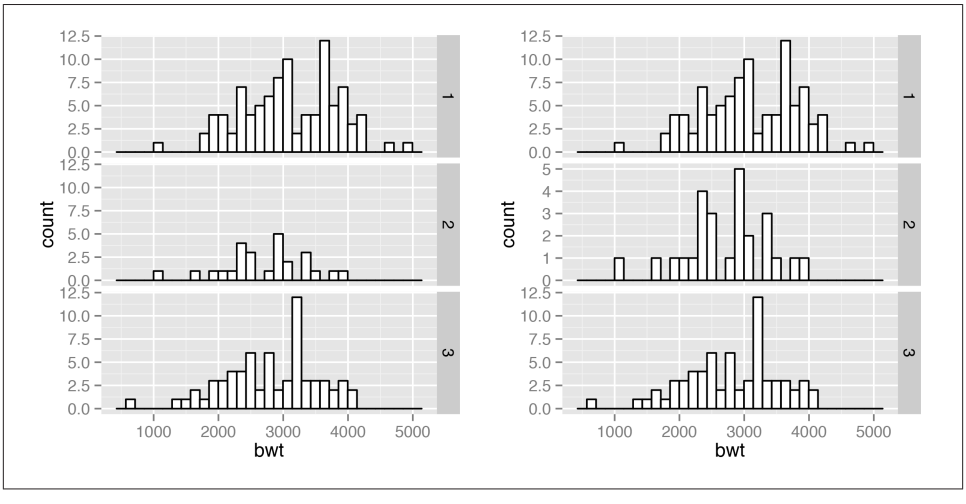


Figure 6-5. Left: histograms with the default fixed scales; right: with scales="free"

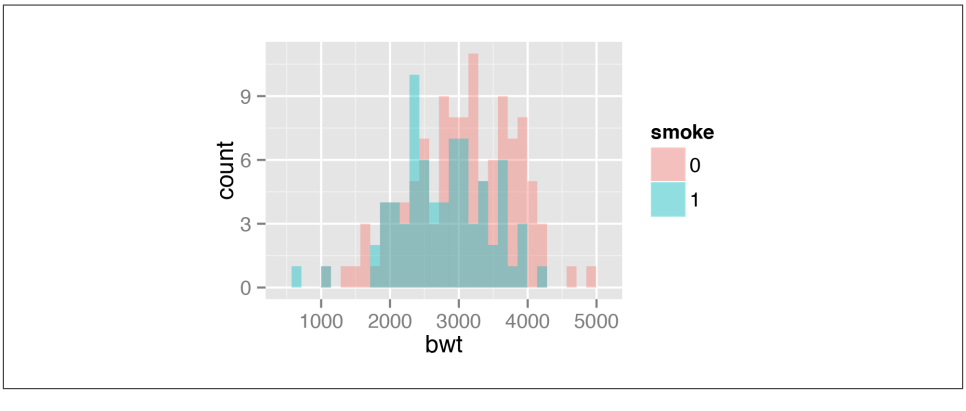


Figure 6-6. Multiple histograms with different fill colors

The `position="identity"` is important. Without it, `ggplot()` will stack the histogram bars on top of each other vertically, making it much more difficult to see the distribution of each group.

6.3. Making a Density Curve

Problem

You want to make a kernel density curve.

Solution

Use `geom_density()` and map a continuous variable to `x` (Figure 6-7):

```
ggplot(faithful, aes(x=waiting)) + geom_density()
```

If you don't like the lines along the side and bottom, you can use `geom_line(stat="density")` (see Figure 6-7, right):

```
# The expand_limits() increases the y range to include the value 0
ggplot(faithful, aes(x=waiting)) + geom_line(stat="density") +
    expand_limits(y=0)
```

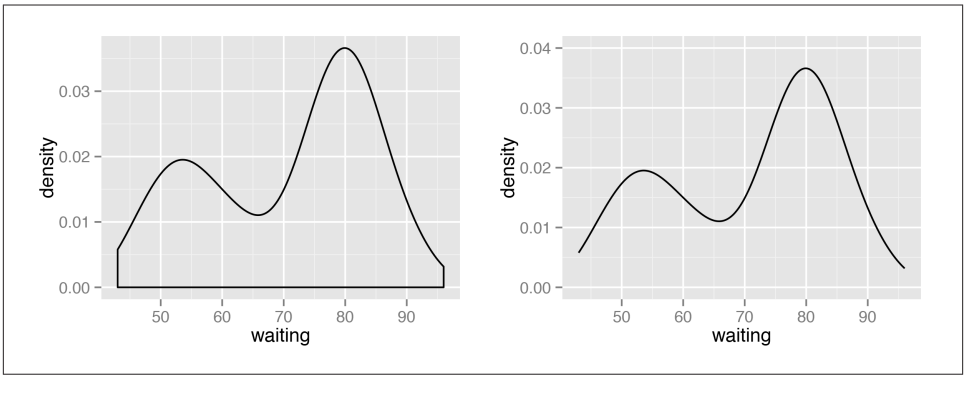


Figure 6-7. Left: a kernel density estimate curve with geom_density(); right: with geom_line()

Discussion

Like `geom_histogram()`, `geom_density()` requires just one column from a data frame. For this example, we'll use the `faithful` data set, which contains data about the Old Faithful geyser in two columns: `eruptions`, which is the length of each eruption, and `waiting`, which is the length of time to the next eruption. We'll only use the `waiting` column in this example:

```
faithful

eruptions waiting
    3.600      79
    1.800      54
    3.333      74
...
```

The second method mentioned earlier uses `geom_line()` and tells it to use the `"density"` statistical transformation. This is essentially the same as the first method, using `geom_density()`, except the former draws it with a closed polygon.

As with `geom_histogram()`, if you just want to get a quick look at data that isn't in a data frame, you can get the same result by passing in `NULL` for the data frame and giving `ggplot()` a vector of values. This would have the same result as the first solution:

```
# Store the values in a simple vector
w <- faithful$waiting

ggplot(NULL, aes(x=w)) + geom_density()
```

A kernel density curve is an estimate of the population distribution, based on the sample data. The amount of smoothing depends on the *kernel bandwidth*: the larger the bandwidth, the more smoothing there is. The bandwidth can be set with the `adjust` parameter, which has a default value of 1. Figure 6-8 shows what happens with a smaller and larger value of `adjust`:

```
ggplot(faithful, aes(x=waiting)) +
    geom_line(stat="density", adjust=.25, colour="red") +
    geom_line(stat="density") +
    geom_line(stat="density", adjust=2, colour="blue")
```

In this example, the *x* range is automatically set so that it contains the data, but this results in the edge of the curve getting clipped. To show more of the curve, set the *x* limits (Figure 6-9). We'll also add an 80% transparent fill with `alpha=.2`:

```
ggplot(faithful, aes(x=waiting)) +
    geom_density(fill="blue", alpha=.2) +
    xlim(35, 105)

# This draws a blue polygon with geom_density(), then adds a line on top
ggplot(faithful, aes(x=waiting)) +
    geom_density(fill="blue", colour=NA, alpha=.2) +
    geom_line(stat="density") +
    xlim(35, 105)
```

If this edge-clipping happens with your data, it might mean that your curve is too smooth—if the curve is wider than your data, it might not be the best model of your data. Or it could be because you have a small data set.

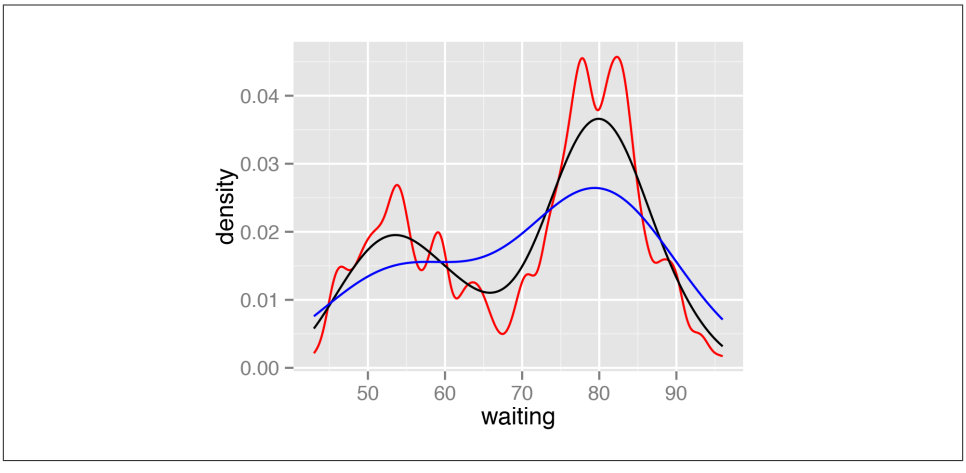


Figure 6-8. Density curves with adjust set to .25 (red), default value of 1 (black), and 2 (blue)

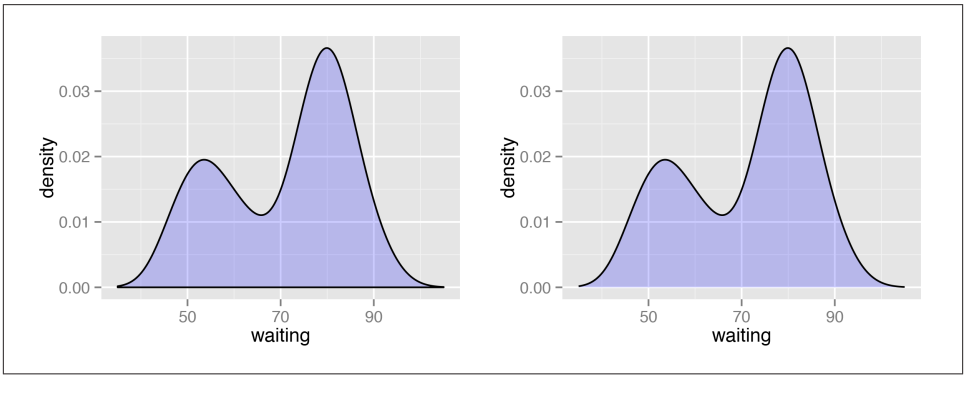


Figure 6-9. Left: density curve with wider x limits and a semitransparent fill; right: in two parts, with geom_density() and geom_line()

To compare the theoretical and observed distributions, you can overlay the density curve with the histogram. Since the *y* values for the density curve are small (the area under the curve always sums to 1), it would be barely visible if you overlaid it on a histogram without any transformation. To solve this problem, you can scale down the histogram to match the density curve with the mapping `y=..density..`. Here we'll add `geom_histogram()` first, and then layer `geom_density()` on top (Figure 6-10):

```
ggplot(faithful, aes(x=waiting, y=..density..)) +
    geom_histogram(fill="cornsilk", colour="grey60", size=.2) +
    geom_density() +
    xlim(35, 105)
```

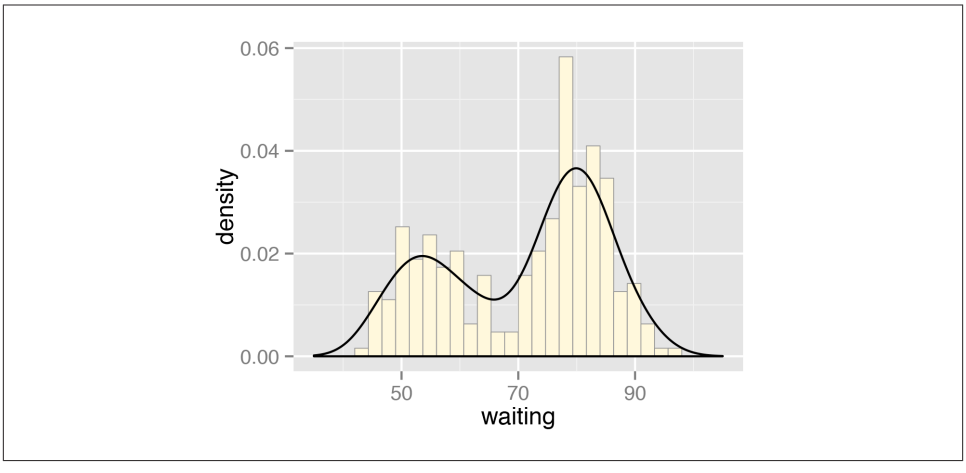


Figure 6-10. Density curve overlaid on a histogram

See Also

See Recipe 6.9 for information on violin plots, which are another way of representing density curves and may be more appropriate for comparing multiple distributions.

6.4. Making Multiple Density Curves from Grouped Data

Problem

You want to make density curves of multiple groups of data.

Solution

Use `geom_density()`, and map the grouping variable to an aesthetic like `colour` or `fill`, as shown in Figure 6-11. The grouping variable must be a factor or character vector. In the `birthwt` data set, the desired grouping variable, `smoke`, is stored as a number, so we have to convert it to a factor first:

```
library(MASS) # For the data set
# Make a copy of the data
birthwt1 <- birthwt

# Convert smoke to a factor
birthwt1$smoke <- factor(birthwt1$smoke)

# Map smoke to colour
```

```
ggplot(birthwt1, aes(x=bwt, colour=smoke)) + geom_density()

# Map smoke to fill and make the fill semitransparent by setting alpha
ggplot(birthwt1, aes(x=bwt, fill=smoke)) + geom_density(alpha=.3)
```

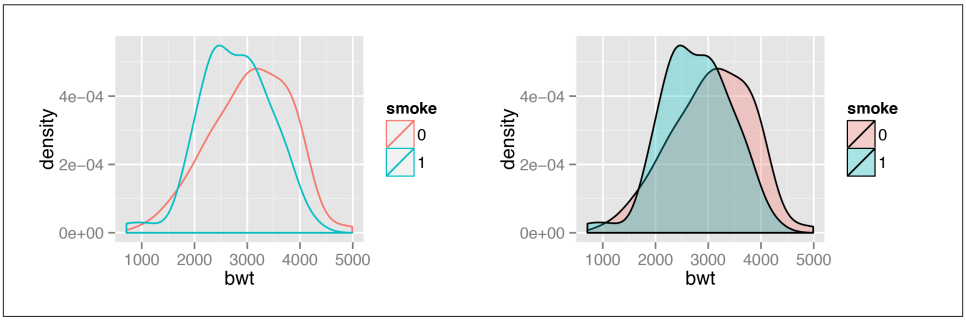


Figure 6-11. Left: different line colors for each group; right: different semitransparent fill colors for each group

Discussion

To make these plots, the data must all be in one data frame, with one column containing a categorical variable used for grouping.

For this example, we used the `birthwt` data set. It contains data about birth weights and a number of risk factors for low birth weight:

```
birthwt

low age lwt race smoke ptl ht ui ftv  bwt
  0  19 182    2     0   0  0  1   0 2523
  0  33 155    3     0   0  0  0   3 2551
  0  20 105    1     1   0  0  0   1 2557
...
```

We looked at the relationship between `smoke` (smoking) and `bwt` (birth weight in grams). The value of `smoke` is either 0 or 1, but since it's stored as a numeric vector, `ggplot()` doesn't know that it should be treated as a categorical variable. To make it so `ggplot()` knows to treat `smoke` as categorical, we can either convert that column of the data frame to a factor, or tell `ggplot()` to treat it as a factor by using `factor(smoke)` inside of the `aes()` statement. For these examples, we converted it to a factor in the data.

Another method for visualizing the distributions is to use facets, as shown in Figure 6-12. We can align the facets vertically or horizontally. Here we'll align them vertically so that it's easy to compare the two distributions:

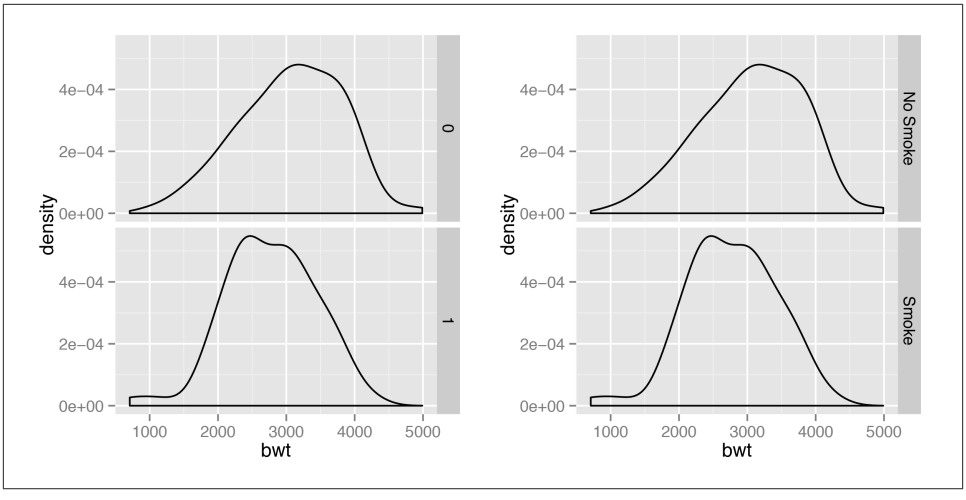


Figure 6-12. Left: density curves with facets; right: with different facet labels

```
ggplot(birthwt1, aes(x=bwt)) + geom_density() + facet_grid(smoke ~ .)
```

One problem with the faceted graph is that the facet labels are just 0 and 1, and there's no label indicating that those values are for `smoke`. To change the labels, we need to change the names of the factor levels. First we'll take a look at the factor levels, then we'll assign new factor level names, in the same order:

```
levels(birthwt1$smoke)
```

```
 "0" "1"
```

```
library(plyr) # For the revalue function
birthwt1$smoke <- revalue(birthwt1$smoke, c("0"="No Smoke", "1"="Smoke"))
```

Now when we plot it again, it shows the new labels (Figure 6-12, right):

```
ggplot(birthwt1, aes(x=bwt)) + geom_density() + facet_grid(smoke ~ .)
```

If you want to see the histograms along with the density curves, the best option is to use facets, since other methods of visualizing both histograms in a single graph can be difficult to interpret. To do this, map `y=..density..`, so that the histogram is scaled down to the height of the density curves. In this example, we'll also make the histogram bars a little less prominent by changing the colors (Figure 6-13):

```
ggplot(birthwt1, aes(x=bwt, y=..density..)) +
    geom_histogram(binwidth=200, fill="cornsilk", colour="grey60", size=.2) +
    geom_density() +
    facet_grid(smoke ~ .)
```

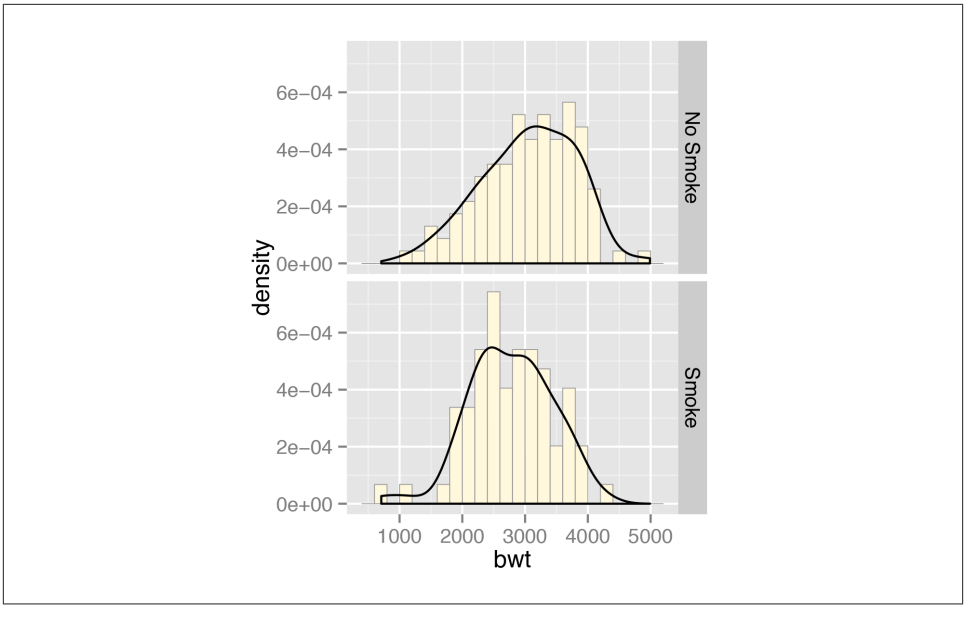


Figure 6-13. Density curves overlaid on histograms

6.5. Making a Frequency Polygon

Problem

You want to make a frequency polygon.

Solution

Use `geom_freqpoly()` (Figure 6-14):

```
ggplot(faithful, aes(x=waiting)) + geom_freqpoly()
```

Discussion

A frequency polygon appears similar to a kernel density estimate curve, but it shows the same information as a histogram. That is, like a histogram, it shows what is in the data, whereas a kernel density estimate is just that—an estimate—and requires you to pick some value for the bandwidth.

Also like a histogram, you can control the bin width for the frequency polygon (Figure 6-14, right):

```
ggplot(faithful, aes(x=waiting)) + geom_freqpoly(binwidth=4)
```

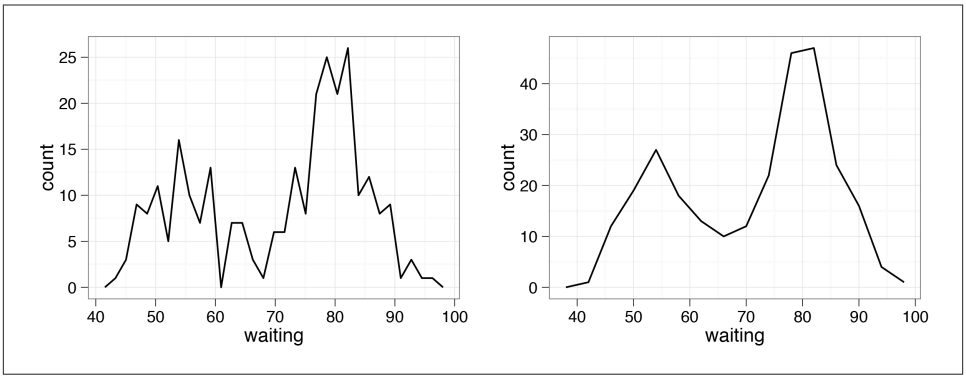


Figure 6-14. Left: a frequency polygon; right: with wider bins

Or, instead of setting the width of each bin directly, you can divide the *x* range into a particular number of bins:

```
# Use 15 bins
binsize <- diff(range(faithful$waiting))/15
ggplot(faithful, aes(x=waiting)) + geom_freqpoly(binwidth=binsize)
```

See Also

Histograms display the same information, but with bars instead of lines. See Recipe 6.1.

6.6. Making a Basic Box Plot

Problem

You want to make a box (or box-and-whiskers) plot.

Solution

Use `geom_boxplot()`, mapping a continuous variable to `y` and a discrete variable to `x` (Figure 6-15):

```
library(MASS) # For the data set

ggplot(birthwt, aes(x=factor(race), y=bwt)) + geom_boxplot()
# Use factor() to convert numeric variable to discrete
```