

O'REILLY®

2nd Edition



R Graphics Cookbook

PRACTICAL RECIPES FOR VISUALIZING DATA

Winston Chang

R Graphics Cookbook

This O'Reilly cookbook provides more than 150 recipes to help scientists, engineers, programmers, and data analysts 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 in this second edition use the updated version of the ggplot2 package, a powerful and flexible way to make graphs in R. You'll also find expanded content about the visual design of graphics. If you have at least a basic understanding of the R language, you're ready to get started with this easy-to-use reference.

- 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 more
- Provide annotations to help viewers interpret data
- Control the overall appearance of graphics
- Explore options for using colors in plots
- Create network graphs, heat maps, and 3D scatter plots
- Get your data into shape using packages from the tidyverse

Winston Chang is a software engineer at RStudio, where he works on data visualization and web-based data analysis tools for R. He created the Cookbook for R website, which contains recipes for handling common tasks in R. In previous lives, he was a philosophy graduate student and a Java developer. Winston has a PhD in psychology from Northwestern University.

“The R Graphics Cookbook is the fastest way to get up and running with visualizations in R. It's jam-packed with useful code and killer tips to make beautiful, effective plots.”

—**Hadley Wickham**

Chief Scientist at RStudio and
coauthor of *R for Data Science*

US \$69.99

CAN \$92.99

ISBN: 978-1-491-97860-3



5 6 9 9 9



Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

R Graphics Cookbook

Practical Recipes for Visualizing Data

Winston Chang

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

R Graphics Cookbook

by Winston Chang

Copyright © 2019 Winston Chang. All rights reserved.

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://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Michele Cronin

Production Editor: Nicholas Adams

Copyeditor: Kim Cofer

Proofreader: Christina Edwards

Indexer: Angela Howard

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

December 2012: First Edition

November 2018: Second Edition

Revision History for the Second Edition

2018-10-25: First Release

2018-11-16: Second Release

2019-01-25: Third Release

2019-02-22: Fourth Release

2019-05-24: Fifth Release

2020-01-24: Sixth Release

2020-07-10: Seventh Release

2020-10-23: Eighth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491978603> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *R Graphics Cookbook*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

R Graphics Cookbook is available under the [Creative Commons Attribution 4.0 International Public License](https://creativecommons.org/licenses/by/4.0/).

978-1-491-97860-3

[GP]

Table of Contents

Preface.....	ix
1. R Basics.....	1
1.1 Installing a Package	2
1.2 Loading a Package	3
1.3 Upgrading Packages	3
1.4 Loading a Delimited Text Data File	4
1.5 Loading Data from an Excel File	5
1.6 Loading Data from SPSS/SAS/Stata Files	6
1.7 Chaining Functions Together with %>%, the Pipe Operator	7
2. Quickly Exploring Data.....	11
2.1 Creating a Scatter Plot	11
2.2 Creating a Line Graph	13
2.3 Creating a Bar Graph	15
2.4 Creating a Histogram	18
2.5 Creating a Box Plot	19
2.6 Plotting a Function Curve	21
3. Bar Graphs.....	23
3.1 Making a Basic Bar Graph	23
3.2 Grouping Bars Together	26
3.3 Making a Bar Graph of Counts	29
3.4 Using Colors in a Bar Graph	31
3.5 Coloring Negative and Positive Bars Differently	33
3.6 Adjusting Bar Width and Spacing	35
3.7 Making a Stacked Bar Graph	37
3.8 Making a Proportional Stacked Bar Graph	40

3.9 Adding Labels to a Bar Graph	43
3.10 Making a Cleveland Dot Plot	49
4. Line Graphs.....	55
4.1 Making a Basic Line Graph	55
4.2 Adding Points to a Line Graph	58
4.3 Making a Line Graph with Multiple Lines	60
4.4 Changing the Appearance of Lines	64
4.5 Changing the Appearance of Points	66
4.6 Making a Graph with a Shaded Area	68
4.7 Making a Stacked Area Graph	70
4.8 Making a Proportional Stacked Area Graph	72
4.9 Adding a Confidence Region	73
5. Scatter Plots.....	77
5.1 Making a Basic Scatter Plot	77
5.2 Grouping Points Together Using Shapes or Colors	79
5.3 Using Different Point Shapes	81
5.4 Mapping a Continuous Variable to Color or Size	84
5.5 Dealing with Overplotting	88
5.6 Adding Fitted Regression Model Lines	94
5.7 Adding Fitted Lines from an Existing Model	99
5.8 Adding Fitted Lines from Multiple Existing Models	103
5.9 Adding Annotations with Model Coefficients	106
5.10 Adding Marginal Rugs to a Scatter Plot	109
5.11 Labeling Points in a Scatter Plot	111
5.12 Creating a Balloon Plot	118
5.13 Making a Scatter Plot Matrix	121
6. Summarized Data Distributions.....	127
6.1 Making a Basic Histogram	127
6.2 Making Multiple Histograms from Grouped Data	130
6.3 Making a Density Curve	134
6.4 Making Multiple Density Curves from Grouped Data	138
6.5 Making a Frequency Polygon	141
6.6 Making a Basic Box Plot	142
6.7 Adding Notches to a Box Plot	145
6.8 Adding Means to a Box Plot	146
6.9 Making a Violin Plot	147
6.10 Making a Dot Plot	151
6.11 Making Multiple Dot Plots for Grouped Data	154
6.12 Making a Density Plot of Two-Dimensional Data	157

7. Annotations.....	161
7.1 Adding Text Annotations	161
7.2 Using Mathematical Expressions in Annotations	164
7.3 Adding Lines	166
7.4 Adding Line Segments and Arrows	169
7.5 Adding a Shaded Rectangle	171
7.6 Highlighting an Item	171
7.7 Adding Error Bars	173
7.8 Adding Annotations to Individual Facets	177
8. Axes.....	181
8.1 Swapping X- and Y-Axes	181
8.2 Setting the Range of a Continuous Axis	183
8.3 Reversing a Continuous Axis	186
8.4 Changing the Order of Items on a Categorical Axis	187
8.5 Setting the Scaling Ratio of the X- and Y-Axes	189
8.6 Setting the Positions of Tick Marks	191
8.7 Removing Tick Marks and Labels	192
8.8 Changing the Text of Tick Labels	193
8.9 Changing the Appearance of Tick Labels	196
8.10 Changing the Text of Axis Labels	198
8.11 Removing Axis Labels	199
8.12 Changing the Appearance of Axis Labels	201
8.13 Showing Lines Along the Axes	202
8.14 Using a Logarithmic Axis	204
8.15 Adding Ticks for a Logarithmic Axis	209
8.16 Making a Circular Plot	211
8.17 Using Dates on an Axis	216
8.18 Using Relative Times on an Axis	220
9. Controlling the Overall Appearance of Graphs.....	223
9.1 Setting the Title of a Graph	223
9.2 Changing the Appearance of Text	225
9.3 Using Themes	228
9.4 Changing the Appearance of Theme Elements	231
9.5 Creating Your Own Themes	235
9.6 Hiding Grid Lines	236
10. Legends.....	239
10.1 Removing the Legend	239
10.2 Changing the Position of a Legend	241
10.3 Changing the Order of Items in a Legend	243

10.4 Reversing the Order of Items in a Legend	246
10.5 Changing a Legend Title	247
10.6 Changing the Appearance of a Legend Title	249
10.7 Removing a Legend Title	250
10.8 Changing the Labels in a Legend	251
10.9 Changing the Appearance of Legend Labels	255
10.10 Using Labels with Multiple Lines of Text	256
11. Facets.....	259
11.1 Splitting Data into Subplots with Facets	259
11.2 Using Facets with Different Axes	262
11.3 Changing the Text of Facet Labels	264
11.4 Changing the Appearance of Facet Labels and Headers	266
12. Using Colors in Plots.....	269
12.1 Setting the Colors of Objects	269
12.2 Representing Variables with Colors	270
12.3 Using a Colorblind-Friendly Palette	272
12.4 Using a Different Palette for a Discrete Variable	275
12.5 Using a Manually Defined Palette for a Discrete Variable	280
12.6 Using a Manually Defined Palette for a Continuous Variable	283
12.7 Coloring a Shaded Region Based on Value	285
13. Miscellaneous Graphs.....	289
13.1 Making a Correlation Matrix	289
13.2 Plotting a Function	293
13.3 Shading a Subregion Under a Function Curve	295
13.4 Creating a Network Graph	297
13.5 Using Text Labels in a Network Graph	300
13.6 Creating a Heat Map	302
13.7 Creating a Three-Dimensional Scatter Plot	304
13.8 Adding a Prediction Surface to a Three-Dimensional Plot	308
13.9 Saving a Three-Dimensional Plot	312
13.10 Animating a Three-Dimensional Plot	313
13.11 Creating a Dendrogram	314
13.12 Creating a Vector Field	317
13.13 Creating a QQ Plot	322
13.14 Creating a Graph of an Empirical Cumulative Distribution Function	323
13.15 Creating a Mosaic Plot	324
13.16 Creating a Pie Chart	329
13.17 Creating a Map	330
13.18 Creating a Choropleth Map	334

13.19 Making a Map with a Clean Background	339
13.20 Creating a Map from a Shapefile	340
14. Output for Presentation.....	345
14.1 Outputting to PDF Vector Files	345
14.2 Outputting to SVG Vector Files	347
14.3 Outputting to WMF Vector Files	347
14.4 Editing a Vector Output File	348
14.5 Outputting to Bitmap (PNG/TIFF) Files	350
14.6 Using Fonts in PDF Files	352
14.7 Using Fonts in Windows Bitmap or Screen Output	355
14.8 Combining Several Plots into the Same Graphic	357
15. Getting Your Data into Shape.....	361
15.1 Creating a Data Frame	362
15.2 Getting Information About a Data Structure	363
15.3 Adding a Column to a Data Frame	365
15.4 Deleting a Column from a Data Frame	366
15.5 Renaming Columns in a Data Frame	367
15.6 Reordering Columns in a Data Frame	368
15.7 Getting a Subset of a Data Frame	369
15.8 Changing the Order of Factor Levels	371
15.9 Changing the Order of Factor Levels Based on Data Values	373
15.10 Changing the Names of Factor Levels	374
15.11 Removing Unused Levels from a Factor	376
15.12 Changing the Names of Items in a Character Vector	377
15.13 Recoding a Categorical Variable to Another Categorical Variable	379
15.14 Recoding a Continuous Variable to a Categorical Variable	381
15.15 Calculating New Columns from Existing Columns	382
15.16 Calculating New Columns by Groups	384
15.17 Summarizing Data by Groups	387
15.18 Summarizing Data with Standard Errors and Confidence Intervals	392
15.19 Converting Data from Wide to Long	395
15.20 Converting Data from Long to Wide	399
15.21 Converting a Time Series Object to Times and Values	400
A. Understanding ggplot2.....	403
Index.....	417

Preface

When the first edition of this book was published five years ago, the phrase “data science” had only recently entered the popular lexicon. Today, the phrase is unavoidable if you’re involved with the sciences, journalism, or high-tech industries. Many inter-related developments have made this possible: there’s a general awareness that understanding quantitative data has tangible benefits; there are better and more widely available educational resources about how to do data science; and finally, the tools have evolved, becoming easier to use and get started with.

The goal of this book is to help you understand your data by visualizing it, and to help you convey that understanding to others. You can think of data analysis as the process of transforming raw data into ideas in somebody’s mind. One of the key techniques for doing this is to create visualizations of the data. Our brains have very highly developed visual pattern detection systems, and data visualizations are a way to efficiently use those visual systems to get quantitative information into a person’s mind.

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.

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` plotting package, and the `dplyr` package for data wrangling. These packages are both part of the *tidyverse*, which is a collection of R packages that make it easier to work with data. Some of the recipes require the most recent version of `ggplot2`, 3.0.0, 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, <http://www.r-project.org>.



You can use the recipes with just a surface understanding of `ggplot2`, but if you want a deeper understanding of how it works, see [Appendix A](#).

Once you've installed R, you can install the necessary packages. In addition to the *tidyverse* packages, you'll also want to install the `gcookbook` package, which contains data sets for many of the examples in this book. You can install the *tidyverse* packages and the `gcookbook` package with:

```
install.packages("tidyverse")
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, load the `ggplot2` and `dplyr` packages in each R session in which you want to use the recipes in this book:

```
library(ggplot2)
library(dplyr)
```

The recipes in this book will assume that you've already loaded `ggplot2` and `dplyr`, so they won't show these lines of code.

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 macOS, 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 element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

O'Reilly

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from

O'Reilly and 200+ other publishers. For more information, please visit <http://oreilly.com>.

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://bit.ly/r-graphics-cookbook-2e>.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For news and more information about our books and courses, 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 have helped make this book possible. I'd like to thank the R community for creating R and for fostering a dynamic ecosystem around it. Thanks to Hadley Wickham and other members of the tidyverse team for creating the software that this book revolves around, and for opening up many opportunities for me to deepen my understanding of R, data analysis, and visualization. I'm grateful that my employer, RStudio, not only makes it possible for me to work with some of the leading lights in the R community, but also pays us to work on software that the entire R community benefits from.

Thanks to the technical reviewers for this book, and the first edition of it: Garrett Grolemond, Thomas Lin Pedersen, Paul Teetor, Hadley Wickham, Dennis Murphy, and Erik Iverson. Their depth of knowledge and attention to detail have resulted in a much better book. Thanks to Jen Wang for her help editing this edition of the 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.

Most of the recipes in this book require the `ggplot2`, `dplyr`, and `gcookbook` packages to be installed on your computer. (The `gcookbook` package contains the data sets used in some of the examples, but is not necessary for doing your real work.) If you want to get started quickly, run:

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

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

```
library(tidyverse)  
library(gcookbook)
```

Running `library(tidyverse)` will load `ggplot2`, `dplyr`, and a number of other packages. If you want to keep your R session more streamlined and load only the packages that are strictly needed, you can load the `ggplot2` and `dplyr` packages individually:

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



If you want a deeper understanding of how `ggplot2` works, see [Appendix A](#), which explains the concepts behind `ggplot2`.

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, which specializes in packages related to genomic data.

If you have spent much time learning R, you may have heard of the *tidyverse*, which is a collection of R packages that share common ideas of how data should be structured and manipulated. This is in contrast to *base R*, which is the set of packages that are included when you just download and install R. The tidyverse is a set of add-ons for R, which make it easier to do many operations related to data manipulation and visualization. This book mostly uses the tidyverse, as I believe that it provides a quicker and simpler (but not less powerful!) way to work with data.

If you haven't used the tidyverse before, there is one recipe in particular that you should read that will help you understand a foreign-looking bit of syntax, `%>%`, also known as the pipe operator. This is [Recipe 1.7](#) in this chapter.

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. It's usually best to use the first choice, <https://cloud.r-project.org/>, as it is a cloud-based mirror with endpoints all over the world.

Discussion

If you want to install multiple packages at once, you can pass it a vector of package names. For example, this will install most of the packages used in this book:

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

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

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 Upgrading Packages

Problem

You want to upgrade a package that is already installed.

Solution

Run `update.packages()`:

```
update.packages()
```

It will prompt you for each package that can be upgraded. If you want it to upgrade all packages without asking, use `ask = FALSE`:

```
update.packages(ask = FALSE)
```

Discussion

Over time, package authors will release new versions of packages with bug fixes and new features, and it's usually a good idea to keep up-to-date. However, keep in mind that occasionally new versions of packages will introduce bugs or have slightly changed behavior.

1.4 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")
```

Alternatively, you can use the `read_csv()` function (note the underscore instead of period) from the `readr` package. This function is significantly faster than `read.csv()`, and has better string and date and time handling.

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 *character vectors* in R terminology). To differentiate this, use `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 "Curren" "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`.

1.5 Loading Data from an Excel File

Problem

You want to load data from an Excel file.

Solution

The `readxl` package has the function `read_excel()` for reading *.xls* and *.xlsx* files from Excel. This will read the first sheet of an Excel spreadsheet:

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

library(readxl)
data <- read_excel("datafile.xlsx", 1)
```

Discussion

With `read_excel()`, you can load from other sheets by specifying a number or name for sheet:

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

data <- read_excel("datafile.xls", sheet = "Revenues")
```

`read_excel()` uses the first row of the spreadsheet for column names. If you don't want to use that row for column names, use `col_names = FALSE`. The columns will instead be named X1, X2, and so on.

By default, `read_excel()` will infer the type of each column, but if you want to specify the type of each column, you can use the `col_types` argument. You can also drop columns if you specify the type as "blank":

```
# Drop the first column, and specify the types of the next three columns
data <- read_excel("datafile.xls",
                   col_types = c("blank", "text", "date", "numeric"))
```

See Also

See `?read_excel` for more options for controlling the reading of these files.

There are other packages for reading Excel files. The `gdata` package has a function `read.xls()` for reading in `.xls` files, and the `xlsx` package has a function `read.xlsx()` for reading in `.xlsx` files. They require external software to be installed on your computer: `read.xls()` requires Java, and `read.xlsx()` requires Perl.

1.6 Loading Data from SPSS/SAS/Stata Files

Problem

You want to load data from an SPSS file, or from other programs like SAS or Stata.

Solution

The `haven` package has the function `read_sav()` for reading SPSS files. To load data from an SPSS file:

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

library(haven)
data <- read_sav("datafile.sav")
```

Discussion

The `haven` package also includes functions to read from other formats:

- `read_sas()`: SAS
- `read_dta()`: Stata

An alternative to `haven` is the `foreign` package. It also supports SPSS and Stata files, but it is not as up-to-date as the functions from `haven`. For example, it only supports

Stata files up to version 12, while haven supports up to version 14 (the current version as of this writing).

The foreign package does support some other formats, including:

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

See Also

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

1.7 Chaining Functions Together with %>%, the Pipe Operator

Problem

You want to call one function, then pass the result to another function, and another, in a way that is easily readable.

Solution

Use %>%, the pipe operator. For example:

```
library(dplyr) # The pipe is provided by dplyr

morley # Look at the morley data set
#>      Expt Run Speed
#> 001     1   1  850
#> 002     1   2  740
#> 003     1   3  900
#> ...<94 more rows>...
#> 098     5  18  800
#> 099     5  19  810
#> 100     5  20  870

morley %>%
  filter(Expt == 1) %>%
  summary()
#>      Expt      Run      Speed
#> Min.   :1  Min.   : 1.00  Min.   : 650
#> 1st Qu.:1  1st Qu.: 5.75  1st Qu.: 850
```

```
#> Median :1      Median :10.50      Median : 940
#> Mean   :1      Mean    :10.50      Mean    : 909
#> 3rd Qu.:1      3rd Qu.:15.25      3rd Qu.: 980
#> Max.   :1      Max.    :20.00      Max.    :1070
```

This takes the `morley` data set, and passes it to the `filter()` function from `dplyr`, keeping only the rows of the data where `Expt` is equal to 1. Then that result is passed to the `summary()` function, which calculates some summary statistics on the data.

Without the pipe operator, the preceding code would be written like this:

```
summary(filter(morley, Expt == 1))
```

In this code, function calls are processed from the inside outward. From a mathematical viewpoint, this makes perfect sense, but from a readability viewpoint, this can be confusing and hard to read, especially when there are many nested function calls.

Discussion

This pattern, with the `%>%` operator, is widely used with tidyverse packages, because they contain many functions that do relatively small things. The idea is that these functions are building blocks that allow users to compose the function calls together to produce the desired result.

To illustrate what's going on, here's a simpler example of two equivalent pieces of code:

```
f(x)

# Equivalent to:
x %>% f()
```

The pipe operator in essence takes the thing that's on the left, and places it as the first argument of the function call that's on the right.

It can be used for multiple function calls, in a *chain*:

```
h(g(f(x)))

# Equivalent to:
x %>%
  f() %>%
  g() %>%
  h()
```

In a function chain, the lexical ordering of the function calls is the same as the order in which they're computed.

If you want to store the final result, you can use the `<-` operator at the beginning. For example, this will replace the original `x` with the result of the function chain:

```
x <- x %>%  
  f() %>%  
  g() %>%  
  h()
```

If there are additional arguments for the function calls, they will be shifted to the right when the pipe operator is used. Going back to the code from the first example, these two are equivalent:

```
filter(morley, Expt == 1)  
  
morley %>% filter(Expt == 1)
```

The pipe operator is actually from the `magrittr` package, but `dplyr` imports it and makes it available when you call `library(dplyr)`.

See Also

For many more examples of how to use `%>%` in data manipulation, see [Chapter 15](#).

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 plot data. 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, straightforward to use in simple cases, and run very quickly.

If you want to do anything beyond very simple plots, 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 `ggplot()` function in `ggplot2`. The previous edition of this book also gave examples using the `qplot()` function from the `ggplot2` package, but now it is recommended to just use `ggplot()` instead.

If you already know how to use R's 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)
```

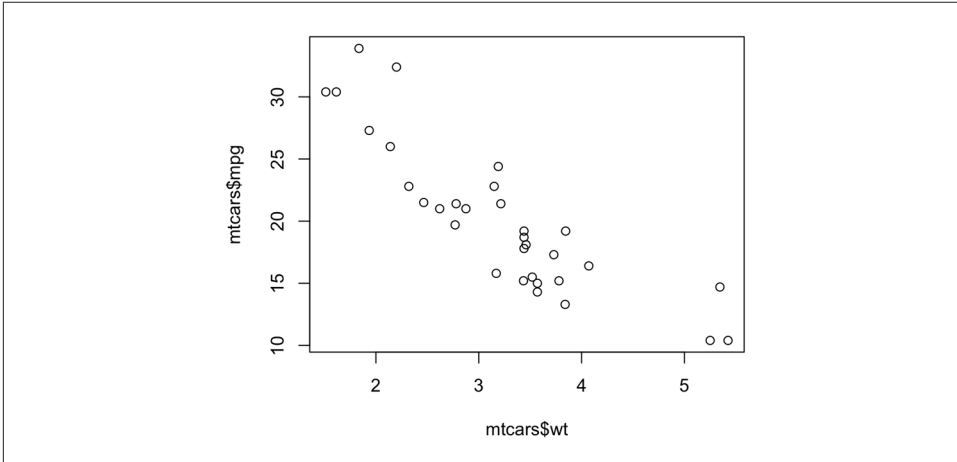


Figure 2-1. Scatter plot with base graphics

The `mtcars$wt` returns the column named `wt` from the `mtcars` data frame, and `mtcars$mpg` is the `mpg` column.

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

```
library(ggplot2)

ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point()
```

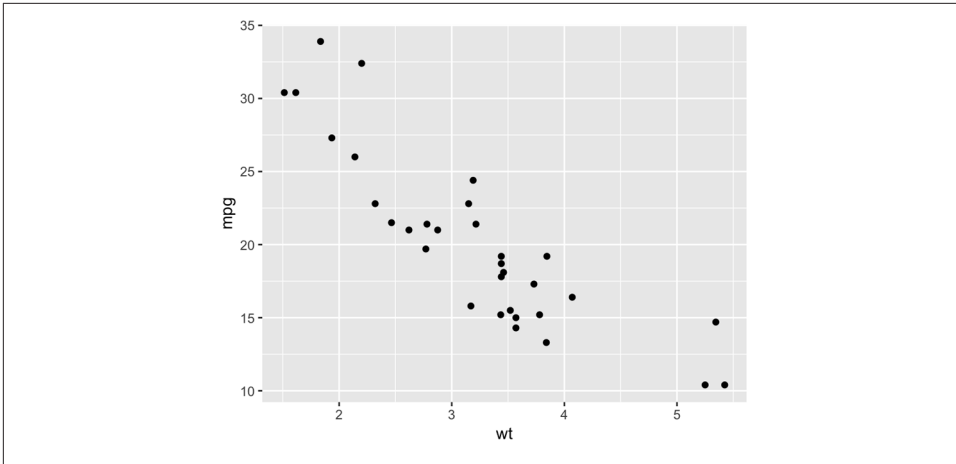


Figure 2-2. Scatter plot with ggplot2

The first part, `ggplot()`, tells it to create a plot object, and the second part, `geom_point()`, tells it to add a layer of points to the plot.

The usual way to use `ggplot()` is to pass it a data frame (`mtcars`) and then tell it which columns to use for the x and y values. If you want to pass it two vectors for x and y values, you can use `data = NULL`, and then pass it the vectors. Keep in mind that `ggplot2` is designed to work with data frames as the data source, not individual vectors, and that using it this way will only allow you to use a limited part of its capabilities:

```
ggplot(data = NULL, aes(x = mtcars$wt, y = mtcars$mpg)) +  
  geom_point()
```

See Also

See [Chapter 5](#) for more in-depth information about creating scatter plots.

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")
```

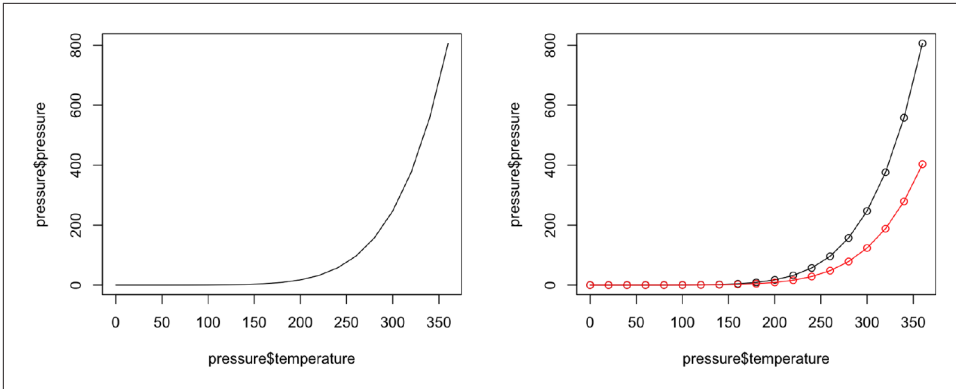


Figure 2-3. Line graph with base graphics (left); With points and another line (right)

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 `geom_line()` (Figure 2-4):

```
library(ggplot2)
ggplot(pressure, aes(x = temperature, y = pressure)) +
  geom_line()
```

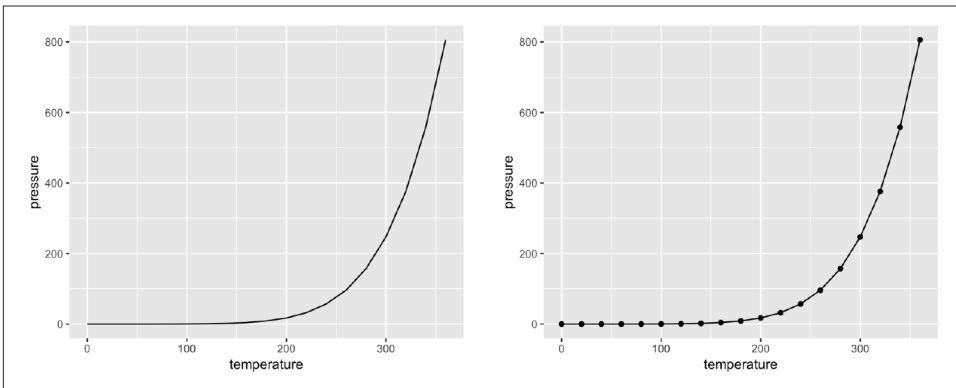


Figure 2-4. Line graph with `ggplot()` (left); With points added (right)

As with scatter plots, you can pass your data in vectors instead of in a data frame (but this will limit the things you can do later with the plot):


```
ggplot(pressure, aes(x = temperature, y = pressure)) +
  geom_line() +
  geom_point()
```



It's common with `ggplot()` to split the command on multiple lines, ending each line with a `+` so that R knows that the command will continue on the next line.

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](#), left), 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:

```
# First, take a look at the BOD data
BOD
#>   Time demand
#> 1     1    8.3
#> 2     2   10.3
#> 3     3   19.0
#> 4     4   16.0
#> 5     5   15.6
#> 6     7   19.8

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

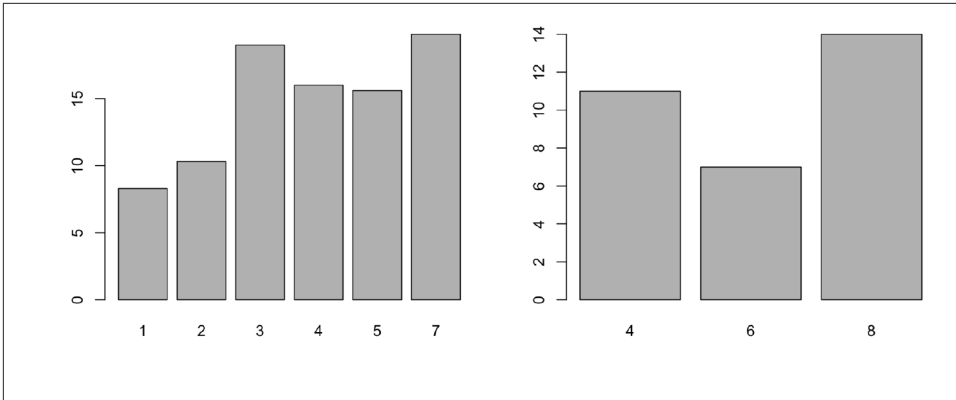


Figure 2-5. Bar graph of values with base graphics (left); Bar graph of counts (right)

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:

```
# There are 11 cases of the value 4, 7 cases of 6, and 14 cases of 8
table(mtcars$cyl)
```

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

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

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

```
library(ggplot2)

# Bar graph of values. This uses the BOD data frame, with the
# "Time" column for x values and the "demand" column for y values.
ggplot(BOD, aes(x = Time, y = demand)) +
  geom_col()

# Convert the x variable to a factor, so that it is treated as discrete
ggplot(BOD, aes(x = factor(Time), y = demand)) +
  geom_col()
```

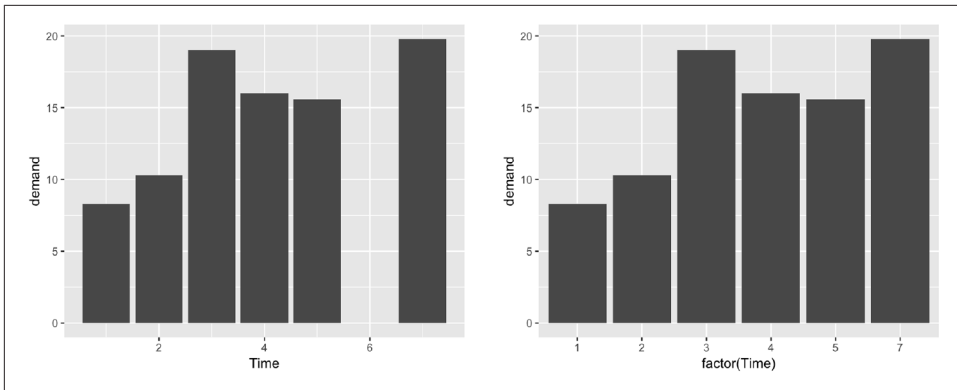


Figure 2-6. Bar graph of values using `geom_col()` with a continuous *x* variable (left); With *x* variable converted to a factor (notice that there is no entry for 6; right)

`ggplot2` can also be used to plot the *count* of the number of data rows in each category (Figure 2-7), by using `geom_bar()` instead of `geom_col()`. Once again, notice the difference between a continuous *x*-axis and a discrete one. For some kinds of data, it may make more sense to convert the continuous *x* variable to a discrete one, with the `factor()` function:

```
# Bar graph of counts. This uses the mtcars data frame, with the "cyl" column for
# x position. The y position is calculated by counting the number of rows for
# each value of cyl.
```

```
ggplot(mtcars, aes(x = cyl)) +
  geom_bar()
```

```
# Bar graph of counts
```

```
ggplot(mtcars, aes(x = factor(cyl))) +
  geom_bar()
```

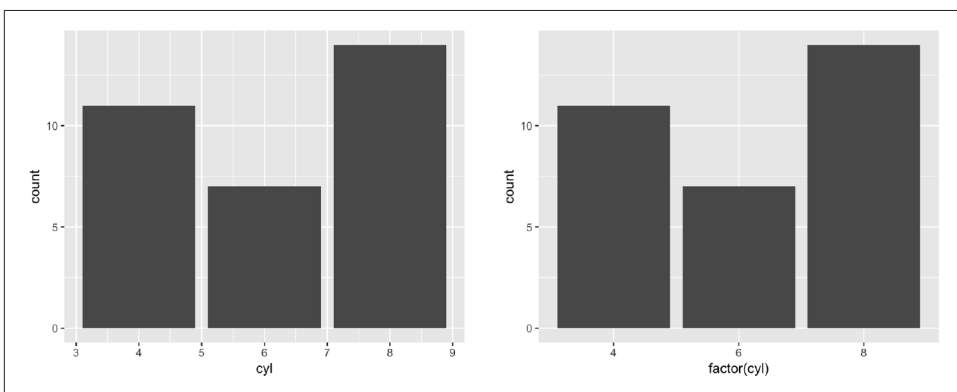


Figure 2-7. Bar graph of counts using `geom_bar()` with a continuous *x* variable (left); With *x* variable converted to a factor (right)



In previous versions of ggplot2, the recommended way to create a bar graph of values was to use `geom_bar(stat = "identity")`. As of ggplot2 2.2.0, there is a `geom_col()` function that does the same thing.

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)
```

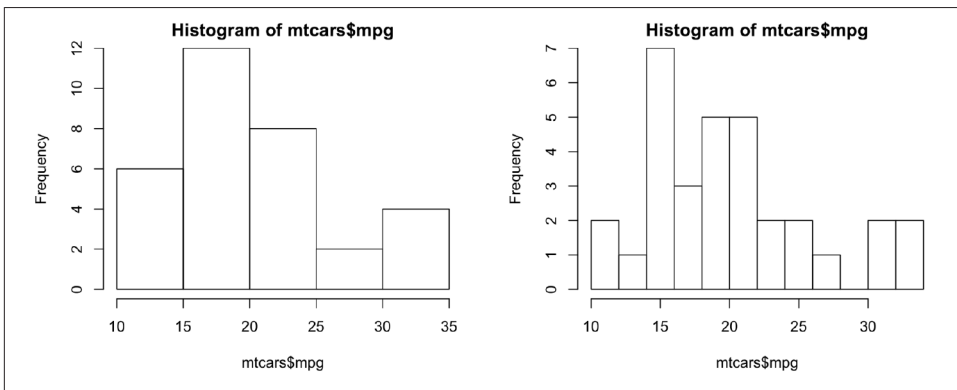


Figure 2-8. Histogram with base graphics (left); With more bins (right). Notice that when the bins are narrower, there are fewer items in each bin.

With the ggplot2, you can get a similar result using `geom_histogram()` ([Figure 2-9](#)):

```
library(ggplot2)
ggplot(mtcars, aes(x = mpg)) +
  geom_histogram()
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
# With wider bins
ggplot(mtcars, aes(x = mpg)) +
  geom_histogram(binwidth = 4)
```

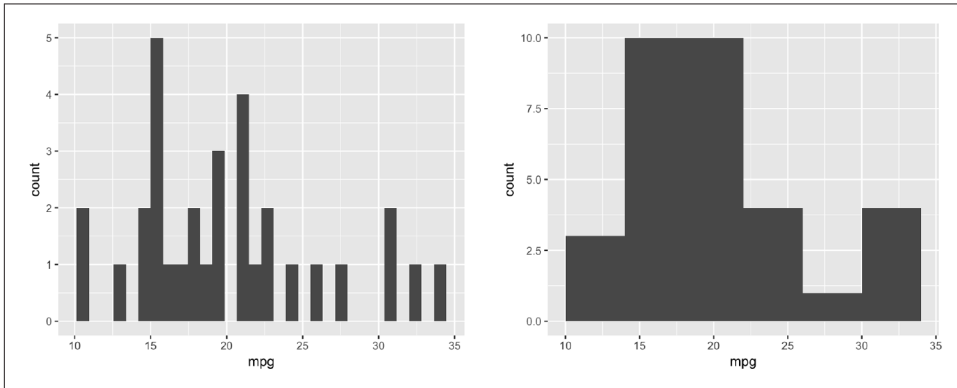


Figure 2-9. ggplot2 histogram with default bin width (left); With wider bins (right)

When you create a histogram without specifying the bin width, `ggplot()` prints out a message telling you that it's defaulting to 30 bins, and to pick a better bin width. This is because it's important to explore your data using different bin widths; the default of 30 may or may not show you something useful about your data.

See Also

For more in-depth information about creating histograms, see Recipes [6.1](#) and [6.2](#).

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)
```

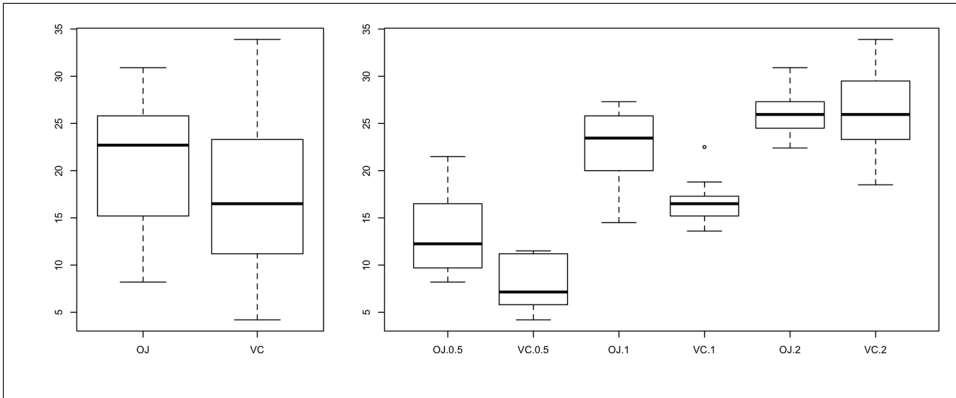


Figure 2-10. Box plot with base graphics (left); With multiple grouping variables (right)

If the two vectors are in the same data frame, you can also use the `boxplot()` function with 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 (Figure 2-11, left) with `geom_boxplot()`:

```
library(ggplot2)
ggplot(ToothGrowth, aes(x = supp, y = len)) +
  geom_boxplot()
```

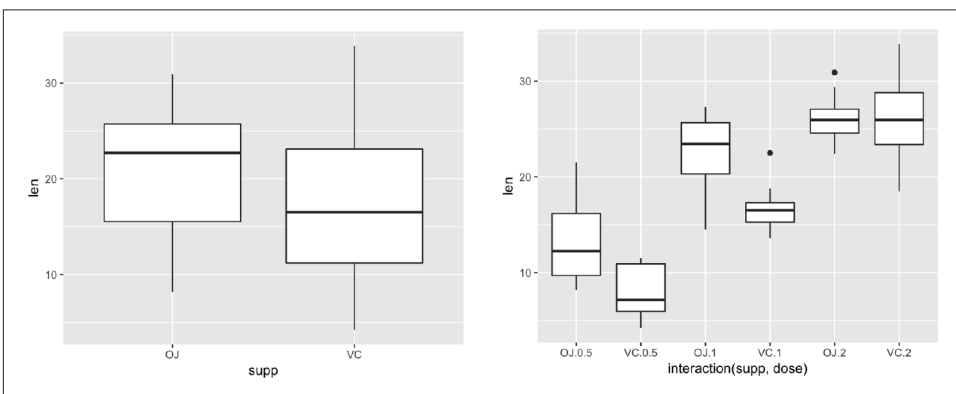


Figure 2-11. Box plot with `ggplot()` (left); With multiple grouping variables (right)

It's also possible to make box plots for multiple variables, by combining the variables with `interaction()`, as in [Figure 2-11](#), right:

```
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)
```

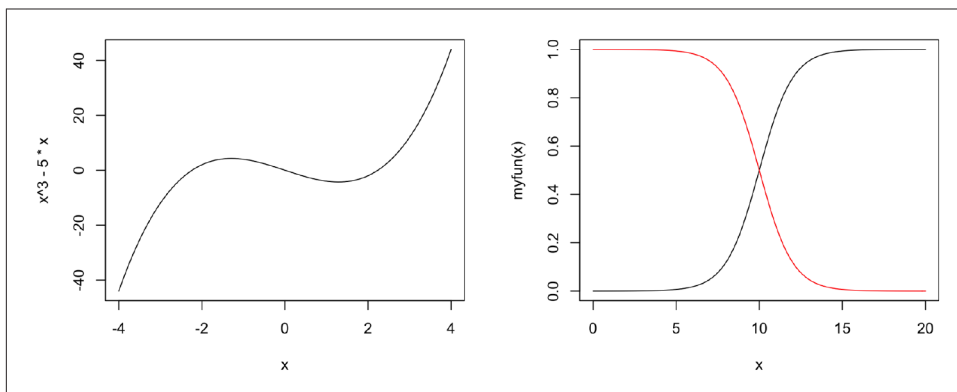


Figure 2-12. Function curve with base graphics (left); With user-defined function (right)

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

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

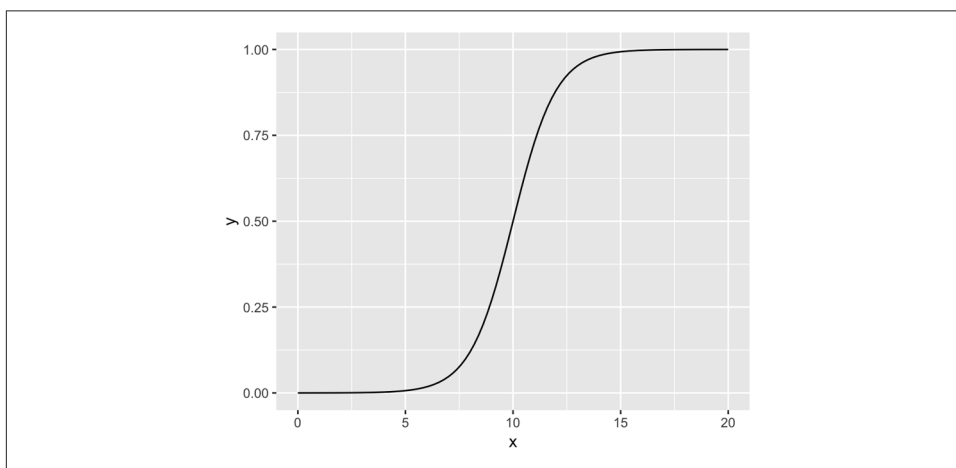


Figure 2-13. A function curve with `ggplot2`

See Also

See [Recipe 13.2](#) for more in-depth information about plotting function curves.

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. In this chapter I'll discuss this more, and present recipes for both types of bar graphs.

From this chapter on, this book will focus on using `ggplot2` instead of base R graphics. Using `ggplot2` will both keep things simpler and make for more sophisticated graphics.

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_col()` and specify what variables you want on the *x*- and *y*-axes (Figure 3-1):

```
library(gcookbook) # Load gcookbook for the pg_mean data set
ggplot(pg_mean, aes(x = group, y = weight)) +
  geom_col()
```

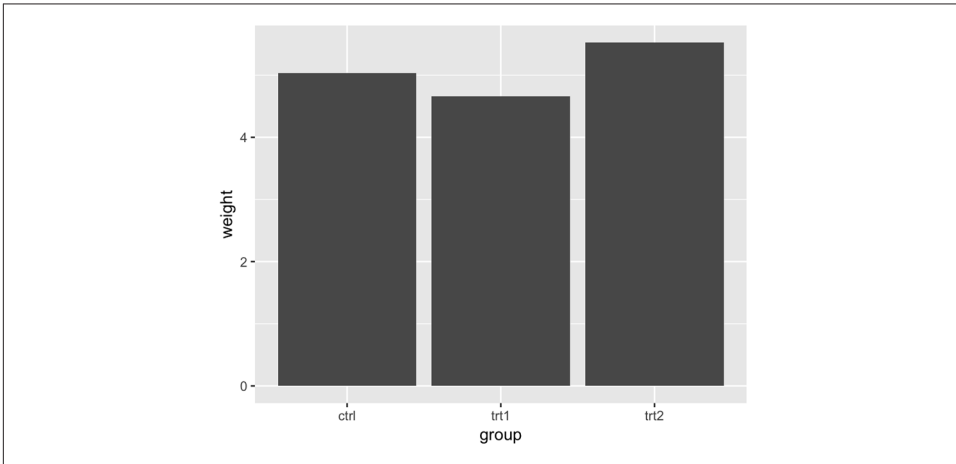


Figure 3-1. Bar graph of values with a discrete x-axis



In previous versions of ggplot2, the recommended way to create a bar graph of values was to use `geom_bar(stat = "identity")`. As of ggplot2 2.2.0, there is a `geom_col()` function that does the same thing.

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     1    8.3
#> 2     2   10.3
#> 3     3   19.0
#> 4     4   16.0
#> 5     5   15.6
#> 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_col()

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

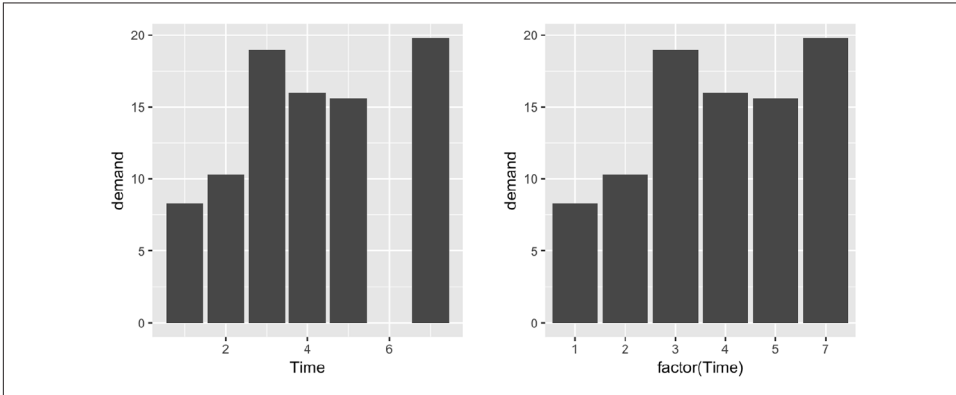


Figure 3-2. Bar graph of values with a continuous x-axis (left); With x variable converted to a factor (notice that the space for 6 is gone; right)

Notice that there was no row in BOD for Time = 6. When the x variable is continuous, ggplot2 will use a numeric axis that will have space for all numeric values within the range—hence the empty space for 6 in the plot. When Time is converted to a factor, ggplot2 uses it as a discrete variable, where the values are treated as arbitrary labels instead of numeric values, and so it won't allocate space on the x-axis for all possible numeric values between the minimum and maximum.

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 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_col(fill = "lightblue", colour = "black")
```

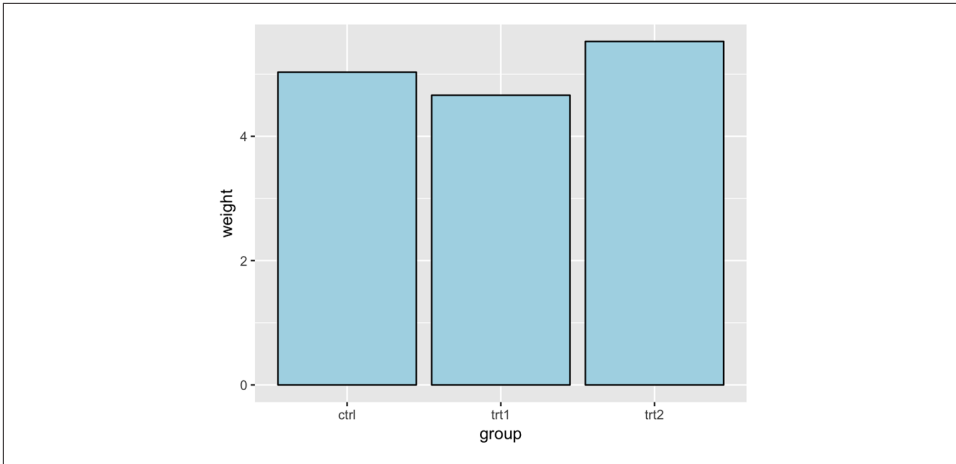


Figure 3-3. A single fill and outline color for all bars



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_col(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) # Load gcookbook for the cabbage_exp data set
cabbage_exp
#>   Cultivar Date Weight      sd    n      se
#> 1    c39  d16   3.18 0.9566144 10 0.30250803
#> 2    c39  d20   2.80 0.2788867 10 0.08819171
#> 3    c39  d21   2.74 0.9834181 10 0.31098410
#> 4    c52  d16   2.26 0.4452215 10 0.14079141
#> 5    c52  d20   3.11 0.7908505 10 0.25008887
#> 6    c52  d21   1.47 0.2110819 10 0.06674995
```

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_col(position = "dodge")
```

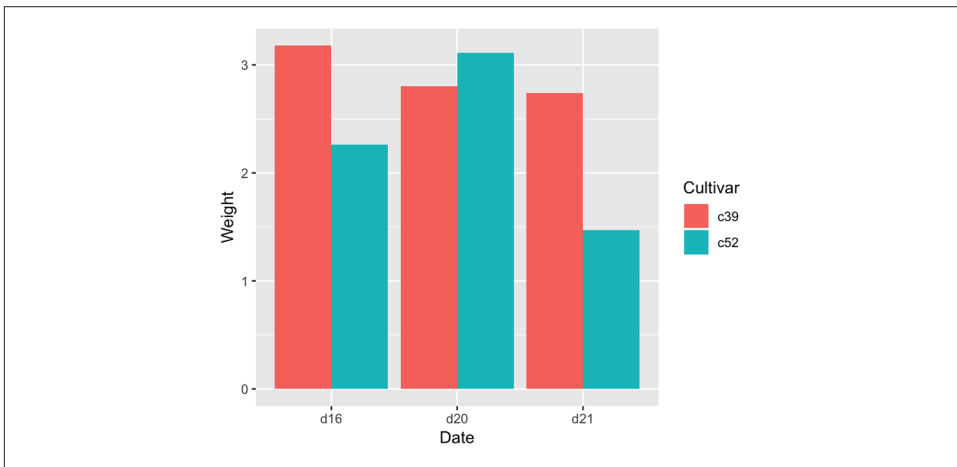


Figure 3-4. Graph with grouped bars

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_col()`. 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_col(position = "dodge", colour = "black") +
  scale_fill_brewer(palette = "Pastel1")
```

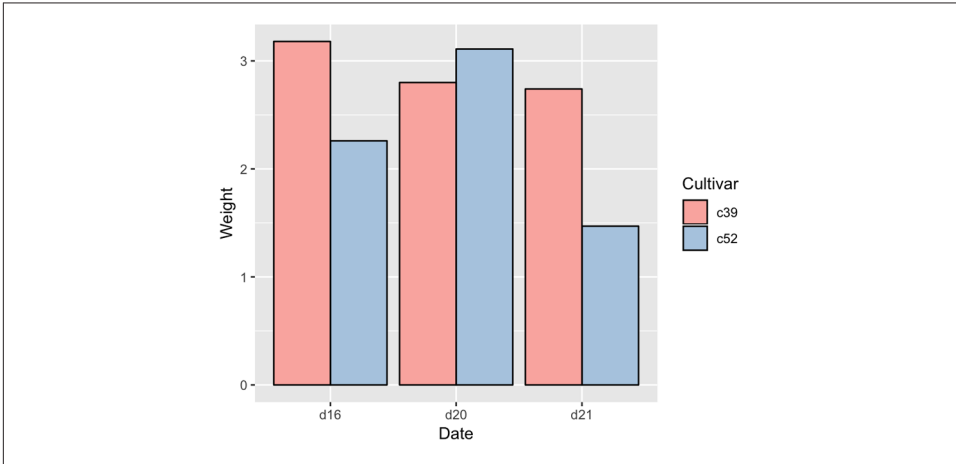


Figure 3-5. Grouped bars with black outline and a different color palette

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, ]
ce
#>   Cultivar Date Weight      sd    n      se
#> 1     c39  d16   3.18 0.9566144  10 0.30250803
#> 2     c39  d20   2.80 0.2788867  10 0.08819171
#> 3     c39  d21   2.74 0.9834181  10 0.31098410
#> 4     c52  d16   2.26 0.4452215  10 0.14079141
#> 5     c52  d20   3.11 0.7908505  10 0.25008887

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

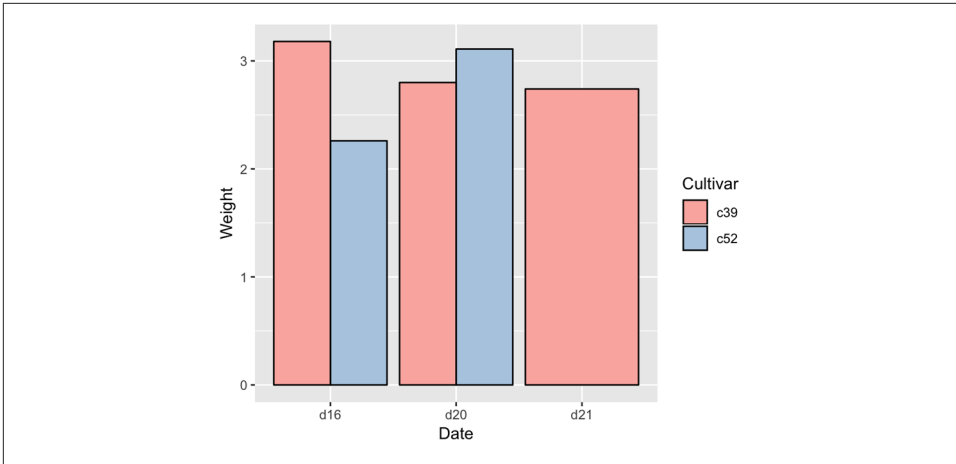


Figure 3-6. Graph with a missing bar—the other bar fills the space

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](#)):

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

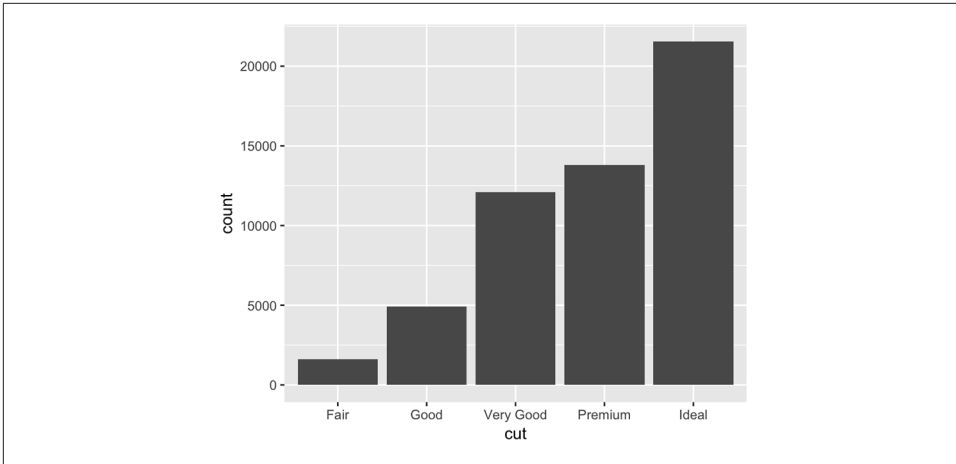


Figure 3-7. Bar graph of counts

Discussion

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

```
diamonds
#> # A tibble: 53,940 x 10
#>   carat cut      color clarity depth table price     x     y     z
#>   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
#> 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
#> 4 0.290 Premium  I      VS2     62.4   58   334   4.2    4.23   2.63
#> 5 0.31 Good     J      SI2     63.3   58   335   4.34   4.35   2.75
#> 6 0.24 Very Good J      VVS2     62.8   57   336   3.94   3.96   2.48
#> # ... with 5.393e+04 more rows
```

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 bar at each unique x value in the data, as shown in [Figure 3-8, top](#).

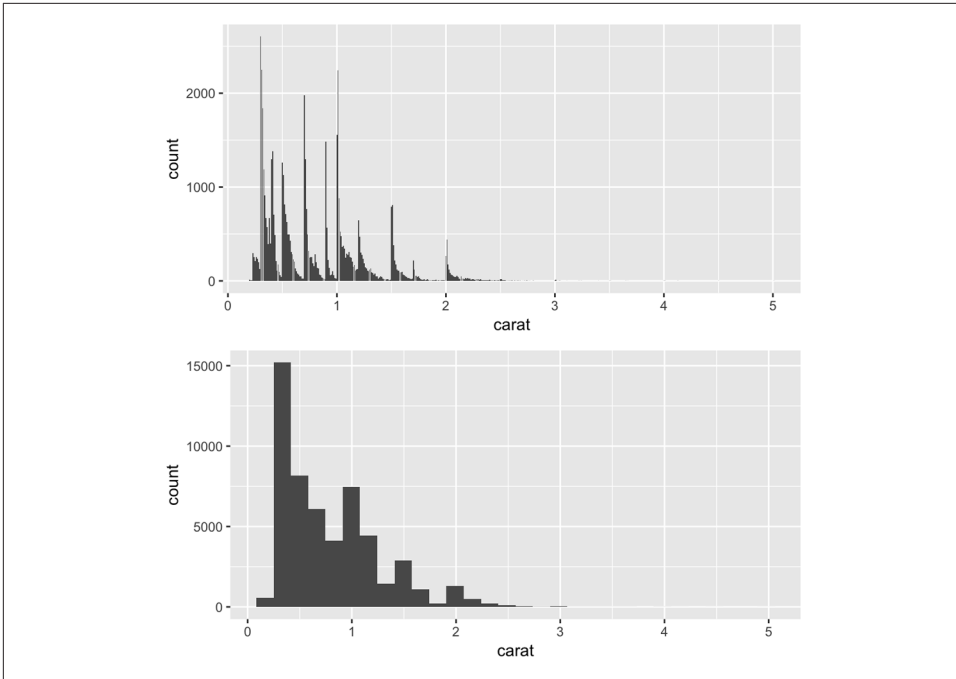


Figure 3-8. Bar graph of counts on a continuous axis (top); A histogram (bottom)

The bar graph with a continuous x-axis is similar to a histogram, but not the same. A histogram is shown on the bottom of [Figure 3-8](#). In this kind of bar graph, each bar represents a unique x value, whereas in a histogram, each bar represents a *range* of x values.

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, use `geom_col()`. 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, take the top 10 states:

```
library(gcookbook) # Load gcookbook for the uspopchange data set
library(dplyr)

upc <- uspopchange %>%
  arrange(desc(Change)) %>%
  slice(1:10)

upc
#>      State Abb Region Change
#> 1     Nevada NV  West   35.1
#> 2    Arizona AZ  West   24.6
#> 3      Utah  UT  West   23.8
#> ...<4 more rows>...
#> 8     Florida FL  South   17.6
#> 9    Colorado CO  West   16.9
#> 10 South Carolina SC South   15.3
```

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

```
ggplot(upc, aes(x = Abb, y = Change, fill = Region)) +
  geom_col()
```

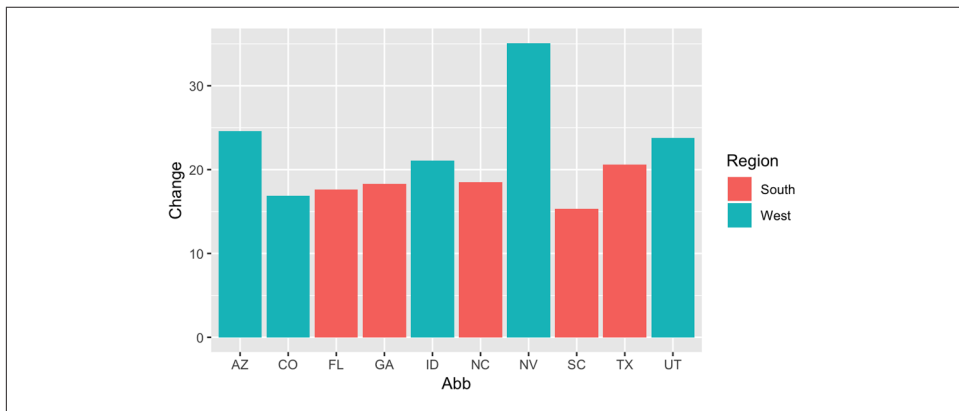


Figure 3-9. A variable mapped to fill

Discussion

The default colors aren't the most 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_col(colour = "black") +  
  scale_fill_manual(values = c("#669933", "#FFCC66")) +  
  xlab("State")
```

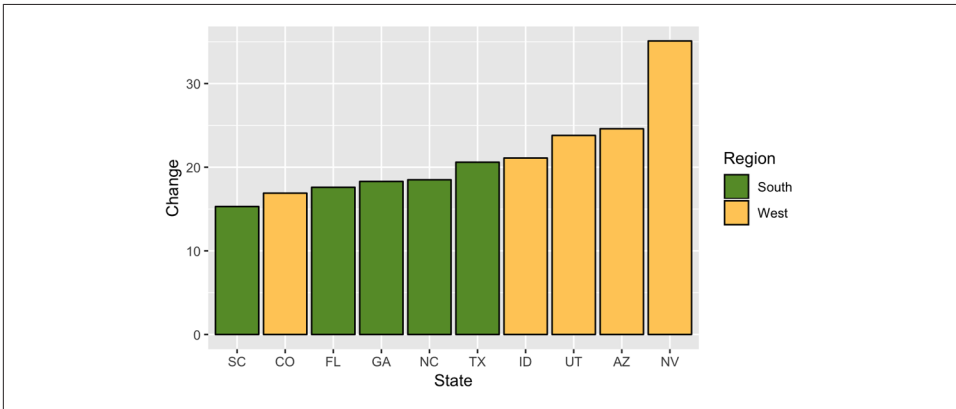


Figure 3-10. Graph with different colors, black outlines, and sorted by percentage change

This example also uses the `reorder()` function to reorder the levels of the factor `Abb` based on the values of `Change`. 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 `reorder()`, 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) # Load gcookbook for the climate data set
library(dplyr)

climate_sub <- climate %>%
  filter(Source == "Berkeley" & Year >= 1900) %>%
  mutate(pos = Anomaly10y >= 0)

climate_sub
#>   Source Year Anomaly1y Anomaly5y Anomaly10y Unc10y   pos
#> 1 Berkeley 1900      NA      NA      -0.171  0.108 FALSE
#> 2 Berkeley 1901      NA      NA      -0.162  0.109 FALSE
#> 3 Berkeley 1902      NA      NA      -0.177  0.108 FALSE
#> ...<99 more rows>...
#> 103 Berkeley 2002      NA      NA       0.856  0.028  TRUE
#> 104 Berkeley 2003      NA      NA       0.869  0.028  TRUE
#> 105 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(climate_sub, aes(x = Year, y = Anomaly10y, fill = pos)) +
  geom_col(position = "identity")
```

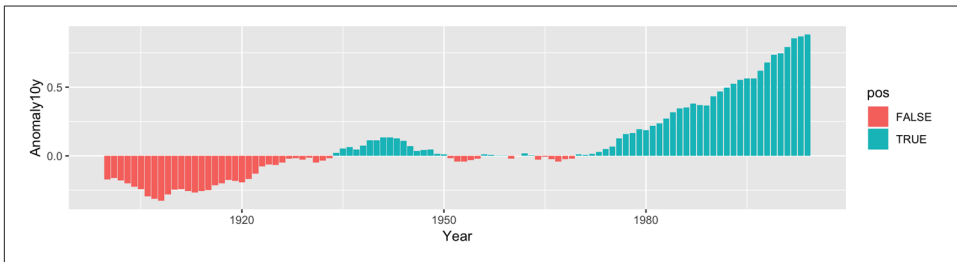


Figure 3-11. Different colors for positive and negative values

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

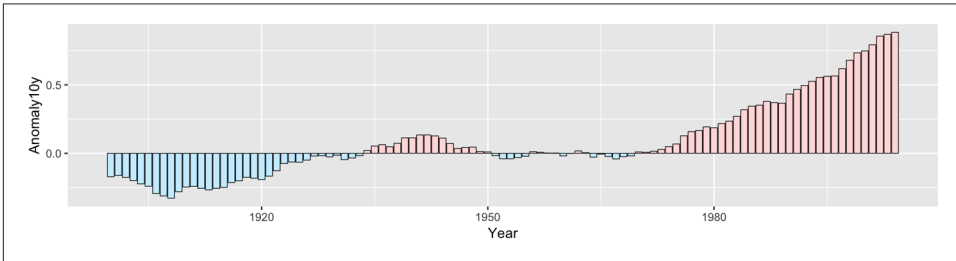


Figure 3-12. Graph with customized colors and no legend

See Also

To change the colors used, see Recipes [12.4](#) and [12.5](#).

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_col()`. 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) # Load gcookbook for the pg_mean data set

ggplot(pg_mean, aes(x = group, y = weight)) +
  geom_col()
```

For narrower bars:

```
ggplot(pg_mean, aes(x = group, y = weight)) +
  geom_col(width = 0.5)
```

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

```
ggplot(pg_mean, aes(x = group, y = weight)) +
  geom_col(width = 1)
```

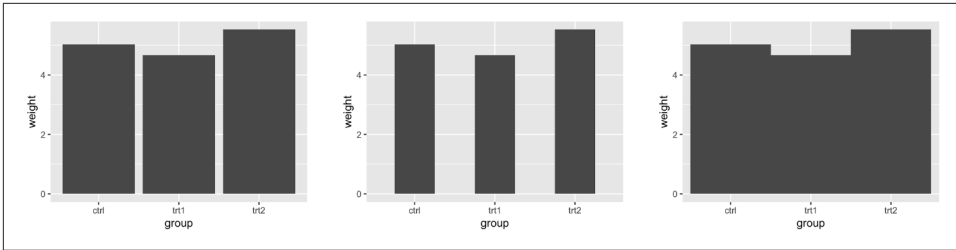


Figure 3-13. Different bar widths

For grouped bars, the default is to have no space between bars within each group. To add space between bars within a group, make the 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_col(width = 0.5, position = "dodge")
```

And with some space between the bars:

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

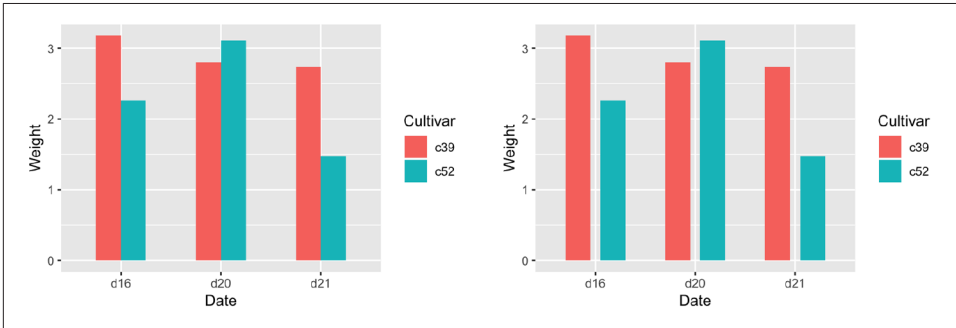


Figure 3-14. Bar graph with narrow grouped bars (left); With space between the bars (right)

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 form.

Discussion

The default width for bars 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 `NULL`, which tells ggplot2 to use the same value as the width from `geom_bar()`.

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 left has a bar width of 0.9, the right has a bar width of 0.2. Despite the different bar widths, the middles of the bars stay aligned.

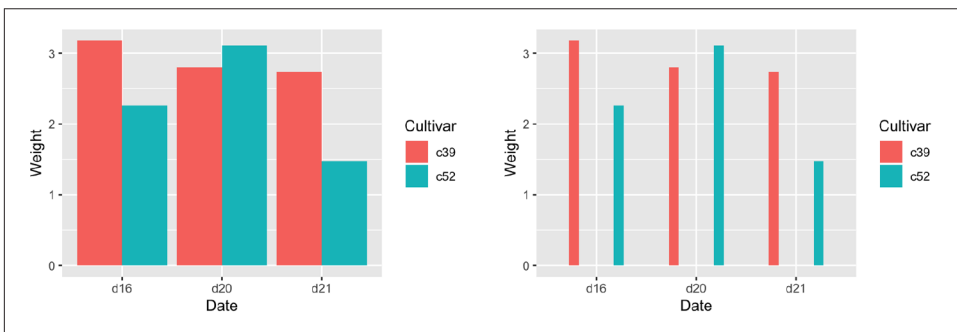


Figure 3-15. Same *dodge* width of 0.9, but different bar widths of 0.9 (left) and 0.2 (right)

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 to `fill`. This will put `Date` on the x-axis and use `Cultivar` for the fill color, as shown in [Figure 3-16](#):

```
library(gcookbook) # Load gcookbook for the cabbage_exp data set

ggplot(cabbage_exp, aes(x = Date, y = Weight, fill = Cultivar)) +
  geom_col()
```

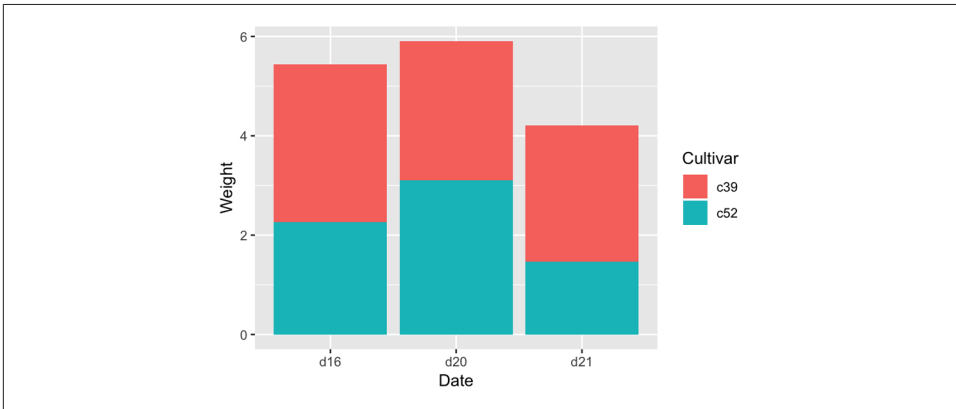


Figure 3-16. Stacked bar graph

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
#> 1     c39  d16   3.18 0.9566144  10 0.30250803
#> 2     c39  d20   2.80 0.2788867  10 0.08819171
#> 3     c39  d21   2.74 0.9834181  10 0.31098410
#> 4     c52  d16   2.26 0.4452215  10 0.14079141
#> 5     c52  d20   3.11 0.7908505  10 0.25008887
#> 6     c52  d21   1.47 0.2110819  10 0.06674995
```

By default, the stacking order of the bars is the same as the order of items in the legend. For some data sets it might make sense to reverse the order of the legend. To do this, you can use the `guides` function and specify the aesthetic for which the legend should be reversed. In this case, it's `fill` ([Figure 3-17](#)):

```
ggplot(cabbage_exp, aes(x = Date, y = Weight, fill = Cultivar)) +
  geom_col() +
  guides(fill = guide_legend(reverse = TRUE))
```

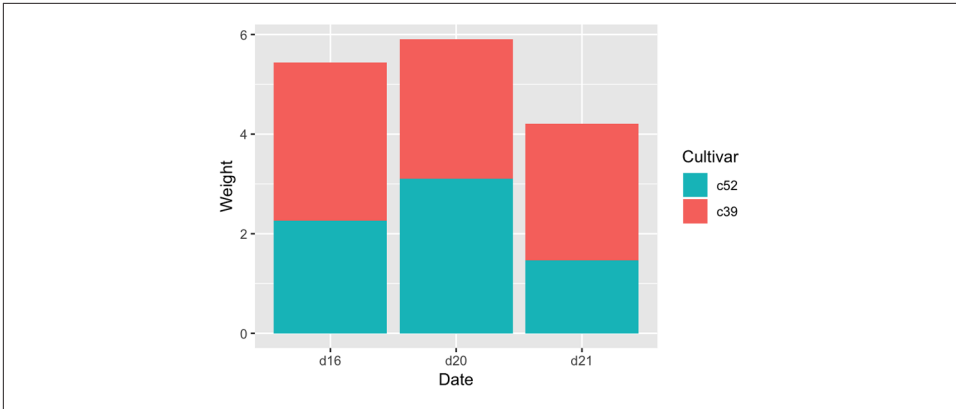



Figure 3-17. Stacked bar graph with reversed legend order

If you'd like to reverse the stacking order of the bars, as in [Figure 3-18](#), use `position_stack(reverse = TRUE)`. You'll also need to reverse the order of the legend for it to match the order of the bars:

```
ggplot(cabbage_exp, aes(x = Date, y = Weight, fill = Cultivar)) +
  geom_col(position = position_stack(reverse = TRUE)) +
  guides(fill = guide_legend(reverse = TRUE))
```

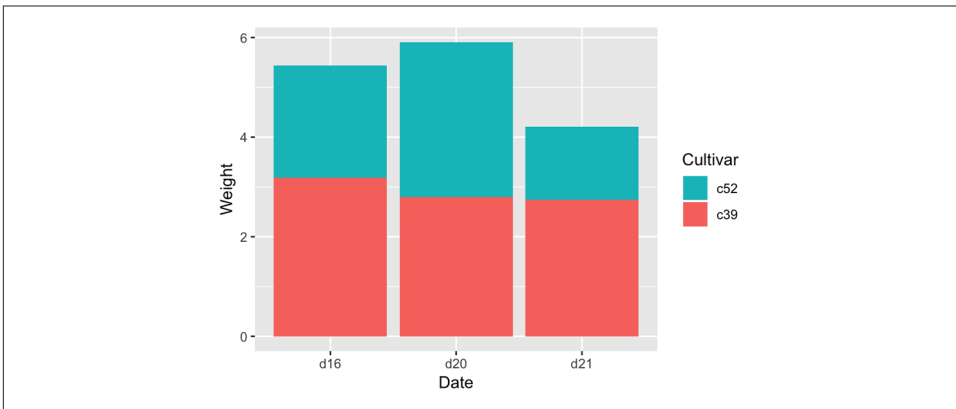


Figure 3-18. Stacked bar graph with reversed stacking order

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 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_col(colour = "black") +
  scale_fill_brewer(palette = "Pastel1")
```

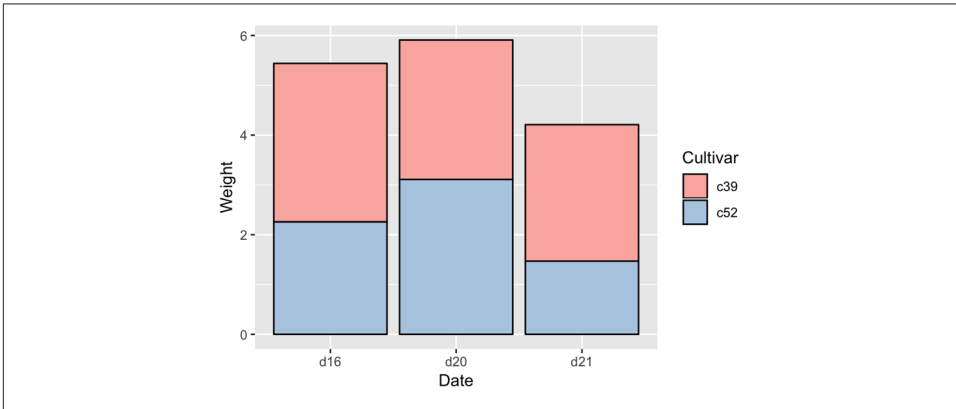


Figure 3-19. Stacked bar graph with reversed legend, new palette, and black outline

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).

Solution

Use `geom_col(position = "fill")` ([Figure 3-20](#)):

```
library(gcookbook) # Load gcookbook for the cabbage_exp data set

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

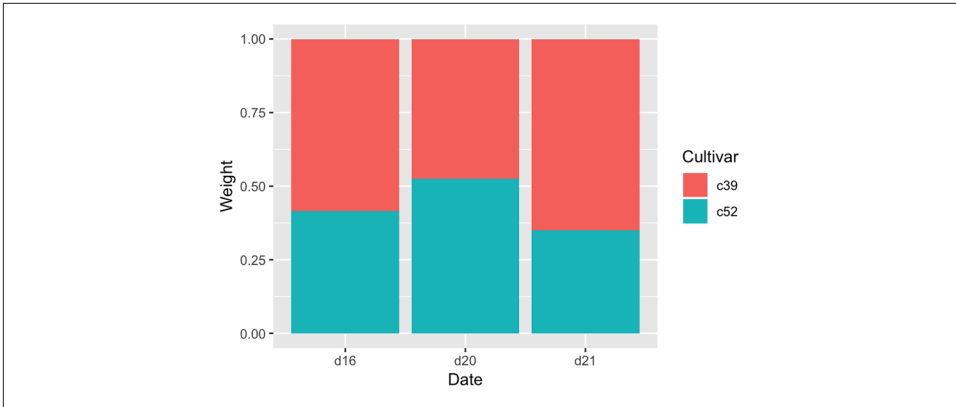


Figure 3-20. Proportional stacked bar graph

Discussion

With `position = "fill"`, the `y` values will be scaled to go from 0 to 1. To print the labels as percentages, use `scale_y_continuous(labels = scales::percent)`:

```
ggplot(cabbage_exp, aes(x = Date, y = Weight, fill = Cultivar)) +
  geom_col(position = "fill") +
  scale_y_continuous(labels = scales::percent)
```



Using `scales::percent` is a way of using the `percent` function from the `scales` package. You could instead do `library(scales)` and then just use `scale_y_continuous(labels = percent)`. This would also make all of the functions from `scales` available in the current R session.

To make the output look a little nicer, you can change the color palette and add an outline. This is shown in (Figure 3-21):

```
ggplot(cabbage_exp, aes(x = Date, y = Weight, fill = Cultivar)) +
  geom_col(colour = "black", position = "fill") +
  scale_y_continuous(labels = scales::percent) +
  scale_fill_brewer(palette = "Pastel1")
```

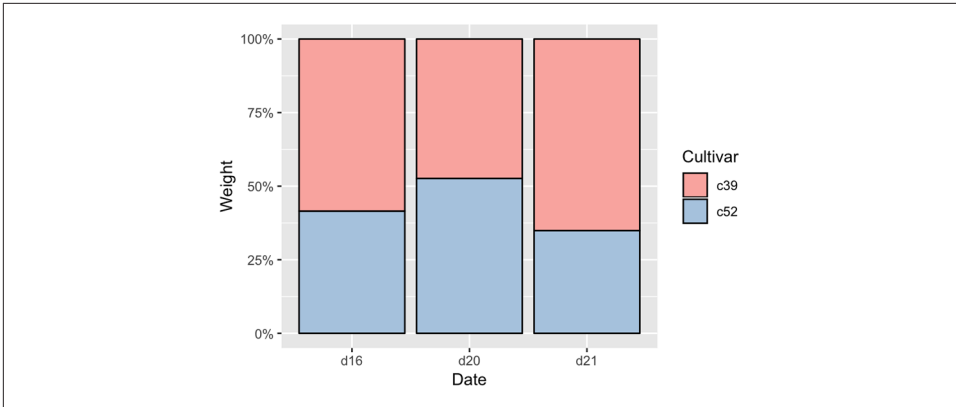


Figure 3-21. Proportional stacked bar graph with reversed legend, new palette, and black outline

Instead of having ggplot2 compute the proportions automatically, you may want to compute the proportional values yourself. This can be useful if you want to use those values in other computations.

To do this, first scale the data to 100% within each stack. This can be done by using `group_by()` together with `mutate()` from the dplyr package:

```
library(gcookbook)
library(dplyr)

cabbage_exp
#>   Cultivar Date Weight      sd    n      se
#> 1    c39  d16   3.18 0.9566144 10 0.30250803
#> 2    c39  d20   2.80 0.2788867 10 0.08819171
#> 3    c39  d21   2.74 0.9834181 10 0.31098410
#> 4    c52  d16   2.26 0.4452215 10 0.14079141
#> 5    c52  d20   3.11 0.7908505 10 0.25008887
#> 6    c52  d21   1.47 0.2110819 10 0.06674995

# Do a group-wise transform(), splitting on "Date"
ce <- cabbage_exp %>%
  group_by(Date) %>%
  mutate(percent_weight = Weight / sum(Weight) * 100)

ce
#> # A tibble: 6 x 7
#> # Groups:   Date [3]
#>   Cultivar Date Weight      sd    n      se percent_weight
#>   <fct>   <fct> <dbl> <dbl> <int> <dbl>         <dbl>
#> 1 c39     d16   3.18 0.957    10 0.303         58.5
#> 2 c39     d20   2.8 0.279    10 0.0882        47.4
#> 3 c39     d21   2.74 0.983    10 0.311         65.1
#> 4 c52     d16   2.26 0.445    10 0.141         41.5
```

```
#> 5 c52      d20      3.11 0.791      10 0.250      52.6
#> 6 c52      d21      1.47 0.211      10 0.0667     34.9
```

To calculate the percentages within each `Weight` group, we used `dplyr`'s `group_by()` and `mutate()` functions. In the example here, the `group_by()` function tells `dplyr` that future operations should operate on the data frame as though it were split up into groups, on the `Date` column. The `mutate()` function tells it to calculate a new column, dividing each row's `Weight` value by the sum of the `Weight` column *within each group*.



You may have noticed that `cabbage_exp` and `ce` print out differently. This is because `cabbage_exp` is a regular data frame, while `ce` is a *tibble*, which is a data frame with some extra properties. The `dplyr` package creates tibbles. For more information, see [Chapter 15](#).

After computing the new column, making the graph is the same as with a regular stacked bar graph:

```
ggplot(ce, aes(x = Date, y = percent_weight, fill = Cultivar)) +
  geom_col()
```

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) # Load gcookbook for the cabbage_exp data set

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

# Above the top
ggplot(cabbage_exp, aes(x = interaction(Date, Cultivar), y = Weight)) +
```

```
geom_col() +
  geom_text(aes(label = Weight), vjust = -0.2)
```

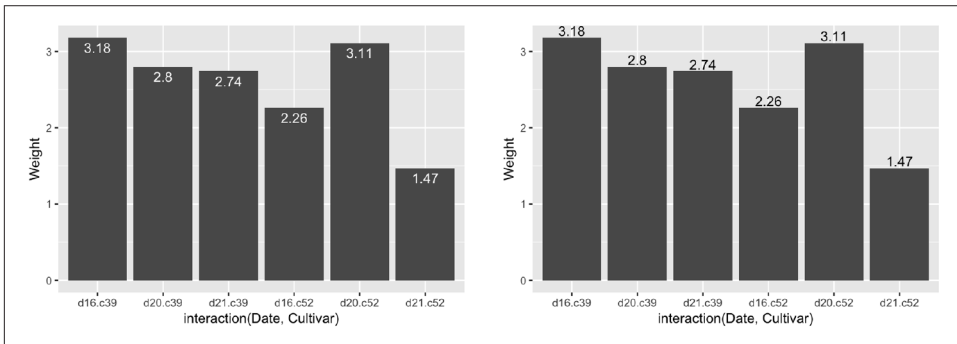


Figure 3-22. Labels under the tops of bars (left); Labels above bars (right)

Notice that when the labels are placed atop the bars, they may be clipped. To remedy this, see [Recipe 8.2](#).

Another common scenario is to add labels for a bar graph of *counts* instead of values. To do this, use `geom_bar()`, which adds bars whose height is proportional to the number of rows, and then use `geom_text()` with counts ([Figure 3-23](#)):

```
ggplot(mtcars, aes(x = factor(cyl))) +
  geom_bar() +
  geom_text(aes(label = ..count..), stat = "count", vjust = 1.5,
    colour = "white")
```

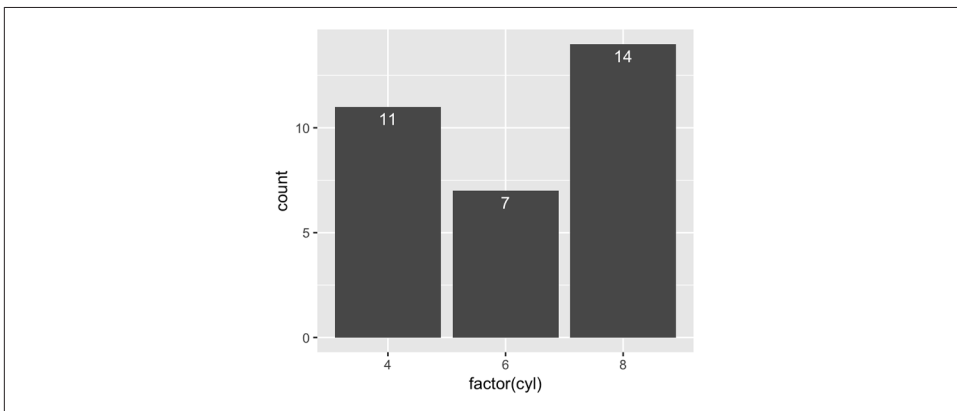


Figure 3-23. Bar graph of counts with labels under the tops of bars

We needed to tell `geom_text()` to use the "count" statistic to compute the number of rows for each x value, and then, to use those computed counts as the labels, we told it to use the aesthetic mapping `aes(label = ..count..)`.

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_col() +
  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_col() +
  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 3-24](#)):

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

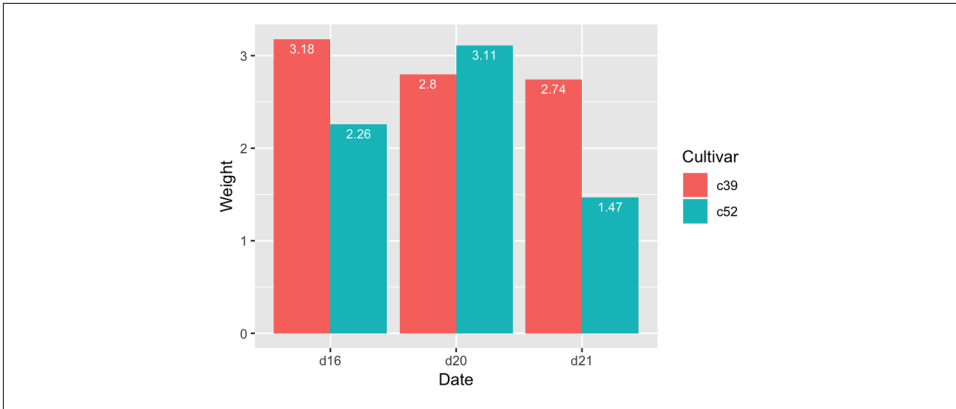


Figure 3-24. Labels on grouped bars

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 `dplyr` package. Note that we have to use the `rev()` function to reverse the order of Cultivar:

```
library(dplyr)

# Sort by the Date and Cultivar columns
ce <- cabbage_exp %>%
  arrange(Date, rev(Cultivar))
```

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

```
# Get the cumulative sum
ce <- ce %>%
  group_by(Date) %>%
  mutate(label_y = cumsum(Weight))

ce
#> # A tibble: 6 x 7
#> # Groups:   Date [3]
#>   Cultivar Date   Weight    sd      n      se label_y
#>   <fct>    <fct>   <dbl> <dbl> <int> <dbl> <dbl>
#> 1 c52     d16     2.26  0.445   10  0.141   2.26
#> 2 c39     d16     3.18  0.957   10  0.303   5.44
#> 3 c52     d20     3.11  0.791   10  0.250   3.11
#> 4 c39     d20     2.8   0.279   10  0.0882  5.91
#> 5 c52     d21     1.47  0.211   10  0.0667  1.47
#> 6 c39     d21     2.74  0.983   10  0.311  4.21

ggplot(ce, aes(x = Date, y = Weight, fill = Cultivar)) +
```



```
geom_col() +
  geom_text(aes(y = label_y, label = Weight), vjust = 1.5, colour = "white")
```

The result is shown in [Figure 3-25](#).

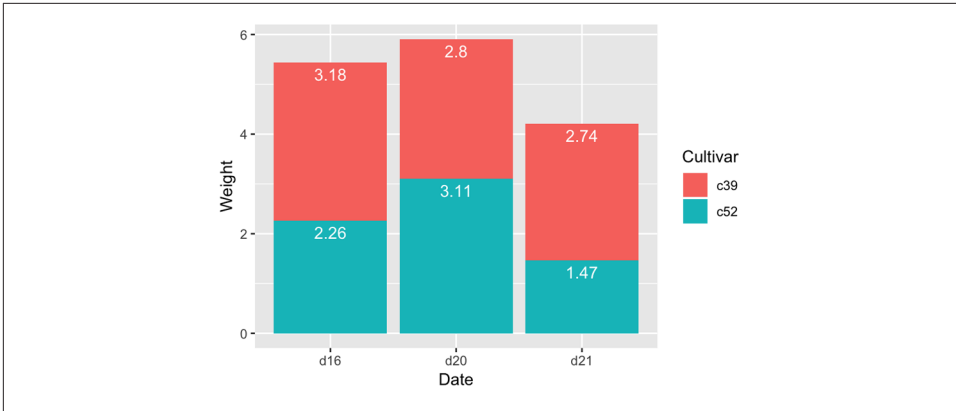


Figure 3-25. Labels on stacked bars

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 3-26](#)), there must be an adjustment to the cumulative sum, and the y offset in `geom_bar()` can be removed:

```
ce <- cabbage_exp %>%
  arrange(Date, rev(Cultivar))

# Calculate y position, placing it in the middle
ce <- ce %>%
  group_by(Date) %>%
  mutate(label_y = cumsum(Weight) - 0.5 * Weight)

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

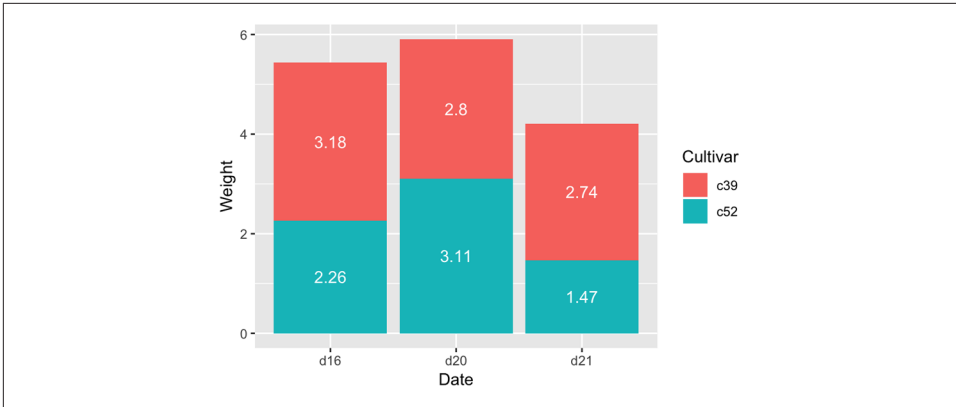


Figure 3-26. Labels in the middle of stacked bars

For a more polished graph (Figure 3-27), we'll change the colors, add labels in the middle with a smaller font using `size`, add a "kg" suffix 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_col(colour = "black") +
  geom_text(aes(y = label_y,
    label = paste(format(Weight, nsmall = 2), "kg")),
    size = 4) +
  scale_fill_brewer(palette = "Pastel1")
```

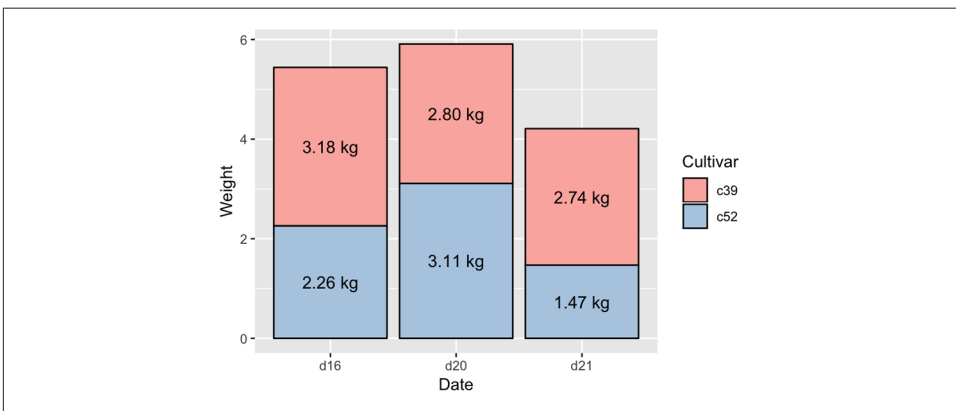


Figure 3-27. Customized stacked bar graph with labels

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 an alternative to bar graphs that reduce visual clutter and can be easier to read.

The simplest way to create a dot plot (as shown in [Figure 3-28](#)) is to use `geom_point()`:

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

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

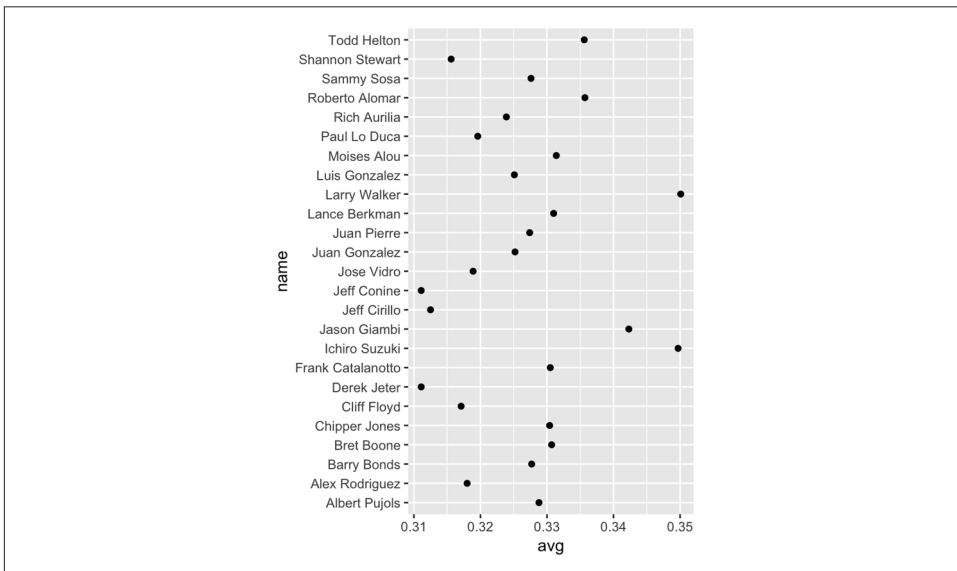


Figure 3-28. Basic dot plot

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
#> 1  Larry Walker NL 0.3501
#> 2  Ichiro Suzuki AL 0.3497
#> 3  Jason Giambi AL 0.3423
#> ...<19 more rows>...
#> 23 Jeff Cirillo NL 0.3125
#> 24 Jeff Conine AL 0.3111
#> 25 Derek Jeter AL 0.3111
```

In [Figure 3-28](#) 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.

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 3-29](#)):

```
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")
  )
```

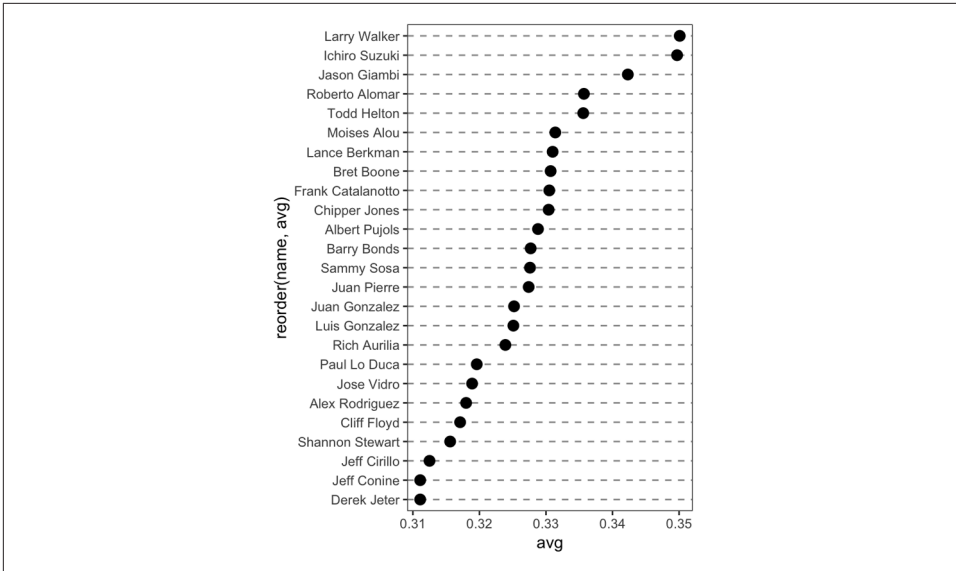


Figure 3-29. Dot plot, ordered by batting average

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 3-30. 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(
    panel.grid.major.y = element_blank(),
    panel.grid.minor.y = element_blank(),
    panel.grid.major.x = element_line(colour = "grey60", linetype = "dashed"),
    axis.text.x = element_text(angle = 60, hjust = 1)
  )
```

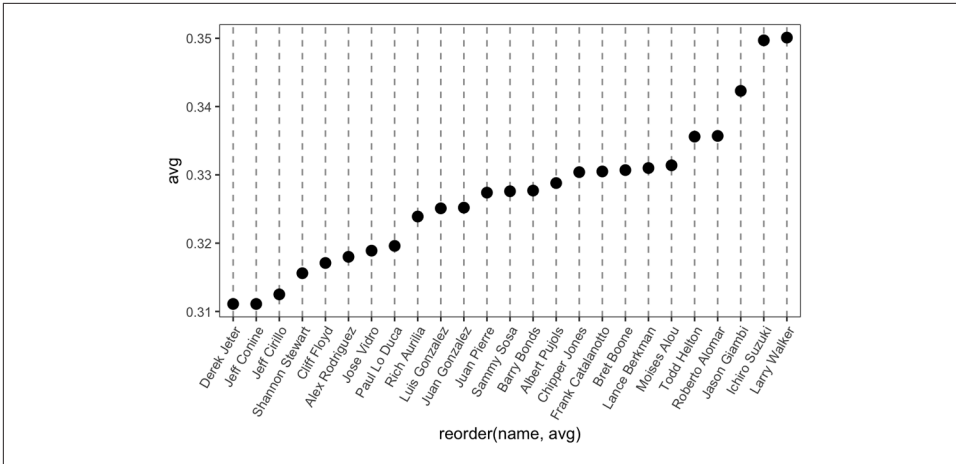


Figure 3-30. Dot plot with names on x-axis and values on y-axis

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:

```
# 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 3-31), 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)
  )
```

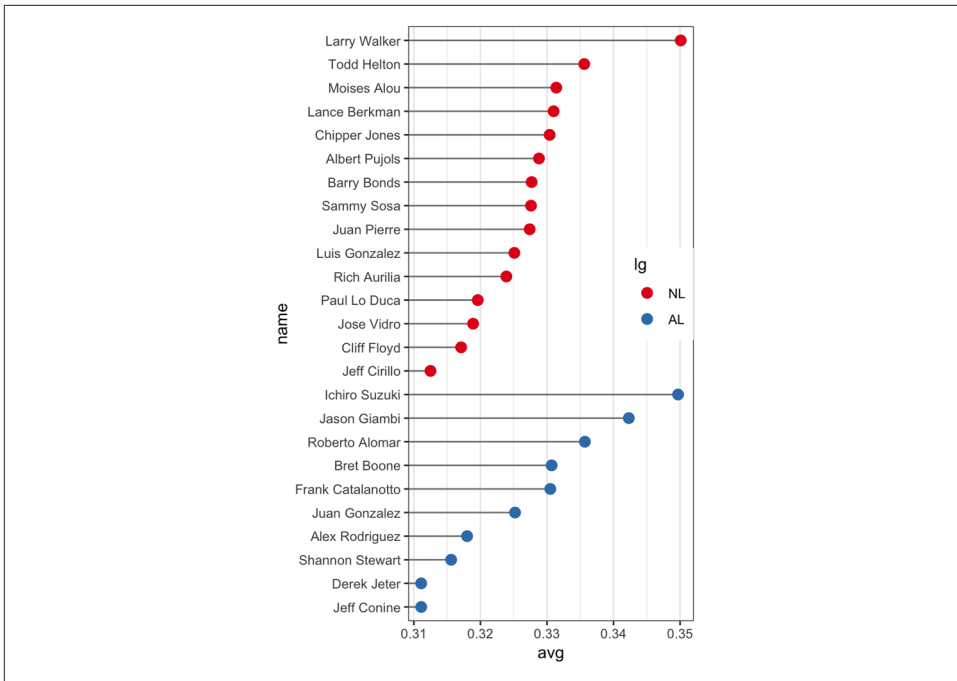


Figure 3-31. Grouped by league, with lines that stop at the point

Another way to separate the two groups is to use facets, as shown in Figure 3-32. The order in which the facets are displayed is different from the sorting order in Figure 3-31; 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")
```

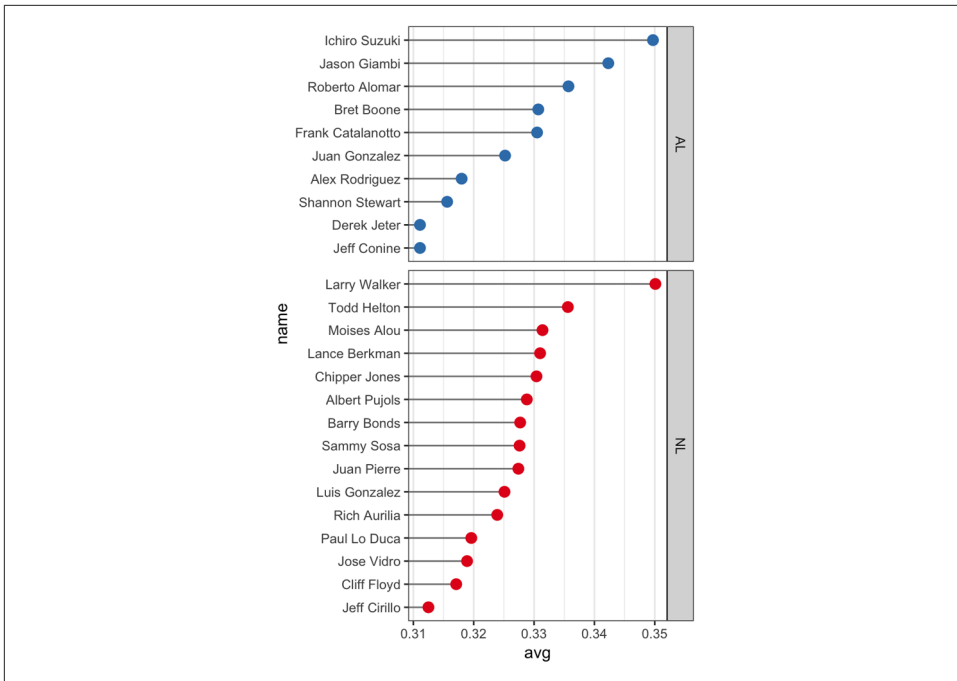


Figure 3-32. Faceted by league

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](#).

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; for example, 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 which variables you mapped to x and y (Figure 4-1):

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

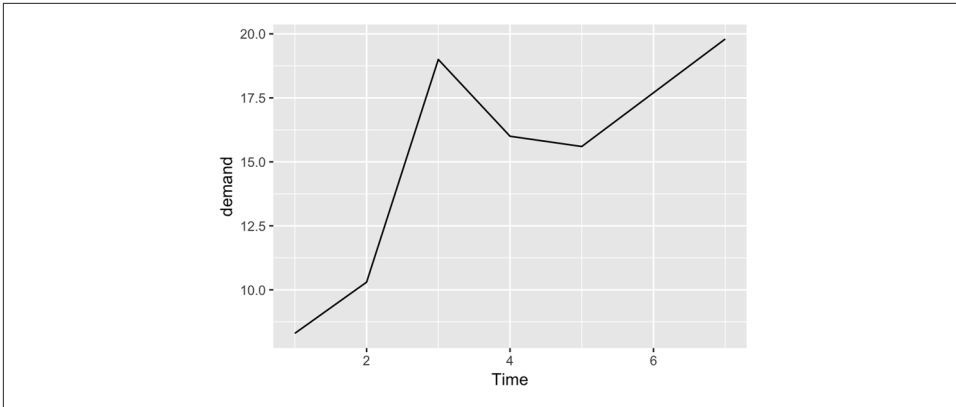


Figure 4-1. Basic line graph

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
#> 6     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()
```

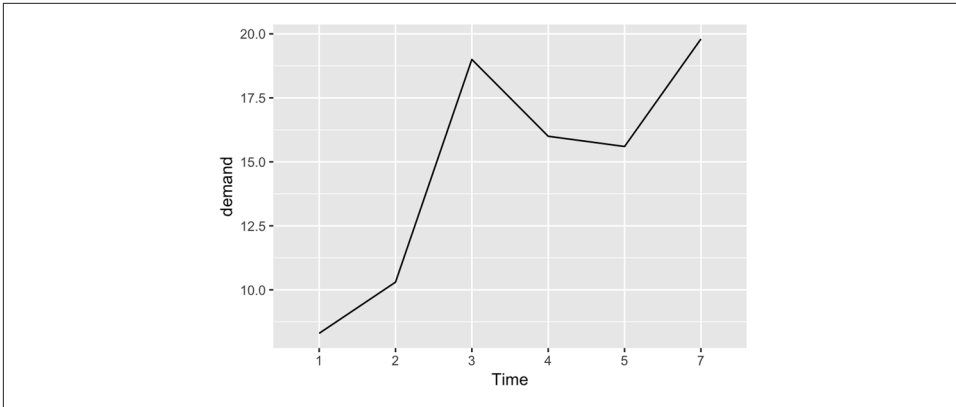


Figure 4-2. Basic line graph with a factor on the x-axis (notice that no space is allocated on the x-axis for 6)

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.

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)
```

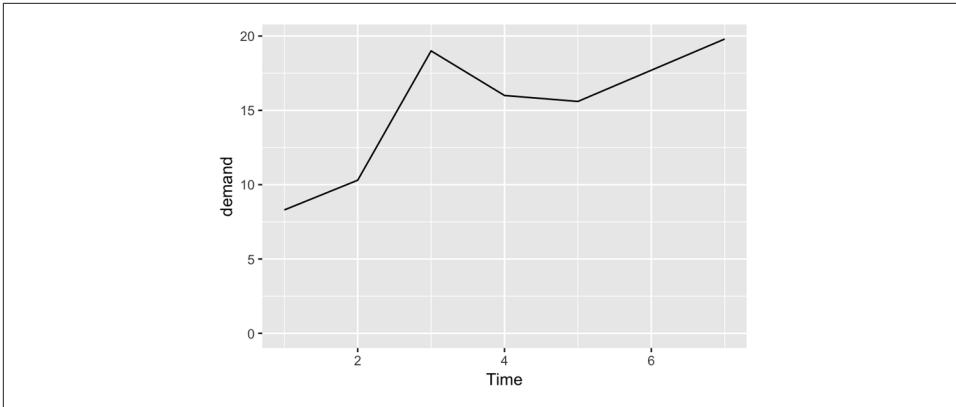


Figure 4-3. Line graph with manually set y range

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()
```

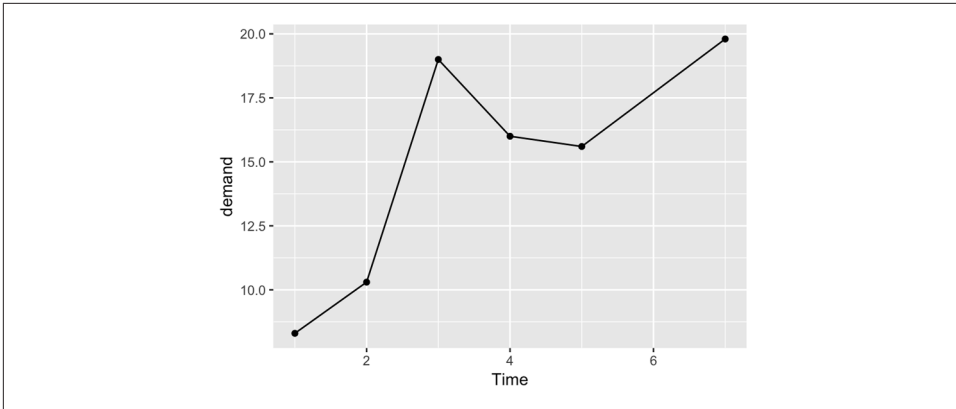


Figure 4-4. Line graph with points

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) # Load gcookbook for the worldpop 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()
```

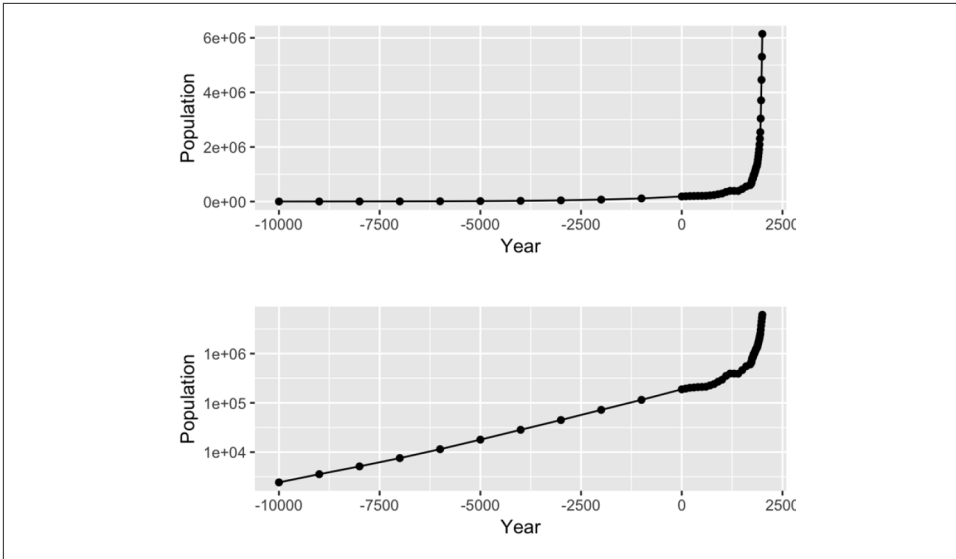


Figure 4-5. Top: points indicate where each data point is; Bottom: the same data with a log y-axis

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](#):

```
library(gcookbook) # Load gcookbook for the tg data set

# 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()
```

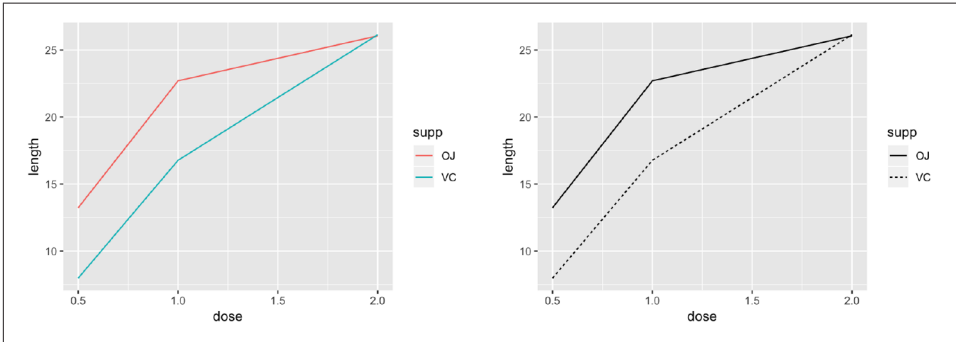


Figure 4-6. A variable mapped to colour (left); A variable mapped to linetype (right)

Discussion

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

```
tg
#>   supp dose length
#> 1   OJ  0.5  13.23
#> 2   OJ  1.0  22.70
#> 3   OJ  2.0  26.06
#> 4   VC  0.5   7.98
#> 5   VC  1.0  16.77
#> 6   VC  2.0  26.14
```



If the x variable is a factor, you must also tell ggplot to group by that same variable, as described next.

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()
```

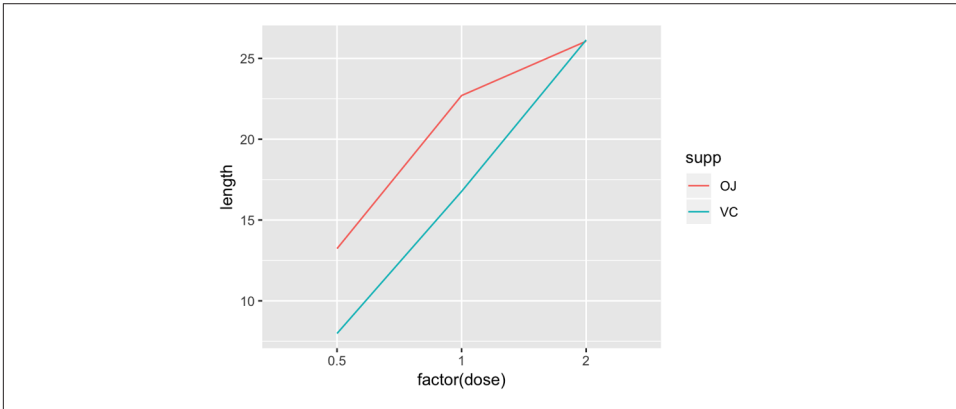


Figure 4-7. Line graph with continuous x variable converted to a factor

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()
```

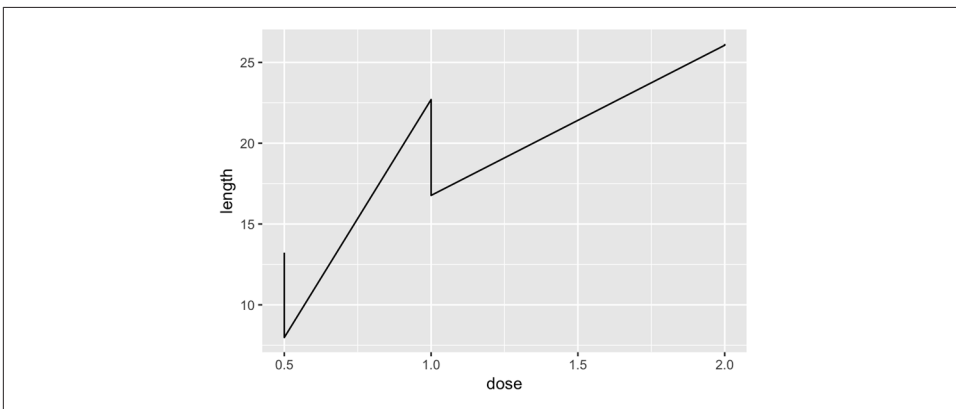


Figure 4-8. A sawtooth pattern indicates improper grouping

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
```

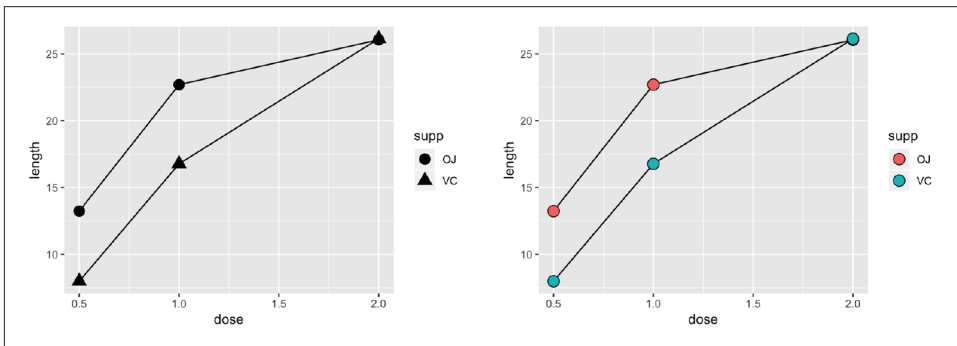


Figure 4-9. Line graph with different shapes (left); With different colors (right)

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
```

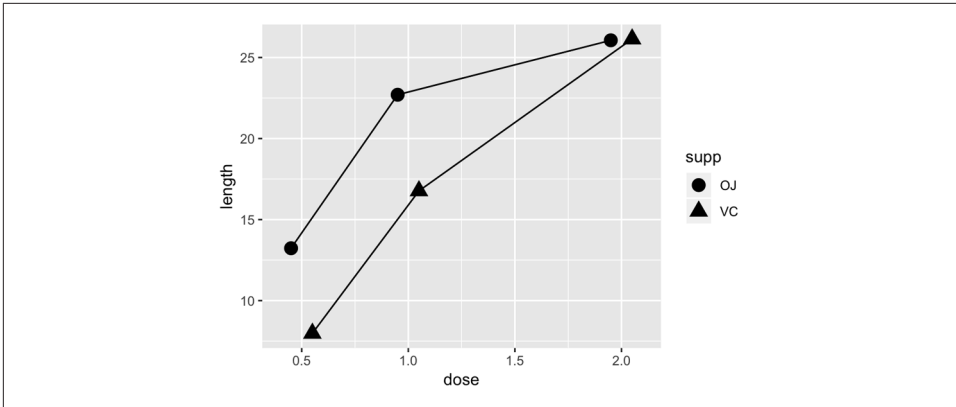


Figure 4-10. Dodging to avoid overlapping points

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` (or `color`).

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")
```

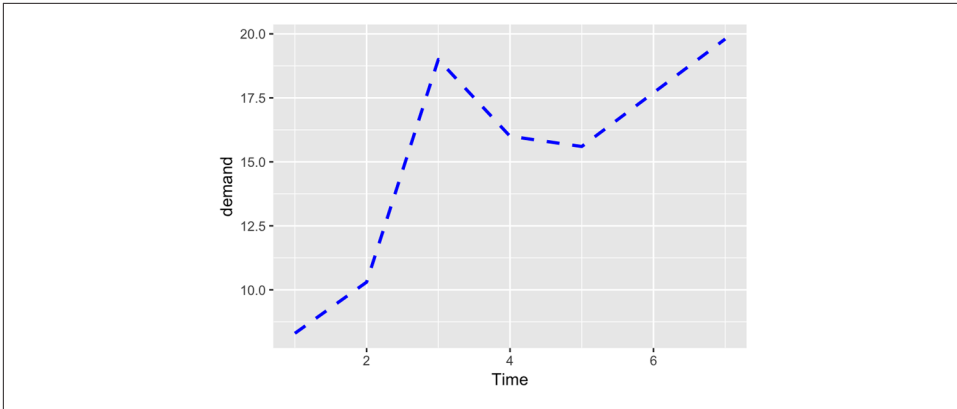


Figure 4-11. Line graph with custom linetype, size, and colour

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()`:

```
library(gcookbook) # Load gcookbook for the tg data set

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

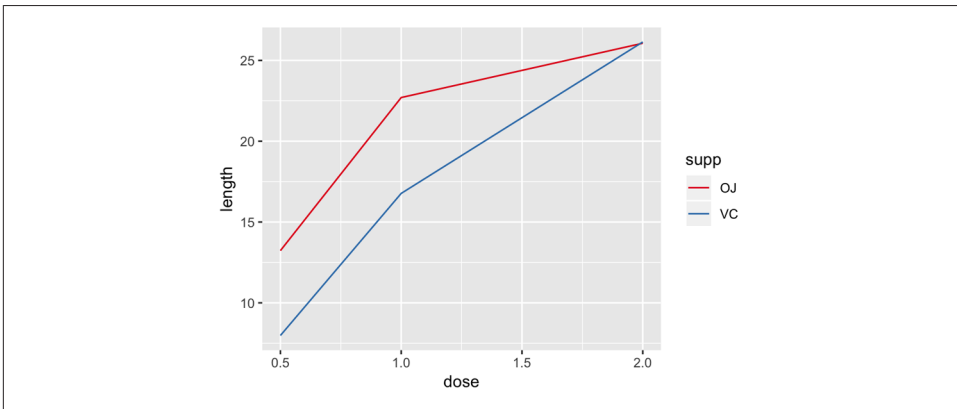


Figure 4-12. Using a palette from RColorBrewer

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")
```

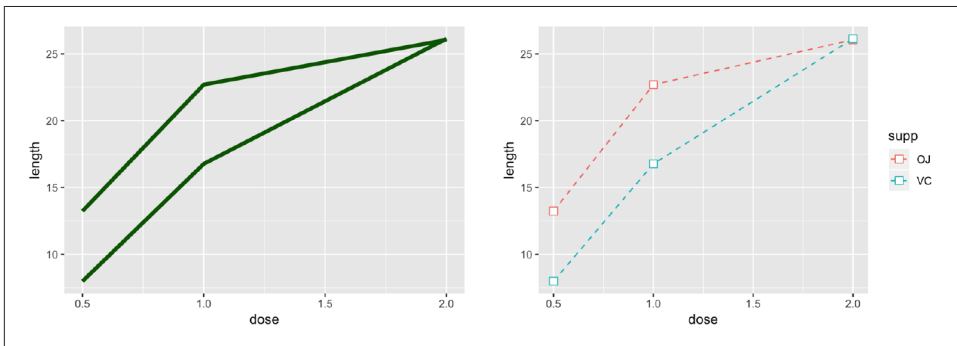


Figure 4-13. Line graph with constant size and color (left); With `supp` mapped to colour, and with points added (right)

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.

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")
```

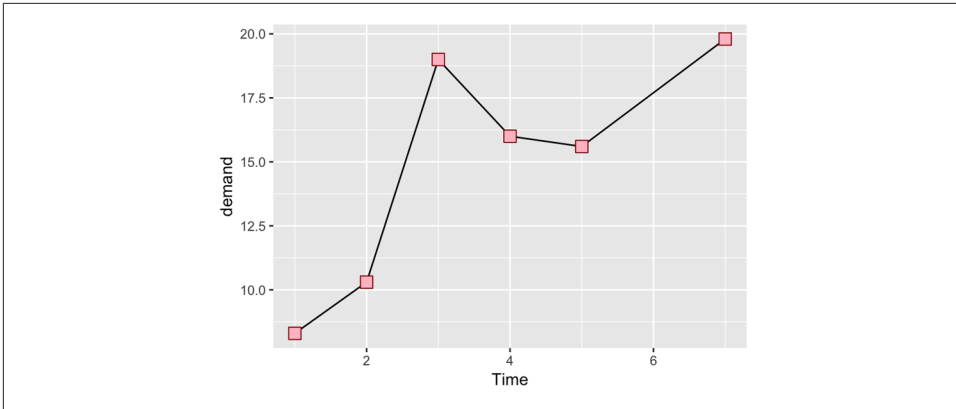


Figure 4-14. Points with custom size, shape, colour, and fill

Discussion

The default shape for points is a solid circle, the default size is 2, and the default color is black. The fill color is relevant only for some point shapes (numbered 21–25), which have separate outlines 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")
```

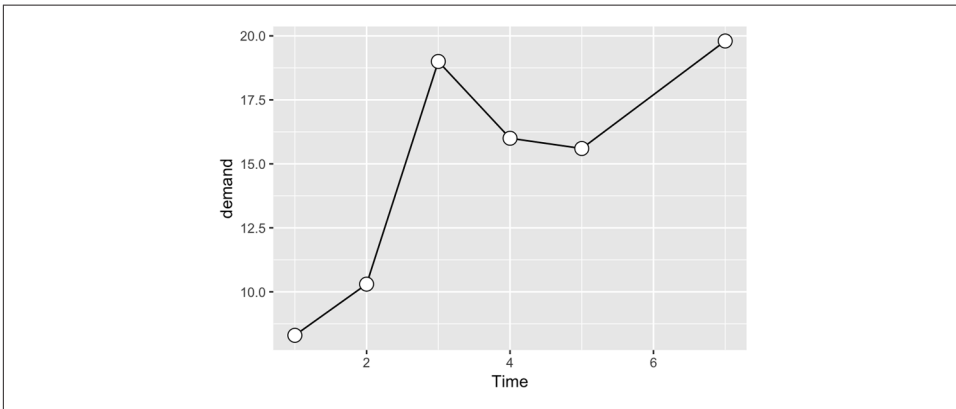


Figure 4-15. Points with a white fill

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()`:

```
library(gcookbook) # Load gcookbook for the tg data set

# 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"))
```

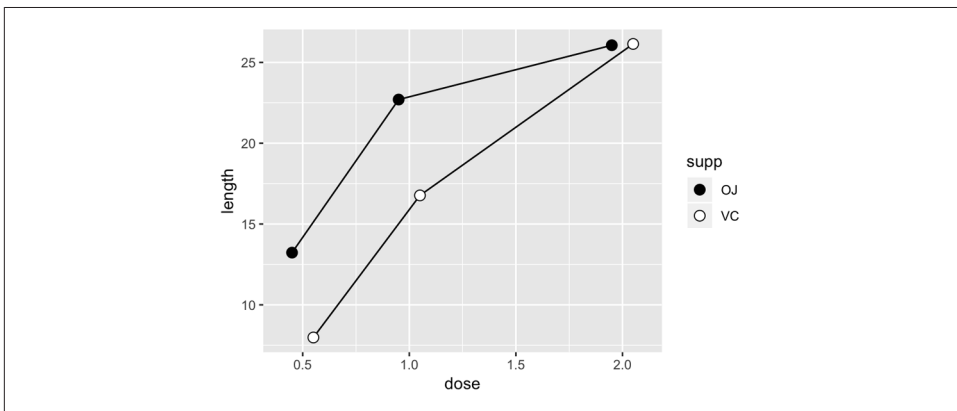


Figure 4-16. Line graph with manually specified fills of black and white, and a slight dodge

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()
```

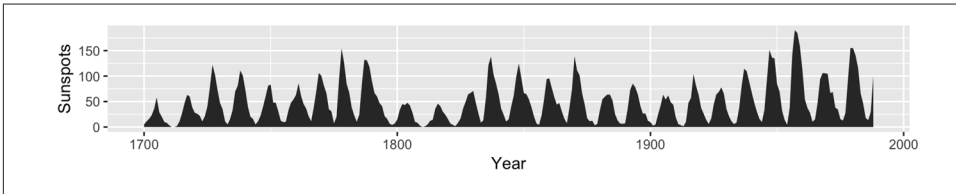


Figure 4-17. Graph with a shaded area

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)
```

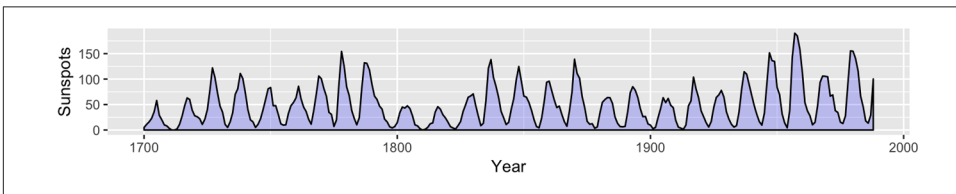


Figure 4-18. Graph with a semitransparent shaded area and an outline

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()
```

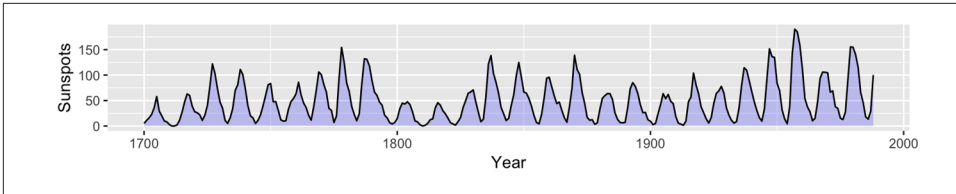


Figure 4-19. Line graph with a line just on top, using `geom_line()`

See Also

See [Chapter 12](#) for more on choosing colors.

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) # Load gcookbook for the uspage data set

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

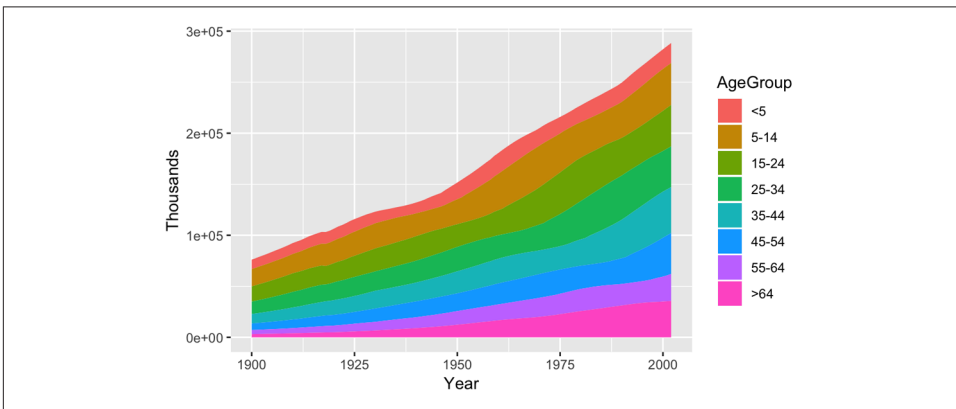


Figure 4-20. Stacked area graph

Discussion

The sort of data that is plotted with a stacked area chart is often provided in a wide format, but ggplot 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
#> 1  1900      <5      9181
#> 2  1900    5-14    16966
#> 3  1900   15-24    14951
#> ...<818 more rows>...
#> 822 2002   45-54   40084
#> 823 2002   55-64   26602
#> 824 2002    >64   35602
```

This version of the chart (Figure 4-21) changes the palette to a range of blues and adds thin (`size = .2`) lines between each area. It also makes the filled areas semi-transparent (`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")
```

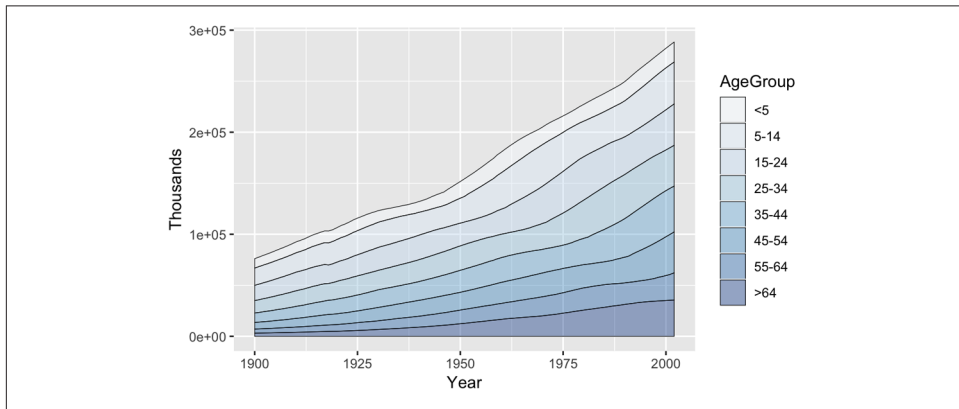


Figure 4-21. Reversed legend order, lines, and a different palette

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 4-22), 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 = dplyr::desc(AgeGroup))) +
  geom_area(colour = NA, alpha = .4) +
  scale_fill_brewer(palette = "Blues") +
  geom_line(position = "stack", size = .2)
```

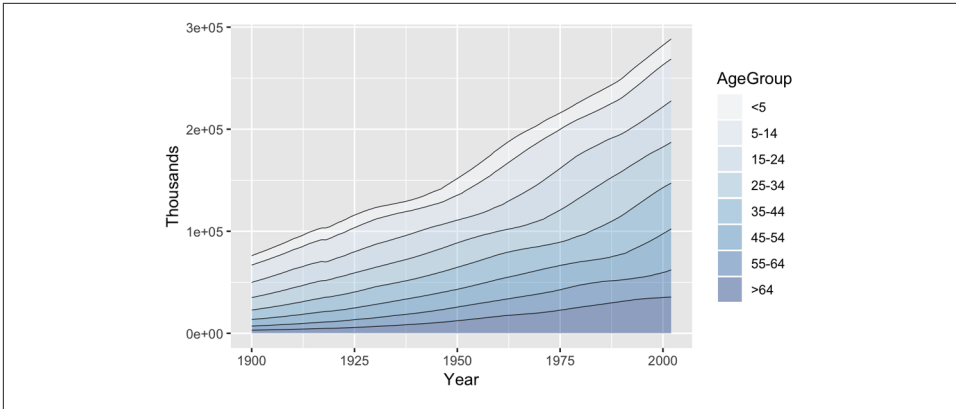


Figure 4-22. No lines on the left and right of the graph

See Also

See [Recipe 15.19](#) for more on converting data from wide to long format.

See [Chapter 12](#) for more on choosing colors.

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

Use `geom_area(position = "fill")`, as in [Figure 4-23](#), top:

```
ggplot(uspope, aes(x = Year, y = Thousands, fill = AgeGroup)) +
  geom_area(position = "fill", colour = "black", size = .2, alpha = .4) +
  scale_fill_brewer(palette = "Blues")
```

Discussion

With `position="fill"`, the `y` values will be scaled to go from 0 to 1. To print the labels as percentages, use `scale_y_continuous(labels = scales::percent)`, as in [Figure 4-23](#), bottom:

```
ggplot(uspope, aes(x = Year, y = Thousands, fill = AgeGroup)) +
  geom_area(position = "fill", colour = "black", size = .2, alpha = .4) +
  scale_fill_brewer(palette = "Blues") +
  scale_y_continuous(labels = scales::percent)
```

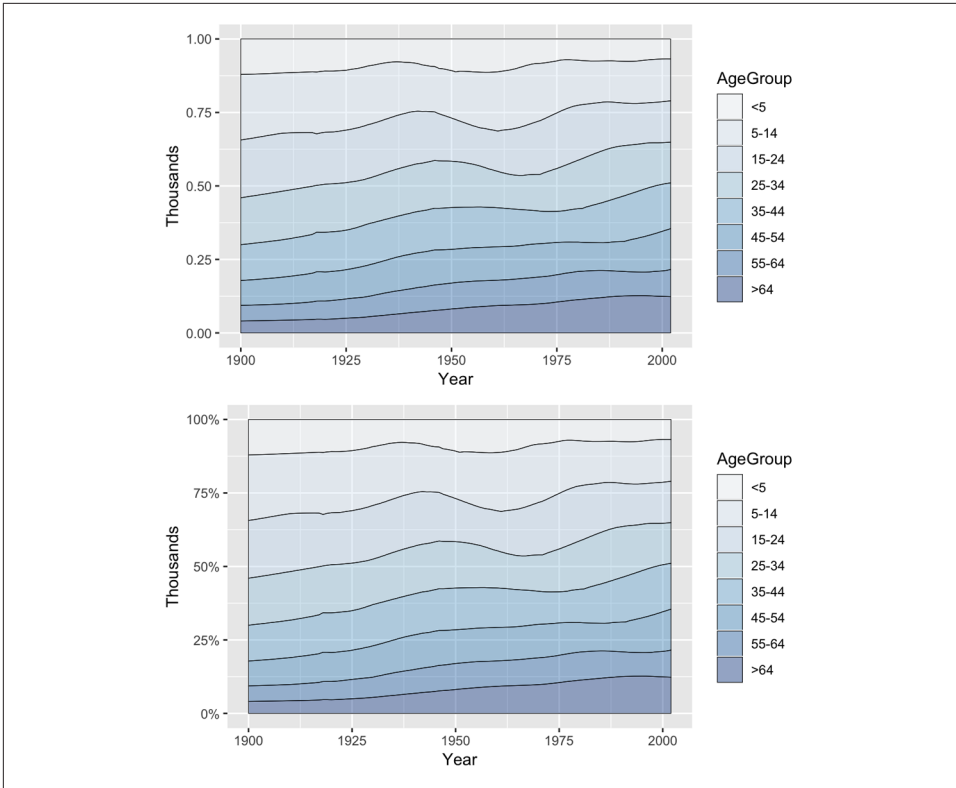


Figure 4-23. A proportional stacked area graph (top); With percent labels on the y axis (bottom)

See Also

Creating a stacked bar graph is done in a similar way. See [Recipe 3.8](#) for information about computing the percentages separately.

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 `ymin` and `ymin` to `Anomaly10y` plus or minus `Unc10y` (Figure 4-24):

```
library(gcookbook) # Load gcookbook for the climate data set
library(dplyr)

# Grab a subset of the climate data
climate_mod <- climate %>%
  filter(Source == "Berkeley") %>%
  select(Year, Anomaly10y, Unc10y)

climate_mod
#>   Year Anomaly10y Unc10y
#> 1  1800    -0.435  0.505
#> 2  1801    -0.453  0.493
#> 3  1802    -0.460  0.486
#> ...<199 more rows>...
#> 203 2002     0.856  0.028
#> 204 2003     0.869  0.028
#> 205 2004     0.884  0.029

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

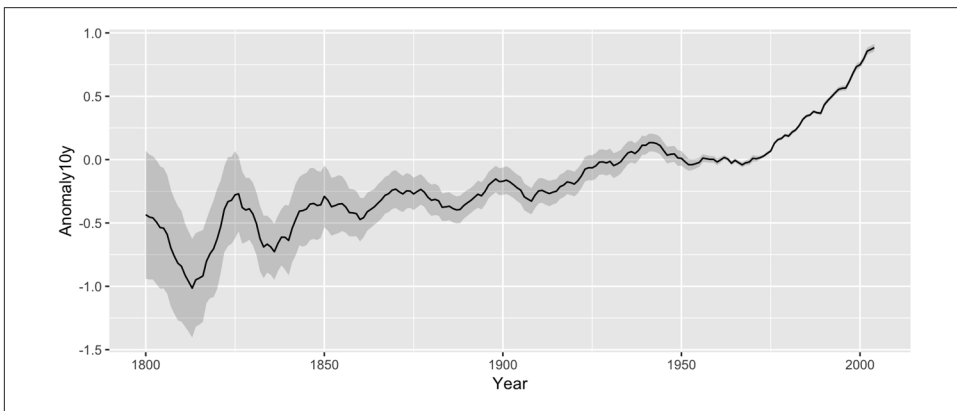


Figure 4-24. A line graph with a shaded confidence region

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()` comes 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 4-25):

```
# With a dotted line for upper and lower bounds
ggplot(climate_mod, 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()
```

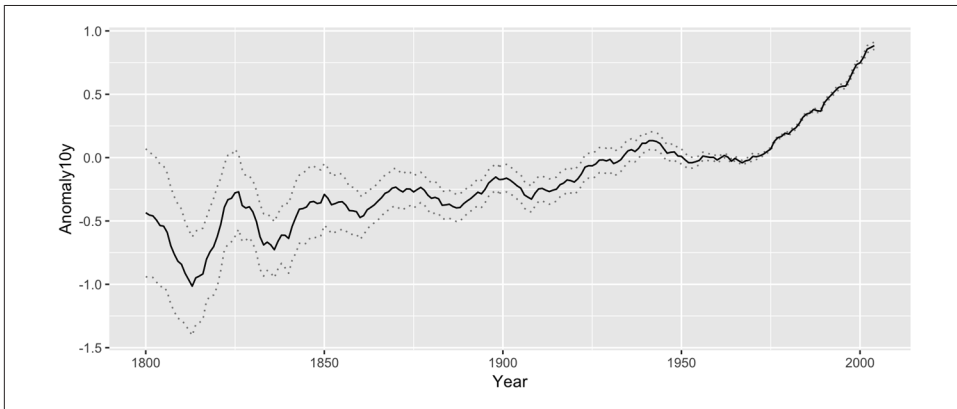


Figure 4-25. A line graph with dotted lines representing a confidence region

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 y_{\min} to y_{\max} .

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. Adding this line is easy to do with R and the `ggplot2` package, and can help to make sense of data when the trends aren't immediately obvious just by looking at the points.

With large data sets, plotting every single observation in the data set can result in overplotting, when points overlap and obscure one another. To deal with the problem of overplotting, 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 using two continuous variables.

Solution

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

We will use the `heightweight` data set. There are a number of columns in this data set, but we'll only use two in this example ([Figure 5-1](#)):

```
library(gcookbook) # Load gcookbook for the heightweight data set
library(dplyr)

# Show the head of the two columns we'll use in the plot
heightweight %>%
```

```

select(ageYear, heightIn)
#>   ageYear heightIn
#> 1    11.92    56.3
#> 2    12.92    62.3
#> 3    12.75    63.3
#> ...<230 more rows>...
#> 235   13.67    61.5
#> 236   13.92    62.0
#> 237   12.58    59.3

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

```

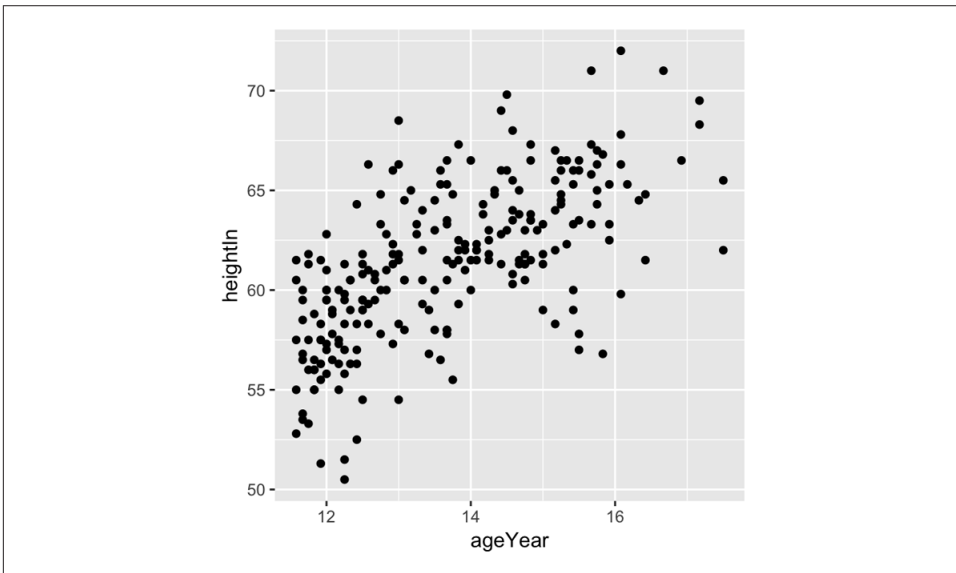


Figure 5-1. A basic scatter plot

Discussion

Instead of points, you can use different shapes for your scatter plot by using the `shape` aesthetic. A common alternative to the default solid circles (`shape #19`) 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 the `size` aesthetic. The default value of `size` is 2 (`size = 2`). The following code will set `size = 1.5` to create smaller points ([Figure 5-2](#), right):

```

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

```

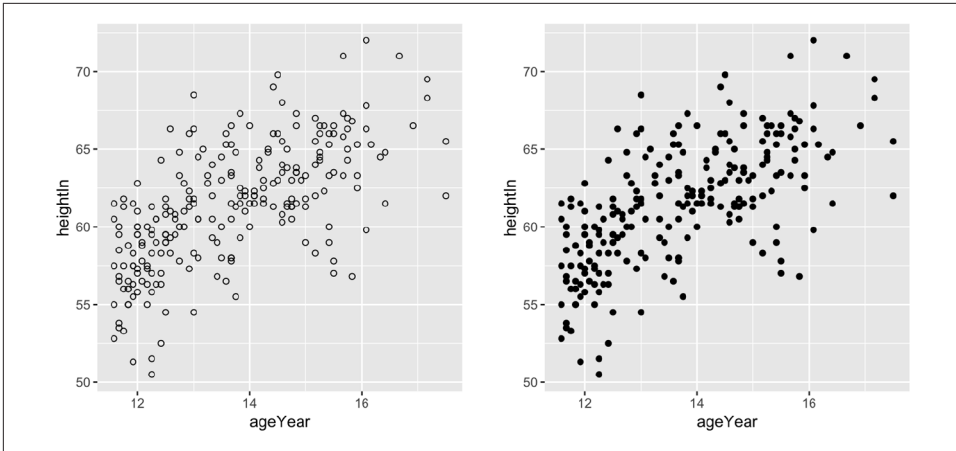



Figure 5-2. Scatter plot with hollow circles (shape #21, left); With smaller points (right)

5.2 Grouping Points Together Using Shapes or Colors

Problem

You want to visually group points by some variable (the grouping variable), using different shapes or colors.

Solution

Map the grouping variable to the aesthetic of shape or colour. We'll use three columns from the `heightweight` data set for this example:

```
library(gcookbook) # Load gcookbook for the heightweight data set

# Show the head of the three columns we'll use
heightweight %>%
  select(sex, ageYear, heightIn)
#>   sex ageYear heightIn
#> 1  f   11.92    56.3
#> 2  f   12.92    62.3
#> 3  f   12.75    63.3
#> ...<230 more rows>...
#> 235 m   13.67    61.5
#> 236 m   13.92    62.0
#> 237 m   12.58    59.3
```

We can use the aesthetics of colour or shape to visually differentiate the data points belonging to different categories of sex. We do this by mapping sex to one of the aesthetics colour or shape (Figure 5-3):

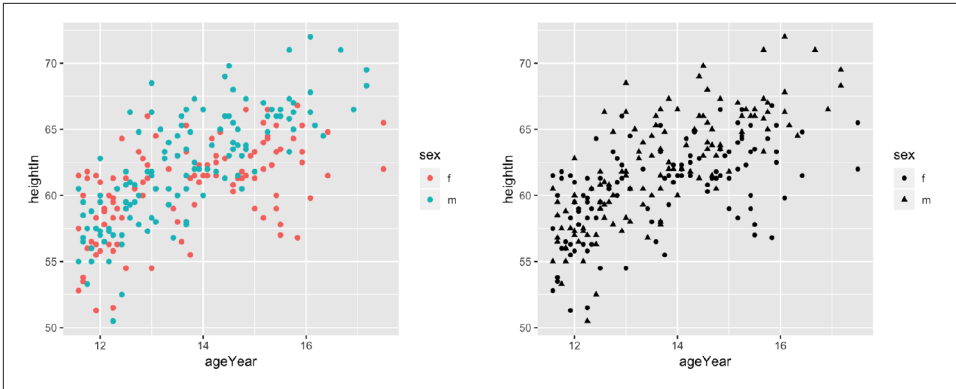


Figure 5-3. Grouping points by a variable mapped to colour (left), or to shape (right)

Discussion

The grouping variable you choose must be categorical—in other words, a factor or character vector. If the grouping variable is a numeric vector, you should convert it to a factor first.

It is possible to map a variable to both shape and colour, or, if you have multiple grouping variables, to map each grouping variable to a different aesthetic. Here, we'll map the variable `sex` to both shape and colour aesthetics (Figure 5-4, left):

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

You may want to use different shapes and colors than are given by the default settings. You can select other shapes for the grouping variables using `scale_shape_manual()`, and select other colors using `scale_colour_brewer()` or `scale_colour_manual()` (Figure 5-4, 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")
```

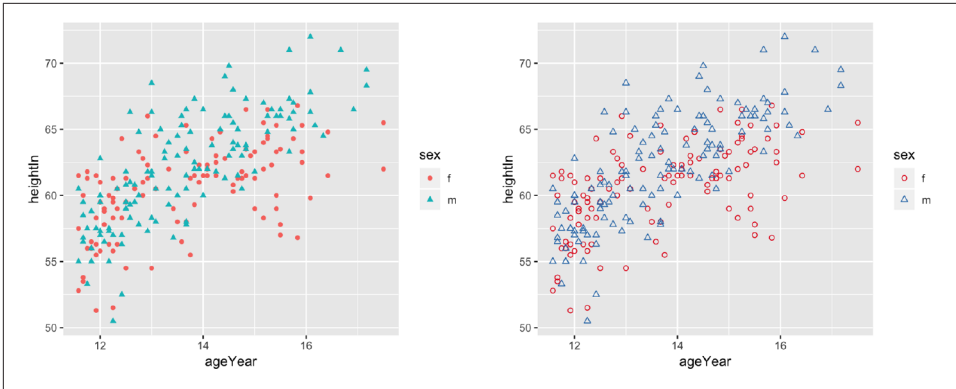


Figure 5-4. Mapping to both shape and colour (left); With manually set shapes and colours (right)

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 change the default scatter plot shapes for the data points.

Solution

You can set the shape of all the data points at once ([Figure 5-5](#), left) by setting a shape in `geom_point()`:

```
library(gcookbook) # Load gcookbook for the heightweight data set

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

If you have mapped a variable to shape, you can use `scale_shape_manual()` to manually change the shapes mapped to the levels of that variable:

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

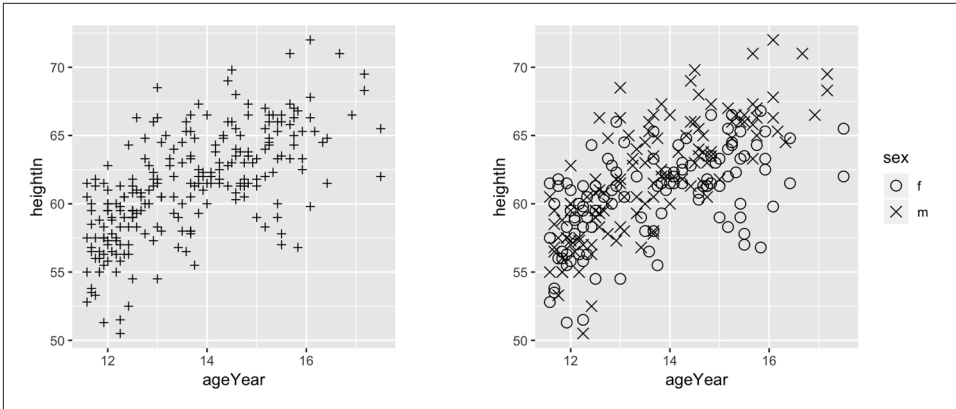


Figure 5-5. Scatter plot with the shape aesthetic set to a custom value (left); With a variable mapped to shape, using a custom shape palette (right)

Discussion

Figure 5-6 shows the shapes that are already built into R. Some of the point shapes (1–14) only have an outline; some (15–20) have solid fill; 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 have solid fill—is controlled by the `colour` aesthetic. For shapes 21–25, the outline is controlled by `colour` and the fill is controlled by `fill`.

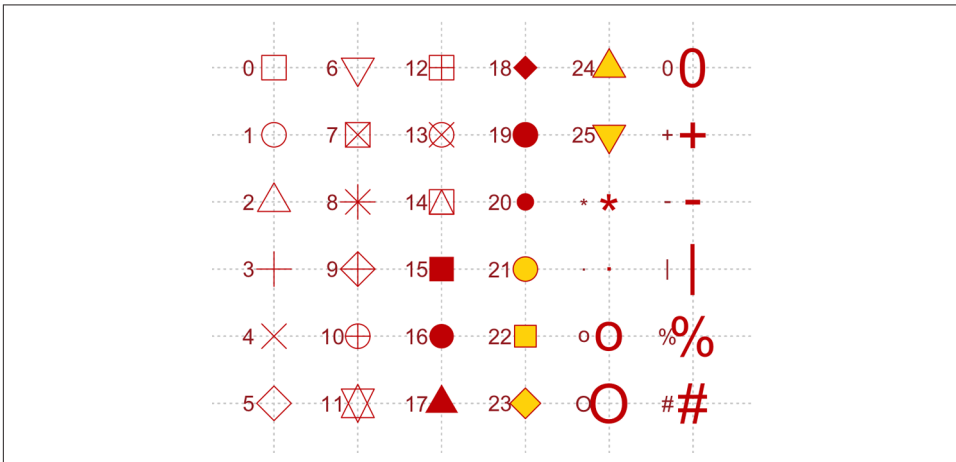


Figure 5-6. Shapes in R graphics

It's possible to have one variable represented by the shape of a point, and another variable represented by the fill (empty or filled) of the point. To do this, you need to first choose point shapes that have both colour and fill, and set these in `scale_shape_manual`. You then need to choose a fill palette that includes NA and another color (the NA will result in a hollow shape) and use these in `scale_fill_manual`.

For example, we'll take the `heightweight` data set and add another column that indicates whether the child weighed 100 pounds or more (Figure 5-7):

```
# Using the heightweight data set, create a new column that indicates if the
# child weighs < 100 or >= 100 pounds. We'll save this modified dataset as 'hw'.
hw <- heightweight %>%
  mutate(weightgroup = ifelse(weightlb < 100, "< 100", ">= 100"))

# Specify shapes with fill and color, and specify fill colors that includes
# an empty (NA) color
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))
  )
)
```

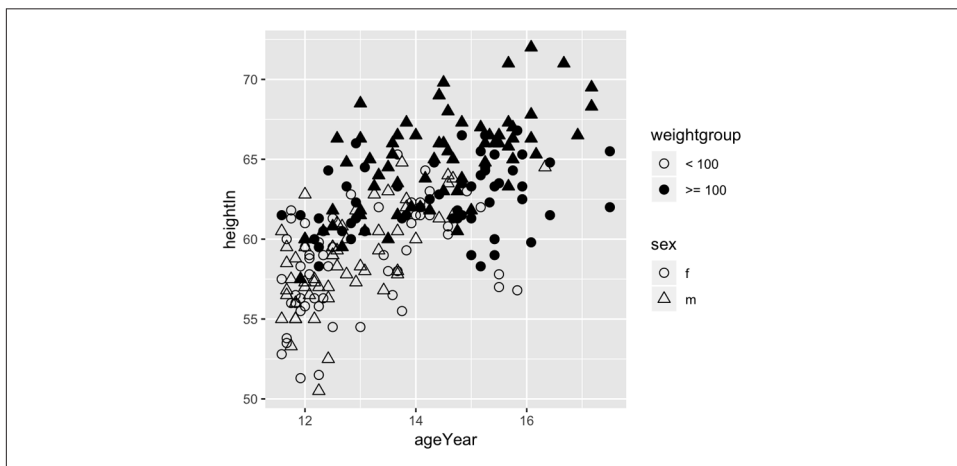


Figure 5-7. A variable mapped to shape and another mapped to fill

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. We will use the `heightweight` data set for this example. There are many columns in this data set, but we'll only use four of them in this example:

```
library(gcookbook) # Load gcookbook for the heightweight data set

# Show the head of the four columns we'll use
heightweight %>%
  select(sex, ageYear, heightIn, weightLb)
#>      sex ageYear heightIn weightLb
#> 1    f   11.92    56.3     85.0
#> 2    f   12.92    62.3    105.0
#> 3    f   12.75    63.3    108.0
#> ...<230 more rows>...
#> 235  m   13.67    61.5    140.0
#> 236  m   13.92    62.0    107.5
#> 237  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`. We can represent a third continuous variable, `weightLb`, by mapping this variable to another aesthetic property, such as colour or size ([Figure 5-8](#)):

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

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

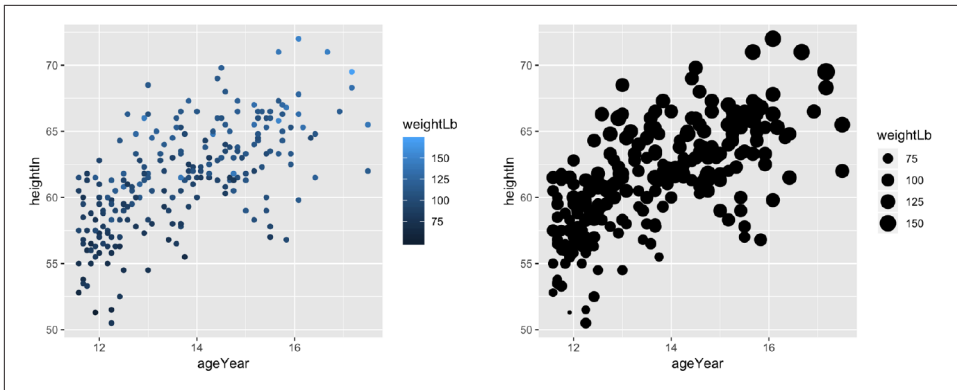


Figure 5-8. A continuous variable mapped to colour (left); Mapped to size (right)

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, these additional variables must be mapped to other aesthetics, like size and colour.

Humans can easily perceive small differences in spatial position, so we can interpret the variables mapped to x and y coordinates with high precision. Humans aren't as good at perceiving small differences in size and colour though, so we will interpret variables mapped to these aesthetic attributes with much lower precision. Therefore, when you map a variable to size or colour, make sure it is a variable where high precision is not very important for correctly interpreting the data.

There is another consideration when mapping a variable to size, which is that the results can be perceptually misleading. While the largest dots in [Figure 5-8](#) are about 36 times the size of the smallest ones, they are only supposed to represent about 3.5 times the weight of the smallest dots.

This relative misrepresentation of size happens because the default values in `ggplot2` for the diameter of points range from 1 to 6 mm, regardless of the actual data values. For example, if the data values range from 0 to 10, the smallest value of 0 will be represented on the plot with a point that is 1 mm wide, while the largest value of 10 will be represented on the plot with a point that is 6 mm wide. Similarly, if the data values range from 100 to 110, the smallest value of 100 will still be represented by a point that is 1 mm wide, and the largest value of 110 will be represented by a point that is 6 mm wide. Thus regardless of the actual data values, the largest point will have a diameter that is 6 times the diameter of the smallest point, and will be 36 times the area.

If it is important for the size of the points to accurately reflect the proportional differences of your data values, you should first decide if you want the diameter of the points to represent the data values, or if you want the area of the points to represent the data values. [Figure 5-9](#) shows the difference between these representations:

```
range(heightweight$weightLb)
#> [1] 50.5 171.5
size_range <- range(heightweight$weightLb) / max(heightweight$weightLb) * 6
size_range
#> [1] 1.766764 6.000000

ggplot(heightweight, aes(x = ageYear, y = heightIn, size = weightLb)) +
  geom_point() +
  scale_size_continuous(range = size_range)

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

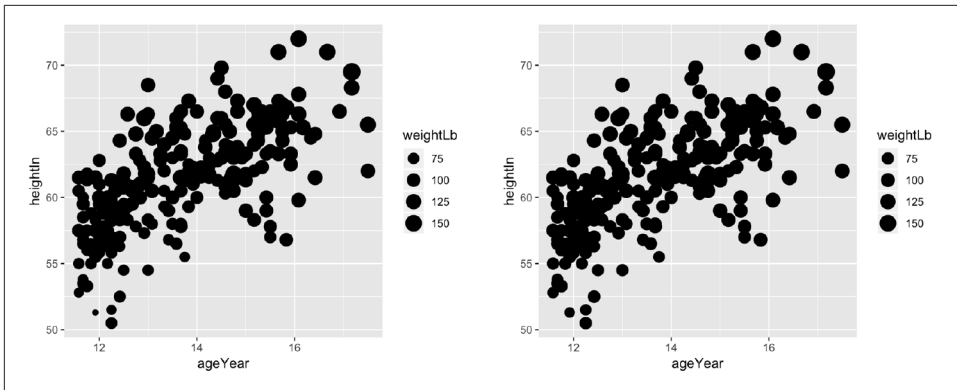


Figure 5-9. Value mapped to diameter of points (left); Value mapped to area of points (right)

See [Recipe 5.12](#) for details on making the area of points proportional to the data values.

When it comes to color, there are actually two aesthetic attributes that can be used: `colour` and `fill`. You will use `colour` for most point shapes. However, shapes 21–25 have an outline with a solid region in the middle where the `colour` 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 the shapes 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 = ageYear, y = heightIn, fill = weightLb)) +
  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
# legend
ggplot(heightweight, aes(x = ageYear, y = heightIn, fill = weightLb)) +
  geom_point(shape = 21, size = 2.5) +
  scale_fill_gradient(
    low = "black", high = "white",
    breaks = seq(70, 170, by = 20),
    guide = guide_legend()
  )
)
```

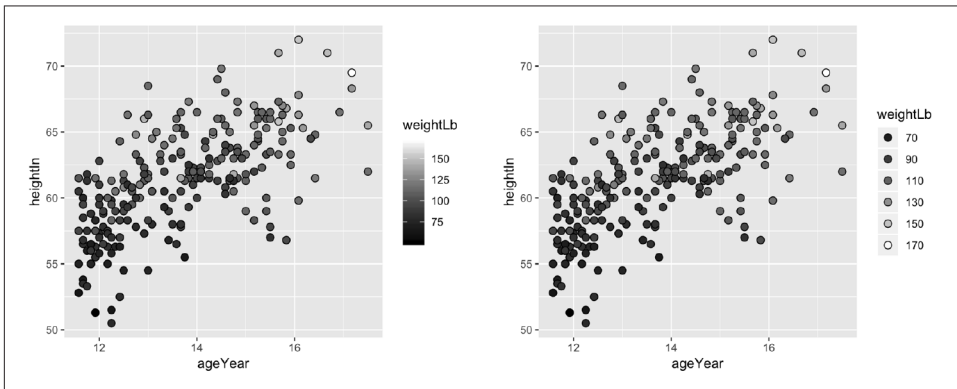


Figure 5-10. Outlined points with a continuous variable mapped to fill (left); With a discrete legend instead of continuous colorbar (right)

Mapping a continuous variable to an aesthetic 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 color. Because there is a fair amount of overplotting (where the points overlap), 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 data values (see [Recipe 5.12](#)), and manually change the color palette.

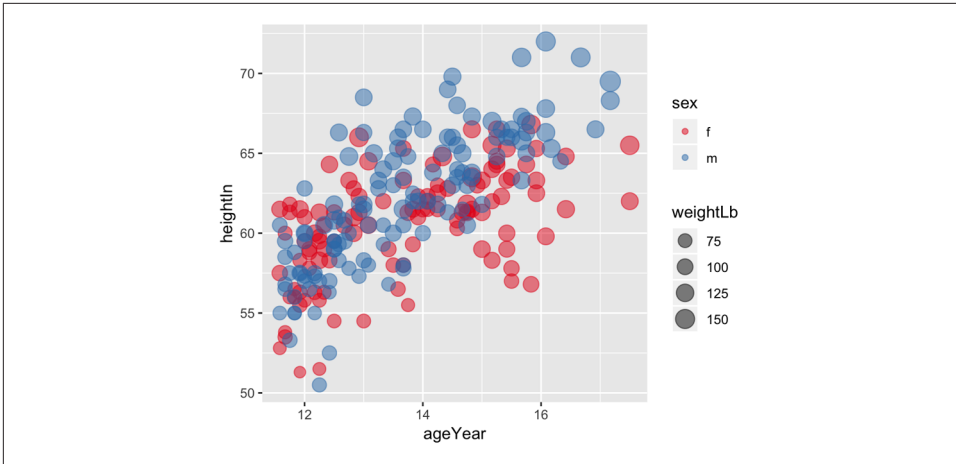


Figure 5-11. Continuous variable mapped to size and categorical variable mapped to colour

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 that overlap and obscure each other when plotted.

Solution

With large data sets, the points in a scatter plot may overlap and 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 the problem 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:

```
# We'll use the diamonds data set and create a base plot called `diamonds_sp`  
diamonds_sp <- ggplot(diamonds, aes(x = carat, y = price))  
  
diamonds_sp +  
  geom_point()
```

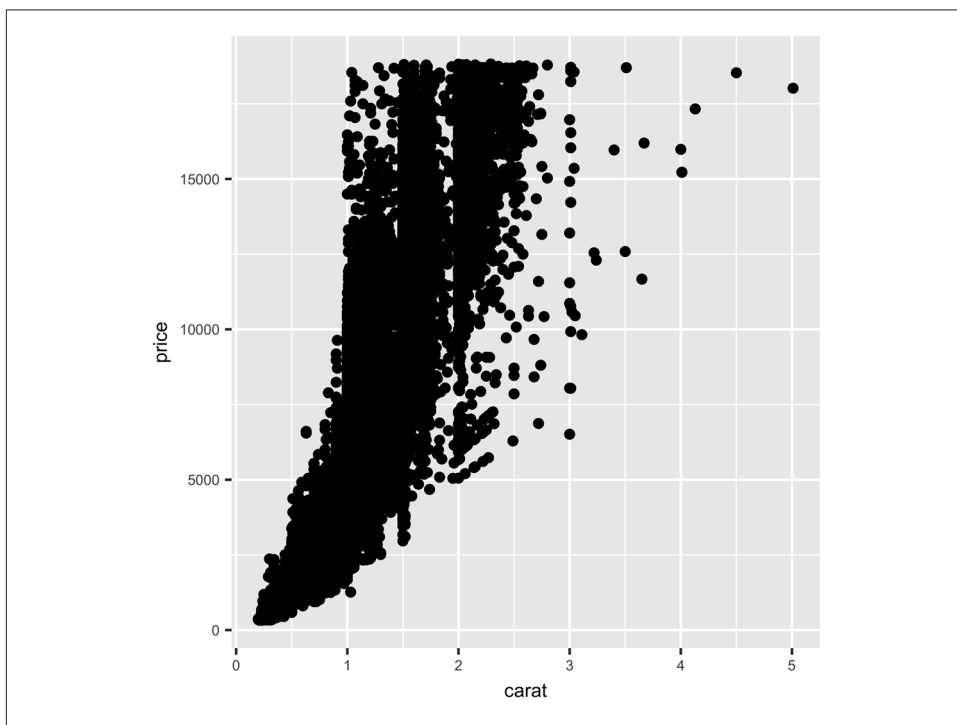


Figure 5-12. Overplotting, with about 54,000 points

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

```
diamonds_sp +  
  geom_point(alpha = .1)  
  
diamonds_sp +  
  geom_point(alpha = .01)
```

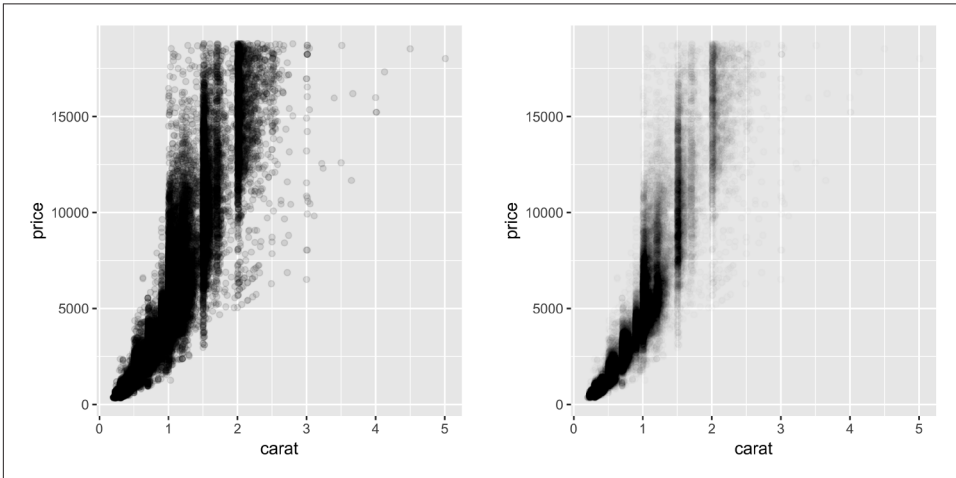


Figure 5-13. Semitransparent points with `alpha=.1` (left); With `alpha=.01` (right)

Now we can see that there appear to be 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 black and the data distribution is still somewhat obscured.



For most plots, 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 a 1.5 MB PDF! 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.

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 by 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):

```
diamonds_sp +
  stat_bin2d()

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

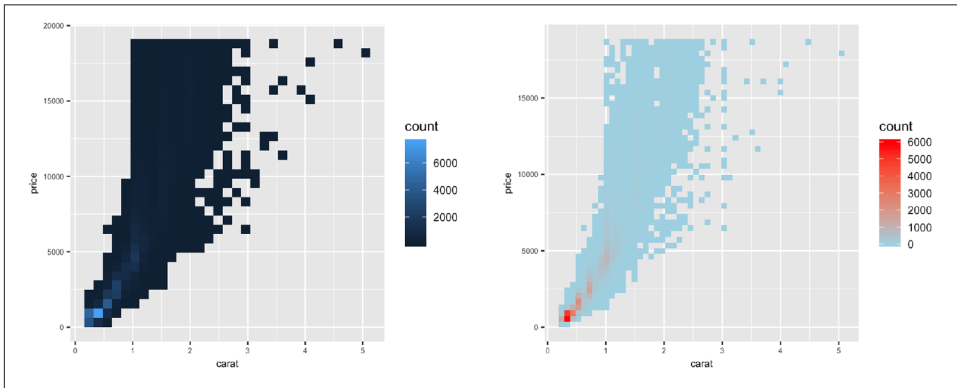


Figure 5-14. Binning data with `stat_bin2d()` (left); With more bins, manually specified colors, and legend breaks (right)

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 `stat_binhex()`, you must first install the `hexbin` package, with the command `install.packages("hexbin")`:

```
library(hexbin) # Load the hexbin library to access stat_binhex()

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

diamonds_sp +
```

```
stat_binhex() +
scale_fill_gradient(low = "lightblue", high = "red", limits = c(0, 5000))
```

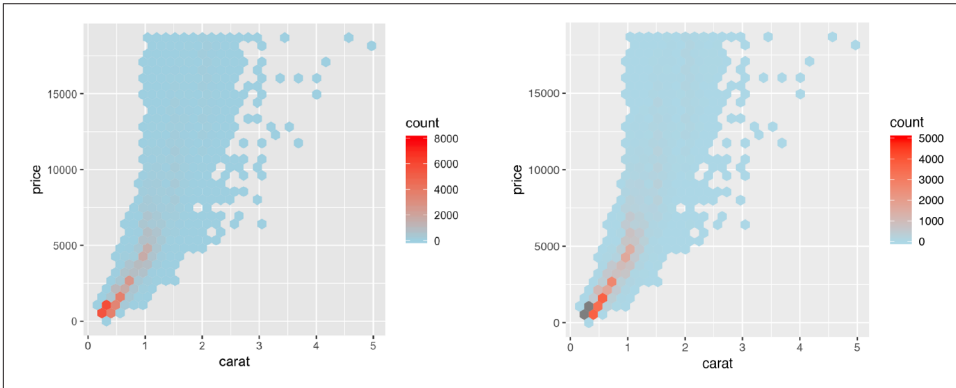


Figure 5-15. Binning data with `stat_binhex()` (left); Cells outside of the range shown in grey (right)

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](#).

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`:

```
# We'll use the ChickWeight data set and create a base plot called `cw_sp`
cw_sp <- ggplot(ChickWeight, aes(x = Time, y = weight))

cw_sp +
  geom_point()

cw_sp +
  geom_point(position = "jitter") # Equivalent to using geom_jitter()

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

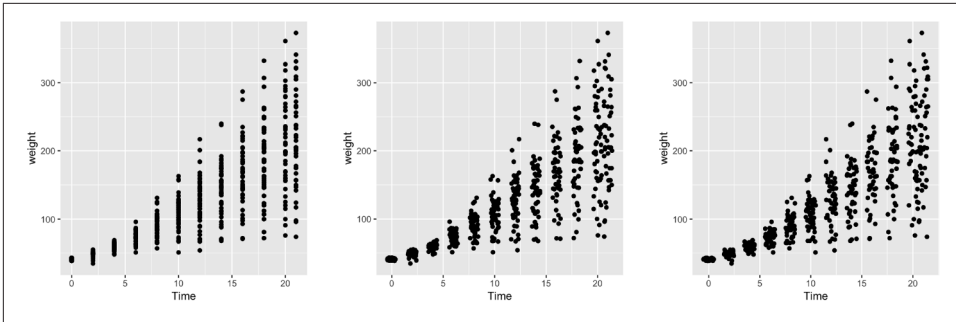


Figure 5-16. Data with a discrete *x* variable (left); Jittered (middle); Jittered horizontally only (right)

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 a box plot 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.

When we look at the `ChickWeight` data we know that we conceptually want to treat `Time` as a discrete variable. However, since `Time` is taken as a numerical variable by default, `ggplot` doesn't know to group the data to form each boxplot box. If you don't tell `ggplot` 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`:

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

cw_sp +
  geom_boxplot() # Without groups
```

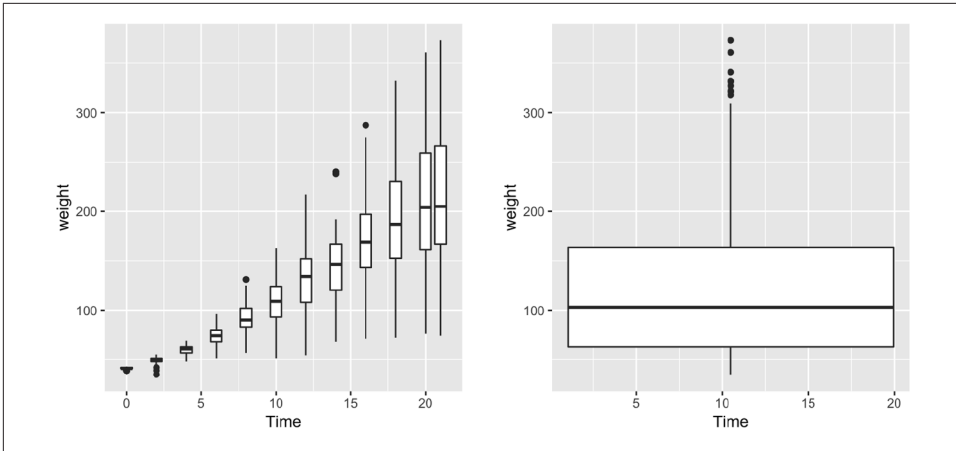


Figure 5-17. Grouping into box plots (left); What happens if you don't specify groups (right)

See Also

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

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 ggplot 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) # Load gcookbook for the heightweight data set

# We'll use the heightweight data set and create a base plot called `hw_sp`
hw_sp <- ggplot(heightweight, aes(x = ageYear, y = heightIn))

hw_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 modifying the value for `level`, or it can be disabled with `se = FALSE` (Figure 5-18):

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

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

The default color of the fit line is blue. This can be changed 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 changing the `colour` of the points (Figure 5-18, bottom right):

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

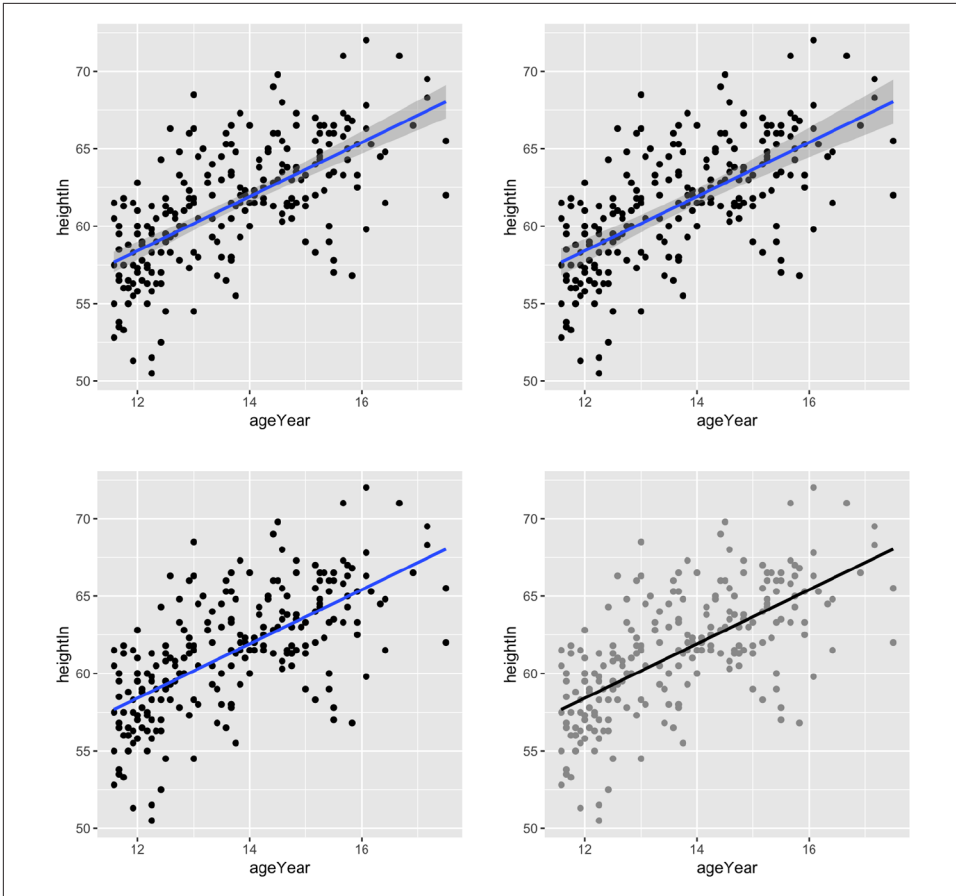


Figure 5-18. An *lm* fit with the default 95% confidence region (top left); A 99% confidence region (top right); No confidence region (bottom left); In black with grey points (bottom right)

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 by default, as shown in Figure 5-19:

```
hw_sp +
  geom_point(colour = "grey60") +
  stat_smooth()

# Equivalent to:
hw_sp +
```

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

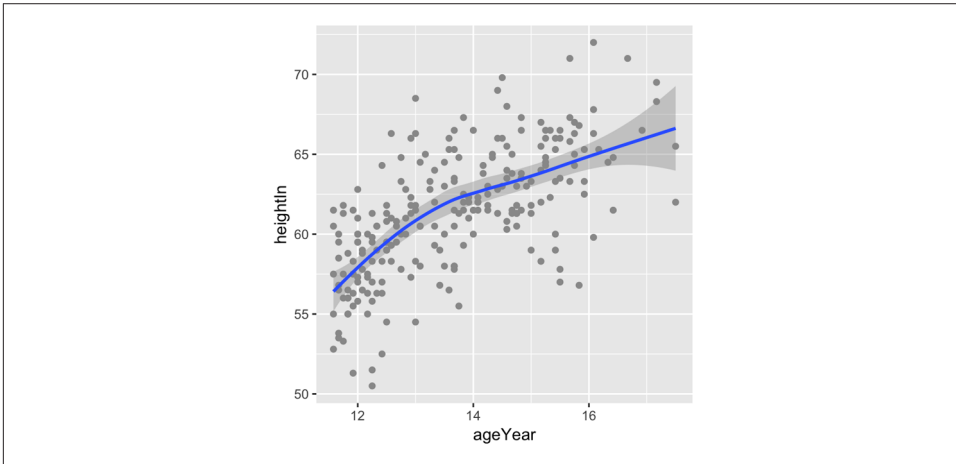


Figure 5-19. A LOESS fit

It may be useful to specify additional parameters for the modeling function, which in this case is `loess()`. If, for example, you wanted to use `loess(degree = 1)`, you would call `stat_smooth(method = loess, method.args = list(degree = 1))`. The same could be done for other modeling functions like `lm()` or `glm()`.

Another common type of model fit is a logistic regression. Logistic regression isn't appropriate for heightweight, but it's perfect for the biopsy data set in the MASS package. In the biopsy data, 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 called `biopsy_mod`, then store the numeric coded class in a column called `classn`:

```
library(MASS) # Load MASS for the biopsy data set

biopsy_mod <- biopsy %>%
  mutate(classn = recode(class, benign = 0, malignant = 1))

biopsy_mod
#>   ID V1 V2 V3 V4 V5 V6 V7 V8 V9   class classn
#> 1 1000025 5 1 1 1 2 1 3 1 1  benign      0
#> 2 1002945 5 4 4 5 7 10 3 2 1  benign      0
#> 3 1015425 3 1 1 1 2 2 3 1 1  benign      0
#> ...<693 more rows>...
#> 697 888820 5 10 10 3 7 3 8 10 2 malignant      1
```

```
#> 698 897471 4 8 6 4 3 4 10 6 1 malignant 1
#> 699 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 `family = binomial`:

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

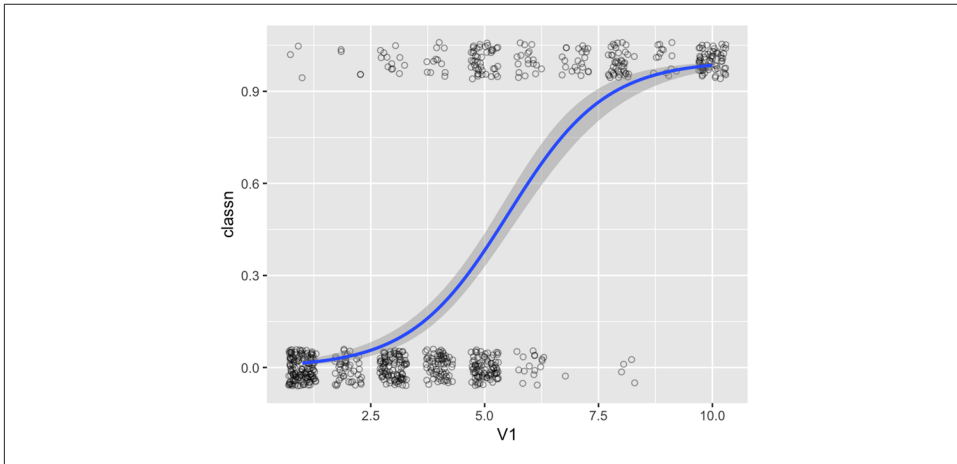


Figure 5-20. A logistic model

If your scatter plot has points grouped by a factor, and that factor is mapped to an aesthetic such as `colour` or `shape`, one fit line will be drawn for each factor level. First, we'll make the base plot object `hw_sp`, then we'll add the LOESS lines to it. We'll also make the points less prominent by making them semitransparent, using `alpha = .4` (Figure 5-21):

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

hw_sp +
  geom_smooth()
```

Notice that the blue line, for males, doesn't run all the way to the right side of the plot. 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 righthand image of [Figure 5-21](#), you must use a model method that allows extrapolation, like `lm()`, and pass `stat_smooth()` the option `fullrange = TRUE`:

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

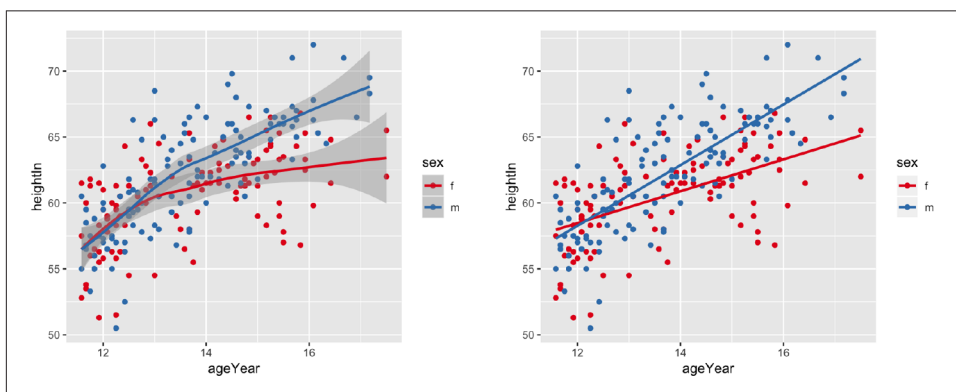


Figure 5-21. LOESS fit lines for each group (left); Extrapolated linear fit lines (right)

In this example with the `heightweight` data set, the default settings for `stat_smooth()` (with `loess` and no extrapolation) may make more sense than the extrapolated linear predictions, because humans 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) # Load gcookbook for the heightweight 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
#> 1   11.5800 56.82624
#> 2   11.6398 57.00047
#> 3   11.6996 57.17294
#> ...<94 more rows>...
#> 98  17.3804 65.47641
#> 99  17.4402 65.47875
#> 100 17.5000 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](#)):

```
# Create a base plot called `hw_sp` (for heightweight scatter plot)
hw_sp <- ggplot(heightweight, aes(x = ageYear, y = heightIn)) +
  geom_point(colour = "grey40")

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

Discussion

Any model object (e.g., `lm`) 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 in the following code. If you simply pass a model to `predictvals()`, the function will do the work of finding the variable names and the 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 function 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")

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

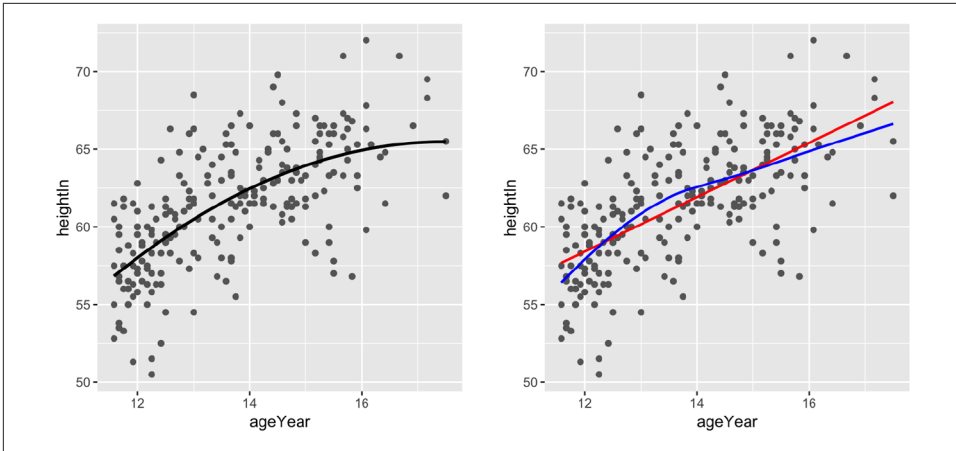


Figure 5-22. A quadratic prediction line from an `lm` object (left); Prediction lines from linear (in red) and LOESS (in blue) models (right)

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 of `glm` 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` package. As we did in [Recipe 5.6](#), we'll use `V1` to predict `class`. Since logistic regressions require numeric values from 0 to 1, we need to convert the factor `class` to 0s and 1s:

```
library(MASS) # Load MASS for the biopsy data set

# Using the biopsy data set, create a new column that stores the factor `class`
# as a numeric variable named `classn`. If `class` == "benign", set `classn`
# to 0. If `class` == "malignant", set `classn` to 1. Save this new dataset as
# `biopsy_mod`.

biopsy_mod <- biopsy %>%
  mutate(classn = recode(class, benign = 0, malignant = 1))

biopsy_mod
#>      ID V1 V2 V3 V4 V5 V6 V7 V8 V9   class classn
#> 1  1000025 5 1 1 1 2 1 3 1 1  benign      0
#> 2  1002945 5 4 4 5 7 10 3 2 1  benign      0
#> 3  1015425 3 1 1 1 2 2 3 1 1  benign      0
#> ...<693 more rows>...
#> 697 888820 5 10 10 3 7 3 8 10 2 malignant    1
#> 698 897471 4 8 6 4 3 4 10 6 1 malignant    1
#> 699 897471 4 8 8 5 4 5 10 4 1 malignant    1
```

Next, we'll perform the logistic regression:


```
fitlogistic <- glm(classn ~ V1, biopsy_mod, 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(biopsy_mod, 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)
```

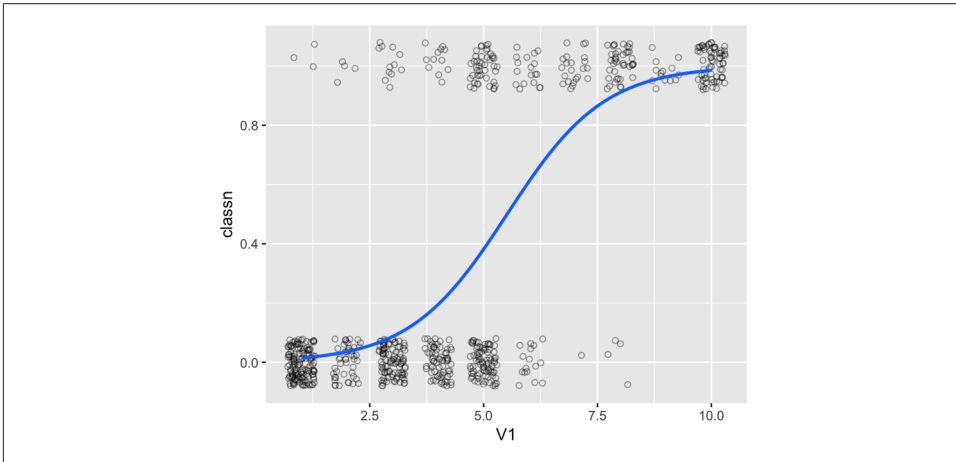


Figure 5-23. A fitted logistic model

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 (Recipe 5.7) along with functions from the `dplyr` package, including `group_by()` and `do()`.

With the `heightweight` data set, we'll make a linear model for each of the levels of `sex`. The model building is done for each subset of the data frame by specifying the model computation we want within the `do()` function.

The following code splits the `heightweight` data frame into two data frames, by the grouping variable `sex`. This creates a data frame subset for males and a data frame subset for females. We then apply `lm(heightIn ~ ageYear, .)` to each subset. The `.` in `lm(heightIn ~ ageYear)` represents the data frame we are piping in from the previous line—in this case, the two data frame subsets we have just created. Finally, this code returns a data frame that contains a column, `model`, which is a list of the model outputs corresponding to each level of the grouping variable `sex`, female and male:

```
library(gcookbook) # Load gcookbook for the heightweight data set
library(dplyr)

# Create an lm model object for each value of sex; this returns a data frame
models <- heightweight %>%
  group_by(sex) %>%
  do(model = lm(heightIn ~ ageYear, .)) %>%
  ungroup()

# Print the data frame
models
#> # A tibble: 2 x 2
#>   sex    model
#>   * <fct> <list>
#> 1 f      <S3: lm>
#> 2 m      <S3: lm>

# Print out the model column of the data frame
models$model
#> [[1]]
#>
#> Call:
#> lm(formula = heightIn ~ ageYear, data = .)
#>
#> Coefficients:
#> (Intercept)      ageYear
#>    43.963         1.209
#>
#>
#> [[2]]
#>
#> Call:
#> lm(formula = heightIn ~ ageYear, data = .)
#>
#> Coefficients:
#> (Intercept)      ageYear
#>    30.658         2.301
```

Now that we have the list of model objects, we can run the `predictvals()` as defined in [Recipe 5.7](#) to get the predicted values from each model:

```
predvals <- models %>%
  group_by(sex) %>%
  do(predictvals(.$model[[1]], xvar = "ageYear", yvar = "heightIn"))
```

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)

# Using facets instead of colors for the groups
ggplot(heightweight, aes(x = ageYear, y = heightIn)) +
  geom_point() +
  geom_line(data = predvals) +
  facet_grid(. ~ sex)
```

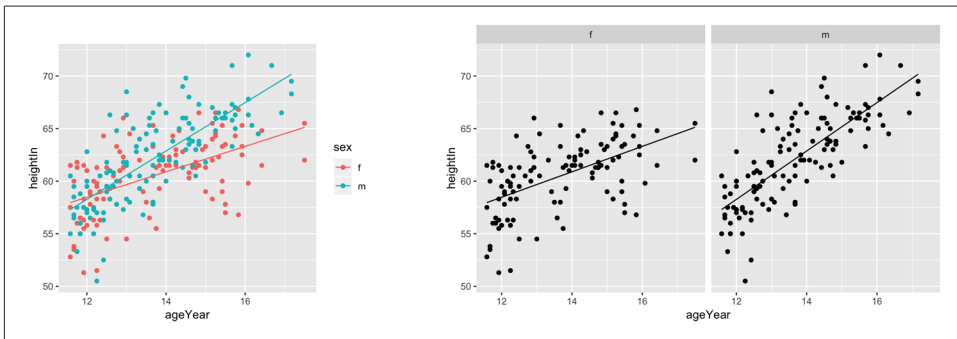


Figure 5-24. Predictions from two separate `lm` objects, one for each subset of data (left); With facets (right)

Discussion

The `group_by()` and `do()` 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 <- models %>%
  group_by(sex) %>%
  do(predictvals(
    .$model[[1]],
    xvar = "ageYear",
    yvar = "heightIn",
    xrange = range(ageYear)
```

```
xrange = range(heightweight$ageYear))
)
```

Then we can plot it, the same as we did before (Figure 5-25):

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

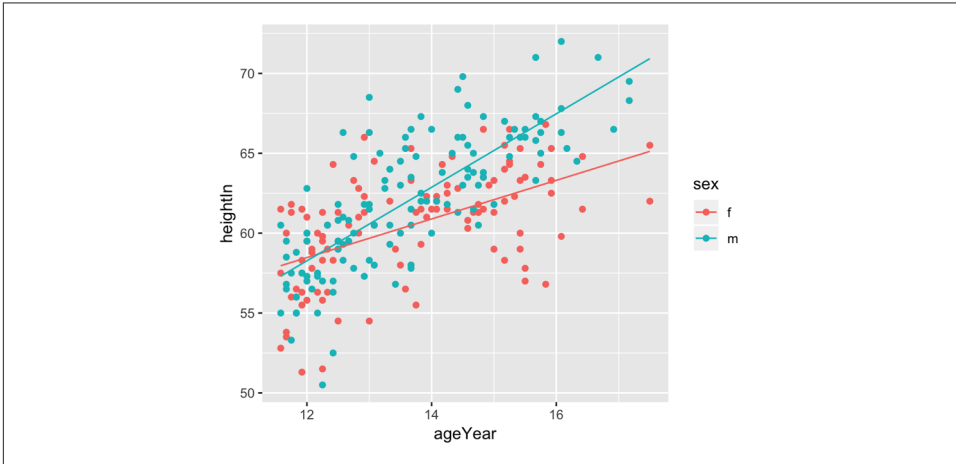


Figure 5-25. Predictions for each group extend to the full x range of all groups together

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.

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) # Load gcookbook for the heightweight 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")

# Save a base plot
hw_sp <- ggplot(heightweight, aes(x = ageYear, y = heightIn)) +
  geom_point() +
  geom_line(data = pred)

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

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

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

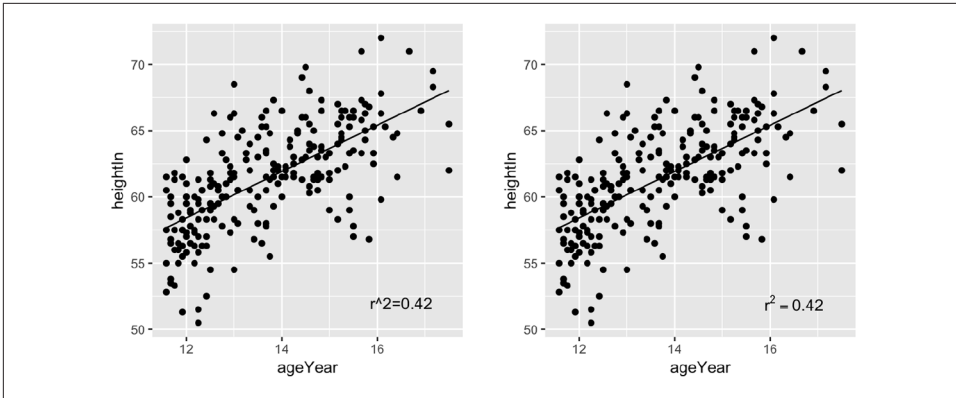


Figure 5-26. Plain text (left); Math expression (right)

Discussion

Text geoms in ggplot do not take expression objects directly; instead, they take character strings that can be turned into expressions with R's `parse()` function.

If you use a mathematical expression, the syntax must be correct for the expression to be a valid R expression object. You can test the validity by wrapping the object in the `expression()` function 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) # 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, yields a valid expression:

```
# Use sprintf() to construct our string.
# The %.3g and %.2g are replaced with numbers with 3 significant digits and 2
# significant digits, respectively. The numbers are supplied after the string.

eqn <- sprintf(
  "italic(y) == %.3g + %.3g * italic(x) * ', ' ~~ italic(r)^2 ~ '=' ~ %.2g",
  coef(model)[1],
  coef(model)[2],
  summary(model)$r.squared
)

eqn
#> [1] "italic(y) == 37.4 + 1.75 * italic(x) * ', ' ~~ italic(r)^2 ~ '=' ~ 0.42"

# Test validity by using parse()
```

```

parse(text = eqn)
#> 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):

```

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

```

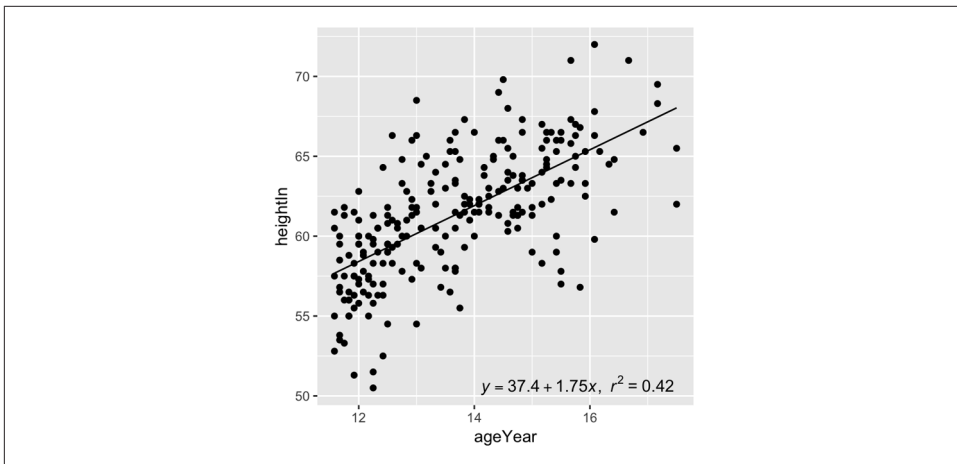


Figure 5-27. Scatter plot with automatically generated expression

See Also

The math expression syntax in R can be a bit tricky. See [Recipe 7.2](#) for more information.

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. This data set has two columns with data about the Old Faithful geyser: `eruptions`, which is the length of each eruption, and `waiting`, which is the length of time until the next eruption:

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

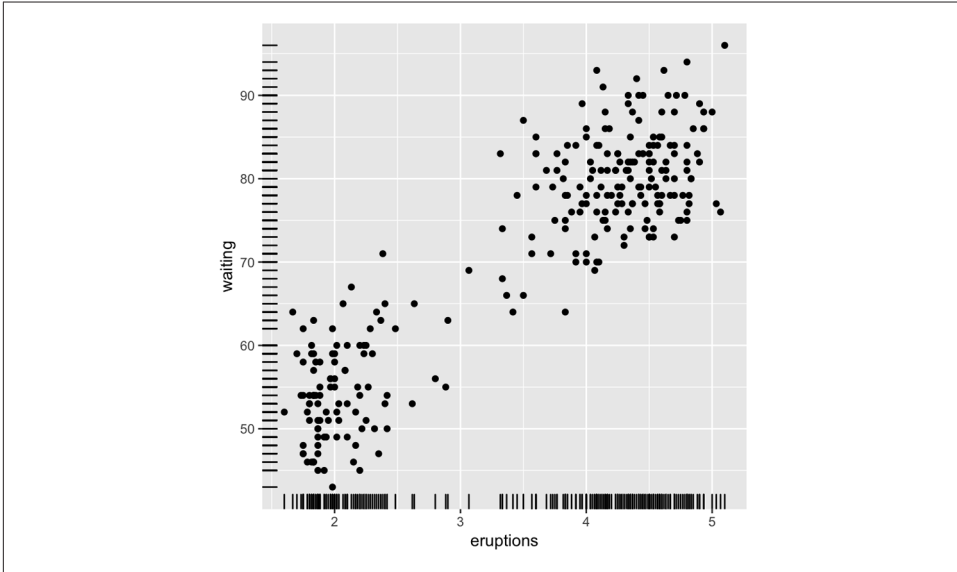


Figure 5-28. Marginal rug added to a scatter plot

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.

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 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 = 0.2)
```

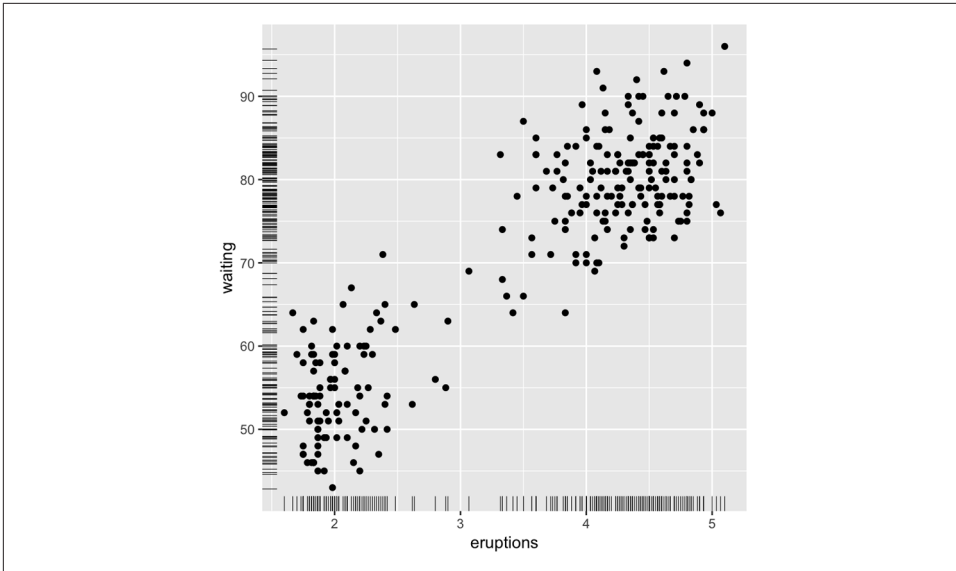



Figure 5-29. Marginal rug with thinner, jittered lines

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.

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 filter the data to only look at data from 2009 for a subset of countries that spent more than \$2,000 USD per capita:

```
library(gcookbook) # Load gcookbook for the countries data set
library(dplyr)

# Filter the data to only look at 2009 data for countries that spent > 2000 USD
# per capita
countries_sub <- countries %>%
  filter(Year == 2009 & healthexp > 2000)
```

We'll save the basic scatter plot object in `countries_sp` (for countries scatter plot) and add then add our annotations 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 the labels positioned just right:

```
countries_sp <- ggplot(countries_sub, aes(x = healthexp, y = infmortality)) +
  geom_point()

countries_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:

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

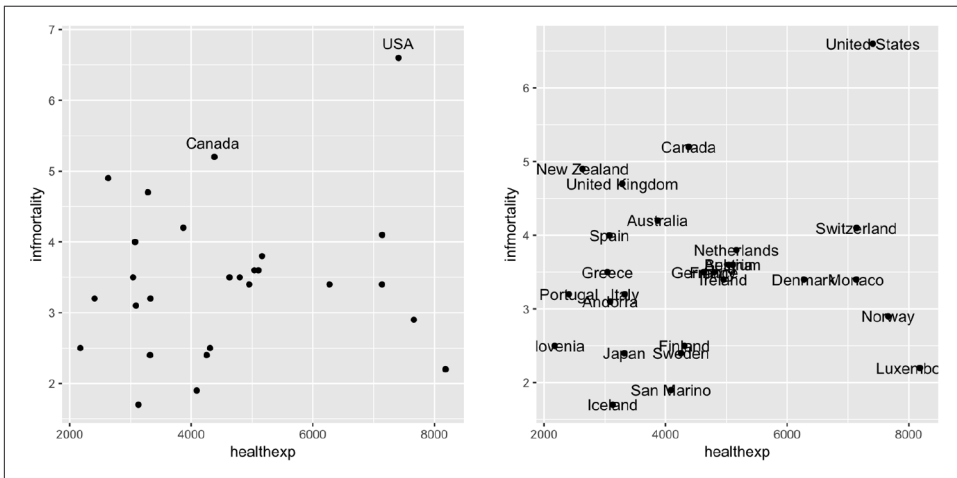


Figure 5-30. A scatter plot with manually labeled points (left); With automatically labeled points and a smaller font (right)

As you can see in the center of Figure 5-30 (right), you may find yourself with a plot where labels are overlapping. To automatically adjust point labels so that they don't overlap, we can use `geom_text_repel` (Figure 5-31, left) or `geom_label_repel` (which adds a box around the label, Figure 5-31, right) from the `ggrepel` package, which functions similarly to `geom_text`:

```
# Make sure to have installed ggrepel with install.packages("ggrepel")
library(ggrepel)
countries_sp +
  geom_text_repel(aes(label = Name), size = 3)

countries_sp +
  geom_label_repel(aes(label = Name), size = 3)
```

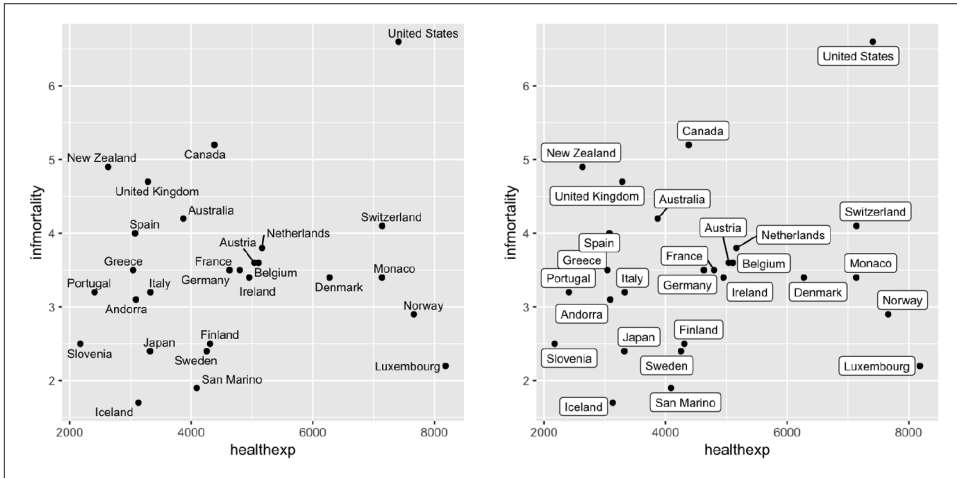


Figure 5-31. A scatter plot labeled with `geom_text_repel` (left); Labeled with `geom_label_repel` (right)

Discussion

Using `geom_text_repel` or `geom_label_repel` is the easiest way to have nicely placed labels on a plot. It makes automatic (and random) decisions about label placement, so if you need exact control over where each label is placed, you should use `annotate()` or `geom_text()`.

The automatic method for placing annotations using `geom_text()` 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 5-32, left), and setting `vjust = 1` will make the top of the text level with the point. This usually isn't enough, though—you can 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 5-32, right):

```
countries_sp +
  geom_text(aes(label = Name), size = 3, vjust = 0)

# Add a little extra to y
countries_sp +
  geom_text(aes(y = infmortality + .1, label = Name), size = 3)
```

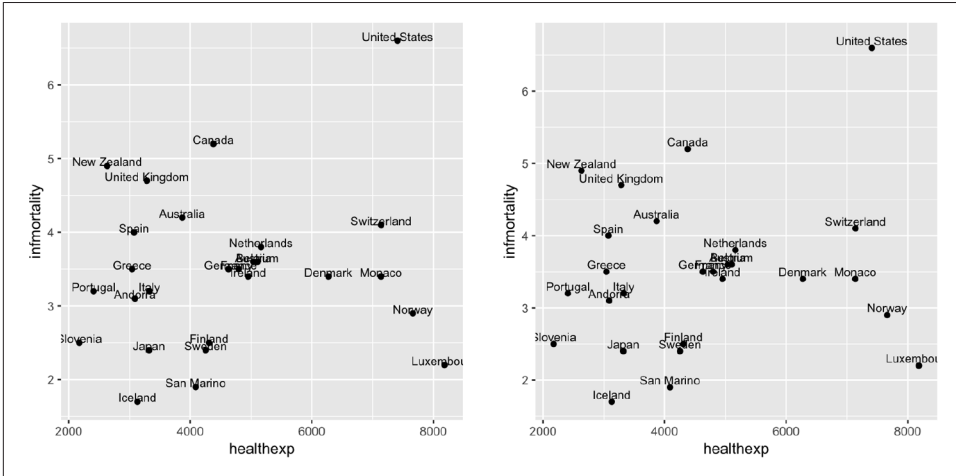


Figure 5-32. A scatter plot with *vjust*=0 (left); With a little extra added to *y* (right)

It often makes sense to right- or left-justify the labels relative to the points. To left-justify, set *hjust* = 0 (Figure 5-33, 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 5-33, right):

```
countries_sp +
  geom_text(
    aes(label = Name),
    size = 3,
    hjust = 0
  )

countries_sp +
  geom_text(
    aes(x = healthexp + 100, label = Name),
    size = 3,
    hjust = 0
  )
```

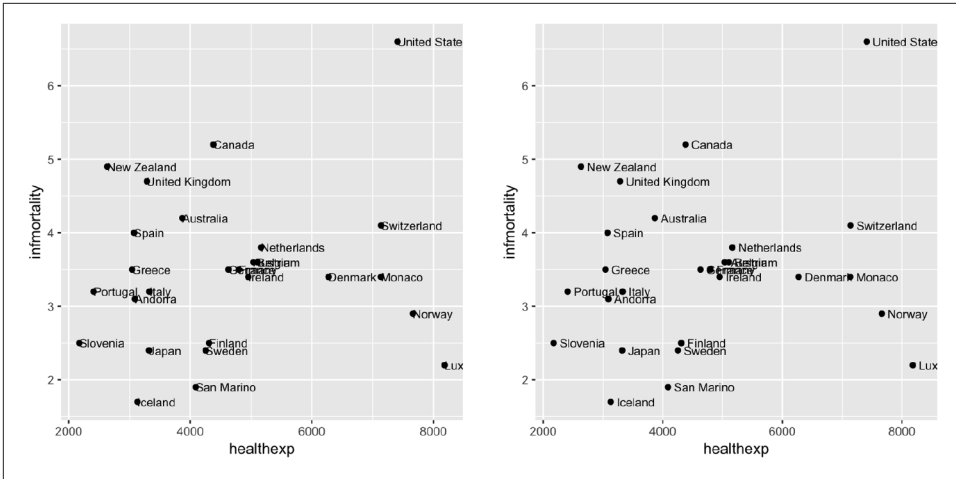


Figure 5-33. A scatter plot with $hjust=0$ (left); With a little extra added to x (right)



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.

Besides right- or left-justifying all of your labels, you can also adjust the position of all of the labels at once by using `position = position_nudge()`. This allows you to specify the amount of vertical or horizontal distance you want to move the labels. As you can see from the following figure (Figure 5-34), this strategy works best when there are fewer labels, or fewer points that can cause overlap with labels. Note that the units you specify with `x = ...` and `y = ...` correspond to the units of the x - and y -axes:

```
countries_sp +
  geom_text(
    aes(x = healthexp + 100, label = Name),
    size = 3,
    hjust = 0
  )

countries_sp +
  geom_text(
    aes(x = healthexp + 100, label = Name),
    size = 3,
    hjust = 0,
    position = position_nudge(x = 100, y = -0.2)
  )
```

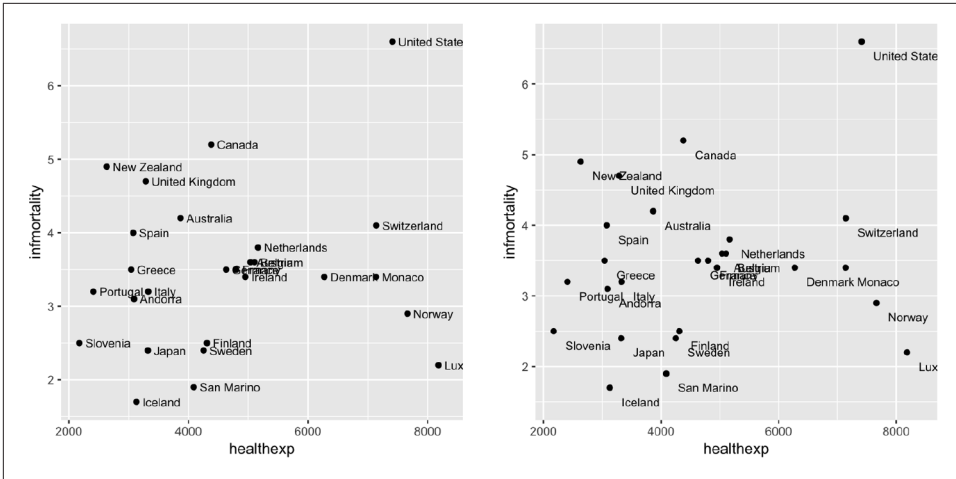


Figure 5-34. Original scatter plot (left); Scatter plot with labels nudged down and to the right (right)

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 copy the `Name` column into `plotname`, converting from a factor to a character vector, for reasons we'll see soon:

```
cdat <- countries %>%
  filter(Year == 2009, healthexp > 2000) %>%
  mutate(plotname = as.character(Name))
```

Now that `plotname` is a character vector, we can use an `ifelse()` function and the `%in%` operator to identify if each row of `plotname` matches the list of names we want to show on our plot, which we have specified manually in the following code. The `%in%` operator returns a logical vector that allows us to specify within the `ifelse()` function that we want to replace all values of `plotname` that do not match one of our specified names with a blank string:

```
countrylist <- c("Canada", "Ireland", "United Kingdom", "United States",
  "New Zealand", "Iceland", "Japan", "Luxembourg", "Netherlands", "Switzerland")

cdat <- cdat %>%
  mutate(plotname = ifelse(plotname %in% countrylist, plotname, ""))

# Take a look at the resulting `plotname` variable, as compared to the original
# `Name` variable
cdat %>%
  select(Name, plotname)
#>      Name      plotname
#> 1 Andorra
```

```
#> 2      Australia
#> 3      Austria
#> ...<21 more rows>...
#> 25     Switzerland Switzerland
#> 26 United Kingdom United Kingdom
#> 27 United States United States
```

Now we can make the plot (Figure 5-35). 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 = plotname), size = 4, hjust = 0) +
  xlim(2000, 10000)
```

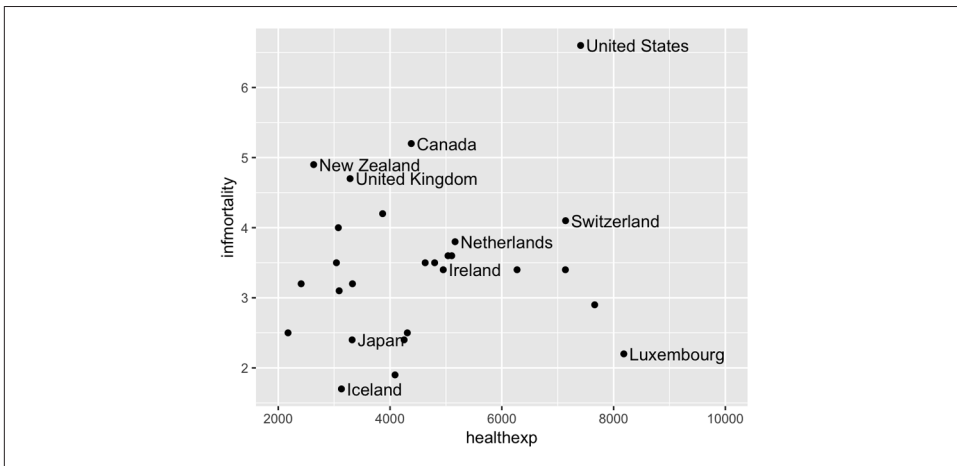


Figure 5-35. Scatter plot with selected labels and expanded *x* range

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!

Finally, 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 filter the data set countries to only include data from the year 2009, for certain countries we have specified in `countrylist`:

```
library(gcookbook) # Load gcookbook for the countries data set

countrylist <- c("Canada", "Ireland", "United Kingdom", "United States",
  "New Zealand", "Iceland", "Japan", "Luxembourg", "Netherlands", "Switzerland")

cdat <- countries %>%
  filter(Year == 2009, Name %in% countrylist)

cdat
#>           Name Code Year      GDP laborrate healthexp infmortality
#> 1      Canada  CAN 2009 39599.04      67.8  4379.761          5.2
#> 2      Iceland ISL 2009 37972.24      77.5  3130.391          1.7
#> 3      Ireland IRL 2009 49737.93      63.6  4951.845          3.4
#> ...<4 more rows>...
#> 8 Switzerland CHE 2009 63524.65      66.9  7140.729          4.1
#> 9 United Kingdom GBR 2009 35163.41      62.2  3285.050          4.7
#> 10 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 5-36, 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 the value of GDP to the *area* of the dots, which we can do using `scale_size_area()` (Figure 5-36, right):

```
# Create a base plot using the cdat data frame.
cdat_sp <- 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)
cdat_sp

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

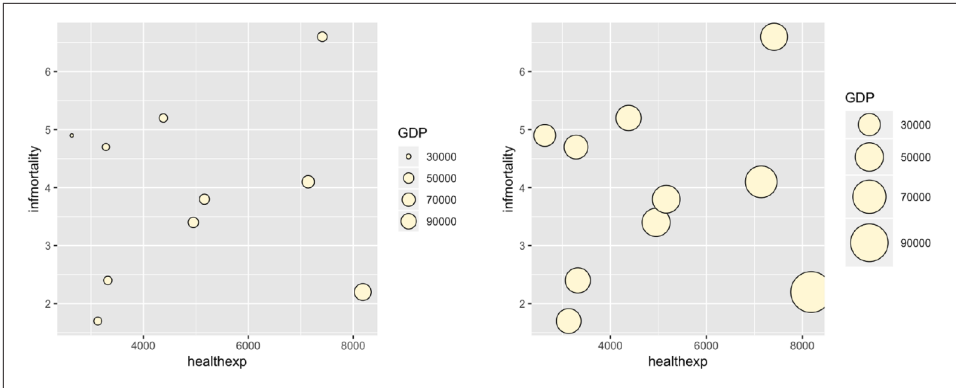



Figure 5-36. Balloon plot with value mapped to radius (left); With value mapped to area (right)

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 balloon plots to represent values on a grid, where the x- and y-axes are categorical, as in [Figure 5-37](#):

```
# Create a data frame that adds up counts for males and females
hec <- HairEyeColor %>%
  # Convert to long format
  as_tibble() %>%
  group_by(Hair, Eye) %>%
  summarize(count = sum(n))

# Create the base balloon plot
hec_sp <- 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(as.factor(Hair)) - sqrt(count)/34, label = count),
    vjust = 1.3,
    colour = "grey60",
    size = 4
  )
)

hec_sp

# Add red guide points
hec_sp +
  geom_point(aes(y = as.numeric(as.factor(Hair)) - sqrt(count)/34),
    colour = "red", size = 1)
```

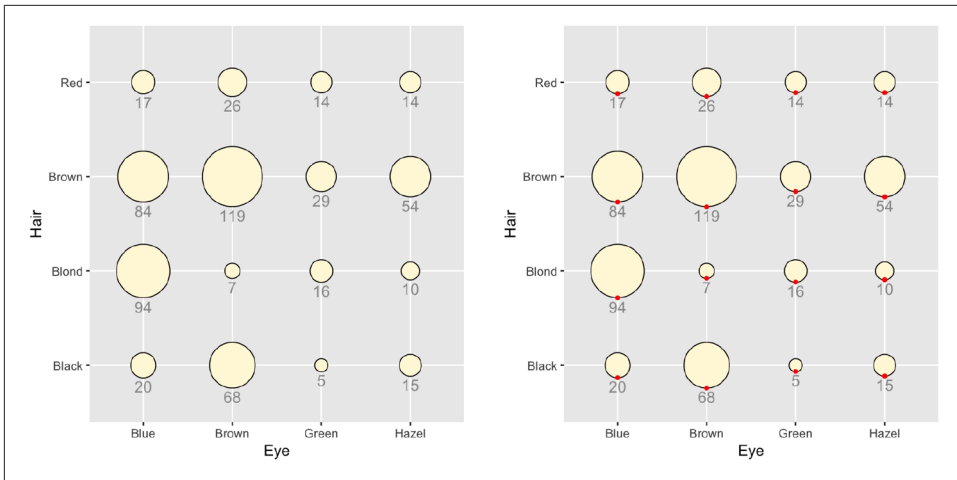


Figure 5-37. Balloon plot with categorical axes and text labels (left); With guide points to help position text (right)

In this example we've used a few tricks to add the text labels under the circles. First, we used `vjust = 1.3` to justify the top of the text slightly below the `y` coordinate. Next, we wanted to set the `y` coordinate so that it is at the bottom of each circle. This requires a little wrangling and arithmetic: we need to first convert the levels of `Hair` and `Eye` into numeric values, which involves converting these variables from being a character vector to being a factor variable, and then converting them again into a numeric variable. We then 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 is divided by (34 in this case) is found by trial and error; it depends on the particular data values, radius, text size, and output image size.

To help find the correct `y` offset, we can add guide points in red and adjust the value until they line up with the bottom of each circle. Once we have the correct value, we can place the text and remove the points.

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.

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. We'll pull out the data for the year 2009, and keep only the columns that are relevant:

```
library(gcookbook) # Load gcookbook for the countries data set

c2009 <- countries %>%
  filter(Year == 2009) %>%
  select(Name, GDP, laborrate, healthexp, infmortality)

c2009
#>      Name      GDP laborrate healthexp infmortality
#> 1  Afghanistan      NA      59.8  50.88597      103.2
#> 2    Albania 3772.6047      59.5 264.60406      17.2
#> 3    Algeria 4022.1989      58.5 267.94653      32.0
#> ...<210 more rows>...
#> 214 Yemen, Rep. 1130.1833      46.8  64.00204      58.7
#> 215    Zambia 1006.3882      69.2  47.05637      71.5
#> 216    Zimbabwe 467.8534      66.8      NA      52.2
```

To make the scatter plot matrix ([Figure 5-38](#)), we'll use all of the variables except for `Name`, since making a scatter plot matrix using the names of the countries wouldn't make sense and would produce strange-looking results:

```
c2009_num <- select(c2009, -Name)
pairs(c2009_num)
```

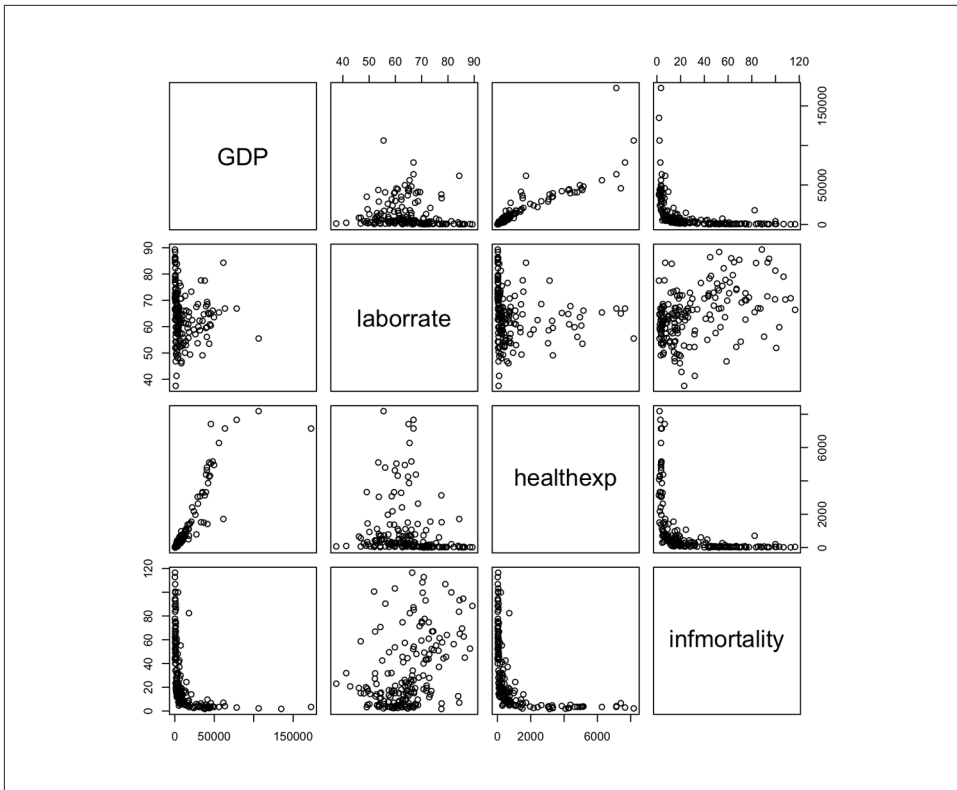


Figure 5-38. A scatter plot matrix

Discussion

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 5-39](#). (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_num,  
  upper.panel = panel.cor,  
  diag.panel = panel.hist,  
  lower.panel = panel.smooth  
)
```

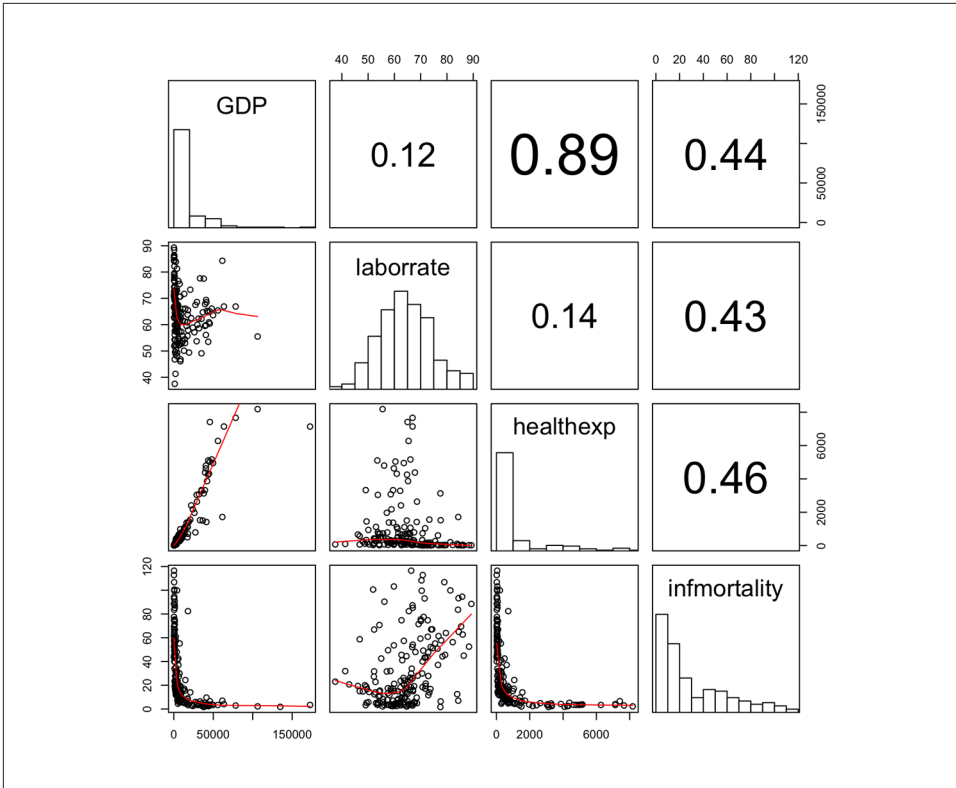


Figure 5-39. Scatter plot with correlations in the upper triangle, smoothing lines in the lower triangle, and histograms on the diagonal

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 5-40). This is done by setting `pch = "."`:

```

pairs(
  c2009_num,
  upper.panel = panel.cor,
  diag.panel = panel.hist,
  lower.panel = panel.smooth,
  pch = ".")

```

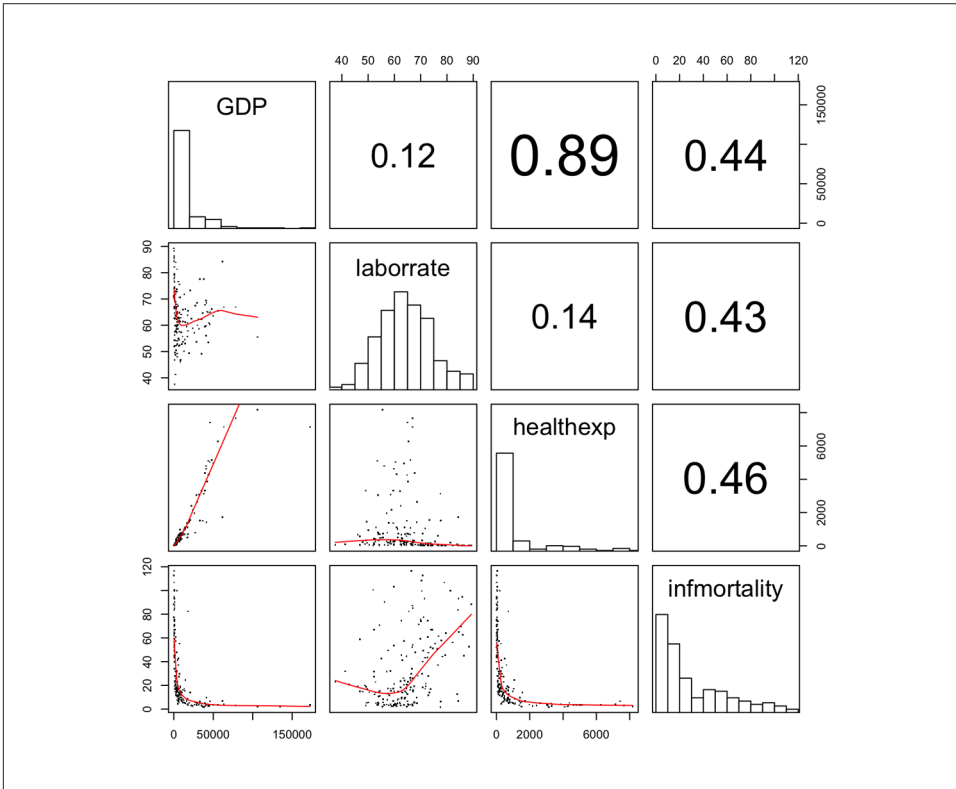


Figure 5-40. Scatter plot matrix with smaller points and linear fit lines

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](#).

It is worth noting that we didn't use `ggplot` here because it doesn't make scatter plot matrices (at least, not well).

Other packages like GGally have been developed as extensions to ggplot to fill in this gap. The `ggpairs()` function from the GGally package makes scatter plot matrices, for example.

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()
```

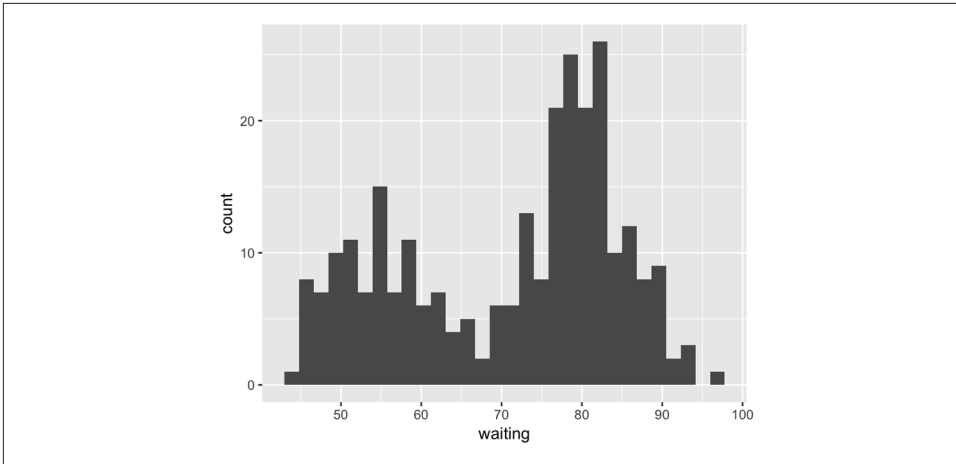


Figure 6-1. A basic histogram

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 two columns with data about the Old Faithful geyser: `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` variable in this example:

```
faithful
#>   eruptions waiting
#> 1      3.600      79
#> 2      1.800      54
#> 3      3.333      74
#> ...<266 more rows>...
#> 270     4.417      90
#> 271     1.817      46
#> 272     4.467      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 number of bins is an arbitrary default value, and may be too fine or too coarse for your data. You can change the size

of the bins by specifying the `binwidth`, or you can divide the range of the data into a specific number of bins.

In addition, 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 (each bin will span 5 x-axis units)
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")
```

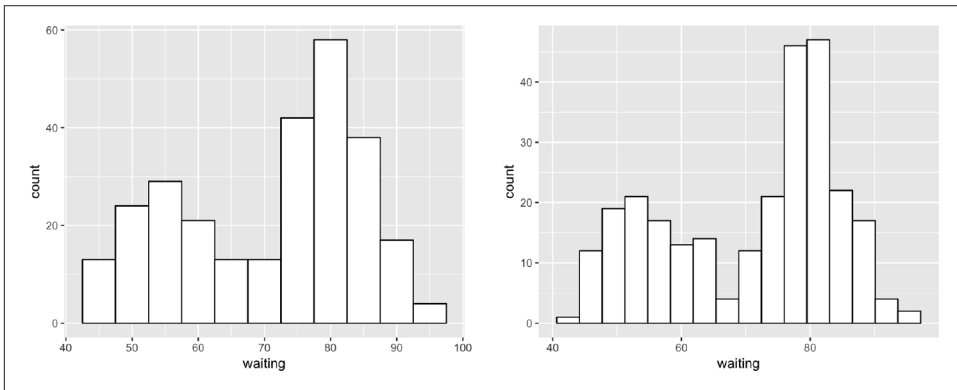


Figure 6-2. Histogram with `binwidth = 5` and with different colors (left); With 15 bins (right)

Sometimes the appearance of the histogram will be very dependent on the width of the bins and where the boundary points between the 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.:

```
# Save a base plot
faithful_p <- ggplot(faithful, aes(x = waiting))

faithful_p +
  geom_histogram(binwidth = 8, fill = "white", colour = "black", boundary = 31)

faithful_p +
  geom_histogram(binwidth = 8, fill = "white", colour = "black", boundary = 35)
```

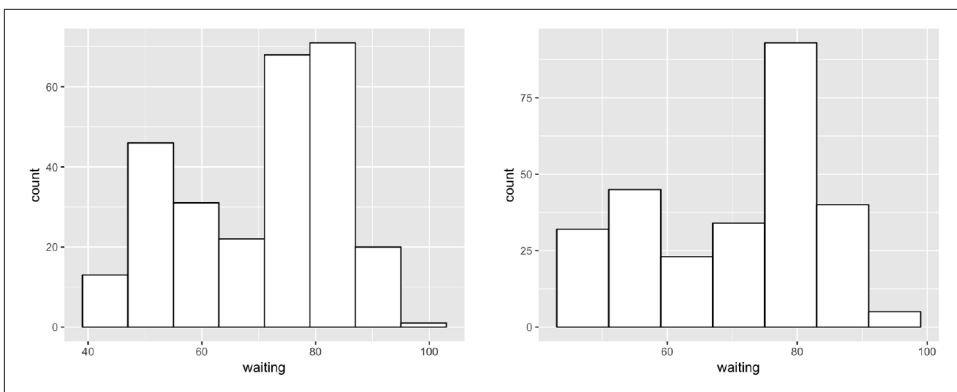


Figure 6-3. Different appearance of histograms with the origin at 31 and 35

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 can be even more of an issue. When visualizing your data, it's a good idea to experiment with different bin sizes and boundary points.

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.

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 have grouped data and want to simultaneously make histograms for each data group.

Solution

Use `geom_histogram()` and use facets for each group, as shown in [Figure 6-4](#):

```
library(MASS) # Load MASS for the birthwt data set

# Use smoke as the faceting variable
ggplot(birthwt, aes(x = bwt)) +
```

```
geom_histogram(fill = "white", colour = "black") +
facet_grid(smoke ~ .)
```

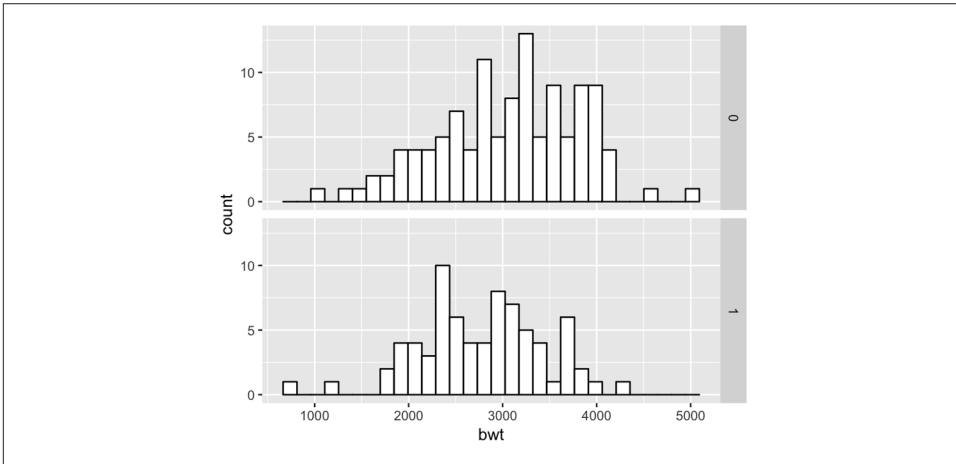


Figure 6-4. A histogram with facets

Discussion

To make multiple histograms from grouped data, 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
#> 85   0  19 182   2    0  0  0  1  0 2523
#> 86   0  33 155   3    0  0  0  0  3 2551
#> 87   0  20 105   1    1  0  0  0  1 2557
#> ...<183 more rows>...
#> 82   1  23  94   3    1  0  0  0  0 2495
#> 83   1  17 142   2    0  0  1  0  0 2495
#> 84   1  21 130   1    1  0  1  0  3 2495
```

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 whether or not smoking is a risk factor that is present. To change the labels, we 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, and save this new data set as `birthwt_mod`:

```
birthwt_mod <- birthwt
# Convert smoke to a factor and reassign new names
birthwt_mod$smoke <- recode_factor(birthwt_mod$smoke,
                                   '0' = 'No Smoke',
                                   '1' = 'Smoke')
```

Now when we plot our modified data frame, our desired labels appear (Figure 6-5):

```
ggplot(birthwt_mod, aes(x = bwt)) +  
  geom_histogram(fill = "white", colour = "black") +  
  facet_grid(smoke ~ .)
```

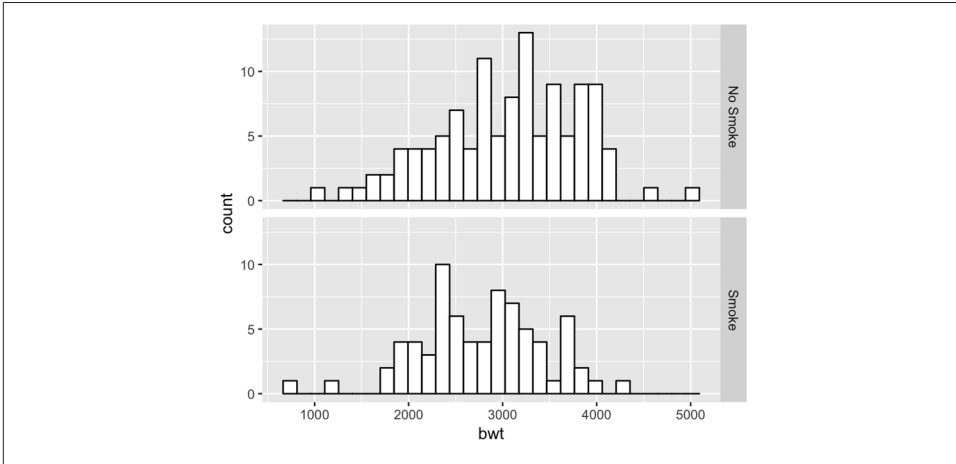


Figure 6-5. Histograms with new facet labels

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 6-6, 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 6-6, 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")
```

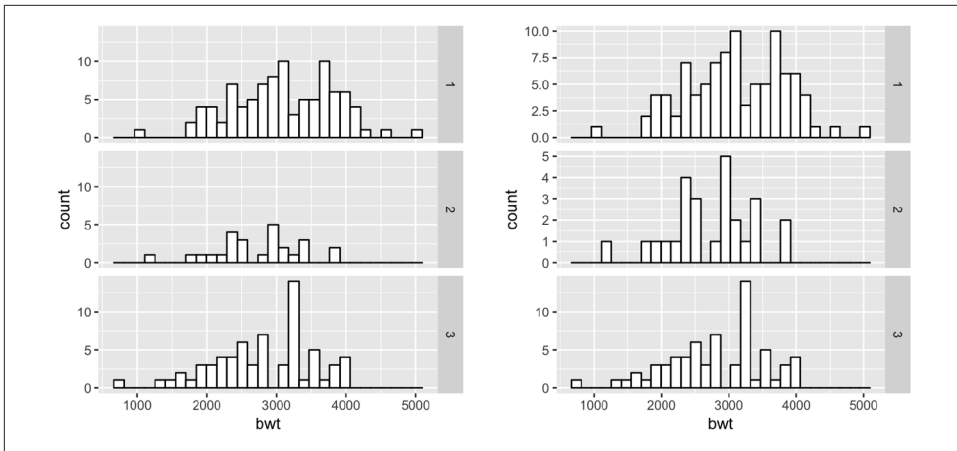


Figure 6-6. Histograms with the default fixed scales (left); With scales = "free" (right)

Another approach is to map the grouping variable to fill, as shown in [Figure 6-7](#). The grouping variable must be a factor or a character vector. In the `birthwt` data set, the desired grouping variable, `smoke`, is stored as a number, so we'll use the `birthwt_mod` data set we created previously, in which `smoke` is a factor:

```
# Map smoke to fill, make the bars NOT stacked, and make them semitransparent
ggplot(birthwt_mod, aes(x = bwt, fill = smoke)) +
  geom_histogram(position = "identity", alpha = 0.4)
```

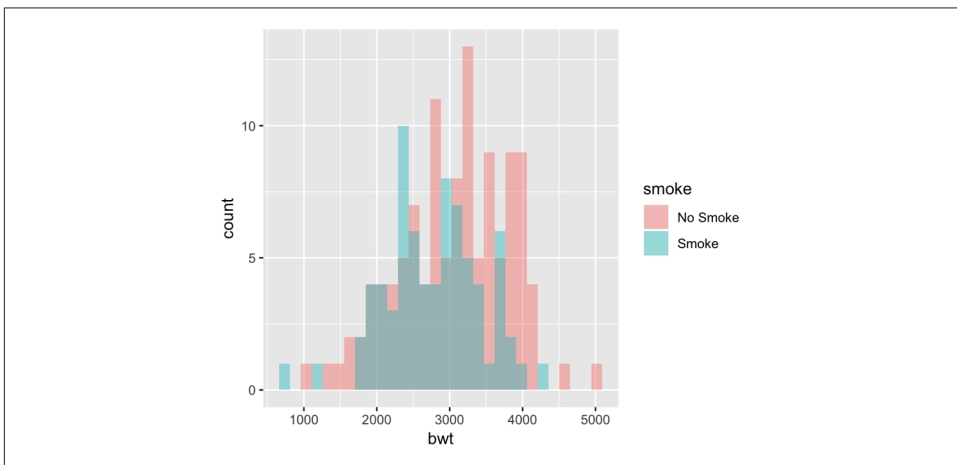


Figure 6-7. Multiple histograms with different fill colors

Specifying `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 estimate curve.

Solution

Use `geom_density()` and map a continuous variable to `x` (Figure 6-8):

```
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 6-8, right):

```
# expand_limits() increases the y range to include the value 0  
ggplot(faithful, aes(x = waiting)) +  
  geom_line(stat = "density") +  
  expand_limits(y = 0)
```

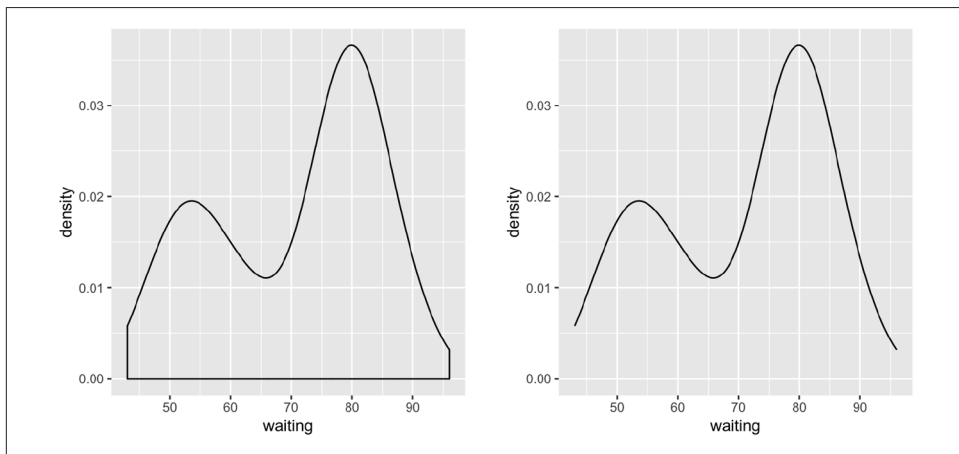


Figure 6-8. A kernel density estimate curve with `geom_density()` (left); With `geom_line()` (right)

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 two columns of data about the Old Faithful geyser: `eruptions`, which is the length of each eruption, and `waiting`, which is the length of time until the next eruption. We'll only use the `waiting` column in this example:


```
faithful
#>      eruptions waiting
#> 1      3.600      79
#> 2      1.800      54
#> 3      3.333      74
#> ...<266 more rows>...
#> 270     4.417      90
#> 271     1.817      46
#> 272     4.467      74
```

The second method of using `geom_line(stat = "density")` tells `geom_line()` 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 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 6-9](#) shows what happens with a smaller and larger value of `adjust`:

```
ggplot(faithful, aes(x = waiting)) +
  geom_line(stat = "density") +
  geom_line(stat = "density", adjust = .25, colour = "red") +
  geom_line(stat = "density", adjust = 2, colour = "blue")
```

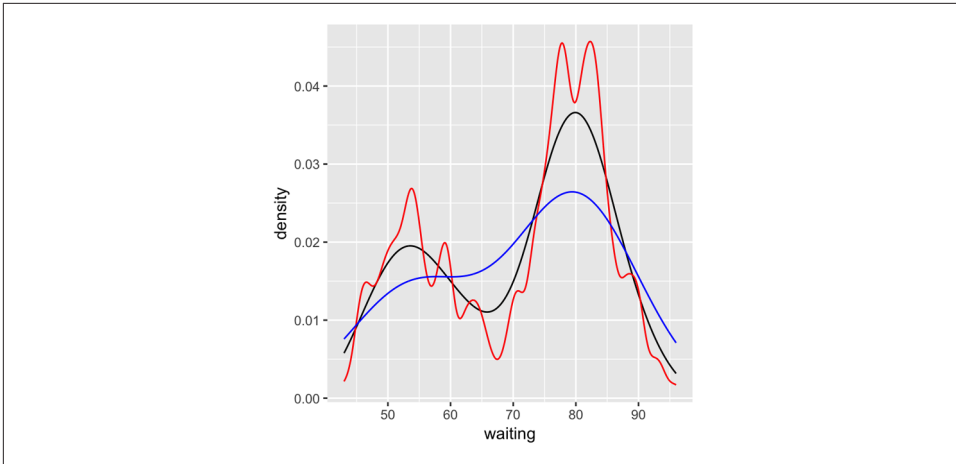


Figure 6-9. Density curves with *adjust* set to .25 (red), default value of 1 (black), and 2 (blue)

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 6-10). 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", alpha = .2, colour = NA) +
  xlim(35, 105) +
  geom_line(stat = "density")
```

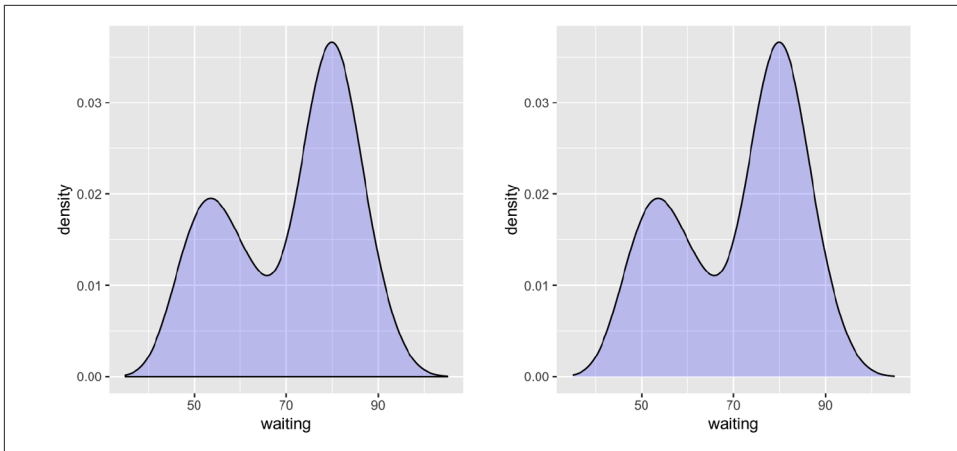


Figure 6-10. Density curve with wider x limits and a semitransparent fill (left); In two parts, with `geom_density()` and `geom_line()` (right)

If this edge-clipping happens with your data, it might mean that your curve is too smooth. If the curve is much 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.

To compare the theoretical and observed distributions of your data, 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 6-11):

```
ggplot(faithful, aes(x = waiting, y = ..density..)) +
  geom_histogram(fill = "cornsilk", colour = "grey60", size = .2) +
  geom_density() +
  xlim(35, 105)
```

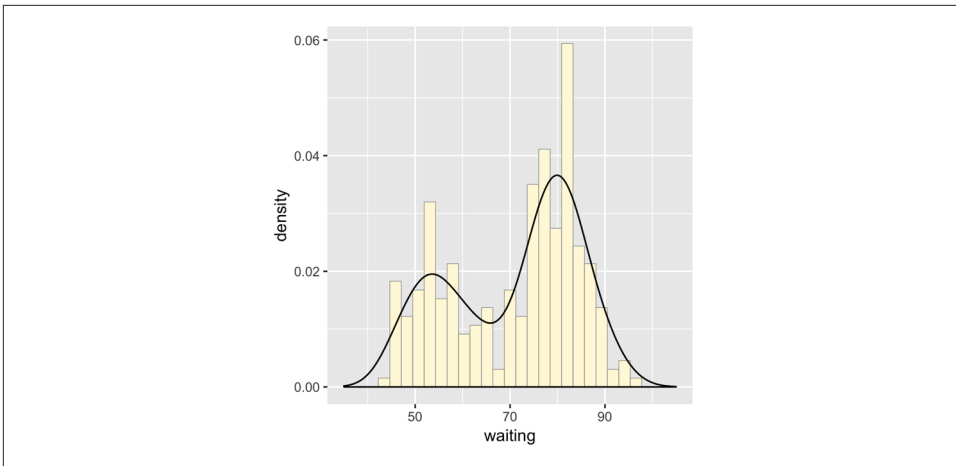


Figure 6-11. Density curve overlaid on a histogram

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 6-12](#). The grouping variable must be a factor or a 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) # Load MASS for the birthwt data set

birthwt_mod <- birthwt %>%
  mutate(smoke = as.factor(smoke)) # Convert smoke to a factor

# Map smoke to colour
ggplot(birthwt_mod, aes(x = bwt, colour = smoke)) +
  geom_density()

# Map smoke to fill and make the fill semitransparent by setting alpha
ggplot(birthwt_mod, aes(x = bwt, fill = smoke)) +
  geom_density(alpha = .3)
```

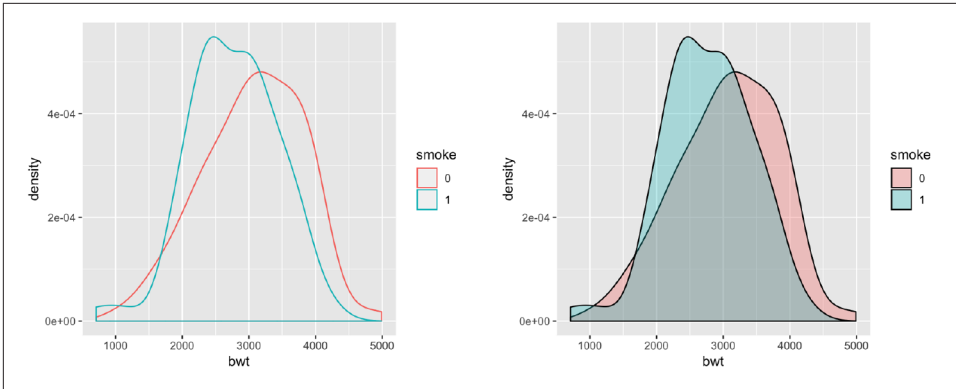


Figure 6-12. Different line colors for each group (left); Different semitransparent fill colors for each group (right)

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
#> 85    0  19 182   2    0  0  0  1   0 2523
#> 86    0  33 155   3    0  0  0  0   3 2551
#> 87    0  20 105   1    1  0  0  0   1 2557
#> ...<183 more rows>...
#> 82    1  23  94   3    1  0  0  0   0 2495
#> 83    1  17 142   2    0  0  1  0   0 2495
#> 84    1  21 130   1    1  0  1  0   3 2495
```

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 `smoke` to a factor.

Another method for visualizing the distributions is to use facets, as shown in [Figure 6-13](#). We can align the facets vertically or horizontally. Here we'll align them vertically so that it's easy to compare the two distributions:

```
ggplot(birthwt_mod, 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:

```
levels(birthwt_mod$smoke)
#> [1] "0" "1"
```

```
birthwt_mod$smoke <- recode(birthwt_mod$smoke, '0' = 'No Smoke', '1' = 'Smoke')
```

Now when we plot our modified data frame, our desired labels appear (Figure 6-13, right):

```
ggplot(birthwt_mod, aes(x = bwt)) +
  geom_density() +
  facet_grid(smoke ~ .)
```

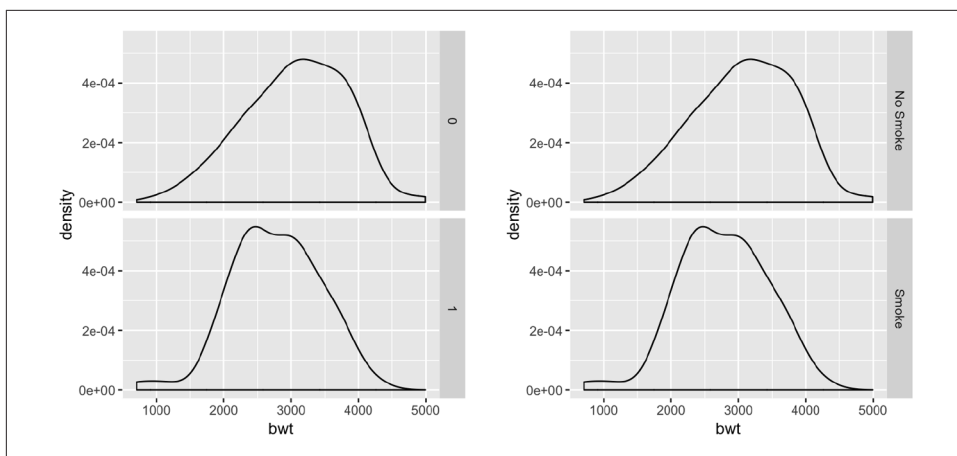


Figure 6-13. Density curves with facets (left); With different facet labels (right)

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 6-14):

```
ggplot(birthwt_mod, aes(x = bwt, y = ..density..)) +
  geom_histogram(binwidth = 200, fill = "cornsilk", colour = "grey60",
    size = .2) +
  geom_density() +
  facet_grid(smoke ~ .)
```

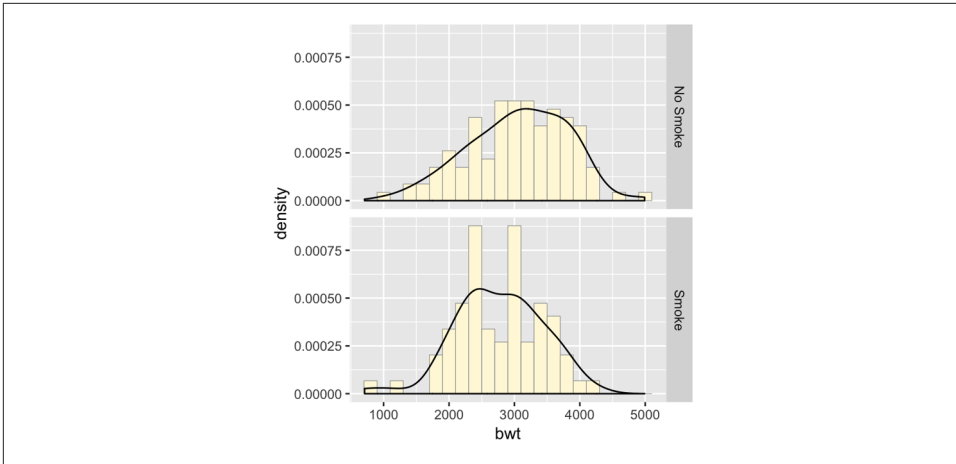


Figure 6-14. Density curves overlaid on histograms

6.5 Making a Frequency Polygon

Problem

You want to make a frequency polygon.

Solution

Use `geom_freqpoly()` (Figure 6-15):

```
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.

Like with a histogram, you can control the bin width for the frequency polygon (Figure 6-15, right):

```
ggplot(faithful, aes(x = waiting)) +  
  geom_freqpoly(binwidth = 4)
```

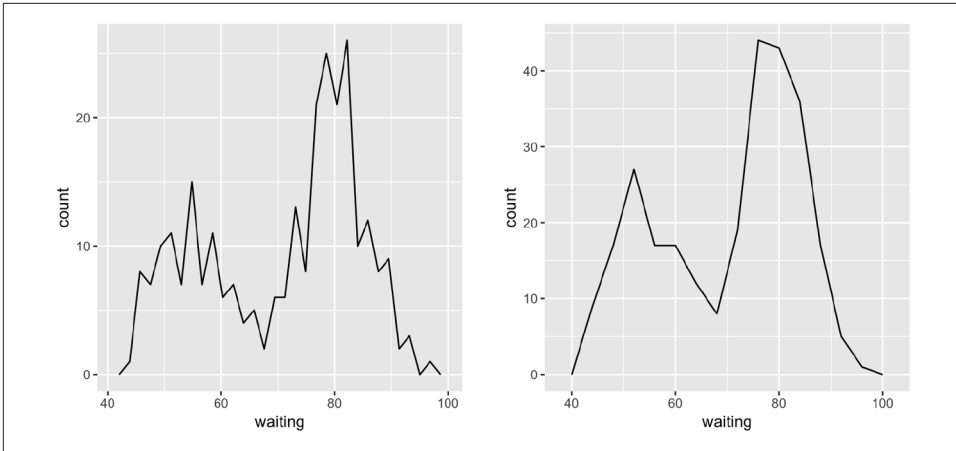


Figure 6-15. A frequency polygon (left); With wider bins (right)

Or, instead of setting the width of each bin directly, you can divide the x range into a particular number of bins:

```
# Divide the x-axis range into 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 6-16](#)):

```
library(MASS) # Load MASS for the birthwt data set

# Use factor() to convert a numeric variable into a discrete variable
ggplot(birthwt, aes(x = factor(race), y = bwt)) +
  geom_boxplot()
```

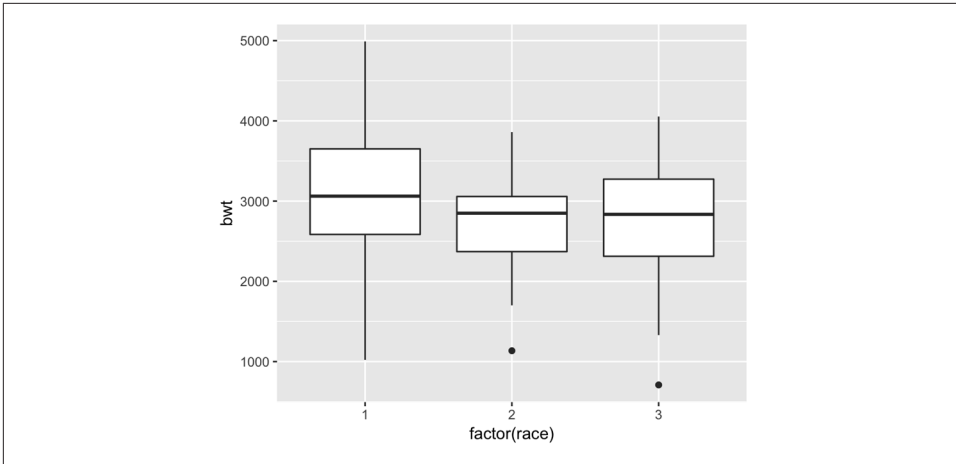



Figure 6-16. A box plot

Discussion

For this example, we used the `birthwt` data set from the `MASS` package. This data set contains data about birth weights (`bwt`) and a number of risk factors for low birth weight:

```
birthwt
#>   low age lwt race smoke ptl ht ui ftv bwt
#> 85   0  19 182   2     0  0  0  1  0 2523
#> 86   0  33 155   3     0  0  0  0  3 2551
#> 87   0  20 105   1     1  0  0  0  1 2557
#> ...<183 more rows>...
#> 82   1  23  94   3     1  0  0  0  0 2495
#> 83   1  17 142   2     0  0  1  0  0 2495
#> 84   1  21 130   1     1  0  1  0  3 2495
```

In [Figure 6-16](#) we have visualized the distributions of `bwt` by each race group. Because `race` is stored as a numeric vector with the values of 1, 2, or 3, `ggplot` doesn't know how to use this numeric version of `race` as a grouping variable. To make this work, we can modify the data frame by converting `race` to a factor, or by telling `ggplot` to treat `race` as a factor by using `factor(race)` inside of the `aes()` statement. In the preceding example, we used `factor(race)`.

A box plot consists of a box and “whiskers.” The box goes from the 25th percentile to the 75th percentile of the data, also known as the *inter-quartile range* (IQR). There's a line indicating the median, or the 50th percentile of the data. The whiskers start from the edge of the box and extend to the furthest data point that is within 1.5 times the IQR. Any data points that are past the ends of the whiskers are considered outliers

and displayed with dots. **Figure 6-17** shows the relationship between a histogram, a density curve, and a box plot, using a skewed data set.

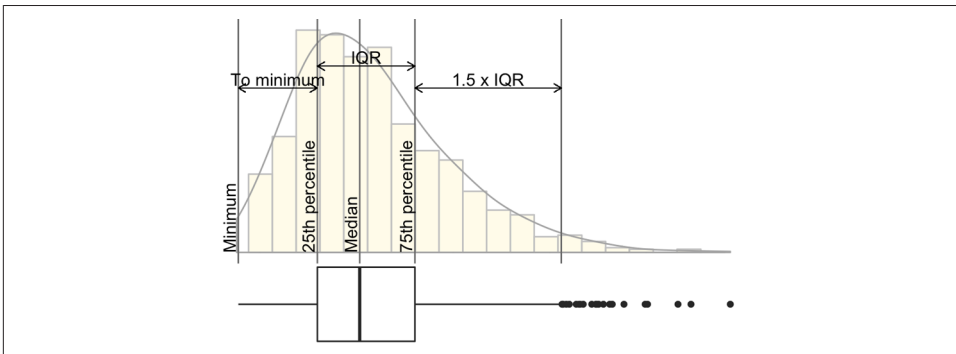


Figure 6-17. Box plot compared to histogram and density curve

To change the width of the boxes, you can set width (**Figure 6-18**, left):

```
ggplot(birthwt, aes(x = factor(race), y = bwt)) +  
  geom_boxplot(width = .5)
```

If there are many outliers and there is overplotting, you can change the size and shape of the outlier points with `outlier.size` and `outlier.shape`. The default size is 2 and the default shape is 16. This will use smaller points, and hollow circles (**Figure 6-18**, right):

```
ggplot(birthwt, aes(x = factor(race), y = bwt)) +  
  geom_boxplot(outlier.size = 1.5, outlier.shape = 21)
```

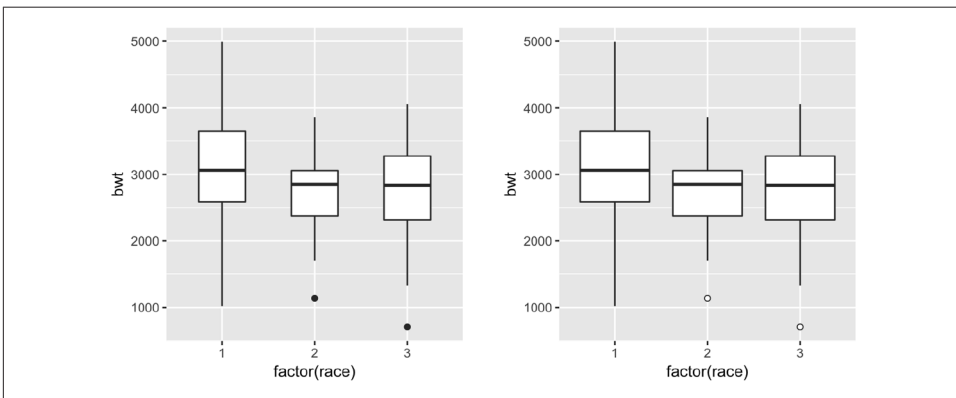


Figure 6-18. Box plot with narrower boxes (left); With smaller, hollow outlier points (right)

To make a box plot of just a single group, we have to provide some arbitrary value for `x`; otherwise, `ggplot` won't know what `x` coordinate to use for the box plot. In this case, we'll set it to 1 and remove the `x`-axis tick markers and label (Figure 6-19):

```
ggplot(birthwt, aes(x = 1, y = bwt)) +  
  geom_boxplot() +  
  scale_x_continuous(breaks = NULL) +  
  theme(axis.title.x = element_blank())
```

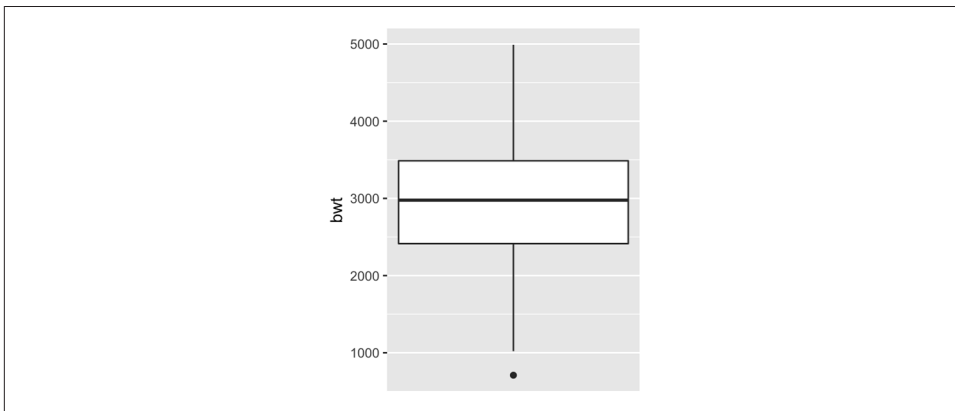


Figure 6-19. Box plot of a single group



The calculation of quantiles works slightly differently from the `boxplot()` function in base R. This can sometimes be noticeable for small sample sizes. See `?geom_boxplot` for detailed information about how the calculations differ.

6.7 Adding Notches to a Box Plot

Problem

You want to add notches to a box plot to assess whether the medians are different.

Solution

Use `geom_boxplot()` and set `notch = TRUE` (Figure 6-20):

```
library(MASS) # Load MASS for the birthwt data set  
  
ggplot(birthwt, aes(x = factor(race), y = bwt)) +  
  geom_boxplot(notch = TRUE)
```

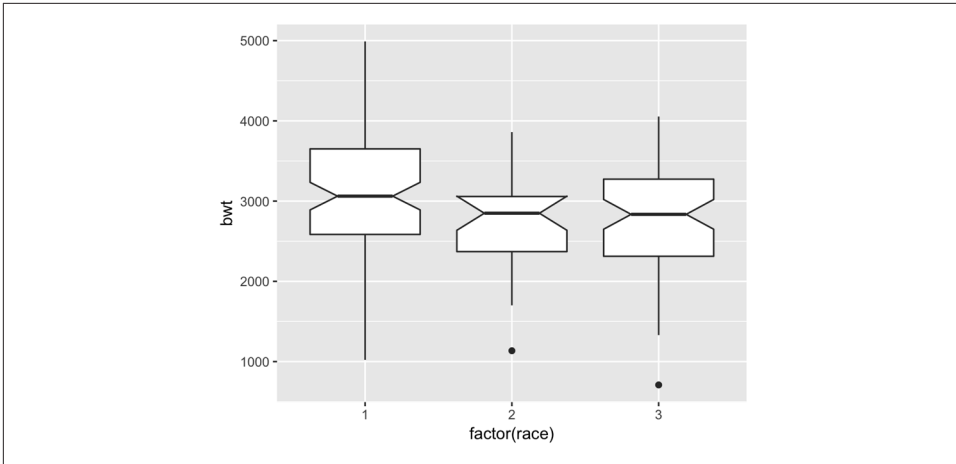


Figure 6-20. A notched box plot

Discussion

Notches are used in box plots to help visually assess whether the medians of distributions differ. If the notches do not overlap, this is evidence that the medians are different.

With this particular data set, you'll see the following message:

```
Notch went outside hinges. Try setting notch=FALSE.
```

This means that the confidence region (the notch) went past the bounds (or hinges) of one of the boxes. In this case, the upper part of the notch in the middle box goes just barely outside the box body, but it's by such a small amount that you can't see it in the final output. There's nothing inherently wrong with a notch going outside the hinges, but it can look strange in more extreme cases.

6.8 Adding Means to a Box Plot

Problem

You want to add markers for the mean to a box plot.

Solution

Use `stat_summary()`. The mean is often shown with a diamond, so we'll use shape 23 with a white fill. We'll also make the diamond slightly larger by setting `size = 3` (Figure 6-21):

```
library(MASS) # Load MASS for the birthwt data set

ggplot(birthwt, aes(x = factor(race), y = bwt)) +
  geom_boxplot() +
  stat_summary(fun.y = "mean", geom = "point", shape = 23, size = 3,
              fill = "white")
```

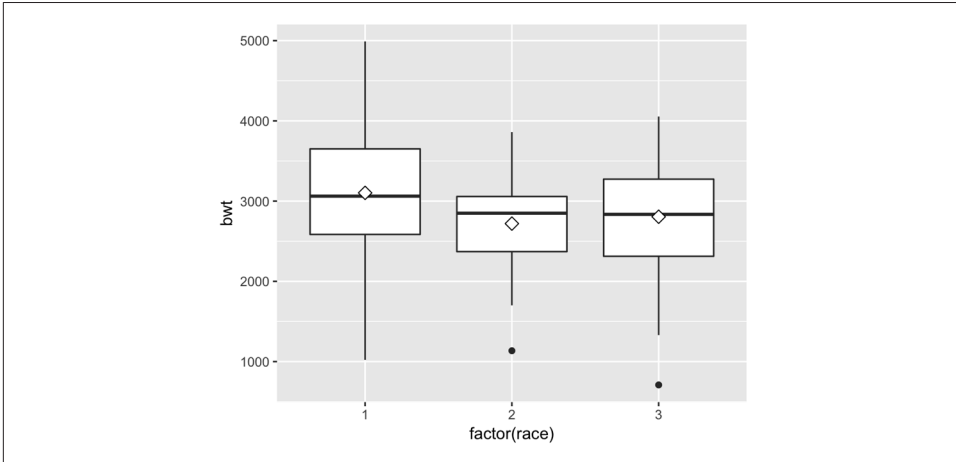


Figure 6-21. Mean markers on a box plot

Discussion

The horizontal line in the middle of a box plot displays the median, not the mean. For data that is normally distributed, the median and mean will be about the same, but for skewed data these values will differ.

6.9 Making a Violin Plot

Problem

You want to make a violin plot to compare density estimates of different groups.

Solution

Use `geom_violin()` (Figure 6-22):

```
library(gcookbook) # Load gcookbook for the heightweight data set

# Create a base plot using the heightweight data set
hw_p <- ggplot(heightweight, aes(x = sex, y = heightIn))

hw_p +
  geom_violin()
```

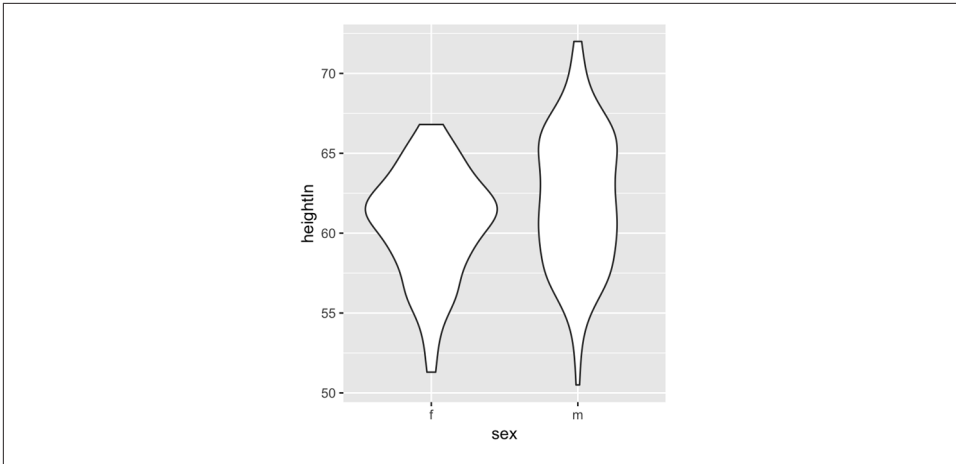


Figure 6-22. A violin plot

Discussion

Violin plots are a way of comparing multiple data distributions. With ordinary density curves, it is difficult to compare more than just a few distributions because the lines visually interfere with each other. With a violin plot, it's easier to compare several distributions since they're placed side by side.

A violin plot is a kernel density estimate, mirrored so that it forms a symmetrical shape. Traditionally, they also have narrow box plots overlaid, with a white dot at the median, as shown in [Figure 6-23](#). Additionally, the box plot outliers are not displayed, which we do by setting `outlier.colour = NA`:

```
hw_p +
  geom_violin() +
  geom_boxplot(width = .1, fill = "black", outlier.colour = NA) +
  stat_summary(fun.y = median, geom = "point", fill = "white", shape = 21,
               size = 2.5)
```

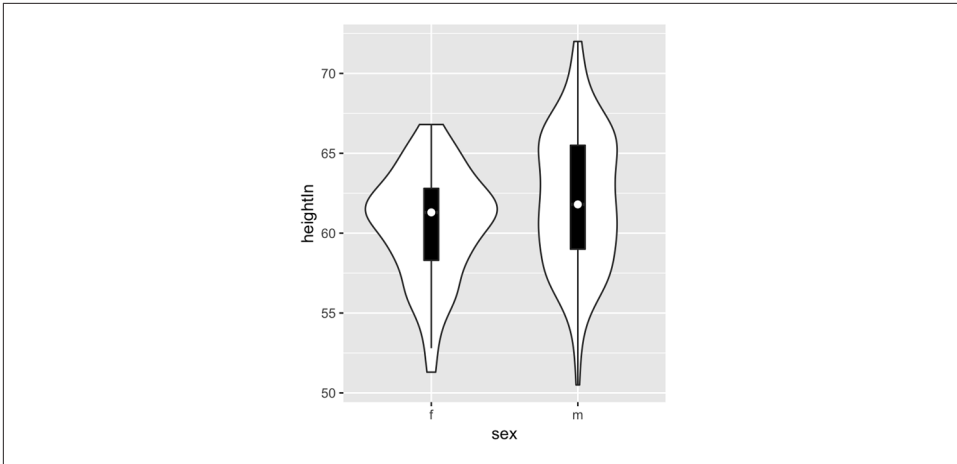


Figure 6-23. A violin plot with box plot overlaid on it

In this example we layered the objects from the bottom up, starting with the violin, then the box plot, then the white dot at the median, which is calculated using `stat_summary()`.

The default range goes from the minimum to maximum data values; the flat ends of the violins are at the extremes of the data. It's possible to keep the tails, by setting `trim = FALSE` (Figure 6-24):

```
hw_p +  
  geom_violin(trim = FALSE)
```

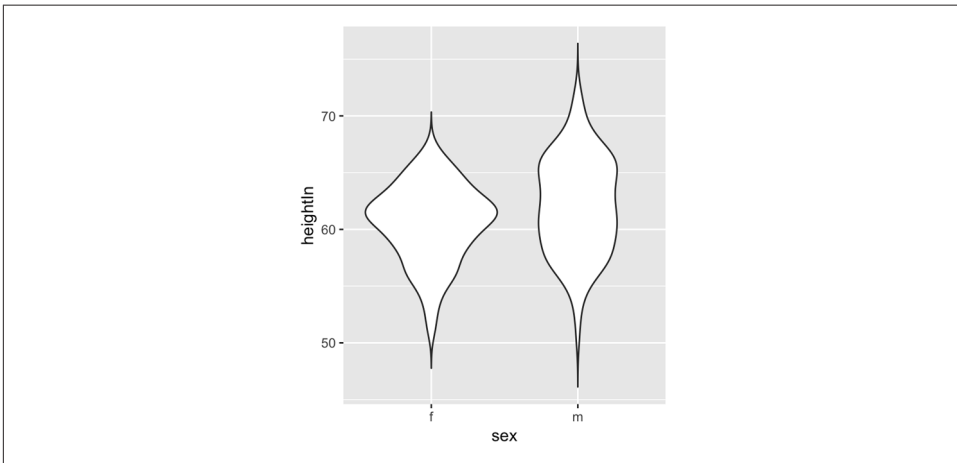


Figure 6-24. A violin plot with tails

By default, the violins are scaled so that the total area of each one is the same (if `trim = TRUE`, then it scales what the area *would be* including the tails). Instead of equal areas, you can use `scale = "count"` to scale the areas proportionally to the number of observations in each group (Figure 6-25). In this example, there are slightly fewer females than males, so the female violin becomes slightly narrower than before:

```
# Scaled area proportional to number of observations
hw_p +
  geom_violin(scale = "count")
```

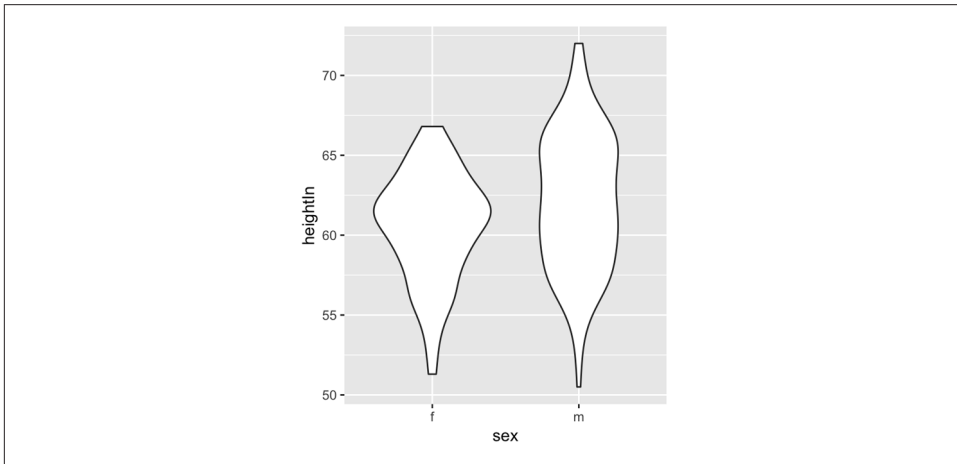


Figure 6-25. Violin plot with area proportional to number of observations

To change the amount of smoothing, use the `adjust` parameter, as described in Recipe 6.3. The default value is 1; use larger values for more smoothing and smaller values for less smoothing (Figure 6-26):

```
# More smoothing
hw_p +
  geom_violin(adjust = 2)

# Less smoothing
hw_p +
  geom_violin(adjust = .5)
```

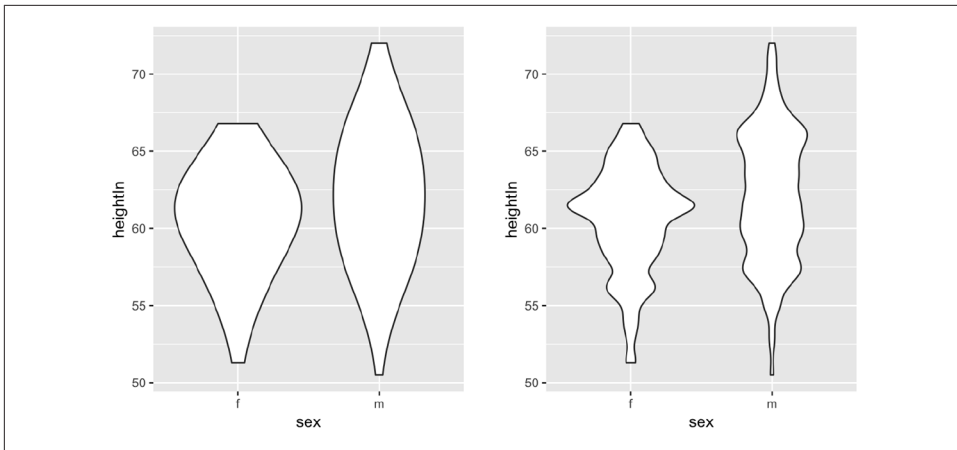



Figure 6-26. Violin plot with more smoothing (left); With less smoothing (right)

See Also

To create a traditional density curve, see [Recipe 6.3](#).

To use different point shapes, see [Recipe 4.5](#).

6.10 Making a Dot Plot

Problem

You want to make a Wilkinson dot plot, which shows each data point.

Solution

Use `geom_dotplot()`. For this example ([Figure 6-27](#)), we'll use a subset of the countries data set:

```
library(gcookbook) # Load gcookbook for the countries data set
library(dplyr)

# Save a modified data set that only includes 2009 data for countries that
# spent > 2000 USD per capita
c2009 <- countries %>%
  filter(Year == 2009 & healthexp > 2000)

# Create a base ggplot object using `c2009`, called `c2009_p` (for c2009 plot)
c2009_p <- ggplot(c2009, aes(x = infmortality))

c2009_p +
  geom_dotplot()
```

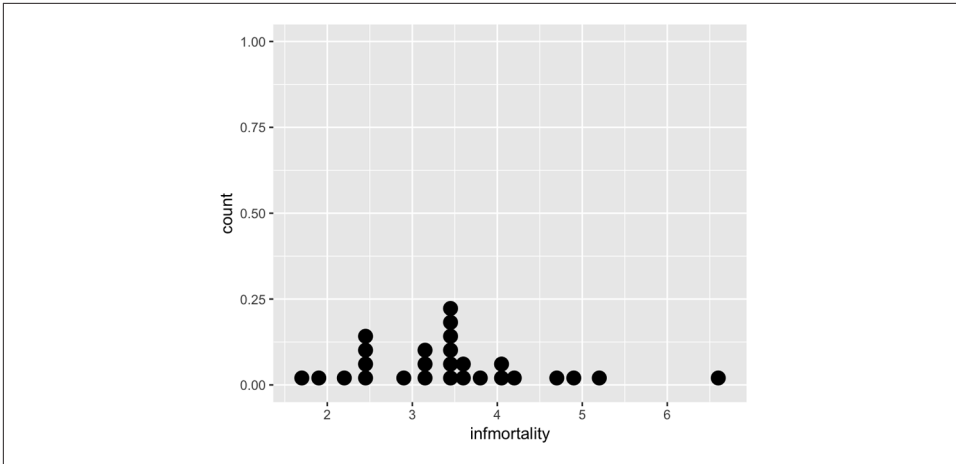


Figure 6-27. A dot plot

Discussion

This kind of dot plot is sometimes called a *Wilkinson* dot plot. It's different from the Cleveland dot plots shown in [Recipe 3.10](#). In these Wilkinson dot plots, the placement of the bins depends on the data, and the width of each dot corresponds to the maximum width of each bin. The maximum bin size defaults to 1/30 of the range of the data, but it can be changed with `binwidth`.

By default, `geom_dotplot()` bins the data along the x-axis and stacks on the y-axis. The dots are stacked visually, and due to technical limitations of `ggplot2`, the resulting graph has y-axis tick marks that aren't meaningful. The y-axis labels can be removed by using `scale_y_continuous()`. In this example, we'll also use `geom_rug()` to show exactly where each data point is ([Figure 6-28](#)):

```
c2009_p +
  geom_dotplot(binwidth = .25) +
  geom_rug() +
  scale_y_continuous(breaks = NULL) + # Remove tick markers
  theme(axis.title.y = element_blank()) # Remove axis label
```

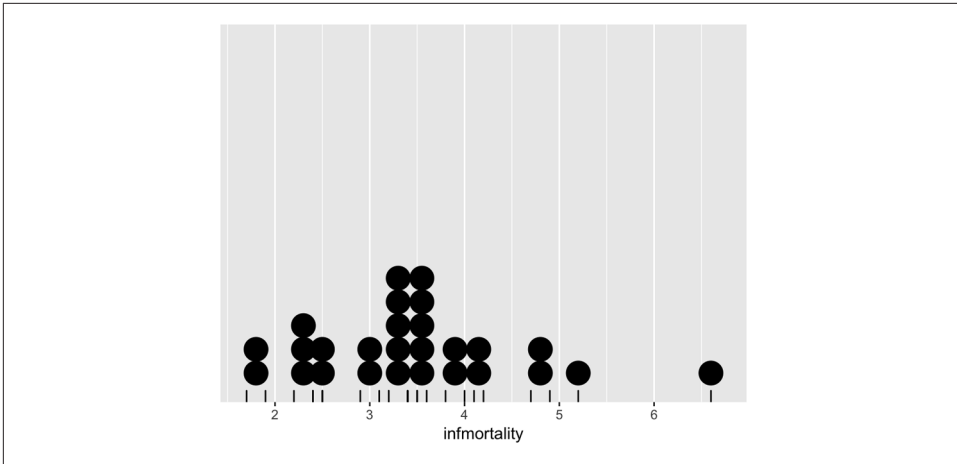


Figure 6-28. Dot plot with no y labels, max bin size of .25, and a rug showing each data point

You may notice that the stacks aren't regularly spaced in the horizontal direction. With the default dot-density binning algorithm, the position of each stack is centered above the set of data points that it represents. To use bins that are arranged with a fixed, regular spacing, like a histogram, use `method = "histodot"`. In [Figure 6-29](#), you'll notice that the stacks *aren't* centered above the data:

```
c2009_p +
  geom_dotplot(method = "histodot", binwidth = .25) +
  geom_rug() +
  scale_y_continuous(breaks = NULL) +
  theme(axis.title.y = element_blank())
```

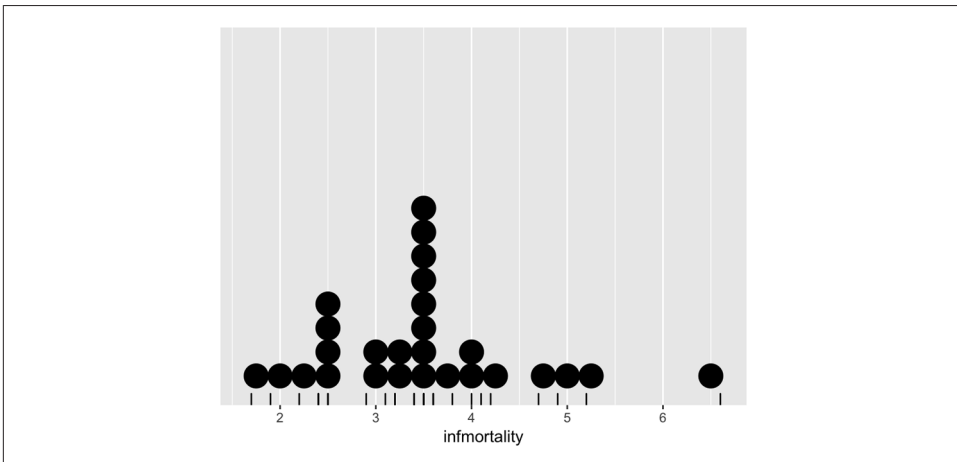


Figure 6-29. Dot plot with *histodot* (fixed-width) binning

The dots can also be stacked centered, or centered in such a way that stacks with even and odd quantities stay aligned. This can be done by setting `stackdir = "center"` or `stackdir = "centerwhole"`, as illustrated in Figure 6-30:

```
c2009_p +
  geom_dotplot(binwidth = .25, stackdir = "center") +
  scale_y_continuous(breaks = NULL) +
  theme(axis.title.y = element_blank())

c2009_p +
  geom_dotplot(binwidth = .25, stackdir = "centerwhole") +
  scale_y_continuous(breaks = NULL) +
  theme(axis.title.y = element_blank())
```

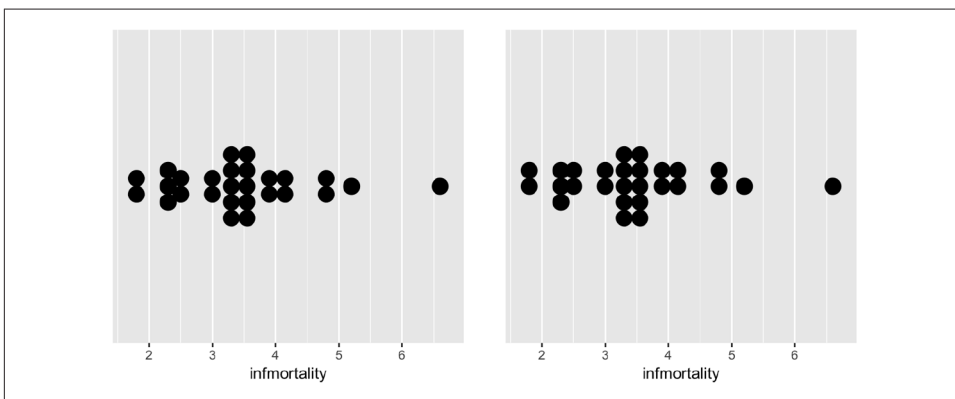


Figure 6-30. Dot plot with `stackdir = "center"` (left); With `stackdir = "centerwhole"` (right)

See Also

Leland Wilkinson, “Dot Plots,” *The American Statistician* 53 (1999): 276–281, <https://www.cs.uic.edu/~wilkinson/Publications/dotplots.pdf>.

6.11 Making Multiple Dot Plots for Grouped Data

Problem

You want to make multiple dot plots from grouped data.

Solution

To compare multiple groups, it’s possible to stack the dots along the y-axis, and group them along the x-axis, by setting `binaxis = "y"`. For this example, we’ll use the heightweight data set (Figure 6-31):

```
library(gcookbook) # Load gcookbook for the heightweight data set

ggplot(heightweight, aes(x = sex, y = heightIn)) +
  geom_dotplot(binaxis = "y", binwidth = .5, stackdir = "center")
```

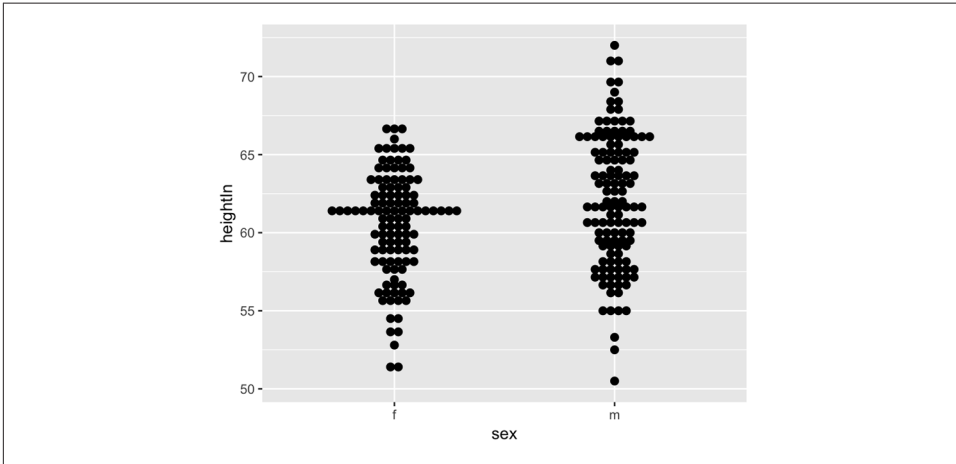


Figure 6-31. Dot plot of multiple groups, binning along the y-axis

Discussion

Dot plots are sometimes overlaid on box plots. In these cases, it may be helpful to make the dots hollow and have the box plots *not* show outliers, since the outlier points will appear to be part of the dot plot (Figure 6-32):

```
ggplot(heightweight, aes(x = sex, y = heightIn)) +
  geom_boxplot(outlier.colour = NA, width = .4) +
  geom_dotplot(binaxis = "y", binwidth = .5, stackdir = "center", fill = NA)
```

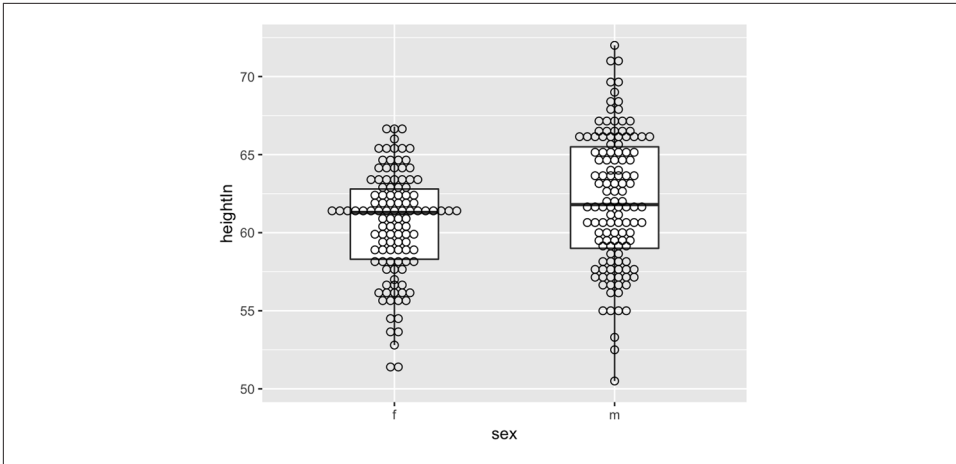


Figure 6-32. Dot plot overlaid on box plot

It's also possible to show the dot plots next to the box plots, as shown in [Figure 6-33](#). This requires using a bit of a hack, by treating the *x* variable as a numeric variable and then subtracting or adding a small quantity to shift the box plots and dot plots left and right. When the *x* variable is treated as numeric you must also specify the group, or else the data will be treated as a single group, with just one box plot and dot plot. Finally, since the *x*-axis is treated as numeric, it will by default show numbers for the *x*-axis tick labels; they must be modified with `scale_x_continuous()` to show *x* tick labels as text corresponding to the factor levels:

```
ggplot(heightweight, aes(x = sex, y = heightIn)) +
  geom_boxplot(aes(x = as.numeric(sex) + .2, group = sex), width = .25) +
  geom_dotplot(
    aes(x = as.numeric(sex) - .2, group = sex),
    binaxis = "y",
    binwidth = .5,
    stackdir = "center"
  ) +
  scale_x_continuous(
    breaks = 1:nlevels(heightweight$sex),
    labels = levels(heightweight$sex)
  )
```

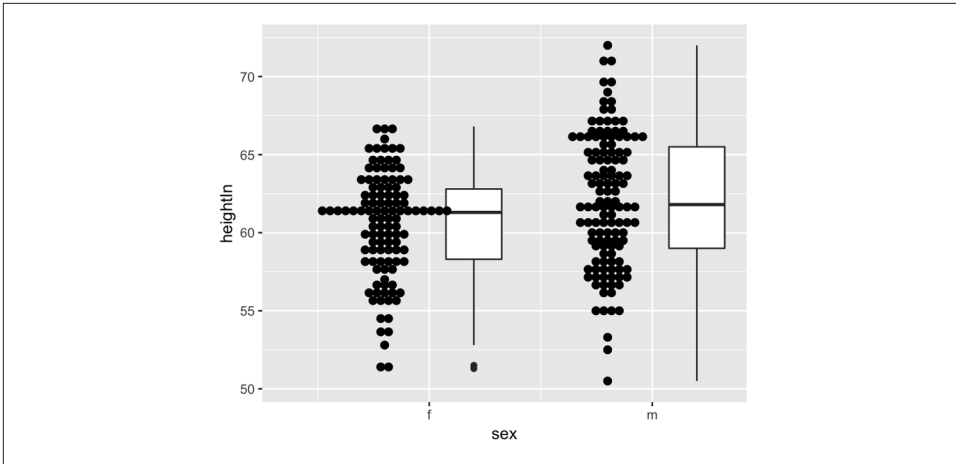


Figure 6-33. Dot plot next to box plot

6.12 Making a Density Plot of Two-Dimensional Data

Problem

You want to plot the density of two-dimensional data.

Solution

Use `stat_density2d()`. This makes a 2D kernel density estimate from the data. First, we'll plot the density contour along with the data points (Figure 6-34, left):

```
# Save a base plot object
faithful_p <- ggplot(faithful, aes(x = eruptions, y = waiting))

faithful_p +
  geom_point() +
  stat_density2d()
```

It's also possible to map the *height* of the density curve to the color of the contour lines, by using `..level..` (Figure 6-34, right):

```
# Contour lines, with "height" mapped to color
faithful_p +
  stat_density2d(aes(colour = ..level..))
```

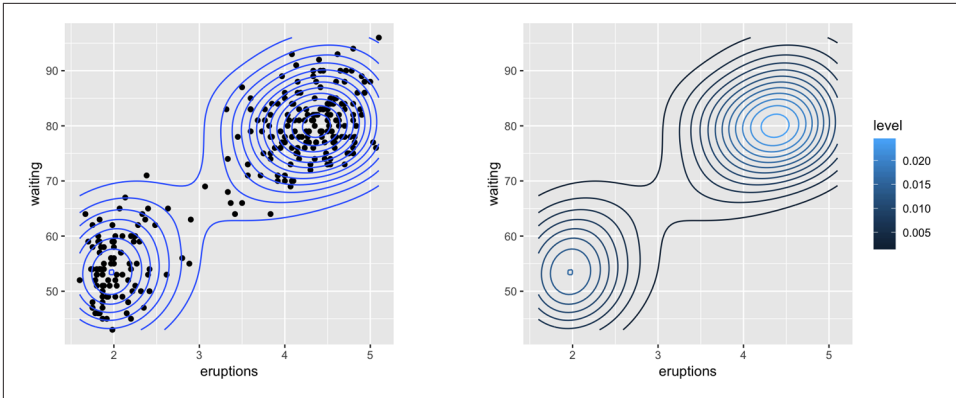


Figure 6-34. Points and density contour (left); With `..level..` mapped to color (right)

Discussion

The two-dimensional kernel density estimate is analogous to the one-dimensional density estimate generated by `stat_density()`, but of course, it needs to be viewed in a different way. The default is to use contour lines, but it's also possible to use tiles and to map the density estimate to the fill color, or to the transparency of the tiles, as shown in Figure 6-35:

```
# Map density estimate to fill color
faithful_p +
  stat_density2d(aes(fill = ..density..), geom = "raster", contour = FALSE)

# With points, and map density estimate to alpha
faithful_p +
  geom_point() +
  stat_density2d(aes(alpha = ..density..), geom = "tile", contour = FALSE)
```

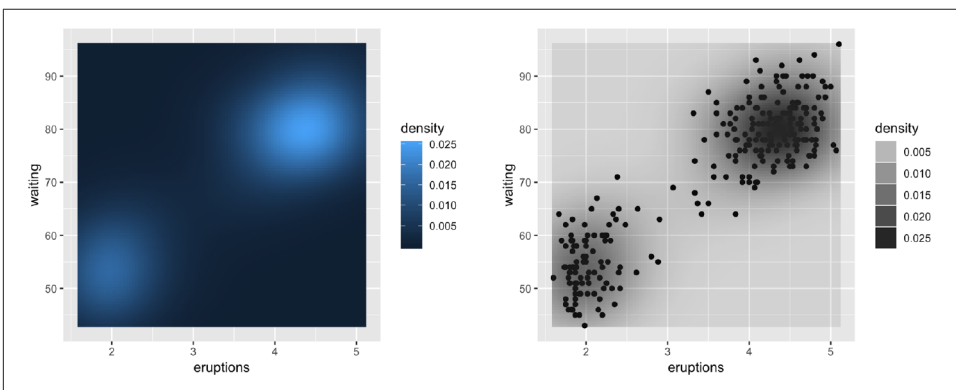


Figure 6-35. With `..density..` mapped to fill (left); With points, and `..density..` mapped to alpha (right)



We used `geom = "raster"` in the first of the preceding examples and `geom = "tile"` in the second. The main difference is that the raster geom renders more efficiently than the tile geom. In theory, they *should* appear the same, but in practice they often do not. If you are writing to a PDF file, the appearance depends on the PDF viewer. On some viewers, when tile is used there may be faint lines between the tiles, and when raster is used the edges of the tiles may appear blurry (although it doesn't matter in this particular case).

As with the one-dimensional density estimate, you can control the bandwidth of the estimate. To do this, pass a vector for the *x* and *y* bandwidths to `h`. This argument gets passed on to the function that actually generates the density estimate, `kde2d()`. In this example (Figure 6-36), we'll use a smaller bandwidth in the *x* and *y* directions, so that the density estimate is more closely fitted (perhaps overfitted) to the data:

```
faithful_p +  
  stat_density2d(  
    aes(fill = ..density..),  
    geom = "raster",  
    contour = FALSE,  
    h = c(.5, .5)  
  )
```

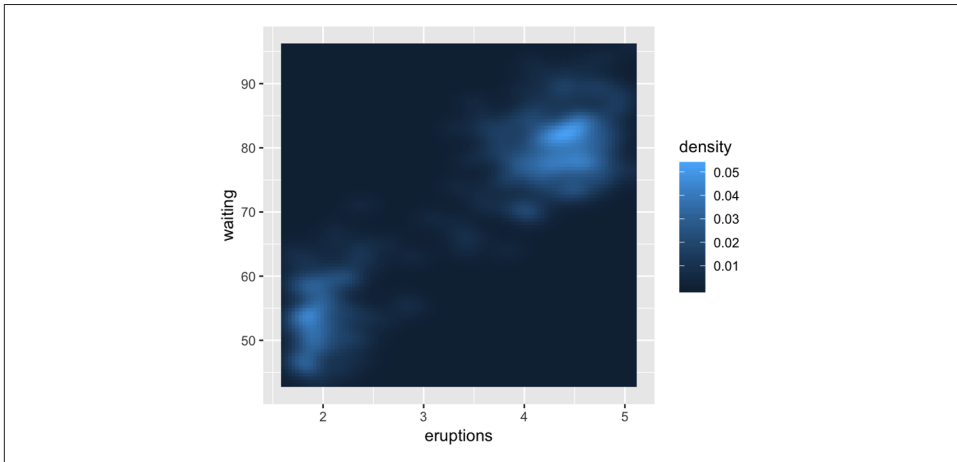


Figure 6-36. Density plot with a smaller bandwidth in the *x* and *y* directions

See Also

The relationship between `stat_density2d()` and `stat_bin2d()` is the same as the relationship between their one-dimensional counterparts, the density curve and the histogram. The density curve is an *estimate* of the distribution under certain assump-

tions, while the binned visualization represents the observed data directly. See [Recipe 5.5](#) for more about binning data.

If you want to use a different color palette, see [Recipe 12.6](#).

`stat_density2d()` passes options to `kde2d()`; see `?kde2d` for information on the available options.

Annotations

Displaying just your data usually isn't enough—there's all sorts of other information that can help the viewer interpret the data. In addition to the standard repertoire of axis labels, tick marks, and legends, you can also add individual graphical or text elements to your plot. These elements can be used to add extra contextual information, highlight an area of the plot, or add some descriptive text about the data.

7.1 Adding Text Annotations

Problem

You want to add a text annotation to a plot.

Solution

Use `annotate()` and a text geom (Figure 7-1):

```
p <- ggplot(faithful, aes(x = eruptions, y = waiting)) +  
  geom_point()  
  
p +  
  annotate("text", x = 3, y = 48, label = "Group 1") +  
  annotate("text", x = 4.5, y = 66, label = "Group 2")
```

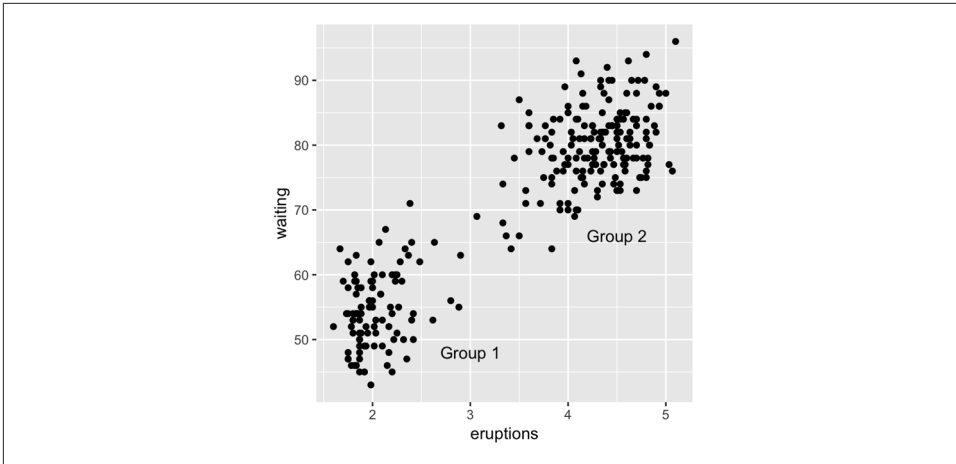


Figure 7-1. Text annotations

Discussion

The `annotate()` function can be used to add any type of geometric object. In this case, we used `geom = "text"`.

Other text properties can be specified, as shown in Figure 7-2:

```
p +
  annotate("text", x = 3, y = 48, label = "Group 1",
    family = "serif", fontface = "italic", colour = "darkred", size = 3) +
  annotate("text", x = 4.5, y = 66, label = "Group 2",
    family = "serif", fontface = "italic", colour = "darkred", size = 3)
```

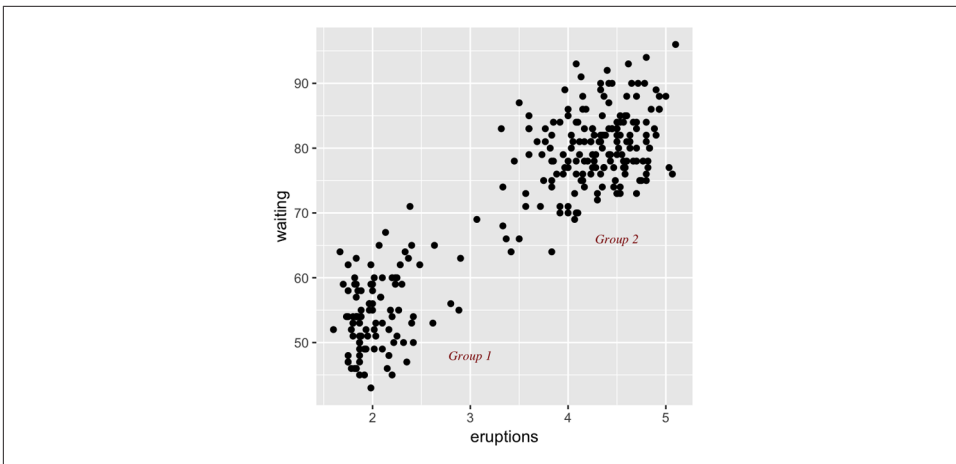


Figure 7-2. Modified text properties

Be careful not to use `geom_text()` when you want to add individual text objects. While `annotate(geom = "text")` will add a single text object to the plot, `geom_text()` will create many text objects based on the data, as discussed in [Recipe 5.11](#).

If you use `geom_text()`, the text will be heavily overplotted on the same location, with one copy per data point ([Figure 7-3](#)):

```
p +
  # Normal
  annotate("text", x = 3, y = 48, label = "Group 1", alpha = .1) +
  # Overplotted
  geom_text(x = 4.5, y = 66, label = "Group 2", alpha = .1)
```

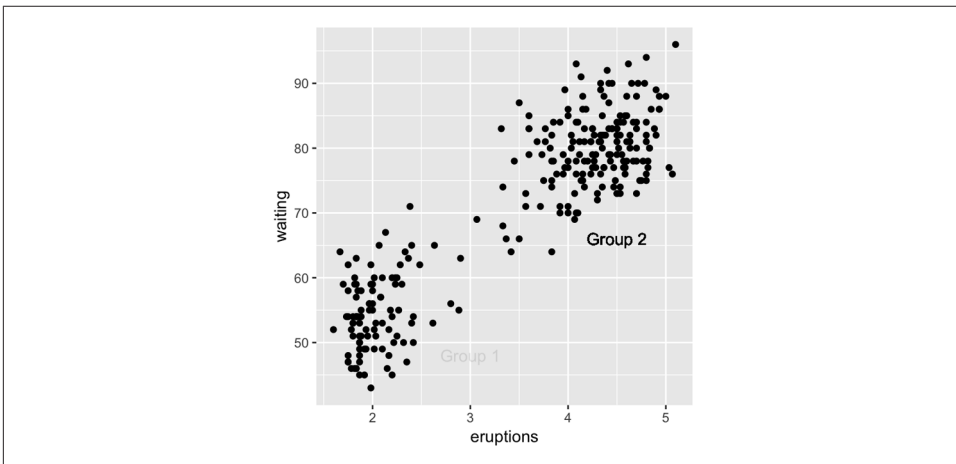


Figure 7-3. Overplotting one of the labels—both should be 90% transparent

In [Figure 7-3](#), each text label is 90% transparent, making it clear which one is overplotted. The overplotting can lead to output with aliased (jagged) edges when outputting to a bitmap.

If the axes are continuous, you can use the special values `Inf` and `-Inf` to place text annotations at the edge of the plotting area, as shown in [Figure 7-4](#). You will also need to adjust the position of the text relative to the corner using `hjust` and `vjust`—if you leave them at their default values, the text will be centered on the edge. It may take a little experimentation with these values to get the text positioned to your liking:

```
p +
  annotate("text", x = -Inf, y = Inf, label = "Upper left", hjust = -.2,
    vjust = 2) +
  annotate("text", x = mean(range(faithful$eruptions)), y = -Inf, vjust = -0.4,
    label = "Bottom middle")
```

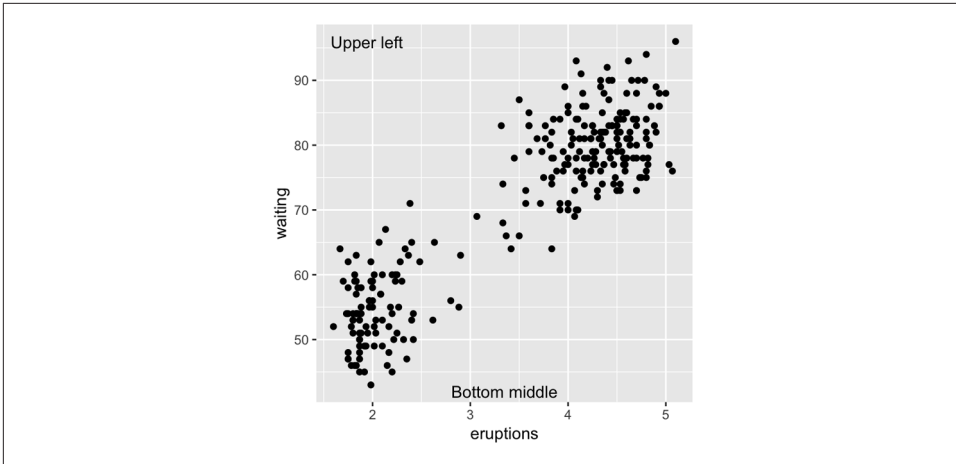


Figure 7-4. Text positioned at the edge of the plotting area

See Also

See [Recipe 5.11](#) for making a scatter plot with text.

For more on controlling the appearance of the text, see [Recipe 9.2](#).

7.2 Using Mathematical Expressions in Annotations

Problem

You want to add a text annotation with mathematical notation.

Solution

Use `annotate(geom = "text")` with `parse = TRUE` ([Figure 7-5](#)):

```
# A normal curve
p <- ggplot(data.frame(x = c(-3,3)), aes(x = x)) +
  stat_function(fun = dnorm)

p +
  annotate("text", x = 2, y = 0.3, parse = TRUE,
    label = "frac(1, sqrt(2 * pi)) * e ^ {-x^2 / 2}")
```

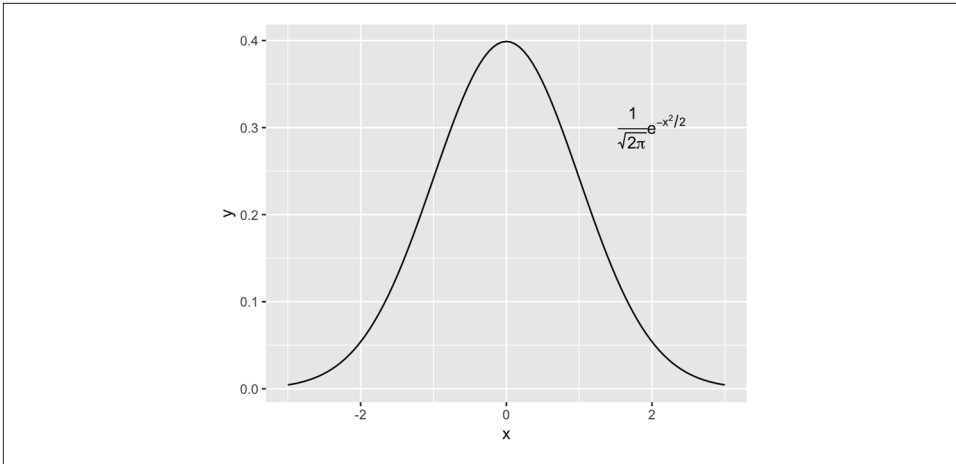


Figure 7-5. Annotation with mathematical expressions

Discussion

Mathematical expressions made with text geoms using `parse = TRUE` in `ggplot2` have a format similar to those made with `plotmath` and `expression` in base R, except that they are stored as strings, rather than as expression objects.

To mix regular text with expressions, use single quotes within double quotes (or vice versa) to mark the plain-text parts. Each block of text enclosed by the inner quotes is treated as a variable in a mathematical expression. Bear in mind that, in R's syntax for mathematical expressions, you can't simply put a variable right next to another without something else in between. To display two variables next to each other, as in [Figure 7-6](#), put a `*` operator between them; when displayed in a graphic, this is treated as an invisible multiplication sign (for a visible multiplication sign, use `%*%`):

```
p +
  annotate("text", x = 0, y = 0.05, parse = TRUE, size = 4,
    label = "'Function: ' * y==frac(1, sqrt(2*pi)) * e^{-x^2/2}")
```

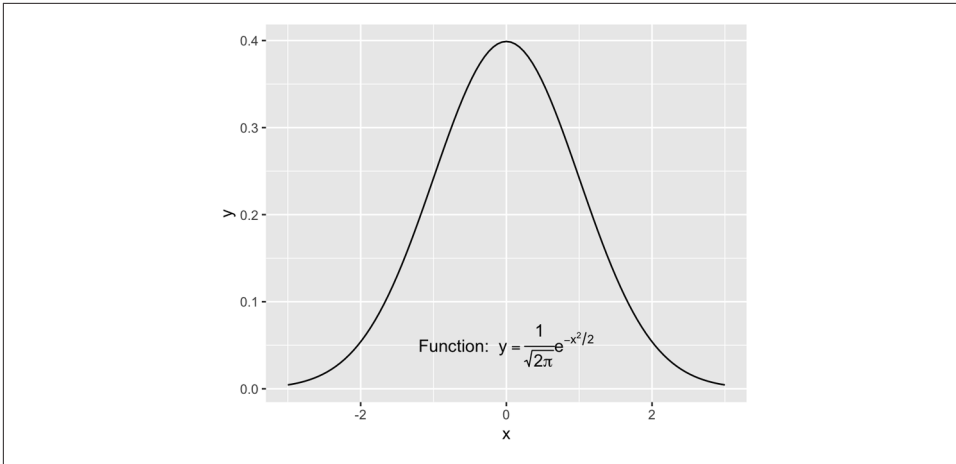


Figure 7-6. Mathematical expression with regular text

See Also

See `?plotmath` for many examples of mathematical expressions, and `?demo(plotmath)` for graphical examples of mathematical expressions.

See [Recipe 5.9](#) for adding regression coefficients to a graph.

For using other fonts in mathematical expressions, see [Recipe 14.6](#).

7.3 Adding Lines

Problem

You want to add lines to a plot.

Solution

For horizontal and vertical lines, use `geom_hline()` and `geom_vline()`, and for angled lines, use `geom_abline()` ([Figure 7-7](#)). For this example, we'll use the height weight data set:

```
library(gcookbook) # Load gcookbook for the heightweight data set

hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn, colour = sex)) +
  geom_point()

# Add horizontal and vertical lines
hw_plot +
  geom_hline(yintercept = 60) +
  geom_vline(xintercept = 14)
```



```
# Add angled line
hw_plot +
  geom_abline(intercept = 37.4, slope = 1.75)
```

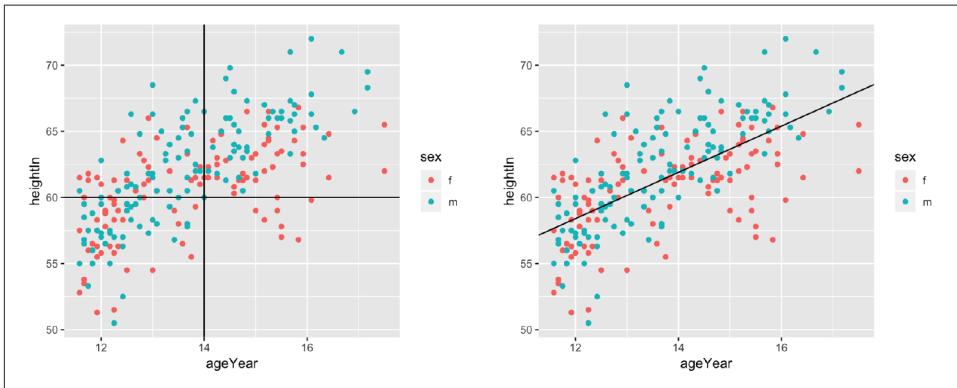


Figure 7-7. Horizontal and vertical lines (left); Angled line (right)

Discussion

The previous examples demonstrate setting the positions of the lines manually, resulting in one line drawn for each geom added. It is also possible to *map* values from the data to `xintercept`, `yintercept`, and so on, and even draw them from another data frame.

Here we'll take the average height for males and females and store it in a data frame, `hw_means`. Then we'll draw a horizontal line for each, and set the `linetype` and `size` (Figure 7-8):

```
library(dplyr)

hw_means <- heightweight %>%
  group_by(sex) %>%
  summarise(heightIn = mean(heightIn))

hw_means
#> # A tibble: 2 x 2
#>   sex   heightIn
#>   <fct>     <dbl>
#> 1 f         60.5
#> 2 m         62.1

hw_plot +
  geom_hline(
    data = hw_means,
    aes(yintercept = heightIn, colour = sex),
    linetype = "dashed",
```

```
size = 1
)
```

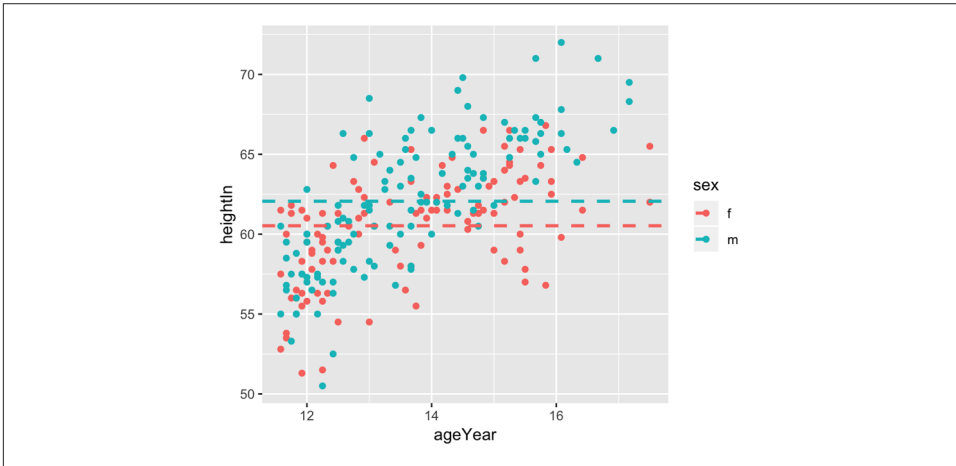


Figure 7-8. Multiple lines, drawn at the mean of each group

If one of the axes is discrete rather than continuous, you can't specify the intercepts as just a character string—they must still be specified as numbers. If the axis represents a factor, the first level has a numeric value of 1, the second level has a value of 2, and so on. You can specify the numerical intercept manually, or calculate the numerical value using `which(levels(...))` (Figure 7-9):

```
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight)) +
  geom_point()

pg_plot +
  geom_vline(xintercept = 2)

pg_plot +
  geom_vline(xintercept = which(levels(PlantGrowth$group) == "ctrl"))
```

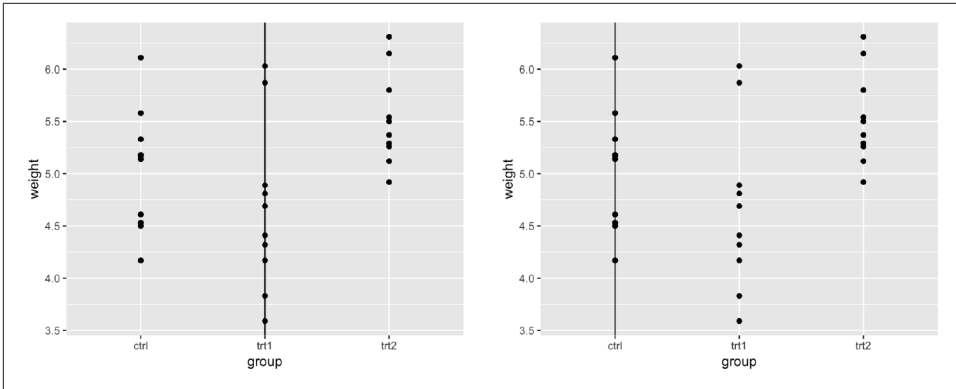


Figure 7-9. Lines with a discrete axis



You may have noticed that adding lines differs from adding other annotations. Instead of using the `annotate()` function, we've used `geom_hline()` and friends. This is because old versions of `ggplot2` didn't have the `annotate()` function. The line geoms had code to handle the special cases where they were used to add a single line, and changing it would break backward compatibility.

See Also

For adding regression lines, see Recipes 5.6 and 5.7.

Lines are often used to indicate summarized information about data. See [Recipe 15.17](#) for more on how to summarize data by groups.

7.4 Adding Line Segments and Arrows

Problem

You want to add line segments or arrows to a plot.

Solution

Use `annotate("segment")`. In this example, we'll use the `climate` data set and use a subset of data from the Berkeley source ([Figure 7-10](#)):

```
library(gcookbook) # Load gcookbook for the climate data set

p <- ggplot(filter(climate, Source == "Berkeley"),
  aes(x = Year, y = Anomaly10y)) +
  geom_line()
```

```
p +
  annotate("segment", x = 1950, xend = 1980, y = -.25, yend = -.25)
```

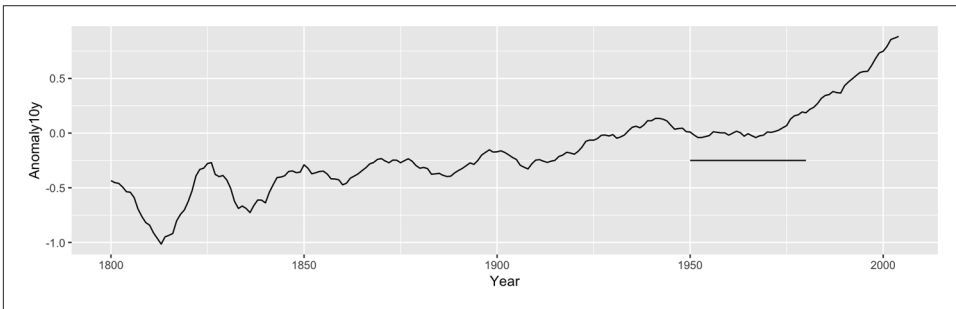


Figure 7-10. Line segment annotation

Discussion

It's possible to add arrowheads or flat ends to the line segments, using `arrow()` from the `grid` package. In this example, we'll do both (Figure 7-11):

```
library(grid)
p +
  annotate("segment", x = 1850, xend = 1820, y = -.8, yend = -.95,
    colour = "blue", size = 2, arrow = arrow()) +
  annotate("segment", x = 1950, xend = 1980, y = -.25, yend = -.25,
    arrow = arrow(ends = "both", angle = 90, length = unit(.2, "cm")))
```

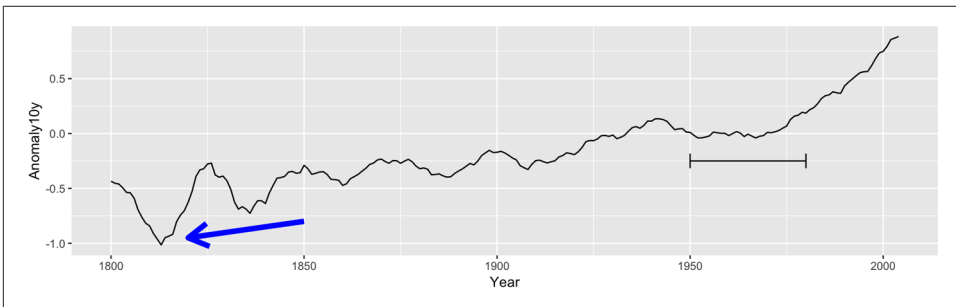


Figure 7-11. Line segments with arrow heads

The default angle is 30, and the default length of the arrowhead lines is 0.2 inches.

If one or both axes are discrete, the x and y positions are such that the categorical items have coordinate values 1, 2, 3, and so on.

See Also

For more information about the parameters for drawing arrows, load the `grid` package and see `?arrow`.

7.5 Adding a Shaded Rectangle

Problem

You want to add a shaded region.

Solution

Use `annotate("rect")` (Figure 7-12):

```
library(gcookbook) # Load gcookbook for the climate data set

p <- ggplot(filter(climate, Source == "Berkeley"),
  aes(x = Year, y = Anomaly10y)) +
  geom_line()

p +
  annotate("rect", xmin = 1950, xmax = 1980, ymin = -1, ymax = 1,
    alpha = .1, fill = "blue")
```

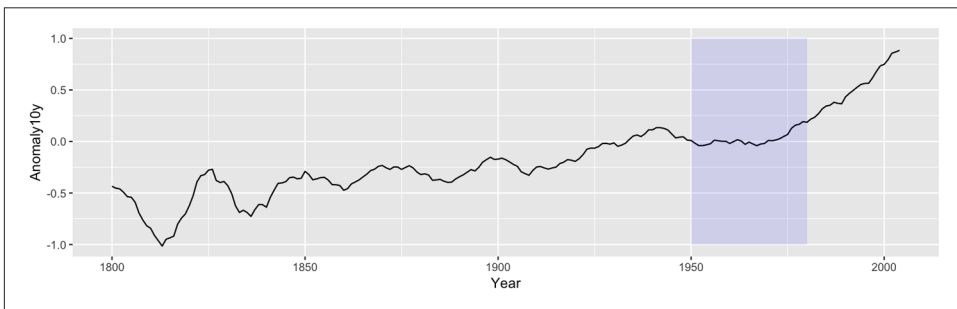


Figure 7-12. A shaded rectangle

Discussion

Each layer is drawn in the order that it's added to the `ggplot` object, so in the preceding example, the rectangle is drawn on top of the line. It's not a problem in that case, but if you'd like to have the line above the rectangle, add the rectangle first, and then the line.

Any geom can be used with `annotate()`, as long as you pass in the proper parameters. In this case, `geom_rect()` requires min and max values for `x` and `y`.

7.6 Highlighting an Item

Problem

You want to change the color of an item to make it stand out.

Solution

To highlight one or more items, create a new column in the data and map it to the color. In this example, we'll create a copy of the `PlantGrowth` data set called `pg_mod` and create a new column, `hl`, which is set to `no` if the case was in the control group or treatment 1 group, and set to `yes` if the case was in the treatment 2 group:

```
library(dplyr)

pg_mod <- PlantGrowth %>%
  mutate(hl = recode(group, "ctrl" = "no", "trt1" = "no", "trt2" = "yes"))
```

Then we'll plot this data with specified colors, and hide the legend (Figure 7-13):

```
ggplot(pg_mod, aes(x = group, y = weight, fill = hl)) +
  geom_boxplot() +
  scale_fill_manual(values = c("grey85", "grey85", "#FFDDCC"), guide = FALSE)
```

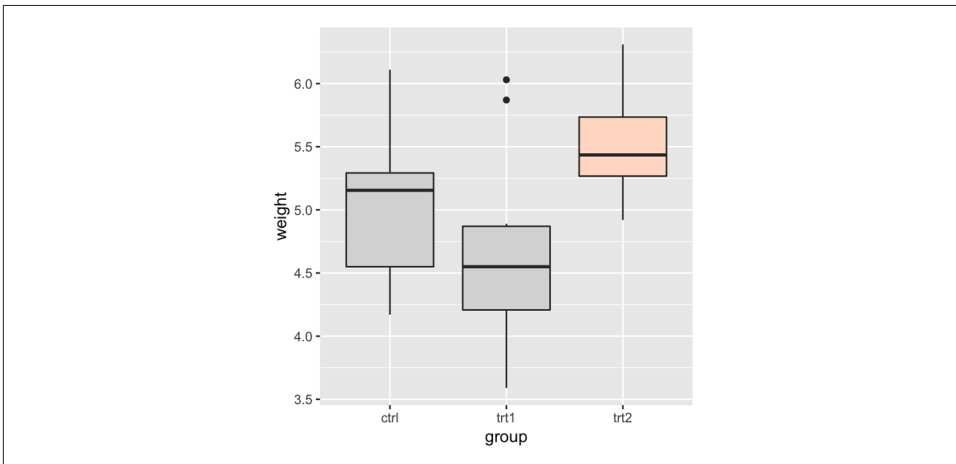


Figure 7-13. Highlighting one item

Discussion

If you have a small number of items, as in this example, instead of creating a new column you could use the original one and specify the colors for every level of that variable. For example, the following code will use the `group` column from `PlantGrowth` and manually set the colors for each of the three levels. The result will appear the same as with the preceding code:

```
ggplot(PlantGrowth, aes(x = group, y = weight, fill = group)) +
  geom_boxplot() +
  scale_fill_manual(values = c("grey85", "grey85", "#FFDDCC"), guide = FALSE)
```

See Also

See [Chapter 12](#) for more information about specifying colors.

For more information about removing the legend, see [Recipe 10.1](#).

7.7 Adding Error Bars

Problem

You want to add error bars to a graph.

Solution

Use `geom_errorbar()` and map variables to the values for `ymin` and `ymax`. Adding the error bars is done the same way for bar graphs and line graphs, as shown in [Figure 7-14](#) (notice that the default `y` range is different for bars and lines, though):

```
library(gcookbook) # Load gcookbook for the cabbage_exp data set
library(dplyr)

# Take a subset of the cabbage_exp data for this example
ce_mod <- cabbage_exp %>%
  filter(Cultivar == "c39")

# With a bar graph
ggplot(ce_mod, aes(x = Date, y = Weight)) +
  geom_col(fill = "white", colour = "black") +
  geom_errorbar(aes(ymin = Weight - se, ymax = Weight + se), width = .2)

# With a line graph
ggplot(ce_mod, aes(x = Date, y = Weight)) +
  geom_line(aes(group = 1)) +
  geom_point(size = 4) +
  geom_errorbar(aes(ymin = Weight - se, ymax = Weight + se), width = .2)
```

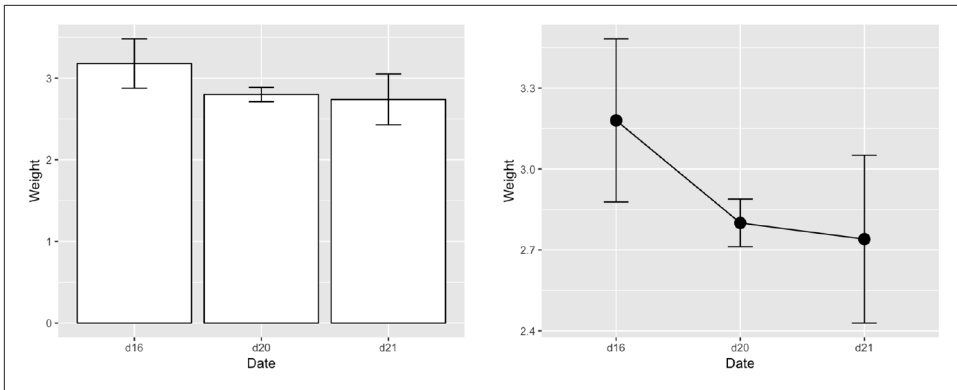


Figure 7-14. Error bars on a bar graph (left); On a line graph (right)

Discussion

In this example, the data already has values for the standard error of the mean (*se*), which we'll use for the error bars (it also has values for the standard deviation, *sd*, but we're not using that here):

```
ce_mod
#>   Cultivar Date Weight      sd    n      se
#> 1    c39  d16   3.18 0.9566144  10 0.30250803
#> 2    c39  d20   2.80 0.2788867  10 0.08819171
#> 3    c39  d21   2.74 0.9834181  10 0.31098410
```

To get the values for *ymax* and *ymin*, we took the *y* variable, *Weight*, and added/subtracted *se*.

We also specified the width of the ends of the error bars, with *width* = .2. It's best to play around with this to find a value that looks good. If you don't set the width, the error bars will be very wide, spanning all the space between items on the *x*-axis.

For a bar graph with groups of bars, the error bars must also be *dodged*; otherwise, they'll have the exact same *x* coordinate and won't line up with the bars. (See [Recipe 3.2](#) for more information about grouped bars and dodging.)

We'll work with the full *cabbage_exp* data set this time:

```
cabbage_exp
#>   Cultivar Date Weight      sd    n      se
#> 1    c39  d16   3.18 0.9566144  10 0.30250803
#> 2    c39  d20   2.80 0.2788867  10 0.08819171
#> 3    c39  d21   2.74 0.9834181  10 0.31098410
#> 4    c52  d16   2.26 0.4452215  10 0.14079141
#> 5    c52  d20   3.11 0.7908505  10 0.25008887
#> 6    c52  d21   1.47 0.2110819  10 0.06674995
```


The default dodge width for `geom_bar()` is 0.9, and you'll have to tell the error bars to be dodged the same width. If you don't specify the dodge width, it will default to dodging by the width of the error bars, which is usually less than the width of the bars (Figure 7-15):

```
# Bad: dodge width not specified
ggplot(cabbage_exp, aes(x = Date, y = Weight, fill = Cultivar)) +
  geom_col(position = "dodge") +
  geom_errorbar(aes(ymin = Weight - se, ymax = Weight + se),
               position = "dodge", width = .2)

# Good: dodge width set to same as bar width (0.9)
ggplot(cabbage_exp, aes(x = Date, y = Weight, fill = Cultivar)) +
  geom_col(position = "dodge") +
  geom_errorbar(aes(ymin = Weight - se, ymax = Weight + se),
               position = position_dodge(0.9), width = .2)
```

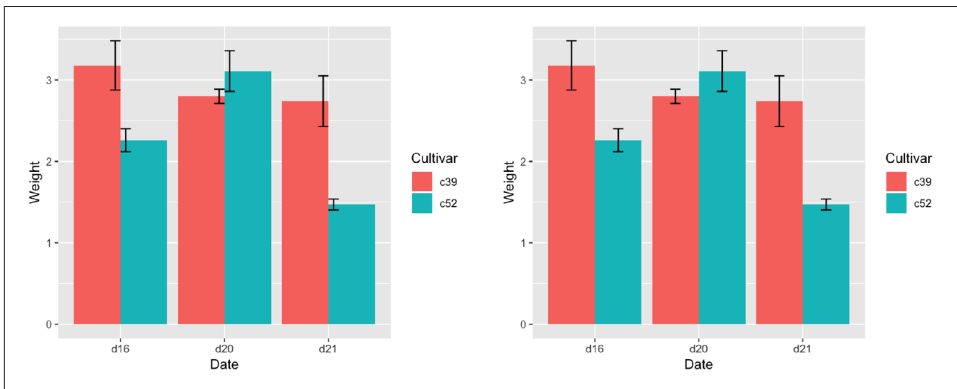


Figure 7-15. Error bars on a grouped bar graph without dodging width specified (left); With dodging width specified (right)



Notice that we used `position = "dodge"`, which is shorthand for `position = position_dodge()`, in the first version. But to pass a specific value, we have to spell it out, as in `position_dodge(0.9)`.

For line graphs, if the error bars are a different color than the lines and points, you should draw the error bars first, so that they are underneath the points and lines. Otherwise the error bars will be drawn on top of the points and lines, which won't look right.

Additionally, you should dodge all the geometric elements so that they will align with the error bars, as shown in Figure 7-16:

```
pd <- position_dodge(.3) # Save the dodge spec because we use it repeatedly

ggplot(cabbage_exp,
  aes(x = Date, y = Weight, colour = Cultivar, group = Cultivar)) +
  geom_errorbar(
    aes(ymin = Weight - se, ymax = Weight + se),
    width = .2,
    size = 0.25,
    colour = "black",
    position = pd
  ) +
  geom_line(position = pd) +
  geom_point(position = pd, size = 2.5)

# Thinner error bar lines with size = 0.25, and larger points with size = 2.5
```

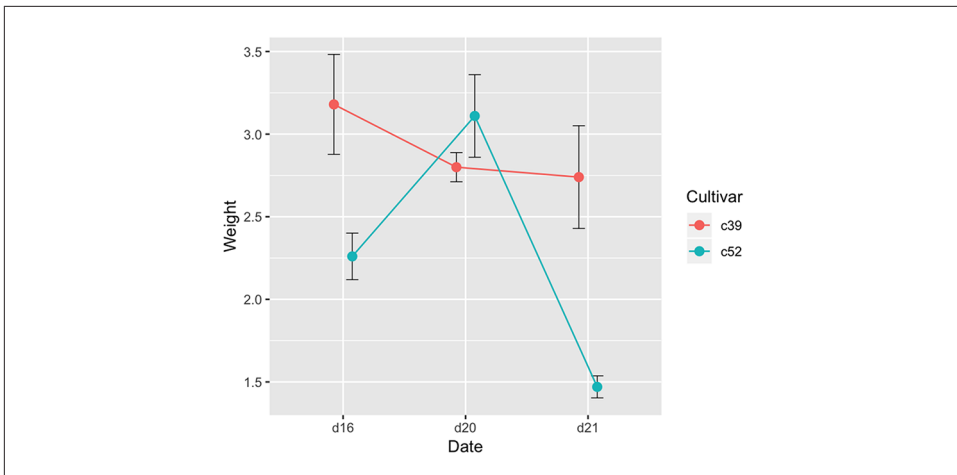


Figure 7-16. Error bars on a line graph, dodged so they don't overlap

Notice that we set `colour = "black"` to make the error bars black; otherwise, they would inherit colour. We also made sure the `Cultivar` was used as a grouping variable by mapping it to `group`.

When a discrete variable is *mapped* to an aesthetic like `colour` or `fill` (as in the case of the bars), that variable is used for grouping the data. But by *setting* the colour of the error bars, we made it so that the variable for colour was not used for grouping, and we needed some other way to inform ggplot that the two data entries at each *x* were in different groups so that they would be dodged.

See Also

See [Recipe 3.2](#) for more about creating grouped bar graphs, and [Recipe 4.3](#) for more about creating line graphs with multiple lines.

See [Recipe 15.18](#) for calculating summaries with means, standard deviations, standard errors, and confidence intervals.

See [Recipe 4.9](#) for adding a confidence region when the data has a higher density along the x-axis.

7.8 Adding Annotations to Individual Facets

Problem

You want to add annotations to each facet in a plot.

Solution

Create a new data frame with the faceting variable(s), and a value to use in each facet. Then use `geom_text()` with the new data frame ([Figure 7-17](#)):

```
# Create the base plot
mpg_plot <- ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  facet_grid(. ~ drv)

# A data frame with labels for each facet
f_labels <- data.frame(drv = c("4", "f", "r"), label = c("4wd", "Front", "Rear"))

mpg_plot +
  geom_text(x = 6, y = 40, aes(label = label), data = f_labels)

# If you use annotate(), the label will appear in all facets
mpg_plot +
  annotate("text", x = 6, y = 42, label = "label text")
```

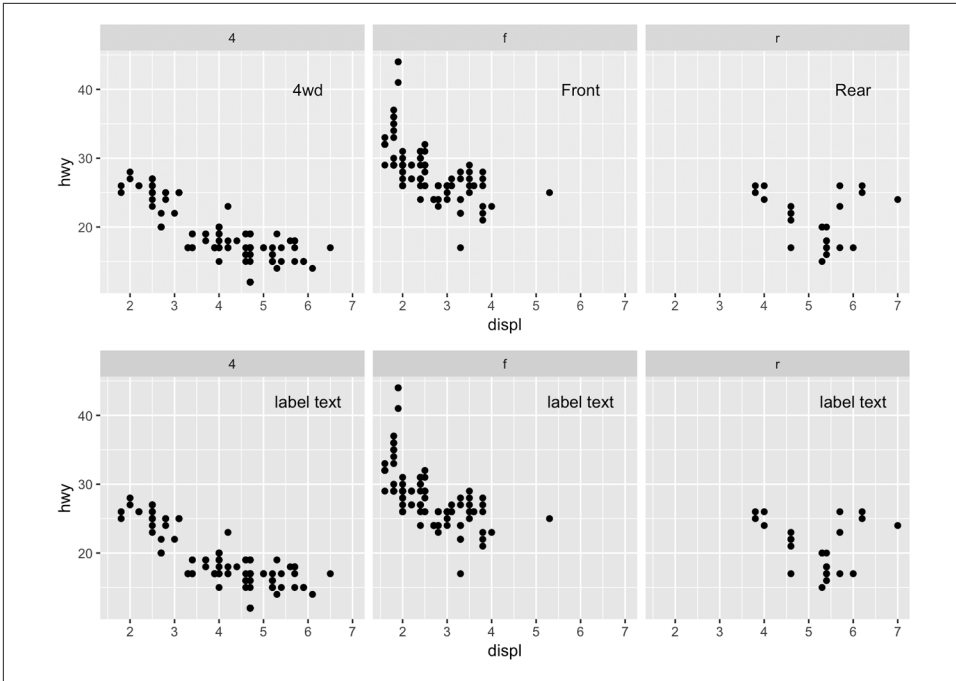


Figure 7-17. Top: different annotations in each facet; Bottom: the same annotation in each facet

Discussion

This method can be used to display information about the data in each facet, as shown in Figure 7-18. For example, in each facet we can show linear regression lines, the formula for each line, and the r^2 value. To do this, we'll write a function that takes a data frame and returns another data frame containing a string for a regression equation, and a string for the r^2 value. Then we'll use dplyr's `do()` function to apply that function to each group of the data:

```
# This function returns a data frame with strings representing the regression
# equation, and the r^2 value.
# These strings will be treated as R math expressions
lm_labels <- function(dat) {
  mod <- lm(hwy ~ displ, data = dat)
  formula <- sprintf("italic(y) == %.2f %+.2f * italic(x)",
                     round(coef(mod)[1], 2), round(coef(mod)[2], 2))
  r <- cor(dat$displ, dat$hwy)
  r2 <- sprintf("italic(R^2) == %.2f", r^2)
  data.frame(formula = formula, r2 = r2, stringsAsFactors = FALSE)
}

library(dplyr)
```

```

labels <- mpg %>%
  group_by(drv) %>%
  do(lm_labels(.))

labels
#> # A tibble: 3 x 3
#> # Groups:   drv [3]
#>   drv formula                                r2
#>   <chr> <chr>                                <chr>
#> 1 4      italic(y) == 30.68 - 2.88 * italic(x) italic(R^2) == 0.65
#> 2 f      italic(y) == 37.38 - 3.60 * italic(x) italic(R^2) == 0.36
#> 3 r      italic(y) == 25.78 - 0.92 * italic(x) italic(R^2) == 0.04

# Plot with formula and R^2 values
mpg_plot +
  geom_smooth(method = lm, se = FALSE) +
  geom_text(data = labels, aes(label = formula), x = 3, y = 40, parse = TRUE,
    hjust = 0) +
  geom_text(x = 3, y = 35, aes(label = r2), data = labels, parse = TRUE,
    hjust = 0)

```

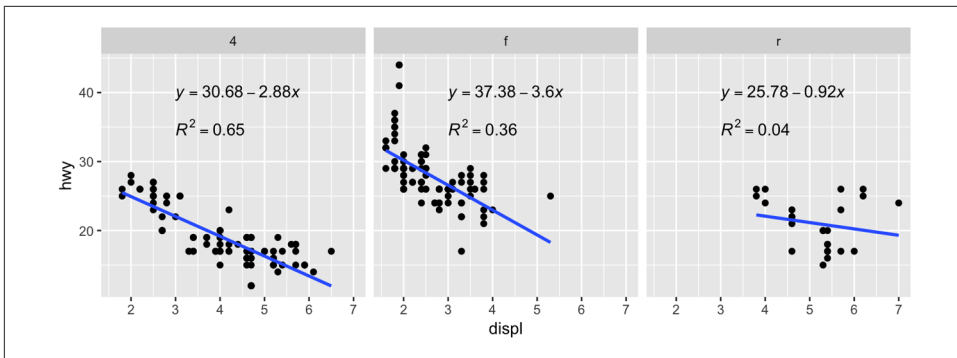


Figure 7-18. Annotations in each facet with information about the data

We needed to write our own function here because generating the linear model and extracting the coefficients requires operating on each subset data frame directly. If you just want to display the r^2 values, it's possible to do something simpler, by using `group_by()` and `summarise()` and passing additional arguments to `summarise()`:

```

# Find r^2 values for each group
labels <- mpg %>%
  group_by(drv) %>%
  summarise(r2 = cor(displ, hwy)^2)

labels$r2 <- sprintf("italic(R^2) == %.2f", labels$r2)
labels
#> # A tibble: 3 x 2
#>   drv    r2
#>   <chr> <chr>

```

```
#> 1 4      italic(R^2) == 0.65  
#> 2 f      italic(R^2) == 0.36  
#> 3 r      italic(R^2) == 0.04
```

Text geoms aren't the only kind that can be added individually for each facet. Any geom can be used, as long as the input data is structured correctly.

See Also

See [Recipe 7.2](#) for more about using math expressions in plots.

If you want to make prediction lines from your own model objects, instead of having `ggplot2` do it for you with `stat_smooth()`, see [Recipe 5.8](#).

The x- and y-axes provide context for interpreting the displayed data. ggplot will display the axes with defaults that look good in most cases, but you might want to control, for example, the axis labels, the number and placement of tick marks, the tick mark labels, and so on. In this chapter, I'll cover how to fine-tune the appearance of the axes.

8.1 Swapping X- and Y-Axes

Problem

You want to swap the x- and y-axes on a graph.

Solution

Use `coord_flip()` to flip the axes (Figure 8-1):

```
ggplot(PlantGrowth, aes(x = group, y = weight)) +  
  geom_boxplot()  
  
ggplot(PlantGrowth, aes(x = group, y = weight)) +  
  geom_boxplot() +  
  coord_flip()
```

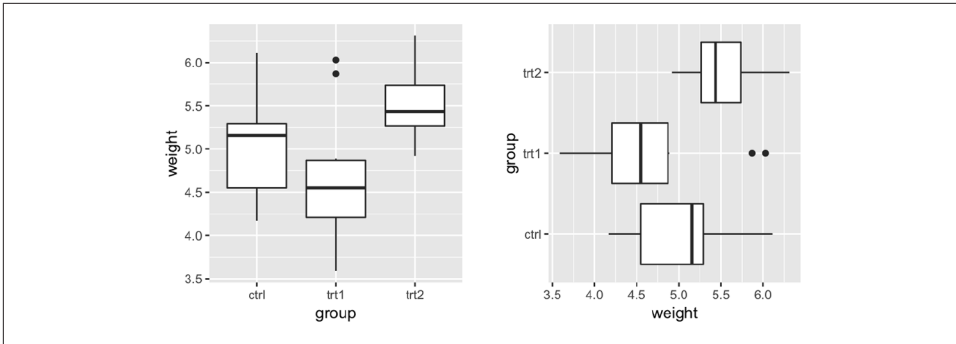


Figure 8-1. A box plot with regular axes (left); With swapped axes (right)

Discussion

For a scatter plot, it is trivial to change what goes on the vertical axis and what goes on the horizontal axis: just exchange the variables mapped to *x* and *y*. But not all the geoms in ggplot treat the *x*- and *y*-axes equally. For example, box plots summarize the data along the *y*-axis, the lines in line graphs move in only one direction along the *x*-axis, error bars have a single *x* value and a range of *y* values, and so on. If you're using these geoms and want them to behave as though the axes are swapped, `coord_flip()` is what you need.

Sometimes when the axes are swapped, the order of items will be the reverse of what you want. On a graph with standard *x*- and *y*-axes, the *x* items start at the left and go to the right, which corresponds to the normal way of reading, from left to right. When you swap the axes, the items still go from the origin outward, which in this case will be from bottom to top—but this conflicts with the normal way of reading, from top to bottom. Sometimes this is a problem, and sometimes it isn't. If the *x* variable is a factor, the order can be reversed by using `scale_x_discrete()` with `limits = rev(levels(...))`, as in [Figure 8-2](#):

```
ggplot(PlantGrowth, aes(x = group, y = weight)) +
  geom_boxplot() +
  coord_flip() +
  scale_x_discrete(limits = rev(levels(PlantGrowth$group)))
```

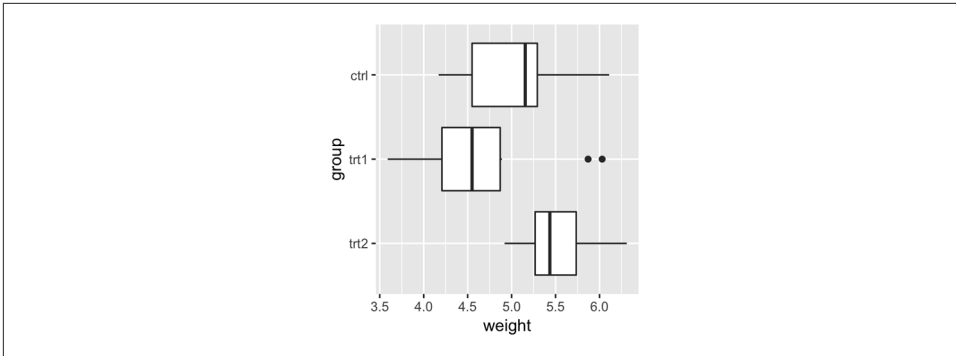



Figure 8-2. A box plot with swapped axes and x-axis order reversed

See Also

If the variable is continuous, see [Recipe 8.3](#) to reverse the direction.

8.2 Setting the Range of a Continuous Axis

Problem

You want to set the range (or limits) of an axis.

Solution

You can use `xlim()` or `ylim()` to set the minimum and maximum values of a continuous axis. [Figure 8-3](#) shows one graph with the default *y* limits, and one with manually set *y* limits:

```
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight)) +
  geom_boxplot()

# Display the basic graph
pg_plot

pg_plot +
  ylim(0, max(PlantGrowth$weight))
```

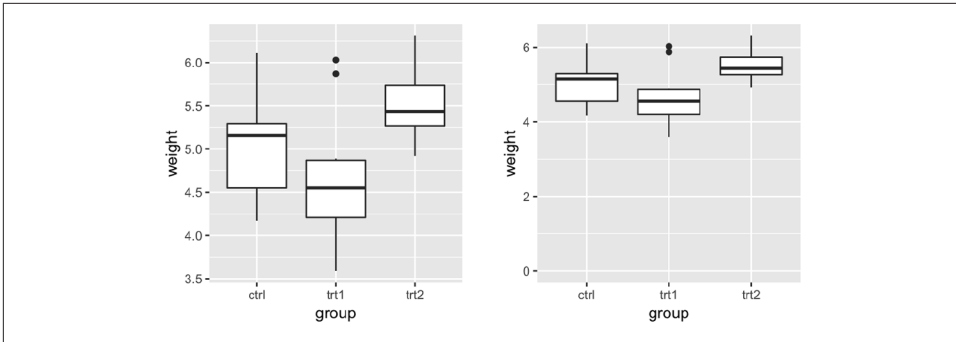


Figure 8-3. Box plot with default range (left); With manually set range (right)

The latter example sets the y range from 0 to the maximum value of the `weight` column, though a constant value (like 10) could instead be used as the maximum.

Discussion

`ylim()` is shorthand for setting the limits with `scale_y_continuous()`. (The same is true for `xlim()` and `scale_x_continuous()`.) The following are equivalent:

```
ylim(0, 10)
scale_y_continuous(limits = c(0, 10))
```

Sometimes you will need to set other properties of `scale_y_continuous()`, and in these cases using `xlim()` and `scale_y_continuous()` together may result in some unexpected behavior, because only the first of the directives will have an effect. In these two examples, `ylim(0, 10)` should set the y range from 0 to 10, and `scale_y_continuous(breaks=c(0, 5, 10))` should put tick marks at 0, 5, and 10. However, in both cases, only the second directive has any effect:

```
pg_plot +
  ylim(0, 10) +
  scale_y_continuous(breaks = NULL)

pg_plot +
  scale_y_continuous(breaks = NULL) +
  ylim(0, 10)
```

To make both changes work, get rid of `ylim()` and set both limits and breaks in `scale_y_continuous()`:

```
pg_plot +
  scale_y_continuous(limits = c(0, 10), breaks = NULL)
```

In `ggplot`, there are two ways of setting the range of the axes. The first way is to modify the *scale*, and the second is to apply a *coordinate transform*. When you modify the limits of the x or y scale, any data outside of the limits is removed—that is, the out-of-

range data is not only not displayed, it is removed from consideration entirely. (It will also print a warning when this happens.)

With the box plots in these examples, if you restrict the y range so that some of the original data is clipped, the box plot statistics will be computed based on clipped data, and the shape of the box plots will change.

With a coordinate transform, the data is not clipped; in essence, it zooms in or out to the specified range. **Figure 8-4** shows the difference between the two methods:

```
pg_plot +  
  scale_y_continuous(limits = c(5, 6.5)) # Same as using ylim()  
#> Warning: Removed 13 rows containing non-finite values (stat_boxplot).  
  
pg_plot +  
  coord_cartesian(ylim = c(5, 6.5))
```

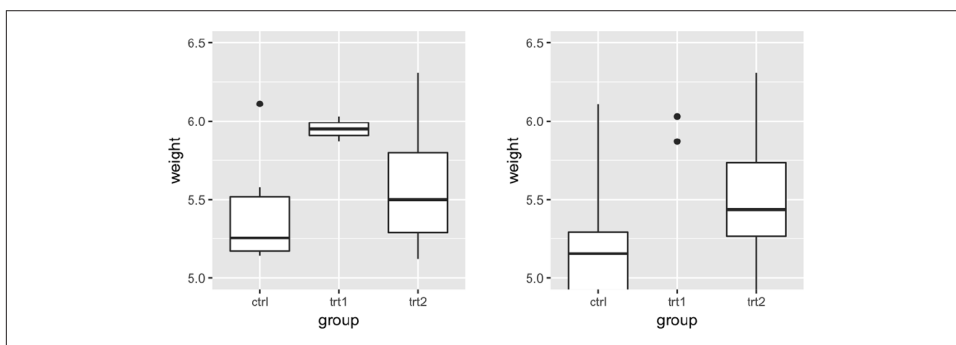


Figure 8-4. Smaller y range using a scale (data has been dropped, so the box plots have changed shape, left); “Zooming in” using a coordinate transform (right)

Finally, it’s also possible to *expand* the range in one direction, using `expand_limits()` (**Figure 8-5**). You can’t use this to shrink the range, however:

```
pg_plot +  
  expand_limits(y = 0)
```

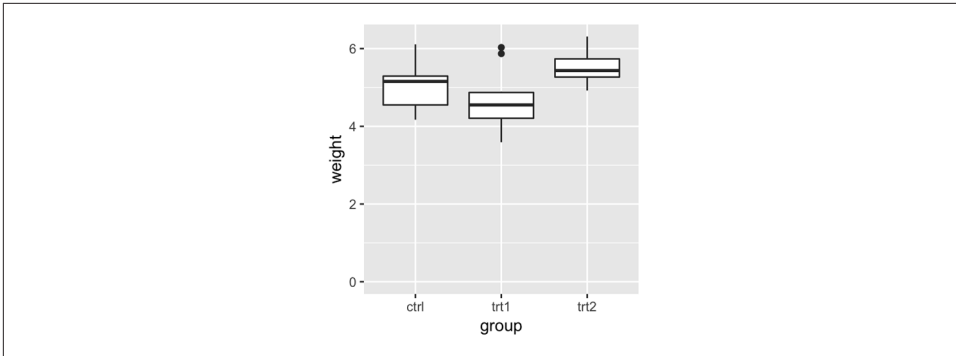


Figure 8-5. Box plot on which y range has been expanded to include 0

8.3 Reversing a Continuous Axis

Problem

You want to reverse the direction of a continuous axis.

Solution

Use `scale_y_reverse()` or `scale_x_reverse()` (Figure 8-6). The direction of an axis can also be reversed by specifying the limits in reversed order, with the maximum first, then the minimum:

```
ggplot(PlantGrowth, aes(x = group, y = weight)) +
  geom_boxplot() +
  scale_y_reverse()

# Similar effect by specifying limits in reversed order
ggplot(PlantGrowth, aes(x = group, y = weight)) +
  geom_boxplot() +
  ylim(6.5, 3.5)
```

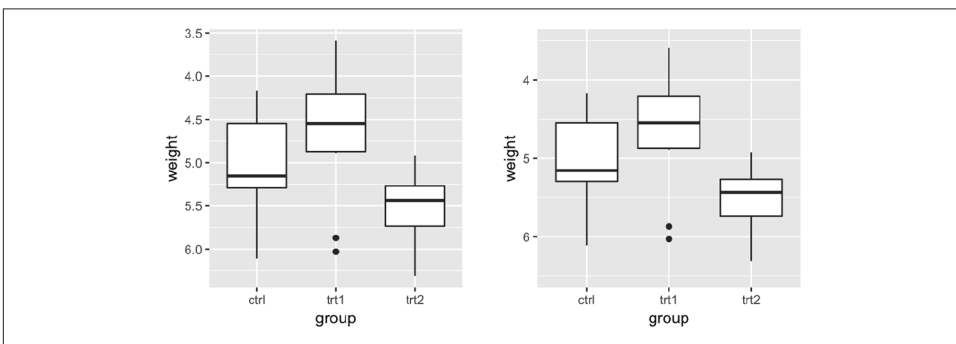


Figure 8-6. Box plot with reversed y-axis

Discussion

Like `scale_y_continuous()`, `scale_y_reverse()` does not work with `ylim()`. (The same is true for the x-axis properties.) If you want to reverse an axis *and* set its range, you must do it within the `scale_y_reverse()` statement, by setting the limits in reversed order (Figure 8-7):

```
ggplot(PlantGrowth, aes(x = group, y = weight)) +  
  geom_boxplot() +  
  scale_y_reverse(limits = c(8, 0))
```

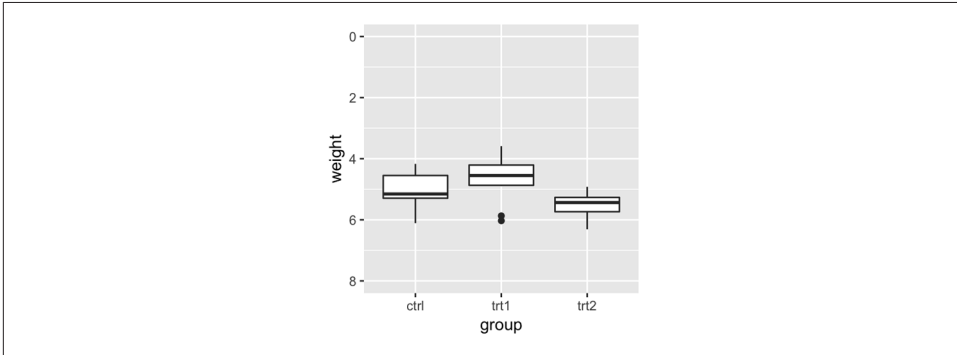


Figure 8-7. Box plot with reversed y-axis with manually set limits

See Also

To reverse the order of items on a *discrete* axis, see [Recipe 8.4](#).

8.4 Changing the Order of Items on a Categorical Axis

Problem

You want to change the order of items on a categorical axis.

Solution

For a categorical (or discrete) axis—one with a factor mapped to it—the order of items can be changed by setting limits in `scale_x_discrete()` or `scale_y_discrete()`.

To manually set the order of items on the axis, specify limits with a vector of the levels in the desired order. You can also omit items with this vector, as shown in [Figure 8-8](#), left:

```
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight)) +  
  geom_boxplot()
```

```
pg_plot +
  scale_x_discrete(limits = c("trt1", "ctrl", "trt2"))
```

Discussion

You can also use this method to display a subset of the items on the axis. This will show only `ctrl` and `trt1` (Figure 8-8, right). Note that because data is removed, it will emit a warning when you do this:

```
pg_plot +
  scale_x_discrete(limits = c("ctrl", "trt1"))
#> Warning: Removed 10 rows containing missing values (stat_boxplot).
```

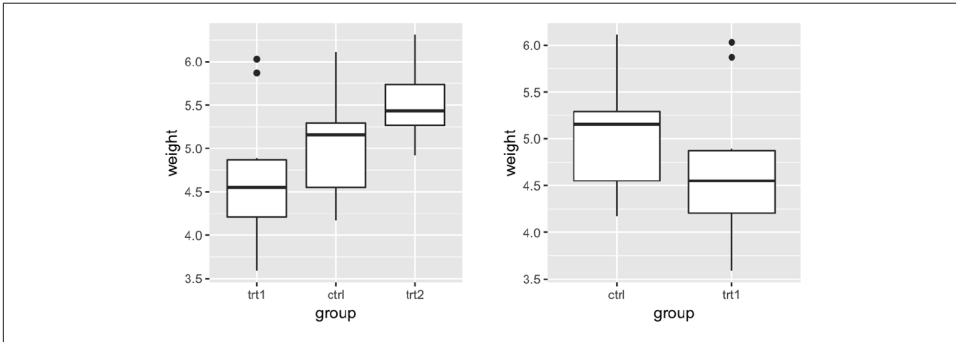


Figure 8-8. Box plot with manually specified items on the x-axis (left); With only two items (right)

To reverse the order, set `limits = rev(levels(...))`, and put the factor inside. This will reverse the order of the `PlantGrowth$group` factor, as shown in Figure 8-9:

```
pg_plot +
  scale_x_discrete(limits = rev(levels(PlantGrowth$group)))
```

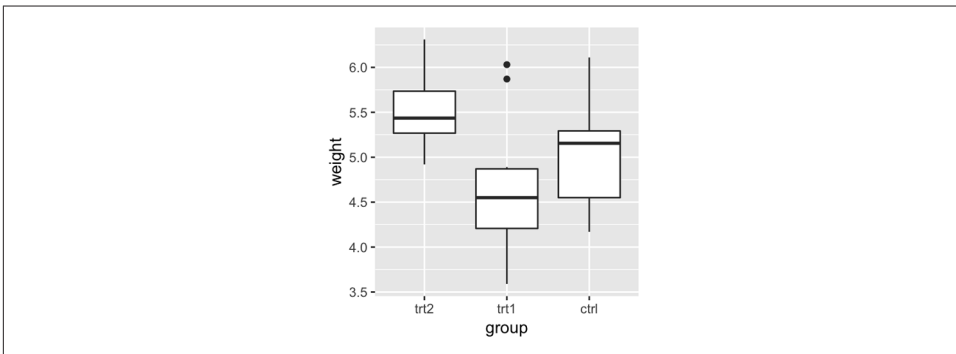


Figure 8-9. Box plot with order reversed on the x-axis

See Also

To reorder factor levels based on data values from another column, see [Recipe 15.9](#).

8.5 Setting the Scaling Ratio of the X- and Y-Axes

Problem

You want to set the ratio at which the x- and y-axes are scaled.

Solution

Use `coord_fixed()`. This will result in a 1:1 scaling between the x- and y-axes, as shown in [Figure 8-10](#):

```
library(gcookbook) # Load gcookbook for the marathon data set

m_plot <- ggplot(marathon, aes(x = Half, y = Full)) +
  geom_point()

m_plot +
  coord_fixed()
```

Discussion

The marathon data set contains runners' marathon and half-marathon times. In this case it might be useful to force the x- and y-axes to have the same scaling.

It's also helpful to set the tick spacing to be the same, by setting breaks in `scale_y_continuous()` and `scale_x_continuous()` (also in [Figure 8-10](#)):

```
m_plot +
  coord_fixed() +
  scale_y_continuous(breaks = seq(0, 420, 30)) +
  scale_x_continuous(breaks = seq(0, 420, 30))
```

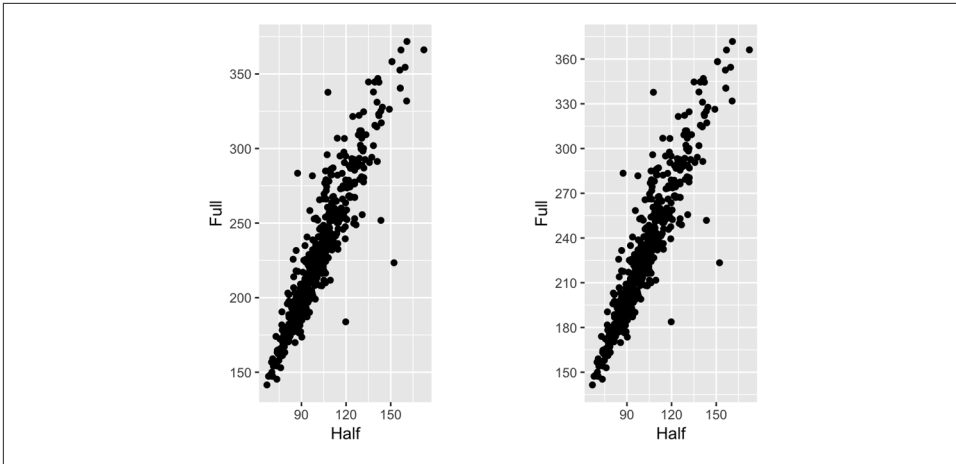


Figure 8-10. Scatter plot with equal scaling of axes (left); With tick marks at specified positions (right)

If, instead of an equal ratio, you want some other fixed ratio between the axes, set the `ratio` parameter. With the marathon data, we might want the axis with half-marathon times stretched out to twice that of the axis with the marathon times (Figure 8-11). We'll also add tick marks twice as often on the x-axis:

```
m_plot +
  coord_fixed(ratio = 1/2) +
  scale_y_continuous(breaks = seq(0, 420, 30)) +
  scale_x_continuous(breaks = seq(0, 420, 15))
```

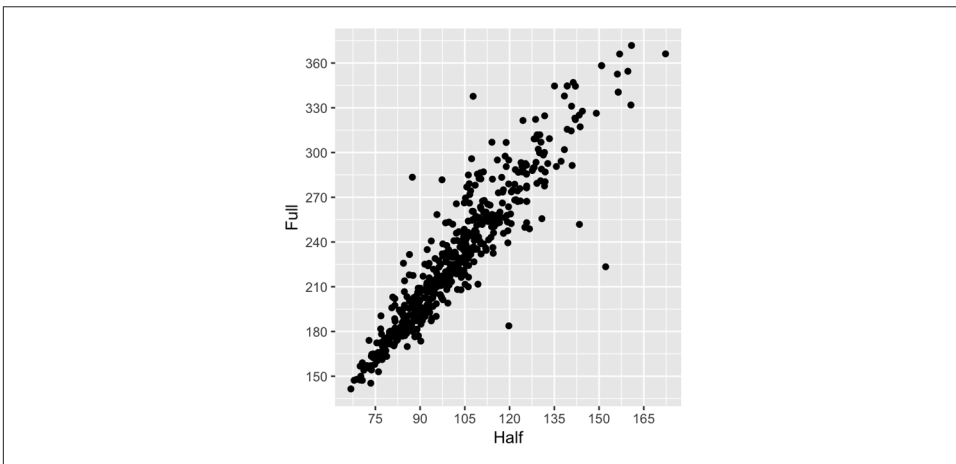


Figure 8-11. Scatter plot with a 1/2 scaling ratio for the axes

8.6 Setting the Positions of Tick Marks

Problem

You want to set where the tick marks appear on the axis.

Solution

Usually ggplot does a good job of deciding where to put the tick marks, but if you want to change them, set breaks in the scale (Figure 8-12):

```
ggplot(PlantGrowth, aes(x = group, y = weight)) +  
  geom_boxplot()  
  
ggplot(PlantGrowth, aes(x = group, y = weight)) +  
  geom_boxplot() +  
  scale_y_continuous(breaks = c(4, 4.25, 4.5, 5, 6, 8))
```

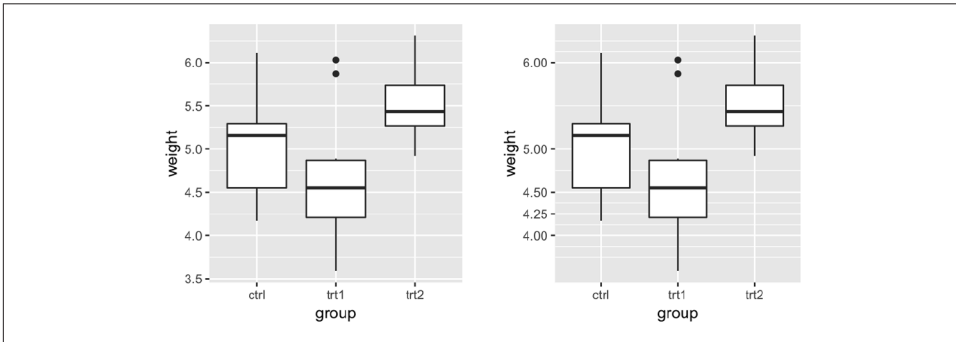


Figure 8-12. Box plot with automatic tick marks (left); With manually set tick marks (right)

Discussion

The location of the tick marks defines where *major* grid lines are drawn. If the axis represents a continuous variable, *minor* grid lines, which are fainter and unlabeled, will by default be drawn halfway between each major grid line.

You can also use the `seq()` function or the `:` operator to generate vectors for tick marks:

```
seq(4, 7, by = .5)  
#> [1] 4.0 4.5 5.0 5.5 6.0 6.5 7.0  
5:10  
#> [1] 5 6 7 8 9 10
```

If the axis is discrete instead of continuous, then there is by default a tick mark for each item. For discrete axes, you can change the order of items or remove them by

specifying the limits (see [Recipe 8.4](#)). Setting breaks will change which of the levels are labeled, but will not remove them or change their order. [Figure 8-13](#) shows what happens when you set limits and breaks (the warning is because we're using only two of the three levels for group and therefore are dropping some rows):

```
# Set both breaks and labels for a discrete axis
ggplot(PlantGrowth, aes(x = group, y = weight)) +
  geom_boxplot() +
  scale_x_discrete(limits = c("trt2", "ctrl"), breaks = "ctrl")
#> Warning: Removed 10 rows containing missing values (stat_boxplot).
```

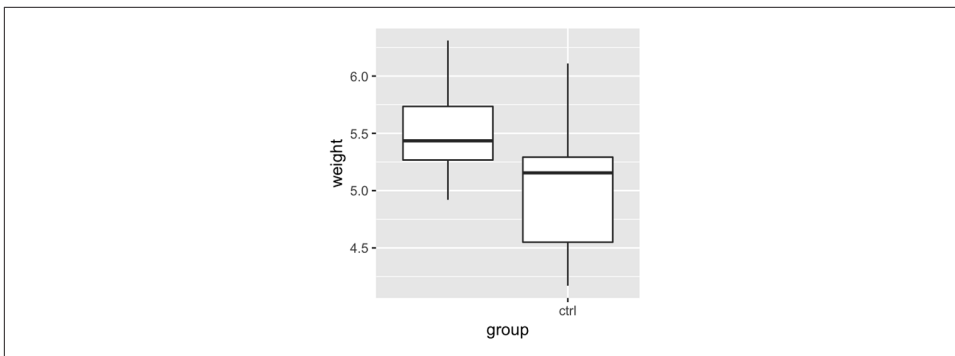


Figure 8-13. For a discrete axis, setting limits reorders and removes items, and setting breaks controls which items have labels

See Also

To remove the tick marks and labels (but not the data) from the graph, see [Recipe 8.7](#).

8.7 Removing Tick Marks and Labels

Problem

You want to remove tick marks and labels.

Solution

To remove just the tick labels, as in [Figure 8-14](#) (left), use `theme(axis.text.y = element_blank())` (or do the same for `axis.text.x`). This will work for both continuous and categorical axes:

```
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight)) +
  geom_boxplot()

pg_plot +
  theme(axis.text.y = element_blank())
```

To remove the tick marks, use `theme(axis.ticks=element_blank())`. This will remove the tick marks on both axes. (It's not possible to hide the tick marks on just one axis.) In this example, we'll hide all tick marks as well as the *y* tick labels (Figure 8-14, center):

```
pg_plot +  
  theme(axis.ticks = element_blank(), axis.text.y = element_blank())
```

To remove the tick marks, the labels, and the grid lines, set `breaks` to `NULL` (Figure 8-14, right):

```
pg_plot +  
  scale_y_continuous(breaks = NULL)
```

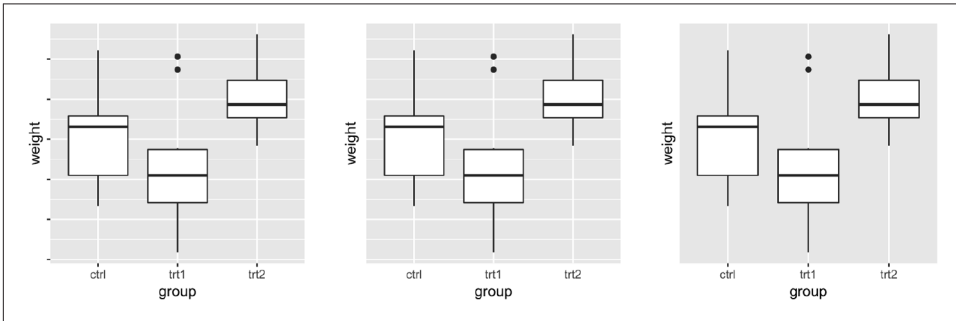


Figure 8-14. No tick labels on *y*-axis (left); No tick marks and no tick labels on *y*-axis (middle); With `breaks=NULL` (right)

This will work for continuous axes only; if you remove items from a categorical axis using limits, as in [Recipe 8.4](#), the data with that value won't be shown at all.

Discussion

There are actually three related items that can be controlled: tick labels, tick marks, and the grid lines. For continuous axes, `ggplot()` normally places a tick label, tick mark, and major grid line at each value of `breaks`. For categorical axes, these things go at each value of limits.

The tick labels on each axis can be controlled independently. However, the tick marks and grid lines must be controlled all together.

8.8 Changing the Text of Tick Labels

Problem

You want to change the text of tick labels.

Solution

Consider the scatter plot in **Figure 8-15**, where height is reported in inches:

```
library(gcookbook) # Load gcookbook for the heightweight data set

hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn)) +
  geom_point()

hw_plot
```

To set arbitrary labels, as in **Figure 8-15** (right), pass values to `breaks` and `labels` in the scale. One of the labels has a newline (`\n`) character, which tells ggplot to put a line break there:

```
hw_plot +
  scale_y_continuous(
    breaks = c(50, 56, 60, 66, 72),
    labels = c("Tiny", "Really\nshort", "Short", "Medium", "Tallish")
  )
```

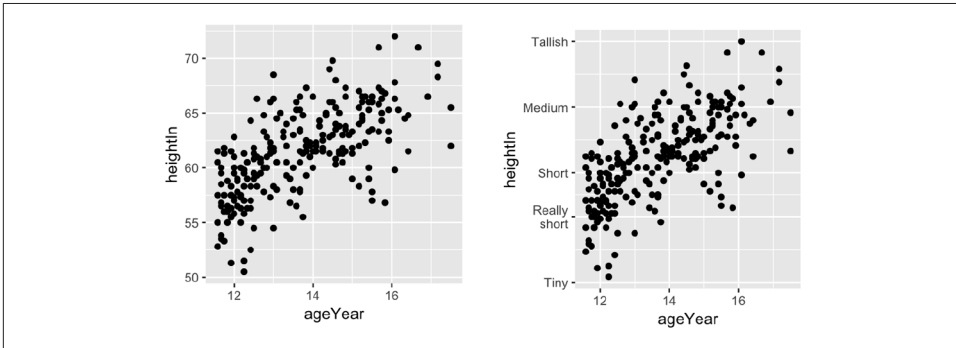


Figure 8-15. Scatter plot with automatic tick labels (left); With manually specified labels on the y-axis (right)

Discussion

Instead of setting completely arbitrary labels, it is more common to have your data stored in one format, while wanting the labels to be displayed in another. We might, for example, want heights to be displayed in feet and inches (like 5'6") instead of just inches. To do this, we can define a *formatter* function, which takes in a value and returns the corresponding string. For example, this function will convert inches to feet and inches:

```
footinch_formatter <- function(x) {
  foot <- floor(x/12)
  inch <- x %% 12
  return(paste(foot, "'", inch, "\"", sep = ""))
}
```

Here's what it returns for values 56–64 (the backslashes are there as escape characters, to distinguish the quotes *in* a string from the quotes that *delimit* a string):

```
footinch_formatter(56:64)
#> [1] "4'8\"" "4'9\"" "4'10\"" "4'11\"" "5'0\"" "5'1\"" "5'2\"" "5'3\""
#> [9] "5'4\""
```

Now we can pass our function to the scale, using the `labels` parameter (Figure 8-16, left):

```
hw_plot +
  scale_y_continuous(labels = footinch_formatter)
```

Here, the automatic tick marks were placed every five inches, but that looks a little off for this data. We can instead have ggplot set tick marks every four inches, by specifying breaks (Figure 8-16, right):

```
hw_plot +
  scale_y_continuous(breaks = seq(48, 72, 4), labels = footinch_formatter)
```

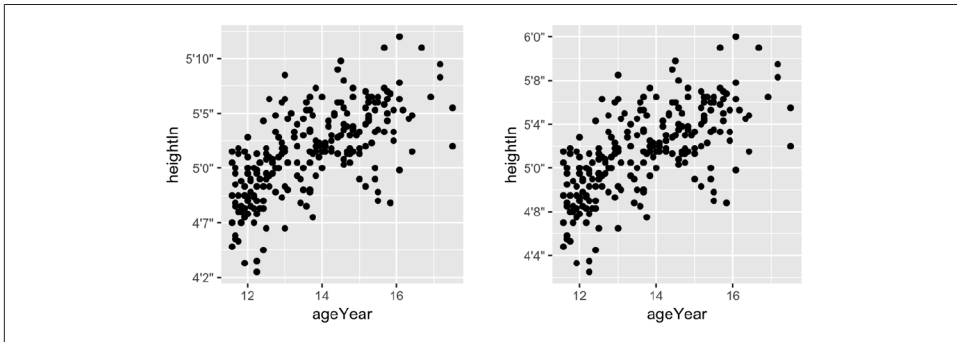


Figure 8-16. Scatter plot with a formatter function (left); With manually specified breaks on the y-axis (right)

Another common task is to convert time measurements to HH:MM:SS format, or something similar. This function will take numeric minutes and convert them to this format, rounding to the nearest second (it can be customized for your particular needs):

```
timeHMS_formatter <- function(x) {
  h <- floor(x/60)
  m <- floor(x %% 60)
  s <- round(60*(x %% 1))
  lab <- sprintf("%02d:%02d:%02d", h, m, s) # Round to nearest second
  lab <- gsub("^00:", "", lab) # Format the strings as HH:MM:SS
  lab <- gsub("^0", "", lab) # Remove leading 00: if present
  return(lab) # Remove leading 0 if present
}
```

Running it on some sample numbers yields:

```
timeHMS_formatter(c(.33, 50, 51.25, 59.32, 60, 60.1, 130.23))
#> [1] "0:20"      "50:00"      "51:15"      "59:19"      "1:00:00"    "1:00:06"    "2:10:14"
```

The scales package, which is installed with ggplot2, comes with some built-in formatting functions:

- `comma()` adds commas to numbers, in the thousand, million, billion, etc., places.
- `dollar()` adds a dollar sign and rounds to the nearest cent.
- `percent()` multiplies by 100, rounds to the nearest integer, and adds a percent sign.
- `scientific()` gives numbers in scientific notation, like `3.30e+05`, for large and small numbers.

If you want to use these functions, you must first load the scales package, with `library(scales)`.

8.9 Changing the Appearance of Tick Labels

Problem

You want to change the appearance of tick labels.

Solution

In [Figure 8-17](#) (left), we've manually set the labels to be long—long enough that they overlap:

```
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight)) +
  geom_boxplot() +
  scale_x_discrete(
    breaks = c("ctrl", "trt1", "trt2"),
    labels = c("Control", "Treatment 1", "Treatment 2")
  )

pg_plot
```

To rotate the text 90 degrees counterclockwise ([Figure 8-17](#), middle), use:

```
pg_plot +
  theme(axis.text.x = element_text(angle = 90, hjust = 1, vjust = .5))
```

Rotating the text 30 degrees ([Figure 8-17](#), right) uses less vertical space and makes the labels easier to read without tilting your head:

```
pg_plot +
  theme(axis.text.x = element_text(angle = 30, hjust = 1, vjust = 1))
```

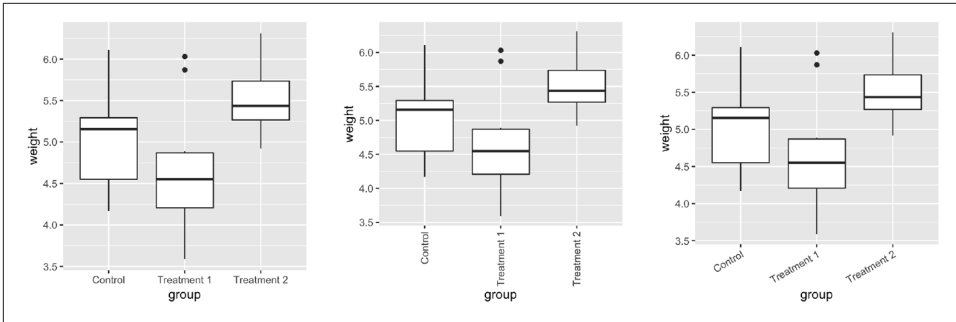


Figure 8-17. X-axis tick labels rotated 0 (left), 90 (middle), and 30 degrees (right)

The `hjust` and `vjust` settings specify the horizontal alignment (left/center/right) and vertical alignment (top/middle/bottom).

Discussion

Besides rotation, other text properties such as size, style (bold/italic/normal), and the font family (such as Times or Helvetica) can be set with `element_text()`, as shown in Figure 8-18:

```
pg_plot +
  theme(
    axis.text.x = element_text(family = "Times", face = "italic",
                               colour = "darkred", size = rel(0.9))
  )
```

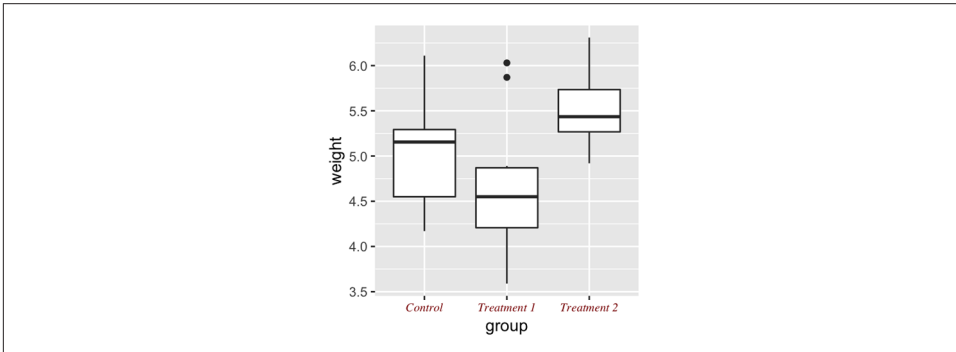


Figure 8-18. X-axis tick labels with manually specified appearance

In this example, the size is set to `rel(0.9)`, which means that it is 0.9 times the size of the base font size for the theme.

These commands control the appearance of only the tick labels, on only one axis. They don't affect the other axis, the axis label, the overall title, or the legend. To control all of these at once, you can use the theming system, as discussed in [Recipe 9.3](#).

See Also

See [Recipe 9.2](#) for more about controlling the appearance of the text.

8.10 Changing the Text of Axis Labels

Problem

You want to change the text of axis labels.

Solution

Use `xlab()` or `ylab()` to change the text of the axis labels ([Figure 8-19](#)):

```
library(gcookbook) # Load gcookbook for the heightweight data set

hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn, colour = sex)) +
  geom_point()

# With default axis labels
hw_plot

# Set the axis labels
hw_plot +
  xlab("Age in years") +
  ylab("Height in inches")
```

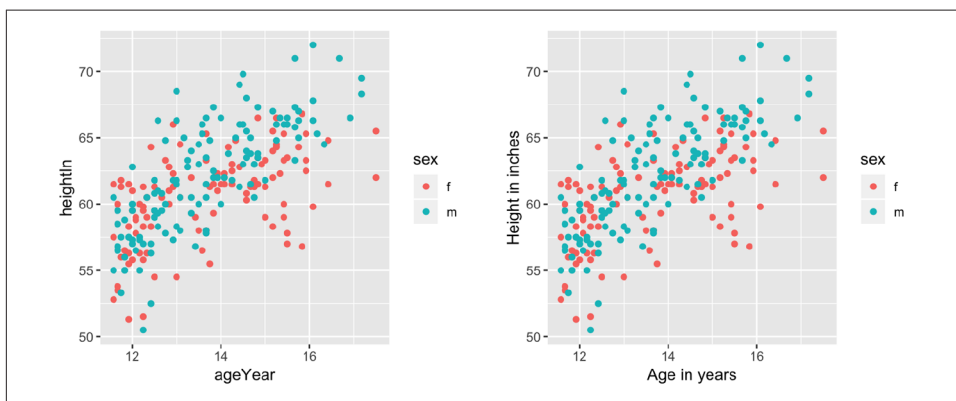


Figure 8-19. Scatter plot with the default axis labels (left); Manually specified labels for the x- and y-axes (right)

Discussion

By default the graphs will just use the column names from the data frame as axis labels. This might be fine for exploring data, but for presenting it, you may want more descriptive axis labels.

Instead of `xlab()` and `ylab()`, you can use `labs()`:

```
hw_plot +  
  labs(x = "Age in years", y = "Height in inches")
```

Another way of setting the axis labels is in the scale specification, like this:

```
hw_plot +  
  scale_x_continuous(name = "Age in years")
```

This may look a bit awkward, but it can be useful if you're also setting other properties of the scale, such as the tick mark placement, range, and so on.

This also applies, of course, to other axis scales, such as `scale_y_continuous()`, `scale_x_discrete()`, and so on.

You can also add line breaks with `\n`, as shown in [Figure 8-20](#):

```
hw_plot +  
  scale_x_continuous(name = "Age\n(years)")
```

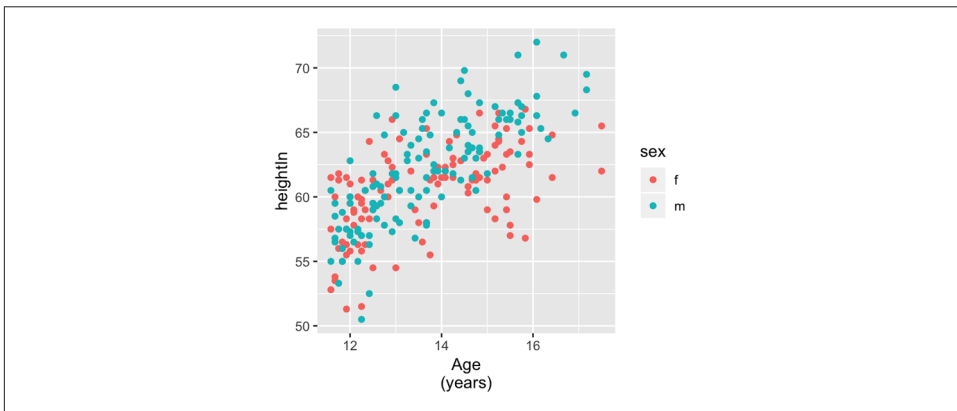


Figure 8-20. X-axis label with a line break

8.11 Removing Axis Labels

Problem

You want to remove the label on an axis.

Solution

For the x-axis label, use `xlab(NULL)`. For the y-axis label, use `ylab(NULL)`.

We'll hide the x-axis in this example (Figure 8-21):

```
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight)) +  
  geom_boxplot()  
  
pg_plot +  
  xlab(NULL)
```

Discussion

Sometimes axis labels are redundant or obvious from the context, and don't need to be displayed. In the example here, the x-axis represents group, but this should be obvious from the context. Similarly, if the y tick labels had *kg* or some other unit in each label, the axis label “weight” would be unnecessary.

Another way to remove the axis label is to set it to an empty string. However, if you do it this way, the resulting graph will still have space reserved for the text, as shown in the graph on the right in Figure 8-21:

```
pg_plot +  
  xlab("")
```

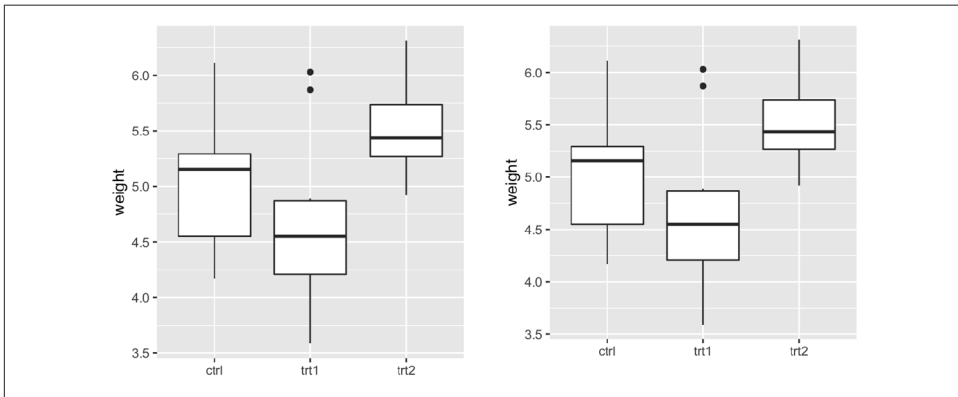


Figure 8-21. X-axis label with `NULL` (left); With the label set to `""` (right)

When you use `theme()` to set `axis.title.x = element_blank()`, the name of the *x* or *y* scale is unchanged, but the text is not displayed and no space is reserved for it. When you set the label to `""`, the name of the scale is changed and the (empty) text does display.

8.12 Changing the Appearance of Axis Labels

Problem

You want to change the appearance of axis labels.

Solution

To change the appearance of the x-axis label (Figure 8-22), use `axis.title.x`:

```
library(gcookbook) # Load gcookbook for the heightweight data set

hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn)) +
  geom_point()

hw_plot +
  theme(axis.title.x = element_text(face = "italic", colour = "darkred",
                                     size = 14))
```

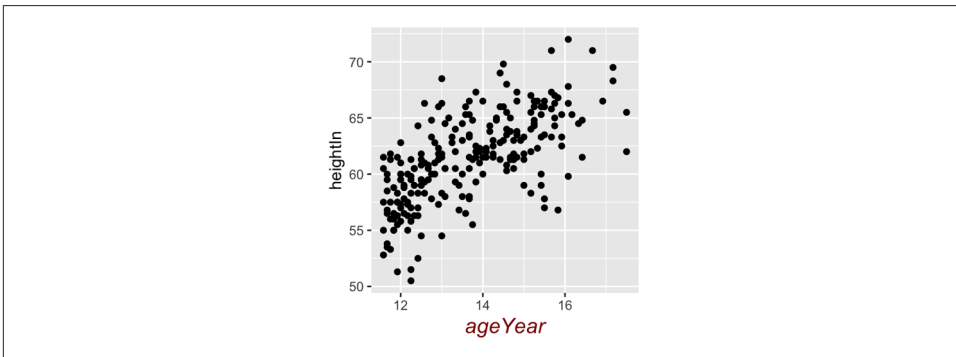


Figure 8-22. X-axis label with customized appearance

Discussion

For the y-axis label, it might also be useful to display the text unrotated, as shown in Figure 8-23 (left). The `\n` in the label represents a newline character:

```
hw_plot +
  ylab("Height\n(inches)") +
  theme(axis.title.y = element_text(angle = 0, face = "italic", size = 14))
```

When you call `element_text()`, the default angle is 0, so if you set `axis.title.y` but don't specify the angle, it will show in this orientation, with the top of the text pointing up. If you change any other properties of `axis.title.y` and want it to be displayed in its usual orientation, rotated 90 degrees, you must manually specify the angle (Figure 8-23, right):

```
hw_plot +
  ylab("Height\n(inches)") +
  theme(axis.title.y = element_text(
    angle = 90,
    face = "italic",
    colour = "darkred",
    size = 14)
  )
```

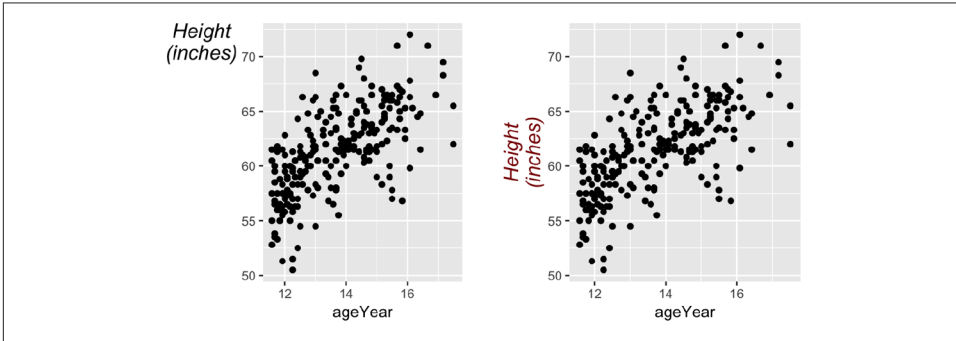


Figure 8-23. Y-axis label with angle = 0 (left); With angle = 90 (right)

See Also

See [Recipe 9.2](#) for more about controlling the appearance of the text.

8.13 Showing Lines Along the Axes

Problem

You want to display lines along the x- and y-axes, but not on the other sides of the graph.

Solution

Using themes, use `axis.line` ([Figure 8-24](#)):

```
library(gcookbook) # Load gcookbook for the heightweight data set

hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn)) +
  geom_point()

hw_plot +
  theme(axis.line = element_line(colour = "black"))
```

Discussion

If you are starting with a theme that has a border around the plotting area, like `theme_bw()`, you will also need to unset `panel.border` (Figure 8-24, right):

```
hw_plot +  
  theme_bw() +  
  theme(panel.border = element_blank(),  
        axis.line = element_line(colour = "black"))
```

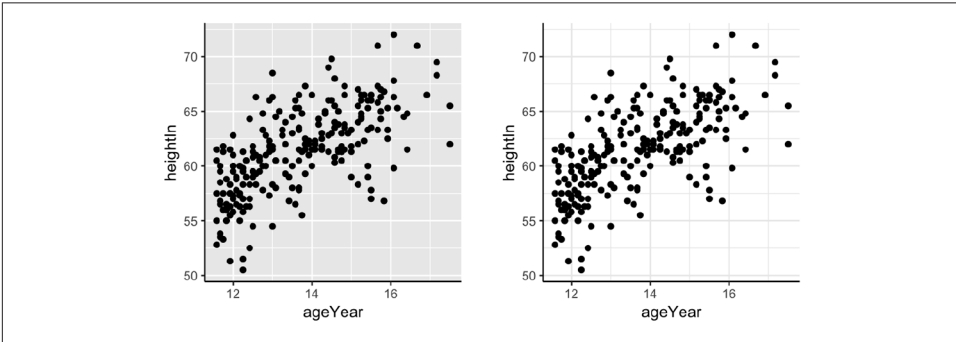


Figure 8-24. Scatter plot with axis lines (left); With `theme_bw()`, `panel.border` must also be made blank (right)

If the lines are thick, the ends will only partially overlap (Figure 8-25, left). To make them fully overlap (Figure 8-25, right), set `lineend = "square"`:

```
# With thick lines, only half overlaps  
hw_plot +  
  theme_bw() +  
  theme(  
    panel.border = element_blank(),  
    axis.line = element_line(colour = "black", size = 4)  
  )  
  
# Full overlap  
hw_plot +  
  theme_bw() +  
  theme(  
    panel.border = element_blank(),  
    axis.line = element_line(colour = "black", size = 4, lineend = "square")  
  )
```

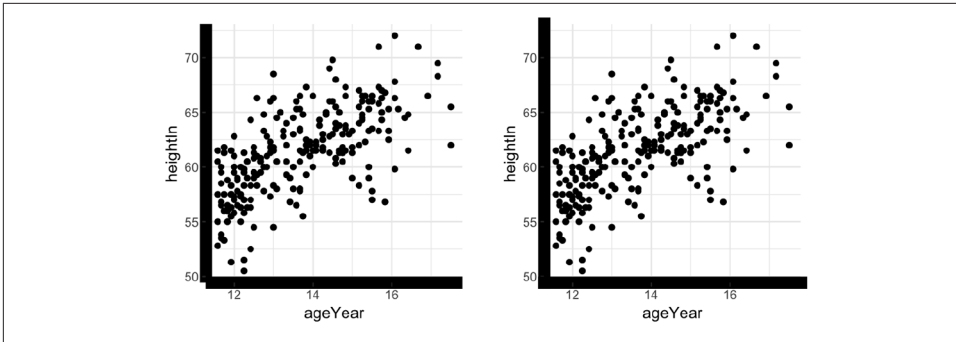


Figure 8-25. With thick lines, the ends don't fully overlap (left); Full overlap with `line-end="square"` (right)

See Also

For more information about how the theming system works, see [Recipe 9.3](#).

8.14 Using a Logarithmic Axis

Problem

You want to use a logarithmic axis for a graph.

Solution

Use `scale_x_log10()` and/or `scale_y_log10()` ([Figure 8-26](#)):

```
library(MASS) # Load MASS for the Animals data set

# Create the base plot
animals_plot <- ggplot(Animals, aes(x = body, y = brain,
                                     label = rownames(Animals))) +
  geom_text(size = 3)

animals_plot

# With logarithmic x and y scales
animals_plot +
  scale_x_log10() +
  scale_y_log10()
```

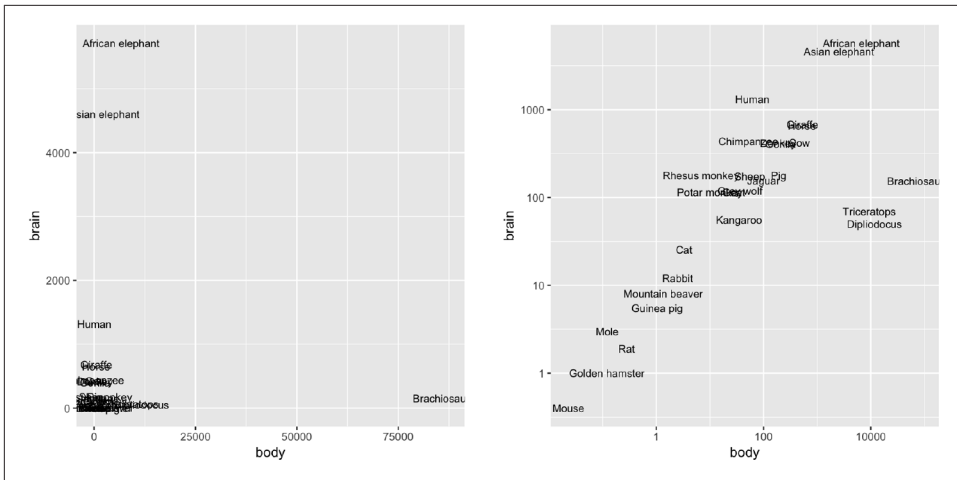


Figure 8-26. Exponentially distributed data with linear-scaled axes (left); With logarithmic axes (right)

Discussion

With a log axis, a given visual distance represents a constant *proportional* change; for example, each centimeter on the y-axis might represent a multiplication of the quantity by 10. In contrast, with a linear axis, a given visual distance represents a constant quantity change; each centimeter might represent adding 10 to the quantity.

Some data sets are exponentially distributed on the x-axis, and others on the y-axis (or both). For example, the `Animals` data set from the `MASS` package contains data on the average brain mass (in g) and body mass (in kg) of various mammals, with a few dinosaurs thrown in for comparison:

```
Animals
#>               body brain
#> Mountain beaver  1.350  8.1
#> Cow             465.000 423.0
#> Grey wolf       36.330 119.5
#> ...<22 more rows>...
#> Brachiosaurus  87000.000 154.5
#> Mole           0.122   3.0
#> Pig            192.000 180.0
```

As shown in [Figure 8-26](#), we can make a scatter plot to visualize the relationship between brain and body mass. With the default linearly scaled axes, it's hard to make much sense of this graph. Because of a few very large animals, the rest of the animals get squished into the lower-left corner—a mouse barely looks different from a triceratops! This is a case where the data is distributed exponentially on both axes.

ggplot will try to make good decisions about where to place the tick marks, but if you don't like them, you can change them by specifying breaks and, optionally, labels. In the example here, the automatically generated tick marks are spaced farther apart than is ideal. For the y-axis tick marks, we can get a vector of every power of 10 from 10^0 to 10^3 like this:

```
10^(0:3)
#> [1] 1 10 100 1000
```

The x-axis tick marks work the same way, but because the range is large, R decides to format the output with scientific notation:

```
10^(-1:5)
#> [1] 1e-01 1e+00 1e+01 1e+02 1e+03 1e+04 1e+05
```

And then we can use those values as the breaks, as in [Figure 8-27](#) (left):

```
animals_plot +
  scale_x_log10(breaks = 10^(-1:5)) +
  scale_y_log10(breaks = 10^(0:3))
```

To instead use exponential notation for the break labels ([Figure 8-27](#), right), use the `trans_format()` function, from the `scales` package:

```
library(scales)
animals_plot +
  scale_x_log10(breaks = 10^(-1:5),
               labels = trans_format("log10", math_format(10^.x))) +
  scale_y_log10(breaks = 10^(0:3),
               labels = trans_format("log10", math_format(10^.x)))
```

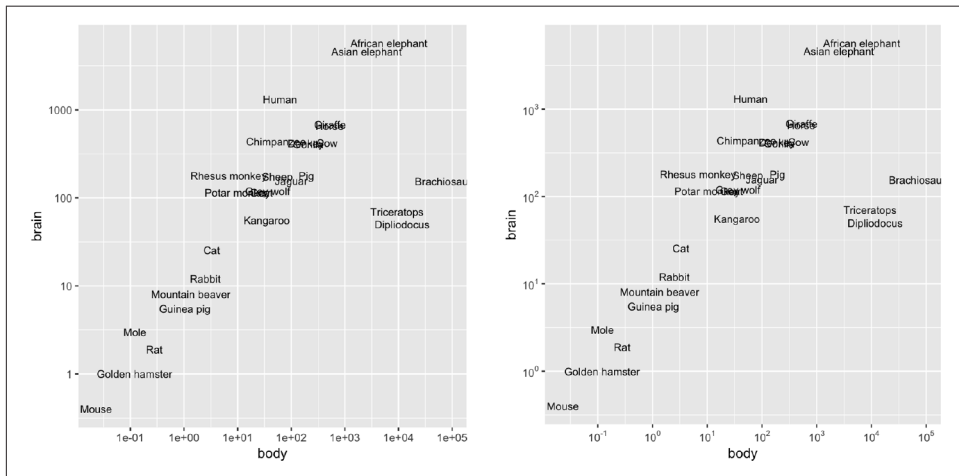


Figure 8-27. Scatter plot with $\log_{10} x$ - and y -axes, and with manually specified breaks (left); With exponents for the tick labels (right)

Another way to use log axes is to transform the data before mapping it to the x and y coordinates (Figure 8-28). Technically, the axes are still linear—it's the quantity that is log-transformed:

```
ggplot(Animals, aes(x = log10(body), y = log10(brain),
                    label = rownames(Animals))) +
  geom_text(size = 3)
```

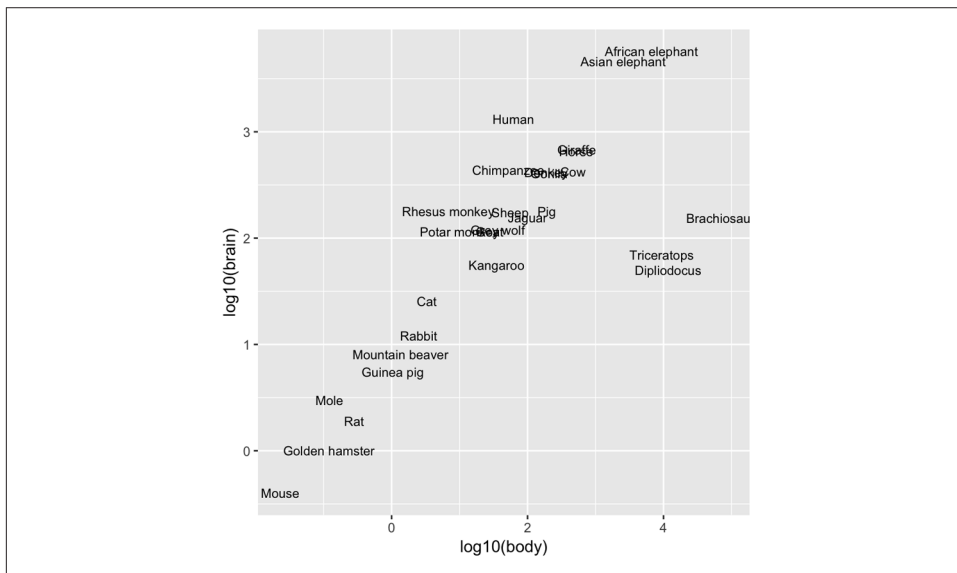


Figure 8-28. Plot with log transform before mapping to x - and y -axes

The previous examples used a \log_{10} transformation, but it is possible to use other transformations, such as \log_2 and natural log, as shown in Figure 8-29. It's a bit more complicated to use these—`scale_x_log10()` is shorthand, but for these other log scales, we need to spell them out:

```
library(scales)

# Use natural log on x, and log2 on y
animals_plot +
  scale_x_continuous(
    trans = log_trans(),
    breaks = trans_breaks("log", function(x) exp(x)),
    labels = trans_format("log", math_format(e^.x))
  ) +
  scale_y_continuous(
    trans = log2_trans(),
    breaks = trans_breaks("log2", function(x) 2^x),
    labels = trans_format("log2", math_format(2^.x))
  )
```

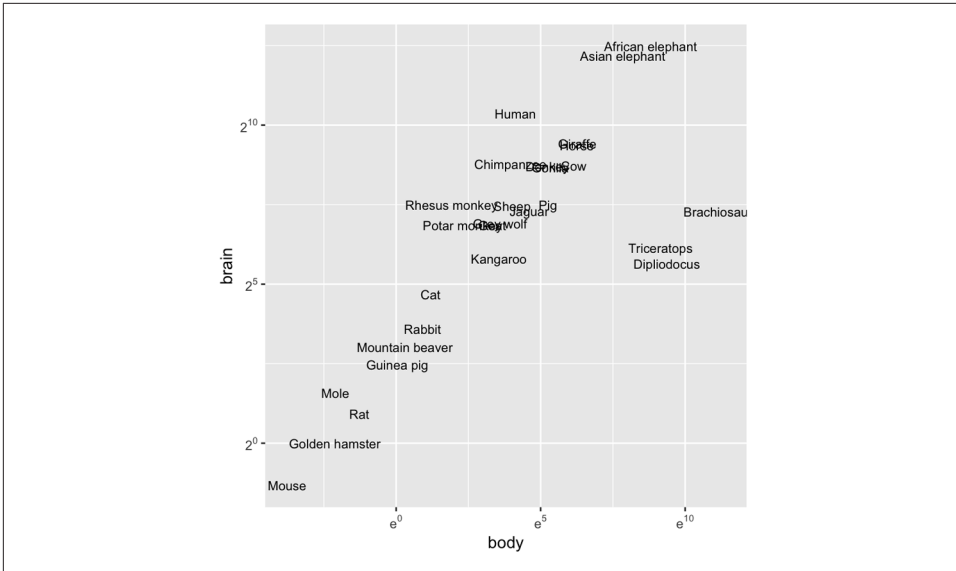


Figure 8-29. Plot with exponents in tick labels. Notice that different bases are used for the x- and y-axes.

It's possible to use a log axis for just one axis. It is often useful to represent financial data this way, because it better represents proportional change. Figure 8-30 shows Apple's stock price with linear and log y-axes. The default tick marks might not be spaced well for your graph; they can be set with the breaks in the scale:

```
library(gcookbook) # Load gcookbook for the aapl data set

ggplot(aapl, aes(x = date, y = adj_price)) +
  geom_line()

ggplot(aapl, aes(x = date, y = adj_price)) +
  geom_line() +
  scale_y_log10(breaks = c(2, 10, 50, 250))
```

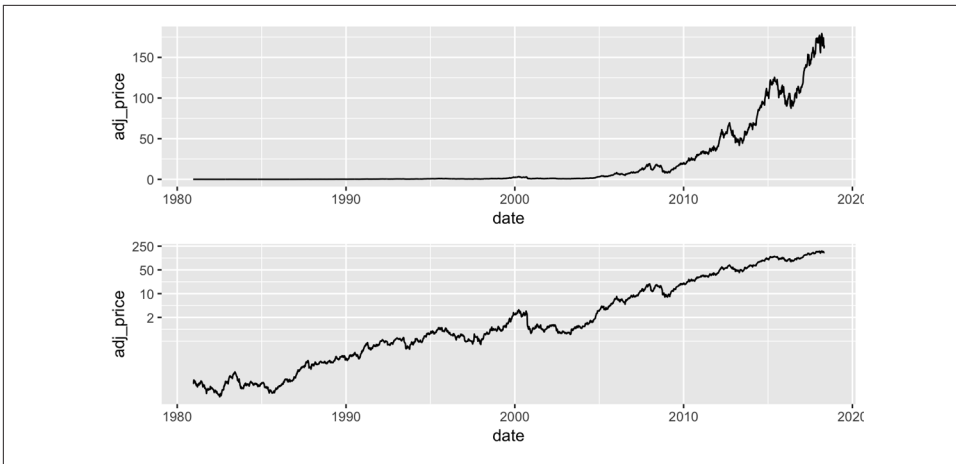


Figure 8-30. Top: a stock chart with a linear x-axis and log y-axis; Bottom: with manual breaks

8.15 Adding Ticks for a Logarithmic Axis

Problem

You want to add tick marks with diminishing spacing for a logarithmic axis.

Solution

Use `annotation_logticks()` (Figure 8-31):

```
library(MASS) # Load MASS for the Animals data set
library(scales) # For the trans_format function

# Given a vector x, return a vector of powers of 10 that encompasses all values
# in x.
breaks_log10 <- function(x) {
  low <- floor(log10(min(x)))
  high <- ceiling(log10(max(x)))

  10^(seq.int(low, high))
}

ggplot(Animals, aes(x = body, y = brain, label = rownames(Animals))) +
  geom_text(size = 3) +
  annotation_logticks() +
  scale_x_log10(breaks = breaks_log10,
               labels = trans_format(log10, math_format(10^.x))) +
  scale_y_log10(breaks = breaks_log10,
               labels = trans_format(log10, math_format(10^.x)))
```

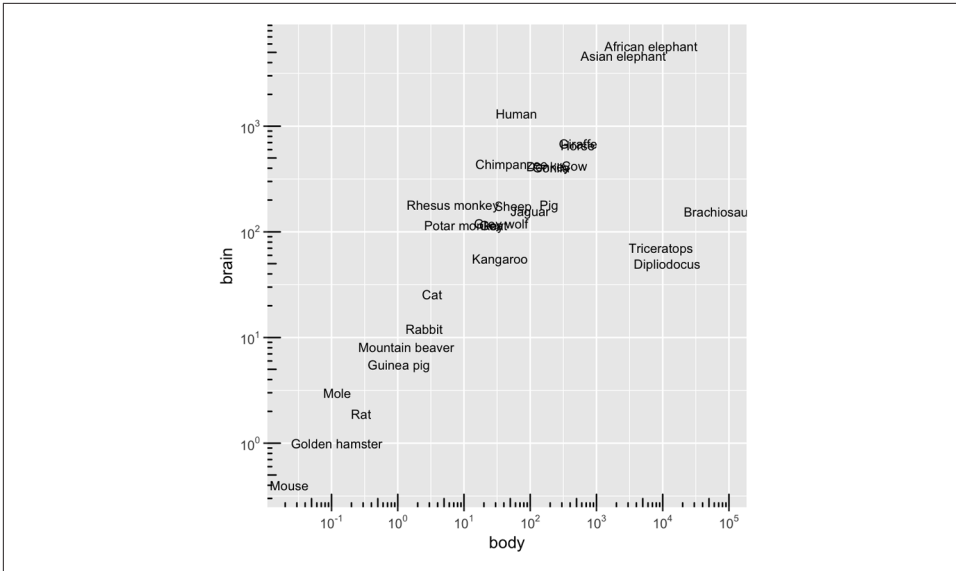


Figure 8-31. Log axes with diminishing tick marks

We also defined a function, `breaks_log10()`, which returns all powers of 10 that encompass the range of values passed to it. This tells `scale_x_log10` where to put the breaks. For example:

```
breaks_log10(c(0.12, 6))
#> [1] 0.1 1.0 10.0
```

Discussion

The tick marks created by `annotation_logticks()` are actually geoms inside the plotting area. There is a long tick mark at each power of 10, and a mid-length tick mark at each 5.

To get the colors of the tick marks and the grid lines to match up a bit better, you can use `theme_bw()`.

By default, the minor grid lines appear visually halfway between the major grid lines, but this is not the same place as the “5” tick marks on a logarithmic scale. To get them to be the same, we can supply a function for the scale’s minor breaks.

We’ll define `breaks_5log10()`, which returns 5 times powers of 10 that encompass the values passed to it:

```
breaks_5log10 <- function(x) {
  low <- floor(log10(min(x)/5))
  high <- ceiling(log10(max(x)/5))
}
```

```

    5 * 10^(seq.int(low, high))
  }

breaks_5log10(c(0.12, 6))
#> [1] 0.05 0.50 5.00 50.00

```

Then we'll use that function for the minor breaks (Figure 8-32):

```

ggplot(Animals, aes(x = body, y = brain, label = rownames(Animals))) +
  geom_text(size = 3) +
  annotation_logticks() +
  scale_x_log10(breaks = breaks_log10,
               minor_breaks = breaks_5log10,
               labels = trans_format(log10, math_format(10^.x))) +
  scale_y_log10(breaks = breaks_log10,
               minor_breaks = breaks_5log10,
               labels = trans_format(log10, math_format(10^.x))) +
  coord_fixed() +
  theme_bw()

```

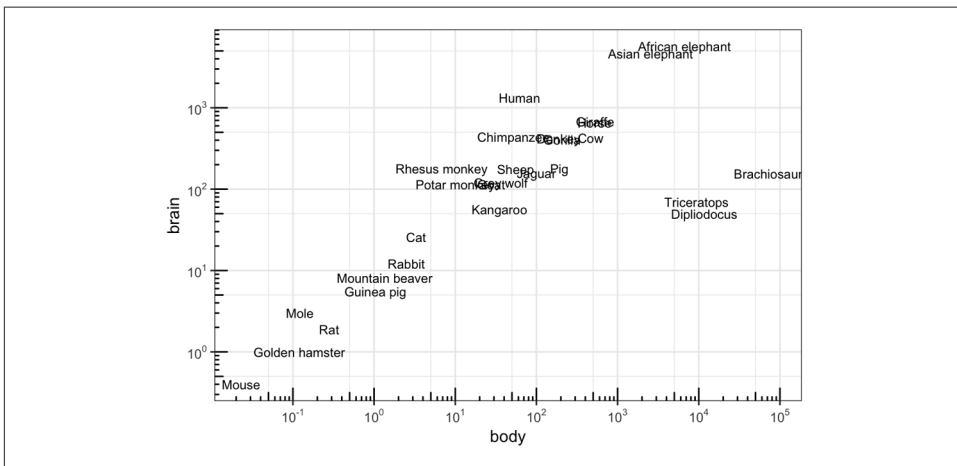


Figure 8-32. Log axes with ticks at each 5, and fixed coordinate ratio

8.16 Making a Circular Plot

Problem

You want to make a circular plot.

Solution

Use `coord_polar()`. For this example we'll use the `wind` data set from `gcookbook`. It contains samples of wind speed and direction for every 5 minutes throughout a day.

The direction of the wind is categorized into 15-degree bins, and the speed is categorized into 5 m/s increments:

```
library(gcookbook) # Load gcookbook for the wind data set
wind
#>      TimeUTC Temp WindAvg WindMax WindDir SpeedCat DirCat
#> 3           0 3.54   9.52   10.39     89    10-15    90
#> 4           5 3.52   9.10   9.90     92     5-10    90
#> 5          10 3.53   8.73   9.51     92     5-10    90
#> ...<280 more rows>...
#> 286       2335 6.74  18.98  23.81    250     >20    255
#> 287       2340 6.62  17.68  22.05    252     >20    255
#> 288       2345 6.22  18.54  23.91    259     >20    255
```

We'll plot a count of the number of samples at each SpeedCat and DirCat using `geom_histogram()` (Figure 8-33). We'll set `binwidth` to 15 and make the origin of the histogram start at `-7.5`, so that each bin is centered around 0, 15, 30, etc.:

```
ggplot(wind, aes(x = DirCat, fill = SpeedCat)) +
  geom_histogram(binwidth = 15, boundary = -7.5) +
  coord_polar() +
  scale_x_continuous(limits = c(0,360))
```

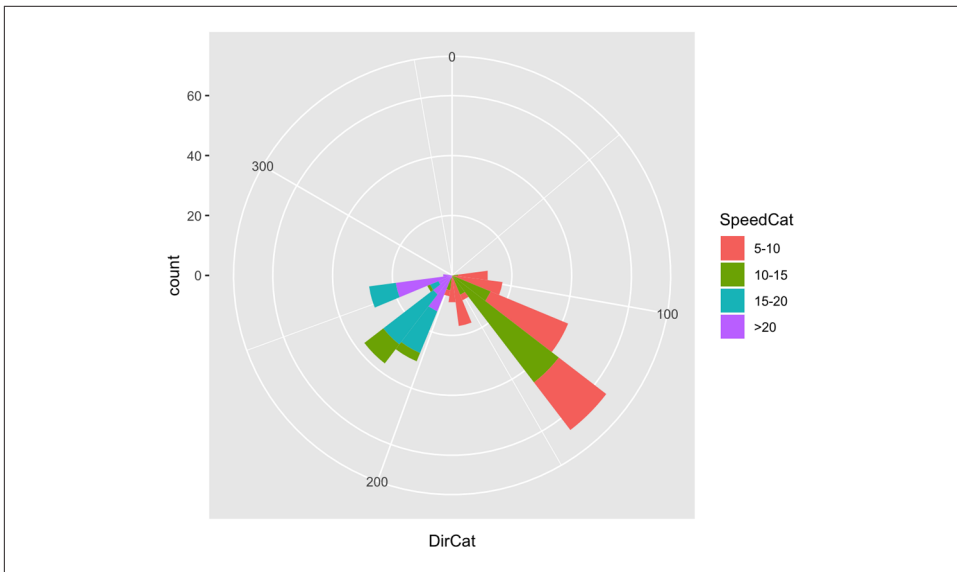


Figure 8-33. Polar plot

Discussion

Be cautious when using polar plots, since they can perceptually distort the data. In the example here, at 210 degrees there are 15 observations with a speed of 15–20 and 13 observations with a speed of >20, but a quick glance at the picture makes it appear

that there are more observations at >20. There are also three observations with a speed of 10–15, but they’re barely visible.

In this example we can make the plot a little prettier by reversing the legend, using a different palette, adding an outline, and setting the breaks to some more familiar numbers (Figure 8-34):

```
ggplot(wind, aes(x = DirCat, fill = SpeedCat)) +
  geom_histogram(binwidth = 15, boundary = -7.5, colour = "black", size = .25) +
  guides(fill = guide_legend(reverse = TRUE)) +
  coord_polar() +
  scale_x_continuous(limits = c(0, 360),
                    breaks = seq(0, 360, by = 45),
                    minor_breaks = seq(0, 360, by = 15)) +
  scale_fill_brewer()
```

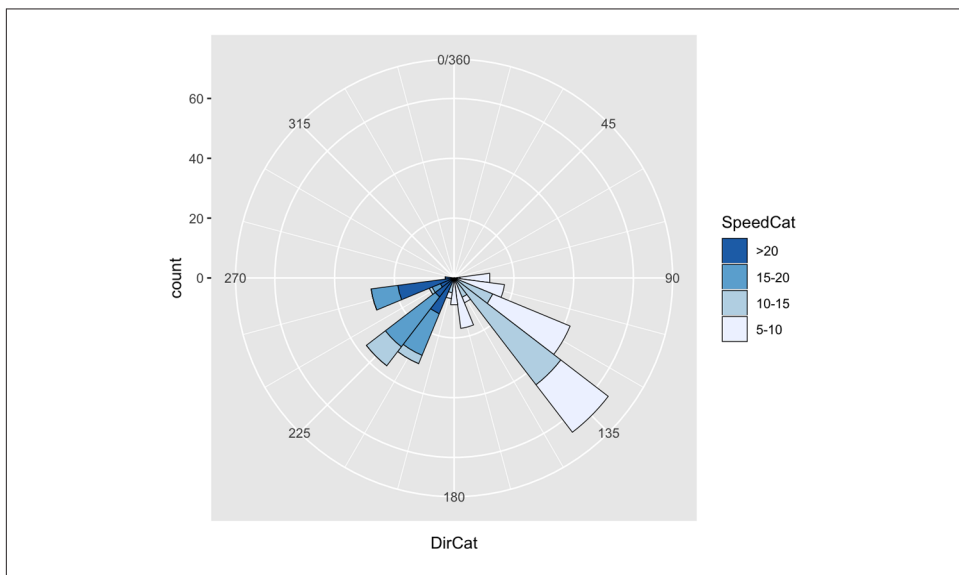


Figure 8-34. Polar plot with different colors and breaks

It may also be useful to set the starting angle with the `start` argument, especially when using a discrete variable for *theta*. The starting angle is specified in radians, so if you know the adjustment in degrees, you’ll have to convert it to radians:

```
coord_polar(start = -45 * pi / 180)
```

Polar coordinates can be used with other geoms, including lines and points. There are a few important things to keep in mind when using these geoms. First, by default, for the variable that is mapped to *y* (or *r*), the smallest actual value gets mapped to the center; in other words, the smallest data value gets mapped to a visual radius value of

0. You may be expecting a data value of 0 to be mapped to a radius of 0, but to make sure this happens, you'll need to set the limits.

Next, when using a continuous x (or *theta*), the smallest and largest data values are merged. Sometimes this is desirable, sometimes not. To change this behavior, you'll need to set the limits.

Finally, the *theta* values of the polar coordinates do not wrap around—it is presently not possible to have a geom that crosses over the starting angle (usually vertical).

I'll illustrate these issues with an example. The following code creates a data frame from the *mdeaths* time series data set and produces the graph shown on the left in [Figure 8-35](#):

```
# Put mdeaths time series data into a data frame
mdeaths_mod <- data.frame(
  deaths = as.numeric(mdeaths),
  month = as.numeric(cycle(mdeaths))
)

# Calculate average number of deaths in each month
library(dplyr)
mdeaths_mod <- mdeaths_mod %>%
  group_by(month) %>%
  summarise(deaths = mean(deaths))

mdeaths_mod
#> # A tibble: 12 x 2
#>   month deaths
#>   <dbl> <dbl>
#> 1     1 2129.833
#> 2     2 2081.333
#> 3     3 1970.500
#> 4     4 1657.333
#> 5     5 1314.167
#> 6     6 1186.833
#> 7     7 1136.667
#> 8     8 1037.667
#> ... with 4 more rows

# Create the base plot
mdeaths_plot <- ggplot(mdeaths_mod, aes(x = month, y = deaths)) +
  geom_line() +
  scale_x_continuous(breaks = 1:12)

# With coord_polar
mdeaths_plot + coord_polar()
```

The first problem is that the data values (ranging from about 1000 to 2100) are mapped to the radius such that the smallest data value is at radius 0. We'll fix this by set-

ting the y (or r) limits from 0 to the maximum data value, as shown in the graph on the right in [Figure 8-35](#):

```
# With coord_polar and y (r) limits going to zero
mdeaths_plot +
  coord_polar() +
  ylim(0, max(mdeaths_mod$deaths))
```

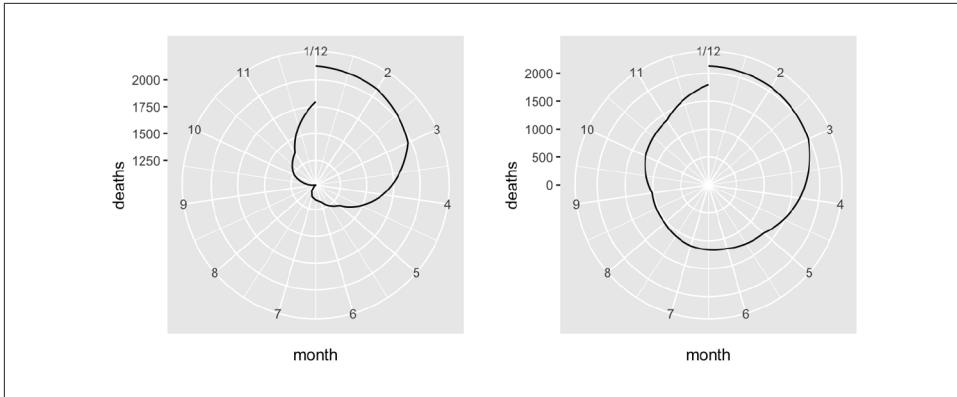


Figure 8-35. Polar plot with line (notice the data range of the radius) (left); With the radius representing a data range starting from zero (right)

The next problem is that the lowest and highest month values, 1 and 12, are shown at the same angle. We'll fix this by setting the x limits from 0 to 12, creating the graph on the left in [Figure 8-36](#) (notice that using `xlim()` overrides the `scale_x_continuous()` in `p`, so it no longer displays breaks for each month; see [Recipe 8.2](#) for more information):

```
mdeaths_plot +
  coord_polar() +
  ylim(0, max(mdeaths_mod$deaths)) +
  xlim(0, 12)
```

There's one last issue, which is that the beginning and end aren't connected. To fix that, we need to modify our data frame by adding one row with a month of 0 that has the same value as the row with month 12. This will make the starting and ending points the same, as in the graph on the right in [Figure 8-36](#) (alternatively, we could add a row with month 13, instead of month 0):

```
# Connect the lines by adding a value for 0 that is the same as 12
mdeaths_x <- mdeaths_mod[mdeaths_mod$month==12, ]
mdeaths_x$month <- 0
mdeaths_new <- rbind(mdeaths_x, mdeaths_mod)

# Make the same plot as before, but with the new data, by using %+%
mdeaths_plot %+%
  mdeaths_new +
```

```
coord_polar() +  
ylim(0, max(mdeaths_mod$deaths))
```

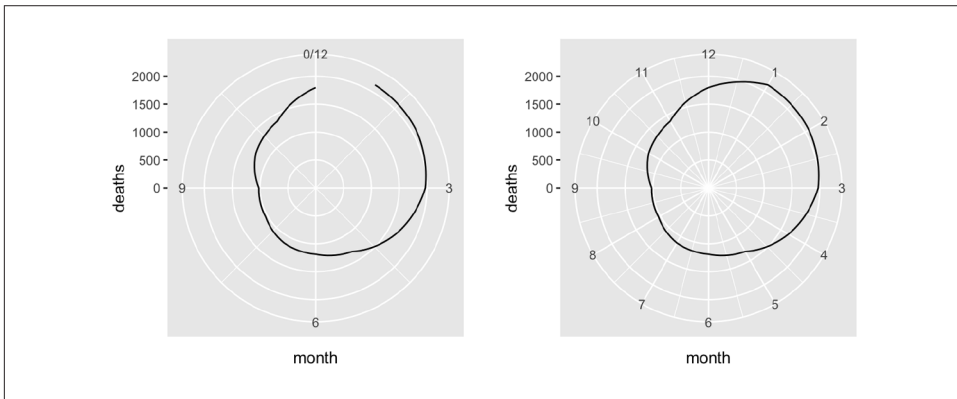


Figure 8-36. Polar plot with theta representing *x* values from 0 to 12 (left); The gap is filled in by adding a dummy data point for month 0 (right)



Notice the use of the `%>%` operator. When you add a data frame to a ggplot object with `%>%`, it replaces the default data frame in the ggplot object. In this case, it changed the default data frame for `p` from `mdeaths_mod` to `mdeaths_new`.

See Also

See [Recipe 10.4](#) for more about reversing the direction of a legend.

See [Recipe 8.6](#) for more about specifying which values will have tick marks (breaks) and labels.

8.17 Using Dates on an Axis

Problem

You want to use dates on an axis.

Solution

Map a column of class `Date` to the x- or y-axis. We'll use the `economics` data set for this example:

```
economics  
#> # A tibble: 574 x 6  
#>   date       pce    pop psavert uempmed unemploy  
#>   <date>     <dbl> <int>   <dbl>   <dbl>   <int>
```

```
#> 1 1967-07-01 507. 198712 12.5 4.5 2944
#> 2 1967-08-01 510. 198911 12.5 4.7 2945
#> 3 1967-09-01 516. 199113 11.7 4.6 2958
#> 4 1967-10-01 513. 199311 12.5 4.9 3143
#> 5 1967-11-01 518. 199498 12.5 4.7 3066
#> 6 1967-12-01 526. 199657 12.1 4.8 3018
#> # ... with 568 more rows
```

The column `date` is an object of class `Date`, and mapping it to `x` will produce the result shown in [Figure 8-37](#):

```
ggplot(economics, aes(x = date, y = psavert)) +
  geom_line()
```

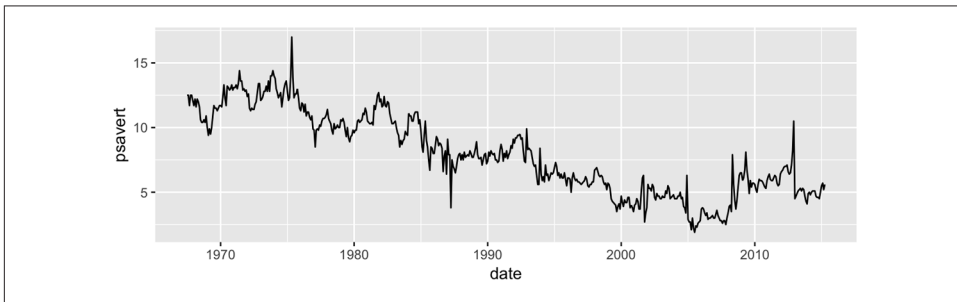


Figure 8-37. Dates on the x-axis

Discussion

`ggplot` handles two kinds of time-related objects: dates (objects of class `Date`) and date-times (objects of class `POSIXt`). The difference between these is that `Date` objects represent dates and have a resolution of one day, while `POSIXt` objects represent moments in time and have a resolution of a fraction of a second.

Specifying the breaks is similar to with a numeric axis—the main difference is in specifying the sequence of dates to use. We’ll use a subset of the `economics` data, ranging from mid-1992 to mid-1993. If breaks aren’t specified, they will be automatically selected, as shown in [Figure 8-38](#) (top):

```
library(dplyr)

# Take a subset of economics
econ_mod <- economics %>%
  filter(date >= as.Date("1992-05-01") & date < as.Date("1993-06-01"))

# Create the base plot, which does not specify the breaks
econ_plot <- ggplot(econ_mod, aes(x = date, y = psavert)) +
  geom_line()

econ_plot
```

The breaks can be created by using the `seq()` function with starting and ending dates, and an interval (Figure 8-38, bottom):

```
# Specify breaks as a Date vector
datebreaks <- seq(as.Date("1992-06-01"), as.Date("1993-06-01"), by = "2 month")

# Use breaks, and rotate text labels
econ_plot +
  scale_x_date(breaks = datebreaks) +
  theme(axis.text.x = element_text(angle = 30, hjust = 1))
```

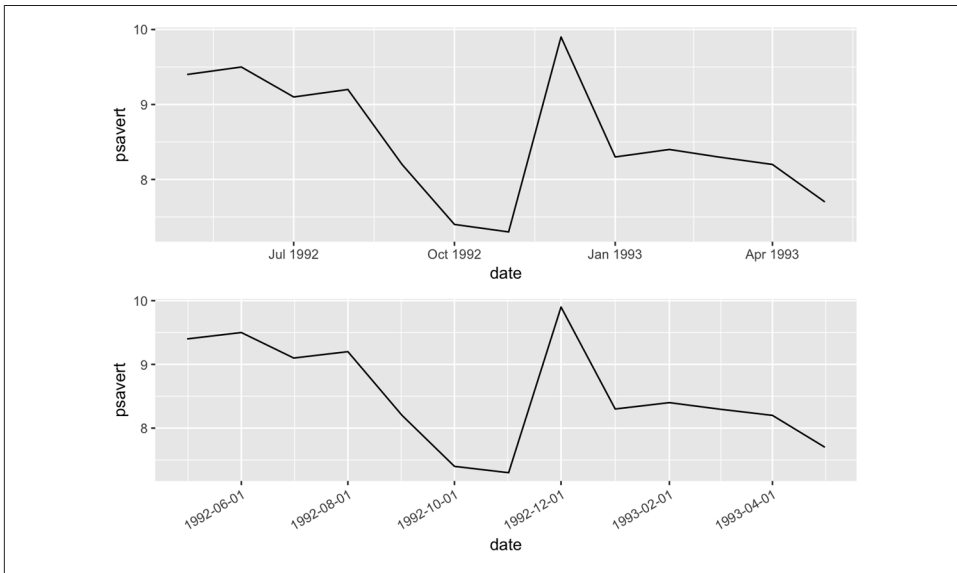


Figure 8-38. Top: with default breaks on the x-axis; Bottom: with breaks specified

Notice that the formatting of the breaks changed. You can specify the formatting by using the `date_format()` function from the `scales` package. Here we'll use `"%Y %b"`, which results in a format like "1992 Jun", as shown in Figure 8-39:

```
library(scales)

econ_plot +
  scale_x_date(breaks = datebreaks, labels = date_format("%Y %b")) +
  theme(axis.text.x = element_text(angle = 30, hjust = 1))
```

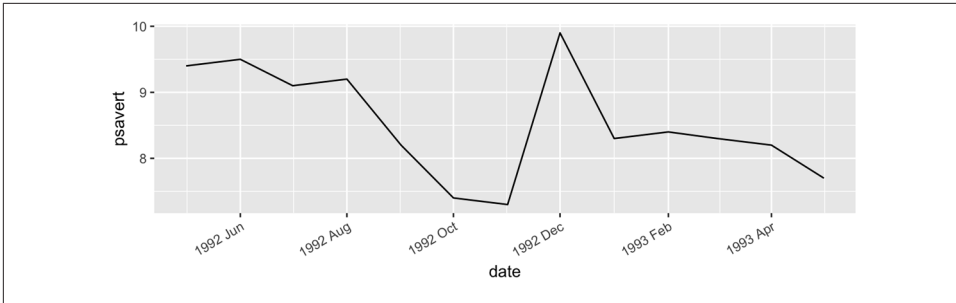


Figure 8-39. Line graph with date format specified

Common date format options are shown in Table 8-1. They are to be put in a string that is passed to `date_format()`, and the format specifiers will be replaced with the appropriate values. For example, if you use "%B %d, %Y", it will result in labels like "June 01, 1992."

Table 8-1. Date format options

Option	Description
%Y	Year with century (2012)
%y	Year without century (12)
%m	Month as a decimal number (08)
%b	Abbreviated month name in current locale (Aug)
%B	Full month name in current locale (August)
%d	Day of month as a decimal number (04)
%U	Week of the year as a decimal number, with Sunday as the first day of the week (00–53)
%W	Week of the year as a decimal number, with Monday as the first day of the week (00–53)
%w	Day of week (0–6, Sunday is 0)
%a	Abbreviated weekday name (Thu)
%A	Full weekday name (Thursday)

Some of these items are specific to the computer's locale. Months and days have different names in different languages (the examples here are generated with a US locale). You can change the locale with `Sys.setlocale()`. For example, this will change the date formatting to use an Italian locale:

```
# Mac and Linux
Sys.setlocale("LC_TIME", "it_IT.UTF-8")

# Windows
Sys.setlocale("LC_TIME", "italian")
```

Note that the locale names may differ between platforms, and your computer must have support for the locale installed at the operating system level.

See Also

See `?Sys.setlocale` for more about setting the locale.

See `?strptime` for information about converting strings to dates, and for information about formatting the date output.

8.18 Using Relative Times on an Axis

Problem

You want to use relative times on an axis.

Solution

Times are commonly stored as numbers. For example, the time of day can be stored as a number representing the hour. Time can also be stored as a number representing the number of minutes or seconds from some starting time. In these cases, you map a value to the x- or y-axis and use a formatter to generate the appropriate axis labels ([Figure 8-40](#)):

```
# Convert WWWusage time-series object to data frame
www <- data.frame(
  minute = as.numeric(time(WWWusage)),
  users  = as.numeric(WWWusage)
)

# Define a formatter function - converts time in minutes to a string
timeHM_formatter <- function(x) {
  h <- floor(x/60)
  m <- floor(x %% 60)
  lab <- sprintf("%d:%02d", h, m) # Format the strings as HH:MM
  return(lab)
}

# Default x axis
ggplot(www, aes(x = minute, y = users)) +
  geom_line()

# With formatted times
ggplot(www, aes(x = minute, y = users)) +
  geom_line() +
  scale_x_continuous(
    name = "time",
    breaks = seq(0, 100, by = 10),
```

```

labels = timeHM_formatter
)

```

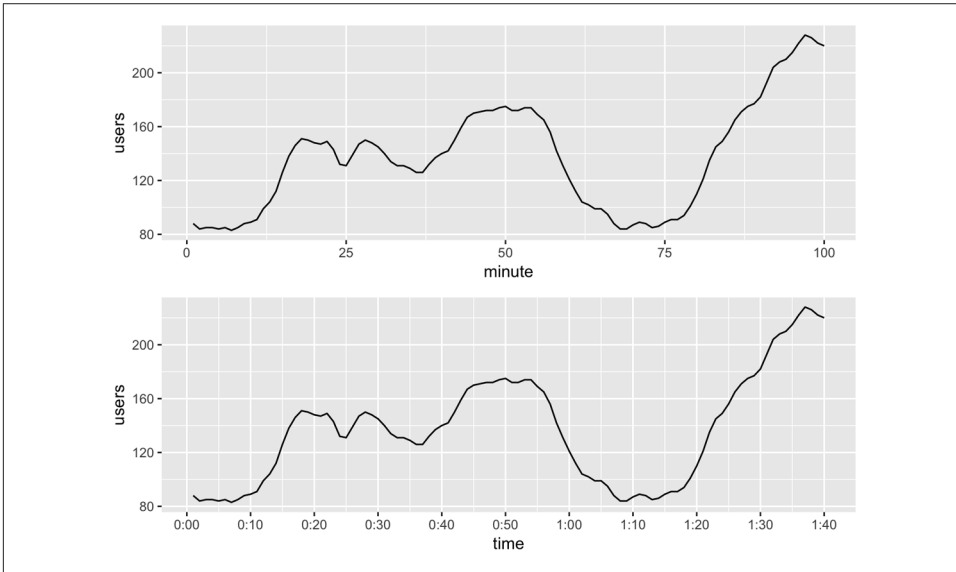


Figure 8-40. Top: relative times on x-axis; Bottom: with formatted times

Discussion

In some cases it might be simpler to specify the breaks and labels manually, with something like this:

```

scale_x_continuous(
  breaks = c(0, 20, 40, 60, 80, 100),
  labels = c("0:00", "0:20", "0:40", "1:00", "1:20", "1:40")
)

```

In the preceding example, we used the `timeHM_formatter()` function to convert the numeric time (in minutes) to a string like "1:10":

```

timeHM_formatter(c(0, 50, 51, 59, 60, 130, 604))
#> [1] "0:00" "0:50" "0:51" "0:59" "1:00" "2:10" "10:04"

```

To convert to HH:MM:SS format, you can use the following formatter function:

```

timeHMS_formatter <- function(x) {
  h <- floor(x/3600)
  m <- floor((x/60) %% 60)
  s <- round(x %% 60)
  lab <- sprintf("%02d:%02d:%02d", h, m, s) # Round to nearest second
  lab <- sub("^00:", "", lab) # Format the strings as HH:MM:SS
  lab <- sub("^0", "", lab) # Remove leading 00: if present
  return(lab) # Remove leading 0 if present
}

```

Running it on some sample numbers yields:

```
timeHMS_formatter(c(20, 3000, 3075, 3559.2, 3600, 3606, 7813.8))  
#> [1] "0:20"      "50:00"      "51:15"      "59:19"      "1:00:00"    "1:00:06"    "2:10:14"
```

See Also

See [Recipe 15.21](#) for information about converting time series objects to data frames.

Controlling the Overall Appearance of Graphs

In this chapter I'll discuss how to control the overall appearance of graphics made by `ggplot2`. The grammar of graphics that underlies `ggplot2` is concerned with how data is processed and displayed—it's not concerned with things like fonts, background colors, and so on. When it comes to presenting your data, there's a good chance that you'll want to tune the appearance of these things. `ggplot2`'s theming system provides control over the appearance of nondata elements. I touched on the theme system in the previous chapter, and here I'll explain a bit more about how it works.

9.1 Setting the Title of a Graph

Problem

You want to add a title to your plot.

Solution

Use `ggtitle()` to add a title, as shown in [Figure 9-1](#):

```
library(gcookbook) # Load gcookbook for the heightweight data set

hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn)) +
  geom_point()

hw_plot +
  ggtitle("Age and Height of Schoolchildren")

# Use \n for a newline
```

```
hw_plot +  
  ggtitle("Age and Height\nof Schoolchildren")
```

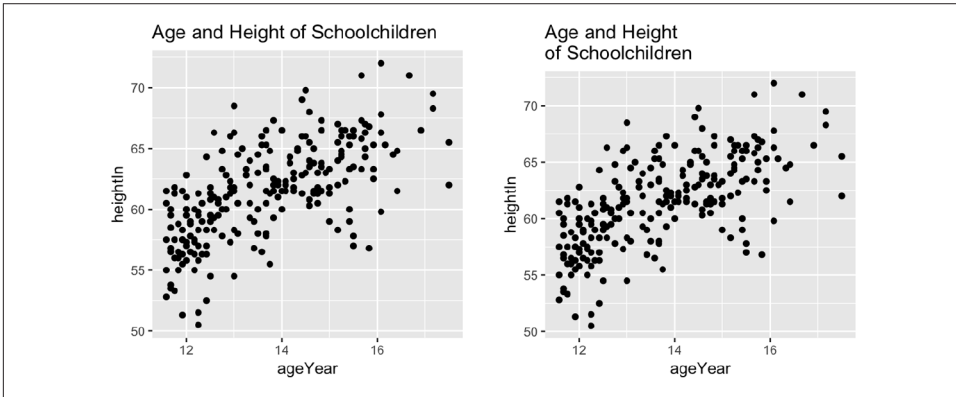


Figure 9-1. Scatter plot with a title added (left); With a `\n` for a newline (right)

Discussion

`ggtitle()` is equivalent to using `labs(title = "Title text")`.

You can add a subtitle by providing a string as the second argument of `ggtitle()`. By default it will display with slightly smaller text than the main title (Figure 9-2):

```
hw_plot +  
  ggtitle("Age and Height of Schoolchildren", "11.5 to 17.5 years old")
```

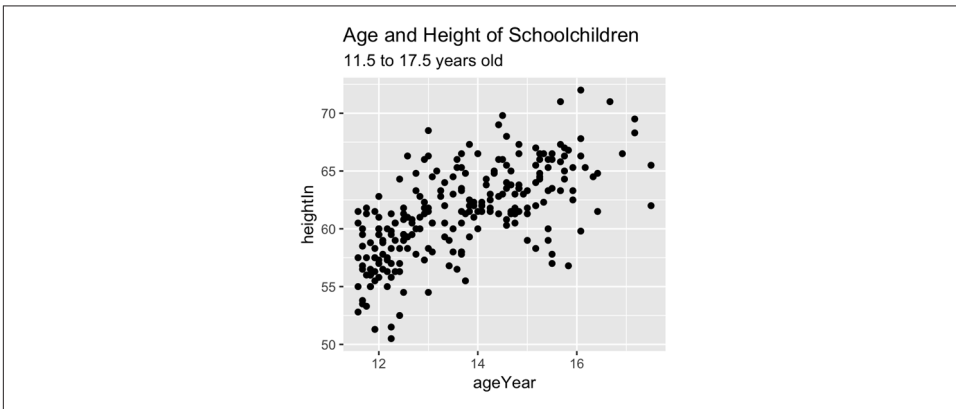


Figure 9-2. Scatter plot with a subtitle

If you want to move the title inside the plotting area, you can use one of two methods, both of which are a little bit of a hack (Figure 9-3). The first method is to use `ggtitle()` with a negative `vjust` value. The drawback of this method is that it still reserves blank space above the plotting region for the title.

The second method is to instead use a text annotation, setting its x position to the middle of the x range and its y position to `Inf`, which places it at the top of the plotting region. This also requires a positive `vjust` value to bring the text fully inside the plotting region:

```
# Move the title inside
hw_plot +
  ggtitle("Age and Height of Schoolchildren") +
  theme(plot.title = element_text(vjust = -8))

# Use a text annotation instead
hw_plot +
  annotate("text", x = mean(range(heightweight$ageYear)), y = Inf,
    label = "Age and Height of Schoolchildren", vjust = 1.5, size = 4.5)
```

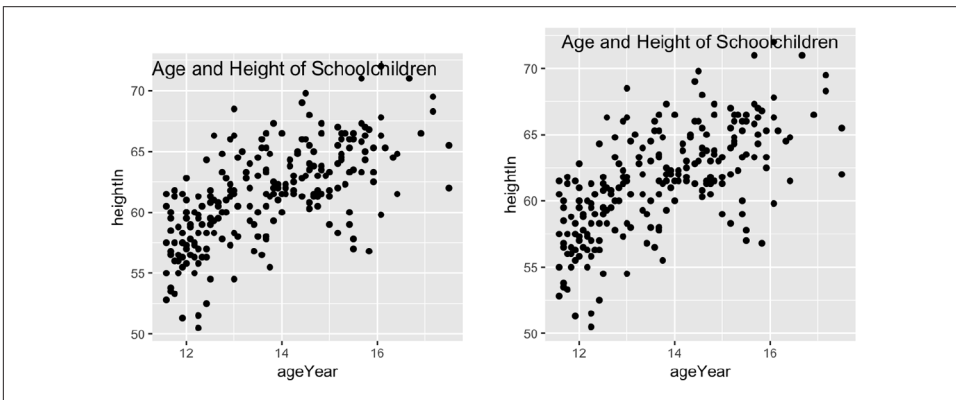


Figure 9-3. Title with `ggtitle` and a negative `vjust` value (note the extra space above the plotting area) (left); With a text annotation at the top of the figure (right)

9.2 Changing the Appearance of Text

Problem

You want to change the appearance of text in a plot.

Solution

To set the appearance of theme items such as the title, axis labels, and axis tick marks, use `theme()` and set the item with `element_text()`. For example, `axis.title.x` controls the appearance of the x-axis label and `plot.title` controls the appearance of the title text (Figure 9-4):

```
library(gcookbook) # Load gcookbook for the heightweight data set

# Base plot
```

```

hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn)) +
  geom_point()

# Controlling appearance of theme items
hw_plot +
  theme(axis.title.x = element_text(
    size = 16, lineheight = .9,
    family = "Times", face = "bold.italic", colour = "red"
  ))

hw_plot +
  ggtitle("Age and Height\nof Schoolchildren") +
  theme(plot.title = element_text(
    size = rel(1.5), lineheight = .9,
    family = "Times", face = "bold.italic", colour = "red"
  ))

# rel(1.5) means that the font will be 1.5 times the base font size of the theme.
# For theme elements, font size is in points.

```

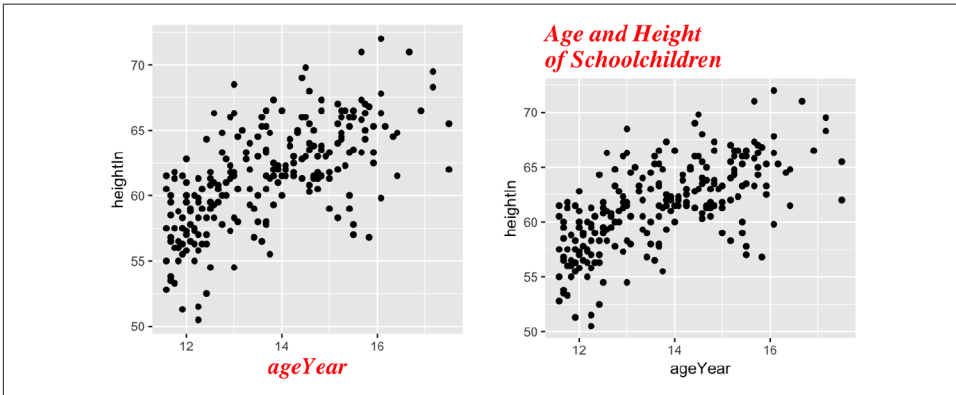


Figure 9-4. *axis.title.x* (left); *plot.title* (right)

To set the appearance of text geoms (text that's in the plot itself, with `geom_text()` or `annotate()`), set the text properties. For example (Figure 9-5):

```

hw_plot +
  annotate("text", x = 15, y = 53, label = "Some text",
    size = 7, family = "Times", fontface = "bold.italic", colour = "red")

hw_plot +
  geom_text(aes(label = weightLb), size = 4, family = "Times", colour = "red")

# For text geoms, font size is in mm

```

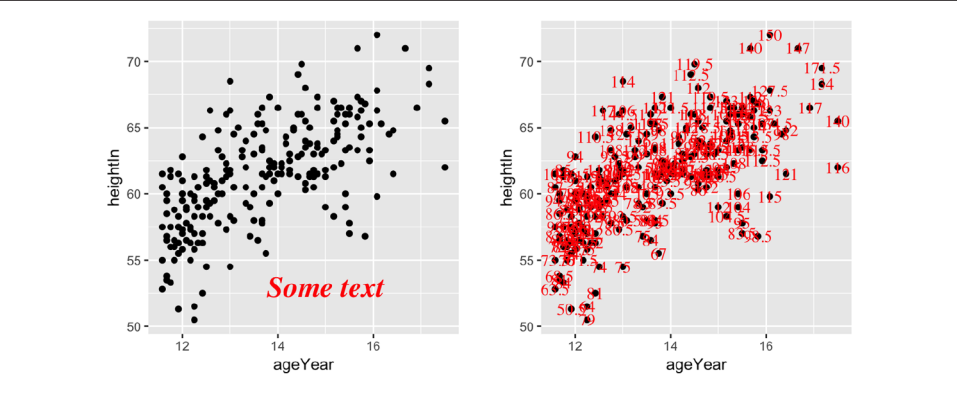


Figure 9-5. `annotate("text")` (left); `geom_text()` (right)

Discussion

There are two kinds of text items in ggplot2: theme elements and text geoms. Theme elements are all the nondata elements in the plot: the title, legends, and axes. Text geoms are things that are part of the plot itself, and reflect the data content.

There are differences in the parameters, as shown in [Table 9-1](#).

Table 9-1. Text properties of theme elements and text geoms

Theme elements	Text geoms	Description
family	family	Helvetica, Times, Courier
face	fontface	plain, bold, italic, bold.italic
colour	colour	Color (name or “#RRGGBB”)
size	size	Font size (in points for theme elements; in mm for geoms)
hjust	hjust	Horizontal alignment: 0 = left, 0.5 = center, 1 = right
vjust	vjust	Vertical alignment: 0 = bottom, 0.5 = middle, 1 = top
angle	angle	Angle in degrees
lineheight	lineheight	Line spacing multiplier

The theme elements are listed in [Table 9-2](#). Most of them are straightforward. Some are shown in [Figure 9-6](#).

Table 9-2. Theme elements that control text appearance in `theme()`

Element name	Description
axis.title	Appearance of axis labels on both axes
axis.title.x	Appearance of x-axis label
axis.title.y	Appearance of y-axis label

Element name	Description
<code>axis.ticks</code>	Appearance of tick labels on both axes
<code>axis.ticks.x</code>	Appearance of x tick labels
<code>axis.ticks.y</code>	Appearance of y tick labels
<code>legend.title</code>	Appearance of legend title
<code>legend.text</code>	Appearance of legend items
<code>plot.title</code>	Appearance of overall plot title
<code>strip.text</code>	Appearance of facet labels in both directions
<code>strip.text.x</code>	Appearance of horizontal facet labels
<code>strip.text.y</code>	Appearance of vertical facet labels

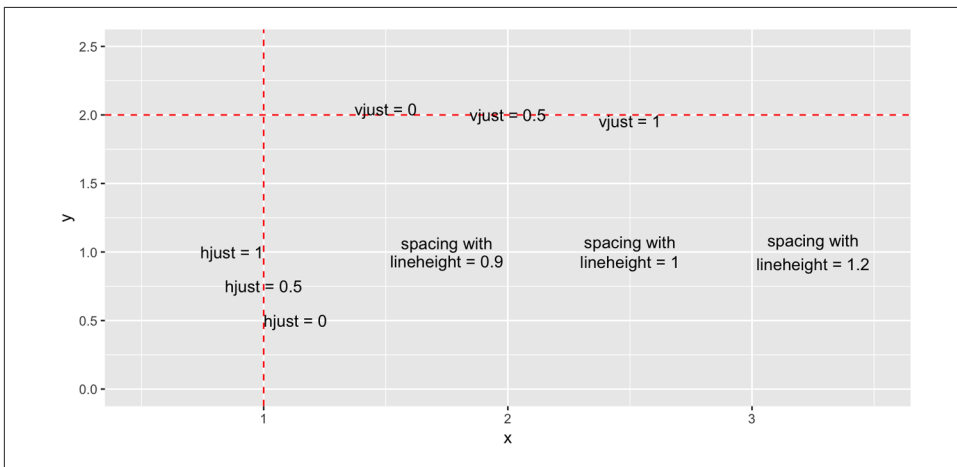


Figure 9-6. Aligning with *hjust* and *vjust*, and spacing with *lineheight*

9.3 Using Themes

Problem

You want to use premade themes to control the overall plot appearance.

Solution

There are many premade themes that are already included in `ggplot2`. The default `ggplot2` theme is `theme_grey()`, but the following examples also showcase `theme_bw()`, `theme_minimal()`, and `theme_classic()`.

To use a premade theme, add `theme_bw()` or another theme to your plot (Figure 9-7):

```
library(gcookbook) # Load gcookbook for the heightweight data set

# Create the base plot
hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn)) +
  geom_point()

# Grey theme (the default)
hw_plot +
  theme_grey()

# Black-and-white theme
hw_plot +
  theme_bw()

# Minimal theme without background annotations
hw_plot +
  theme_minimal()

# Classic theme, with axis lines but no gridlines
hw_plot +
  theme_classic()
```

Another theme included in `ggplot2` is `theme_void()`, which makes all plot elements blank and only shows your data (Figure 9-8). This is especially useful if you don't want any default theme settings, and instead want a blank slate on which to choose your own theme elements:

```
hw_plot +
  theme_void()
```

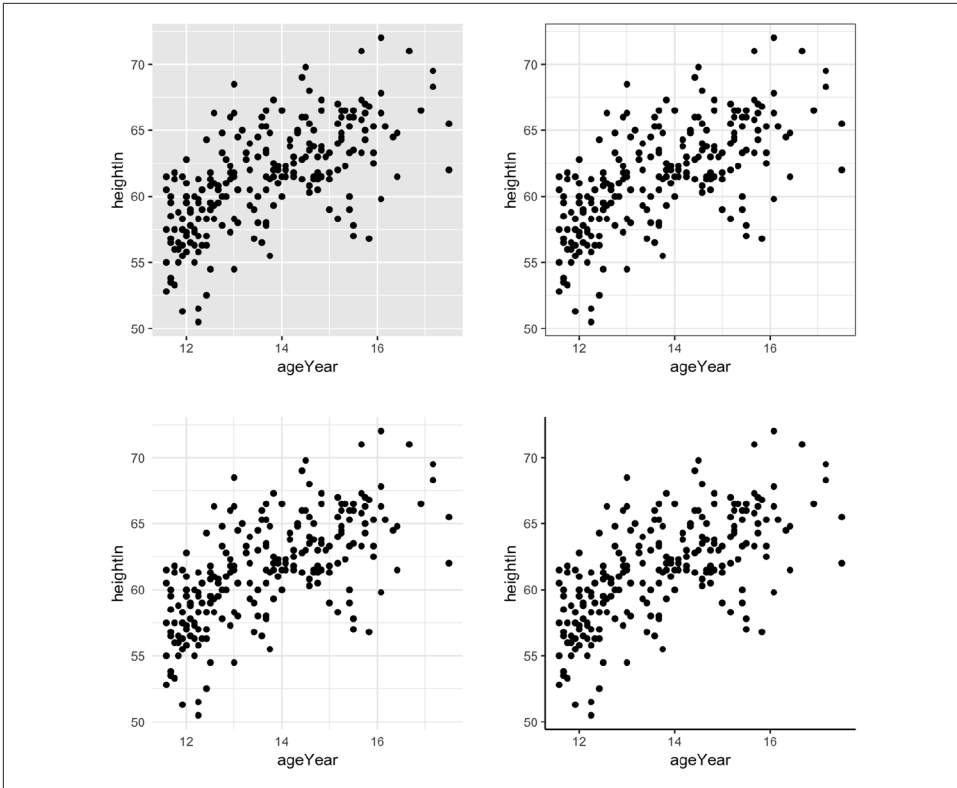


Figure 9-7. Scatter plot with `theme_grey()` (the default, top left); With `theme_bw()` (top right); With `theme_minimal()` (bottom left); With `theme_classic()` (bottom right)

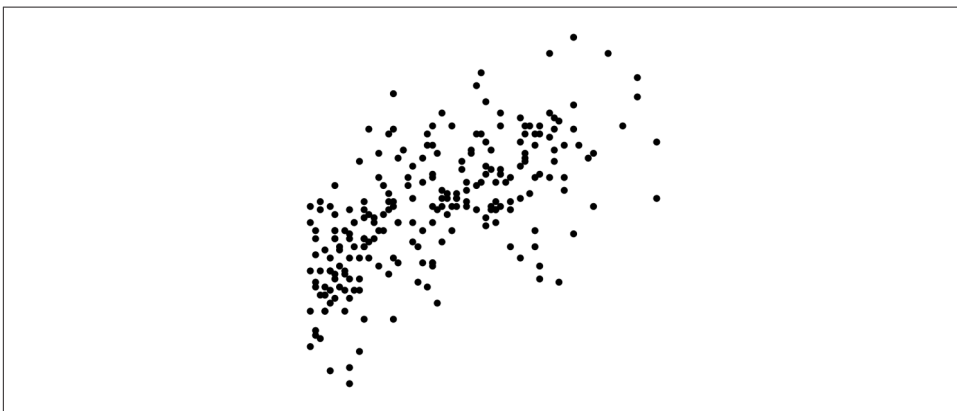


Figure 9-8. Scatter plot with `theme_void()`

Discussion

Some commonly used properties of theme elements in ggplot2 are those things that are controlled by `theme()`. Most of these things, like the title, legend, and axes, are outside the plot area, but some of them are inside the plot area, such as grid lines and the background coloring.

Besides the themes included in ggplot2, it is also possible to create your own.

You can set the base font family and size with either of the included themes (the default base font family is Helvetica, and the default size is 12):

```
hw_plot +  
  theme_grey(base_size = 16, base_family = "Times")
```

You can set the default theme for the current R session with `theme_set()`, although it's generally not a good idea to set options globally because it may affect other plots that are unrelated to your current project:

```
# Set default theme for current session  
theme_set(theme_bw())  
  
# This will use theme_bw()  
hw_plot  
  
# Reset the default theme back to theme_grey()  
theme_set(theme_grey())
```

See Also

To see additional complete themes included in ggplot2, see <https://ggplot2.tidyverse.org/reference/ggtheme.html>.

To modify a theme, see [Recipe 9.4](#).

To create your own themes, see [Recipe 9.5](#).

See `?theme` to see all the available theme properties.

9.4 Changing the Appearance of Theme Elements

Problem

You want to change the appearance of theme elements.

Solution

To modify a theme, add `theme()` with a corresponding `element_xx` object. These include `element_line`, `element_rect`, and `element_text`. The following code shows how to modify many of the commonly used theme properties (Figure 9-9):

```
library(gcookbook) # Load gcookbook for the heightweight data set

# Create the base plot
hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn, colour = sex)) +
  geom_point()

# Options for the plotting area
hw_plot +
  theme(
    panel.grid.major = element_line(colour = "red"),
    panel.grid.minor = element_line(colour = "red", linetype = "dashed",
                                     size = 0.2),
    panel.background = element_rect(fill = "lightblue"),
    panel.border = element_rect(colour = "blue", fill = NA, size = 2)
  )

# Options for the legend
hw_plot +
  theme(
    legend.background = element_rect(fill = "grey85", colour = "red", size = 1),
    legend.title = element_text(colour = "blue", face = "bold", size = 14),
    legend.text = element_text(colour = "red"),
    legend.key = element_rect(colour = "blue", size = 0.25)
  )

# Options for text items
hw_plot +
  ggtitle("Plot title here") +
  theme(
    axis.title.x = element_text(colour = "red", size = 14),
    axis.text.x = element_text(colour = "blue"),
    axis.title.y = element_text(colour = "red", size = 14, angle = 90),
    axis.text.y = element_text(colour = "blue"),
    plot.title = element_text(colour = "red", size = 20, face = "bold")
  )

# Options for facets
hw_plot +
  facet_grid(sex ~ .) +
  theme(
    strip.background = element_rect(fill = "pink"),
    strip.text.y = element_text(size = 14, angle = -90, face = "bold")
  ) # strip.text.x is the same, but for horizontal facets
```

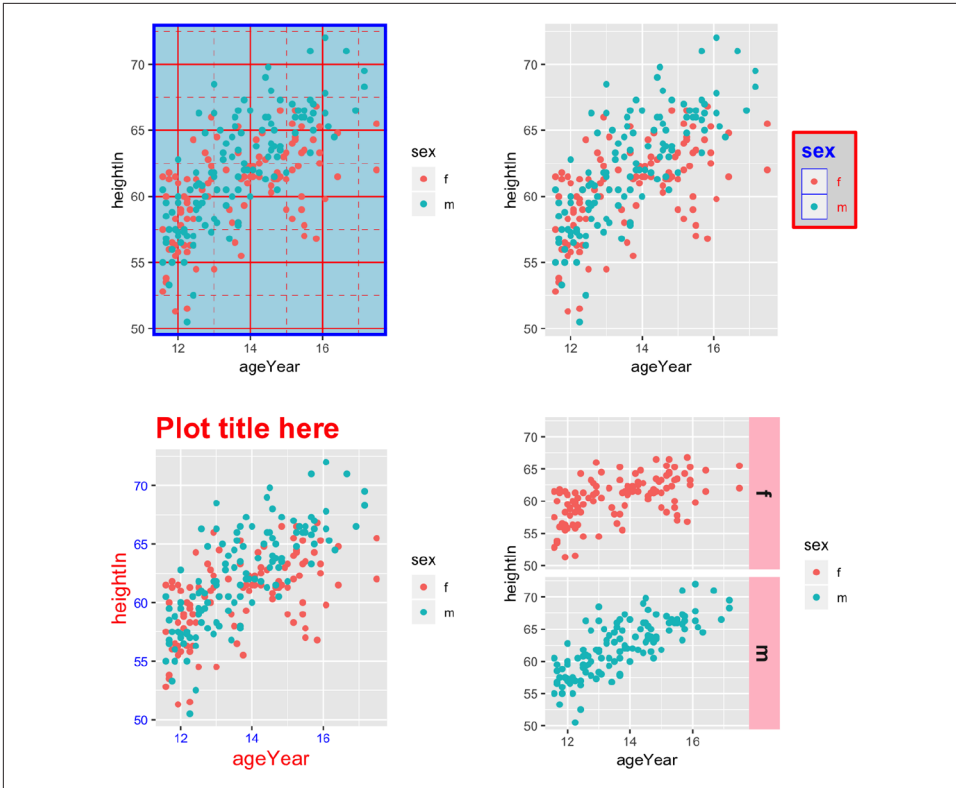


Figure 9-9. Clockwise from top left: modifying theme properties for the plotting area, the legend, the facets, and the text items

Discussion

If you want to use a saved theme and tweak a few parts of it with `theme()`, the `theme()` must come after the theme specification. Otherwise, anything set by `theme()` will be unset by the theme you add:

```
# theme() has no effect if before adding a complete theme
hw_plot +
  theme(axis.title.x = element_text(colour = "red")) +
  theme_bw()

# theme() works if after a complete theme
hw_plot +
  theme_bw() +
  theme(axis.title.x = element_text(colour = "red", size = 12))
```

Many of the commonly used theme properties are shown in [Table 9-3](#).

Table 9-3. Theme items that control text appearance in `theme()`

Name	Description	Element type
<code>text</code>	All text elements	<code>element_text()</code>
<code>rect</code>	All rectangular elements	<code>element_rect()</code>
<code>line</code>	All line elements	<code>element_line()</code>
<code>axis.line</code>	Lines along axes	<code>element_line()</code>
<code>axis.title</code>	Appearance of both axis labels	<code>element_text()</code>
<code>axis.title.x</code>	X-axis label appearance	<code>element_text()</code>
<code>axis.title.y</code>	Y-axis label appearance	<code>element_text()</code>
<code>axis.text</code>	Appearance of tick labels on both axes	<code>element_text()</code>
<code>axis.text.x</code>	X-axis tick label appearance	<code>element_text()</code>
<code>axis.text.y</code>	Y-axis tick label appearance	<code>element_text()</code>
<code>legend.background</code>	Background of legend	<code>element_rect()</code>
<code>legend.text</code>	Legend item appearance	<code>element_text()</code>
<code>legend.title</code>	Legend title appearance	<code>element_text()</code>
<code>legend.position</code>	Position of the legend	"left", "right", "bottom", "top", or two-element numeric vector if you wish to place it inside the plot area (for more on legend placement, see Recipe 10.2)
<code>panel.background</code>	Background of plotting area	<code>element_rect()</code>
<code>panel.border</code>	Border around plotting area	<code>element_rect(line type="dashed")</code>
<code>panel.grid.major</code>	Major grid lines	<code>element_line()</code>
<code>panel.grid.major.x</code>	Major grid lines, vertical	<code>element_line()</code>
<code>panel.grid.major.y</code>	Major grid lines, horizontal	<code>element_line()</code>
<code>panel.grid.minor</code>	Minor grid lines	<code>element_line()</code>
<code>panel.grid.minor.x</code>	Minor grid lines, vertical	<code>element_line()</code>
<code>panel.grid.minor.y</code>	Minor grid lines, horizontal	<code>element_line()</code>
<code>plot.background</code>	Background of the entire plot	<code>element_rect(fill = "white", colour = NA)</code>
<code>plot.title</code>	Title text appearance	<code>element_text()</code>
<code>strip.background</code>	Background of facet labels	<code>element_rect()</code>
<code>strip.text</code>	Text appearance for vertical and horizontal facet labels	<code>element_text()</code>
<code>strip.text.x</code>	Text appearance for horizontal facet labels	<code>element_text()</code>
<code>strip.text.y</code>	Text appearance for vertical facet labels	<code>element_text()</code>

9.5 Creating Your Own Themes

Problem

You want to create your own theme.

Solution

You can create your own theme by adding elements to an existing theme (Figure 9-10):

```
library(gcookbook) # Load gcookbook for the heightweight data set

# Start with theme_bw() and modify a few things
mytheme <- theme_bw() +
  theme(
    text = element_text(colour = "red"),
    axis.title = element_text(size = rel(1.25))
  )

# Create the base plot
hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn)) +
  geom_point()

# Plot with the modified theme we created
hw_plot +
  mytheme
```

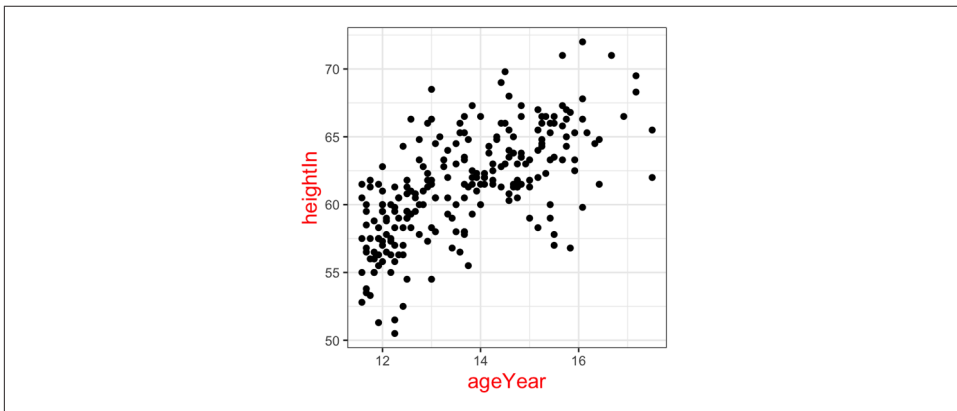


Figure 9-10. A modified default theme

Discussion

With ggplot2, you can not only make use of the default themes, but also modify these themes to suit your needs. You can add new theme elements or change the values of existing ones, and apply your changes globally or to a single plot.

See Also

The options for modifying themes are listed in [Recipe 9.4](#).

9.6 Hiding Grid Lines

Problem

You want to hide the grid lines in a plot.

Solution

The major grid lines (those that align with the tick marks) are controlled with `panel.grid.major`. The minor grid lines (the ones between the major lines) are controlled with `panel.grid.minor`. This will hide them both, as shown in [Figure 9-11](#) (left):

```
library(gcookbook) # Load gcookbook for the heightweight data set

hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn)) +
  geom_point()

hw_plot +
  theme(
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank()
  )
```

Discussion

It's possible to hide just the vertical or horizontal grid lines, as shown in the middle graph and the righthand graph in [Figure 9-11](#), with `panel.grid.major.x`, `panel.grid.major.y`, `panel.grid.minor.x`, and `panel.grid.minor.y`:

```
# Hide the vertical grid lines (which intersect with the x-axis)
hw_plot +
  theme(
    panel.grid.major.x = element_blank(),
    panel.grid.minor.x = element_blank()
  )

# Hide the horizontal grid lines (which intersect with the y-axis)
hw_plot +
  theme(
    panel.grid.major.y = element_blank(),
    panel.grid.minor.y = element_blank()
  )
```

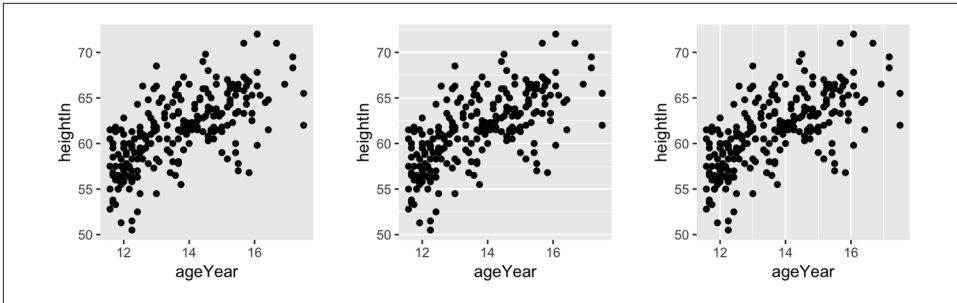


Figure 9-11. No grid lines (left); No vertical lines (middle); No horizontal lines (right)

Like the x- or y-axis, a legend is a guide: it shows people how to map visual (aesthetic) properties back to data values.

10.1 Removing the Legend

Problem

You want to remove the legend from a graph.

Solution

Use `guides()`, and specify the scale that should have its legend removed (Figure 10-1):

```
# Create the base plot (with legend)
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight, fill = group)) +
  geom_boxplot()

pg_plot

# Remove the legend for fill
pg_plot +
  guides(fill = FALSE)
```

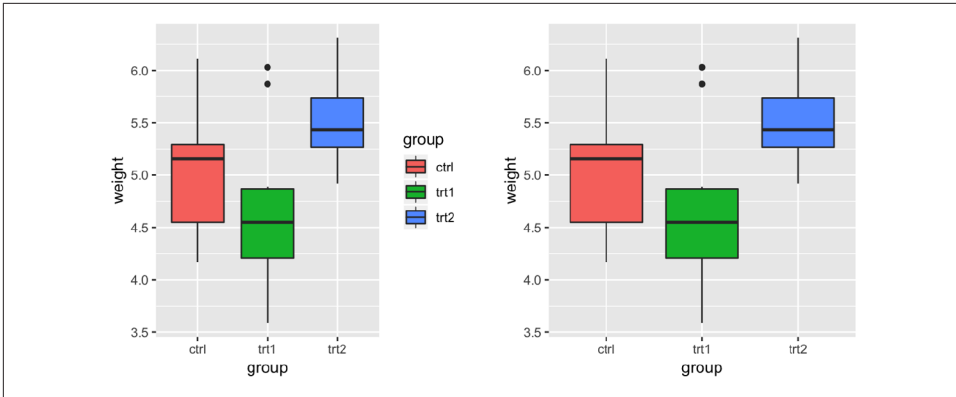


Figure 10-1. Default appearance (left); With legend removed (right)

Discussion

Another way to remove a legend is to set `guide = FALSE` in the scale. This will result in the exact same output as the preceding code:

```
# Remove the legend for fill
pg_plot +
  scale_fill_discrete(guide = FALSE)
```

Yet another way to remove the legend is to use the theming system. If you have more than one aesthetic mapping with a legend (color and shape, for example), this will remove legends for all of them:

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

Sometimes a legend is redundant, or it is supplied in another graph that will be displayed with the current one. In these cases, it can be useful to remove the legend from a graph.

In the example used here, the colors provide the same information that is on the x-axis, so the legend is unnecessary. Notice that with the legend removed, the area used for graphing the data is larger. If you want to achieve the same proportions in the graphing area, you will need to adjust the overall dimensions of the graph.

When a variable is mapped to fill, the default scale used is `scale_fill_discrete()` (equivalent to `scale_fill_hue()`), which maps the factor levels to colors that are equally spaced around the color wheel. There are other scales for fill, such as `scale_fill_manual()`. If you use scales for other aesthetics, such as `colour` (for lines and points) or `shape` (for points), you must use the appropriate scale. Commonly used scales include:

- `scale_fill_discrete()`
- `scale_fill_hue()`
- `scale_fill_manual()`
- `scale_fill_grey()`
- `scale_fill_brewer()`
- `scale_colour_discrete()`
- `scale_colour_hue()`
- `scale_colour_manual()`
- `scale_colour_grey()`
- `scale_colour_brewer()`
- `scale_shape_manual()`
- `scale_linetype()`

10.2 Changing the Position of a Legend

Problem

You want to move the legend from its default place on the right side.

Solution

Use `theme(legend.position = ...)`. It can be put on the top, left, right, or bottom by using one of those strings as the position (Figure 10-2, left).

```
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight, fill = group)) +
  geom_boxplot() +
  scale_fill_brewer(palette = "Pastel2")

pg_plot +
  theme(legend.position = "top")
```

The legend can also be placed inside the plotting area by specifying a coordinate position, as in `legend.position = c(.8, .3)` (Figure 10-2, right). The coordinate space starts at (0, 0) in the bottom left and goes to (1, 1) in the top right.

```
pg_plot +
  theme(legend.position = c(.8, .3))
```

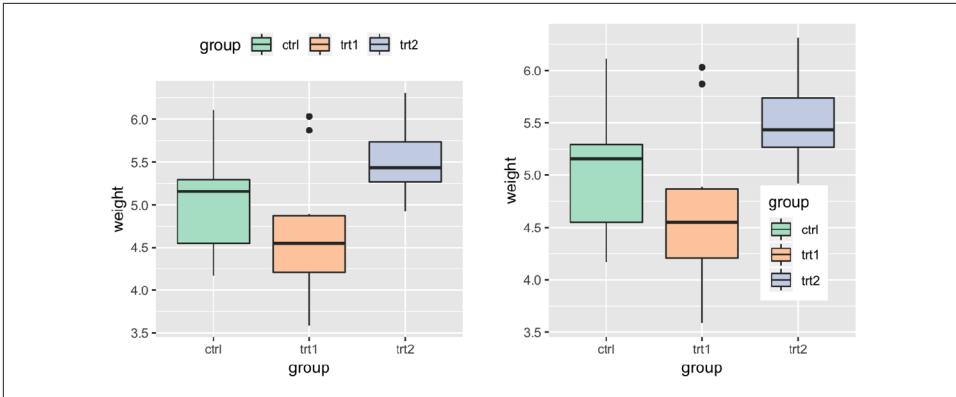


Figure 10-2. Legend on top (left); Legend inside of plotting area (right)

Discussion

You can also use `legend.justification` to set which *part* of the legend box is set to the position at `legend.position`. By default, the center of the legend (.5, .5) is placed at the coordinate, but it is often useful to specify a different point.

For example, this will place the bottom-right corner of the legend (1, 0) in the bottom-right corner of the plotting area (1, 0) (Figure 10-3, left):

```
pg_plot +
  theme(legend.position = c(1, 0), legend.justification = c(1, 0))
```

And this will place the top-right corner of the legend in the top-right corner of the graphing area (Figure 10-3, right):

```
pg_plot +
  theme(legend.position = c(1, 1), legend.justification = c(1, 1))
```

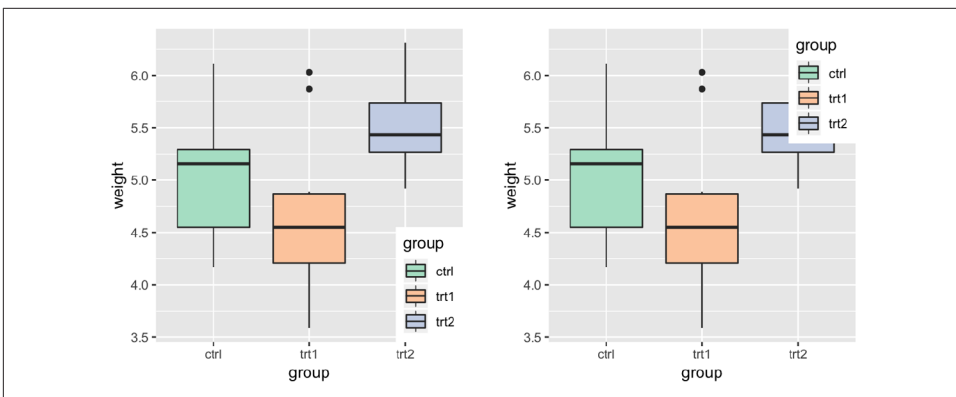


Figure 10-3. Legend in bottom-right corner (left); Legend in top-right corner (right)

When placing the legend inside of the graphing area, it may be helpful to add an opaque border to set it apart (Figure 10-4, left):

```
pg_plot +
  theme(legend.position = c(.85, .2)) +
  theme(legend.background = element_rect(fill = "white", colour = "black"))
```

You can also remove the border around its elements so that it blends in (Figure 10-4, right):

```
pg_plot +
  theme(legend.position = c(.85, .2)) +
  theme(legend.background = element_blank()) + # Remove overall border
  theme(legend.key = element_blank())         # Remove border around each item
```

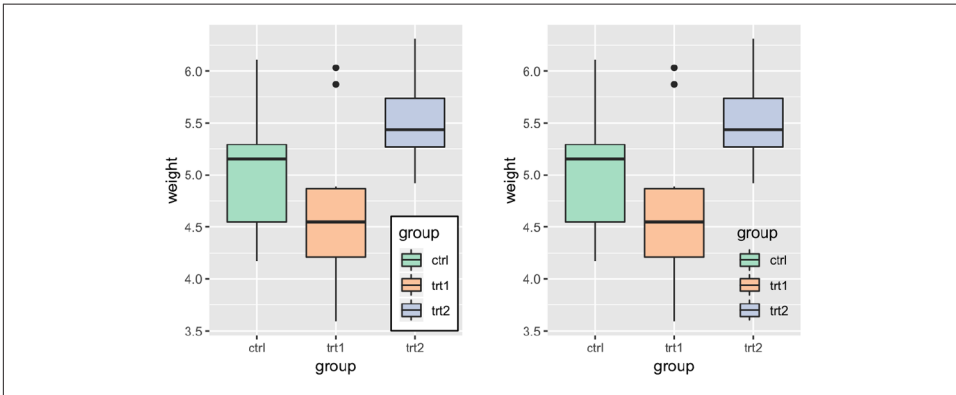


Figure 10-4. Legend with opaque background and outline (left); With no background or outlines (right)

10.3 Changing the Order of Items in a Legend

Problem

You want to change the order of the items in a legend.

Solution

Set the limits in the scale to the desired order (Figure 10-5):

```
# Create the base plot
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight, fill = group)) +
  geom_boxplot()

pg_plot

# Change the order of items
```

```
pg_plot +
  scale_fill_discrete(limits = c("trt1", "trt2", "ctrl"))
```

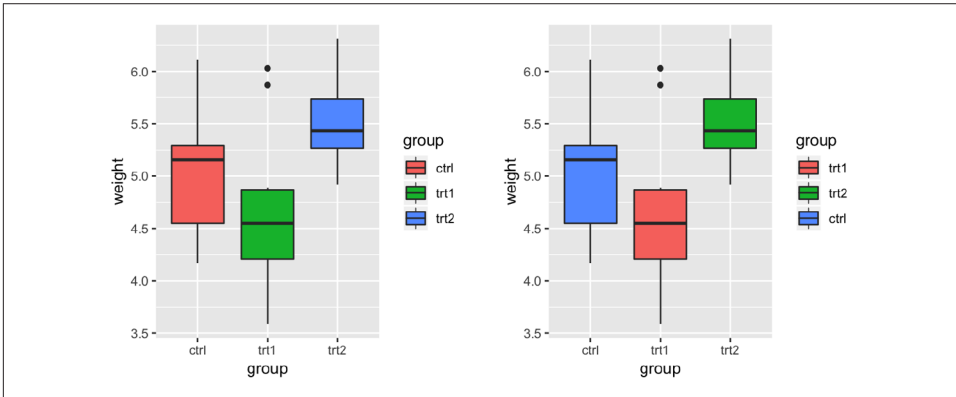


Figure 10-5. Default order for legend (left); Modified order (right)

Discussion

Note that the order of the items on the x-axis did not change. To do that, you would have to set the limits of `scale_x_discrete()` (Recipe 8.4), or change the data to have a different factor level order (Recipe 15.8).

In the preceding example, `group` was mapped to the `fill` aesthetic. By default this uses `scale_fill_discrete()` (which is the same as `scale_fill_hue()`), which maps the factor levels to colors that are equally spaced around the color wheel. We could have used a different `scale_fill_xxx()`, though. For example, we could use a grey palette (Figure 10-6, left):

```
pg_plot +
  scale_fill_grey(start = .5, end = 1, limits = c("trt1", "trt2", "ctrl"))
```

Or we could use a palette from RColorBrewer (Figure 10-6, right):

```
pg_plot +
  scale_fill_brewer(palette = "Pastel2", limits = c("trt1", "trt2", "ctrl"))
```

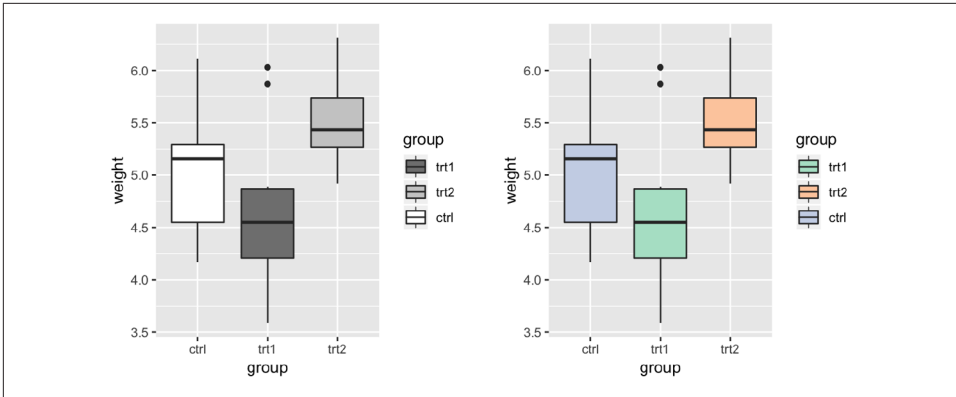


Figure 10-6. Modified order with a grey palette (left); With a palette from RColorBrewer (right)

All the previous examples were for fill. If you use scales for other aesthetics, such as colour (for lines and points) or shape (for points), you must use the appropriate scale. Commonly used scales include:

- `scale_fill_discrete()`
- `scale_fill_hue()`
- `scale_fill_manual()`
- `scale_fill_grey()`
- `scale_fill_brewer()`
- `scale_colour_discrete()`
- `scale_colour_hue()`
- `scale_colour_manual()`
- `scale_colour_grey()`
- `scale_colour_brewer()`
- `scale_shape_manual()`
- `scale_linetype()`

By default, using `scale_fill_discrete()` is equivalent to using `scale_fill_hue()`; the same is true for color scales.

See Also

To reverse the order of the legend, see [Recipe 10.4](#).

To change the order of factor levels, see [Recipe 15.8](#). To order legend items based on values in another variable, see [Recipe 15.9](#).

10.4 Reversing the Order of Items in a Legend

Problem

You want to reverse the order of items in a legend.

Solution

Add `guides(fill = guide_legend(reverse = TRUE))` to reverse the order of the legend, as in [Figure 10-7](#) (for other aesthetics, replace `fill` with the name of the aesthetic, such as `colour` or `size`):

```
# Create the base plot
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight, fill = group)) +
  geom_boxplot()

pg_plot

# Reverse the legend order
pg_plot +
  guides(fill = guide_legend(reverse = TRUE))
```

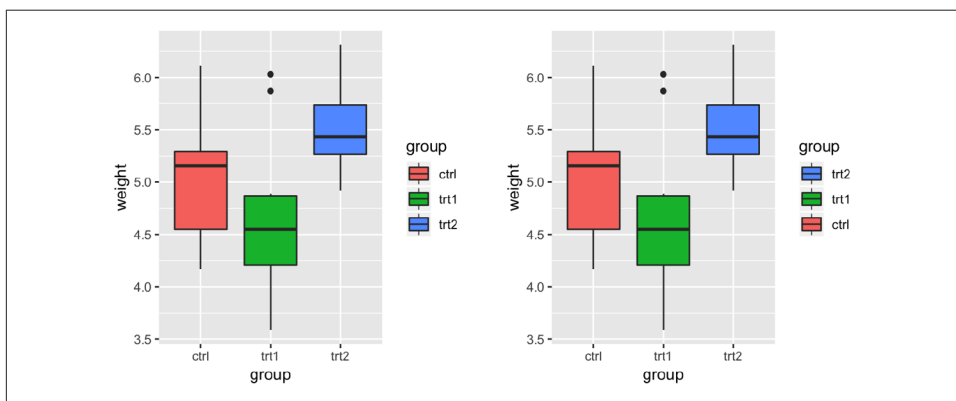


Figure 10-7. Default order for legend (left); Reversed order (right)

Discussion

It is also possible to control the legend when specifying the scale, as in the following:

```
scale_fill_hue(guide = guide_legend(reverse = TRUE))
```


10.5 Changing a Legend Title

Problem

You want to change the text of a legend title.

Solution

Use `labs()` and set the value of `fill`, `colour`, `shape`, or whatever aesthetic is appropriate for the legend (Figure 10-8):

```
# Create the base plot
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight, fill = group)) +
  geom_boxplot()

pg_plot

# Set the legend title to "Condition"
pg_plot + labs(fill = "Condition")
```

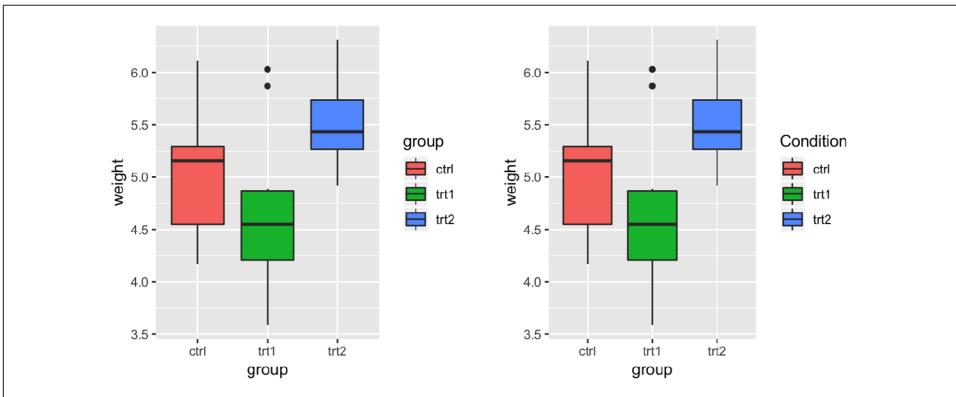


Figure 10-8. With the legend title set to “Condition”

Discussion

It’s also possible to set the title of the legend in the scale specification. Since legends and axes are both guides, this works the same way as setting the title of the x- or y-axis.

This would have the same effect as the previous code:

```
pg_plot + scale_fill_discrete(name = "Condition")
```

If there are multiple variables mapped to aesthetics with a legend (those other than x and y), you can set the title of each individually. In the example here we’ll use `\n` to add a line break in one of the titles (Figure 10-9):

```
library(gcookbook) # Load gcookbook for the heightweight data set

# Load gcookbook for the heightweight data set
hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn, colour = sex)) +
  geom_point(aes(size = weightLb)) +
  scale_size_continuous(range = c(1, 4))

hw_plot

# With new legend titles
hw_plot +
  labs(colour = "Male/Female", size = "Weight\n(pounds)")
```

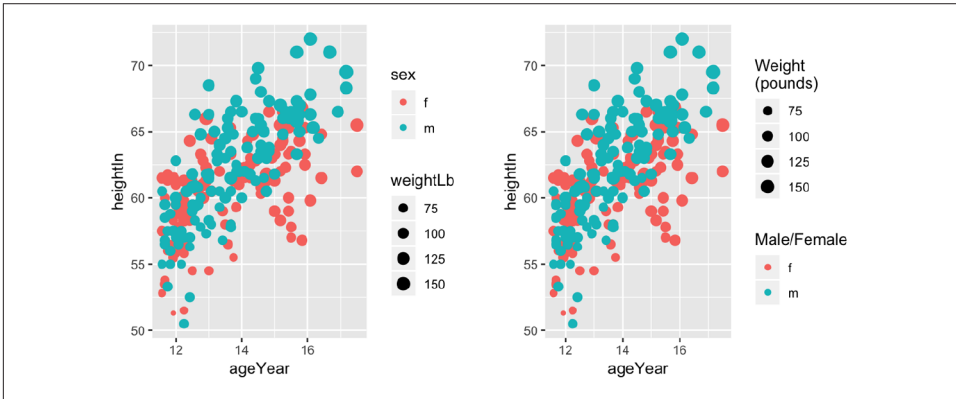


Figure 10-9. Two legends with original titles (left); With new titles (right)

If you have one variable mapped to two separate aesthetics, the default is to have a single legend that combines both. For example, if we map sex to both shape and weight, there will be just one legend (Figure 10-10, left):

```
hw_plot2 <- ggplot(heightweight, aes(x = ageYear, y = heightIn,
                                     shape = sex, colour = sex)) +
  geom_point()

hw_plot2
```

To change the title (Figure 10-10, right), you need to set the name for both of them. If you change the name for just one, it will result in two separate legends (Figure 10-10, middle):

```
# Change just shape
hw_plot2 +
  labs(shape = "Male/Female")

# Change both shape and colour
hw_plot2 +
  labs(shape = "Male/Female", colour = "Male/Female")
```

It is also possible to control the legend title with the `guides()` function. It's a little more verbose, but it can be useful when you're already using it to control other properties:

```
hw_plot +  
  guides(fill = guide_legend(title = "Condition"))
```

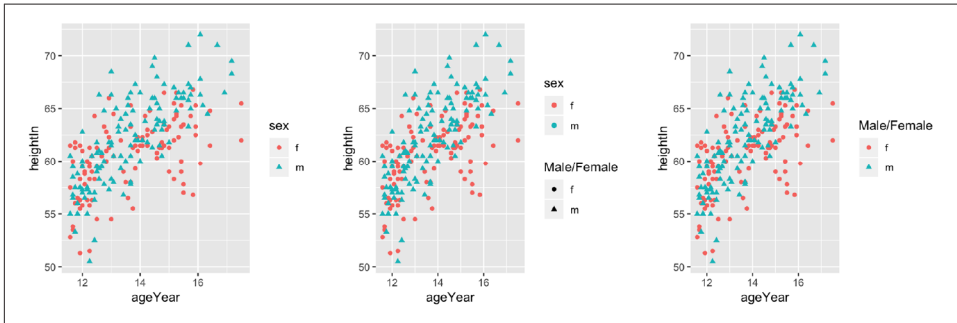


Figure 10-10. Default legend with a variable mapped to shape and colour (left); With shape renamed (middle); With both shape and colour renamed (right)

10.6 Changing the Appearance of a Legend Title

Problem

You want to change the appearance of a legend title's text.

Solution

Use `theme(legend.title = element_text())` (Figure 10-11):

```
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight, fill = group)) +  
  geom_boxplot()  
  
pg_plot +  
  theme(legend.title = element_text(  
    face = "italic",  
    family = "Times",  
    colour = "red",  
    size = 14)  
  )
```

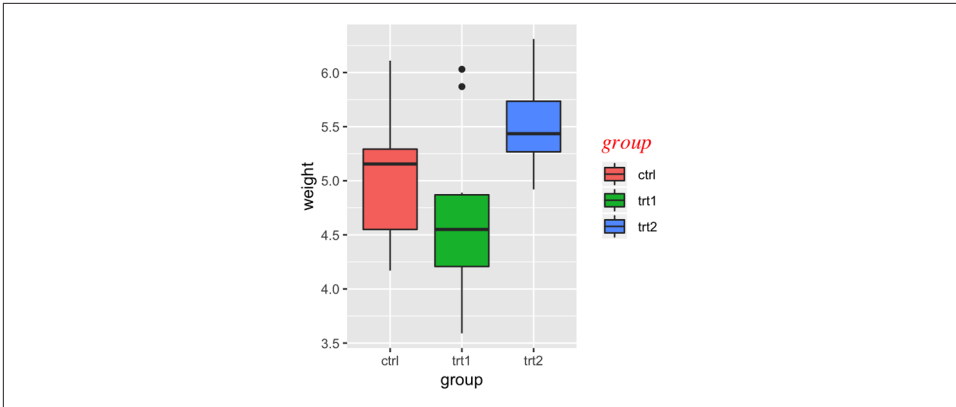


Figure 10-11. Customized legend title appearance

Discussion

It's also possible to specify the legend title's appearance via `guides()`, but this method can be a bit verbose. This has the same effect as the previous code:

```
pg_plot +
  guides(fill = guide_legend(title.theme = element_text(
    face = "italic",
    family = "times",
    colour = "red",
    size = 14))
)
```

See Also

See [Recipe 9.2](#) for more on controlling the appearance of text.

10.7 Removing a Legend Title

Problem

You want to remove a legend title.

Solution

Add `guides(fill = guide_legend(title = NULL))` to remove the title from a legend, as in [Figure 10-12](#) (for other aesthetics, replace `fill` with the name of the aesthetic, such as `colour` or `size`):

```
ggplot(PlantGrowth, aes(x = group, y = weight, fill = group)) +
  geom_boxplot() +
  guides(fill = guide_legend(title = NULL))
```

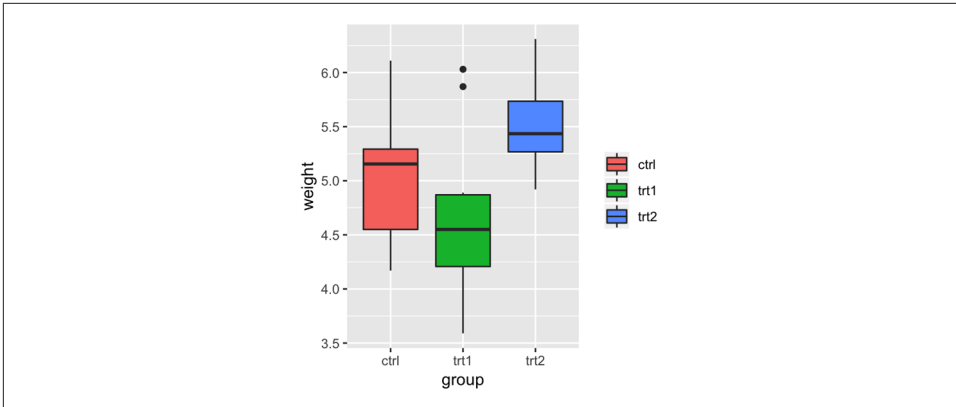


Figure 10-12. Box plot with no legend title

Discussion

It is also possible to control the legend title when specifying the scale. This has the same effect as the preceding code:

```
scale_fill_hue(guide = guide_legend(title = NULL))
```

10.8 Changing the Labels in a Legend

Problem

You want to change the text of labels in a legend.

Solution

Set the labels in the scale (Figure 10-13, left):

```
library(gcookbook) # Load gcookbook for the PlantGrowth data set

# The base plot
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight, fill = group)) +
  geom_boxplot()

# Change the legend labels
pg_plot +
  scale_fill_discrete(labels = c("Control", "Treatment 1", "Treatment 2"))
```

Discussion

Note that the labels on the x-axis did not change. To do that, you would have to set the labels of `scale_x_discrete()` (Recipe 8.10), or change the data to have different factor level names (Recipe 15.10).

In the preceding example, group was mapped to the fill aesthetic. By default this uses `scale_fill_discrete()`, which maps the factor levels to colors that are equally spaced around the color wheel (the same as `scale_fill_hue()`). There are other fill scales we could use, and setting the labels works the same way. For example, to produce the graph on the right in **Figure 10-13**:

```
pg_plot +
  scale_fill_grey(start = .5, end = 1,
                 labels = c("Control", "Treatment 1", "Treatment 2"))
```

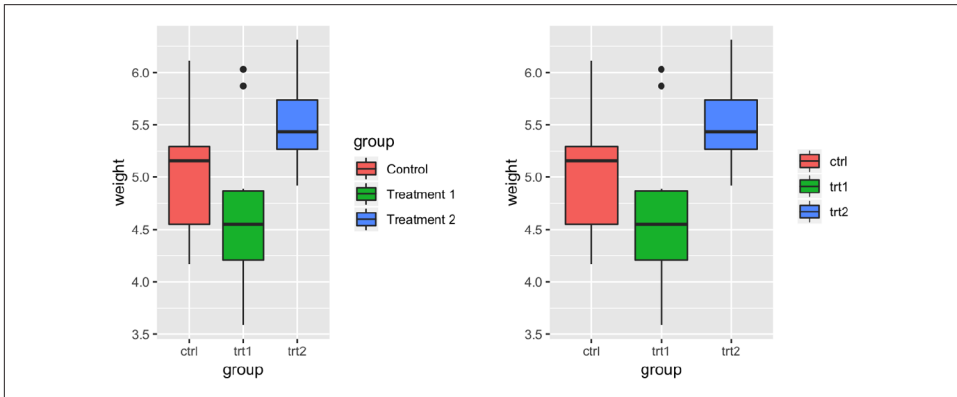


Figure 10-13. Manually specified legend labels with the default discrete scale (left); Manually specified labels with a different scale (right)

If you are also changing the order of items in the legend, the labels are matched to the items by position. In this example we'll change the item order, and make sure to set the labels in the same order (**Figure 10-14**):

```
pg_plot +
  scale_fill_discrete(
    limits = c("trt1", "trt2", "ctrl"),
    labels = c("Treatment 1", "Treatment 2", "Control")
  )
```

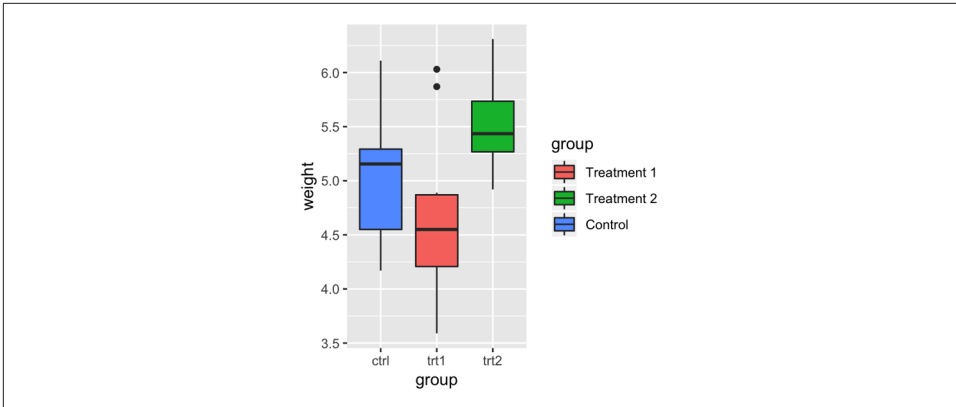


Figure 10-14. Modified legend label order and manually specified labels (note that the x-axis labels and their order are unchanged)

If you have one variable mapped to two separate aesthetics, the default is to have a single legend that combines both. If you want to change the legend labels, you must change them for both scales; otherwise you will end up with two separate legends, as shown in Figure 10-15:

```
# Create the base plot
hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn, shape = sex,
                                     colour = sex)) +
  geom_point()

hw_plot

# Change the labels for one scale
hw_plot +
  scale_shape_discrete(labels = c("Female", "Male"))

# Change the labels for both scales
hw_plot +
  scale_shape_discrete(labels = c("Female", "Male")) +
  scale_colour_discrete(labels = c("Female", "Male"))
```

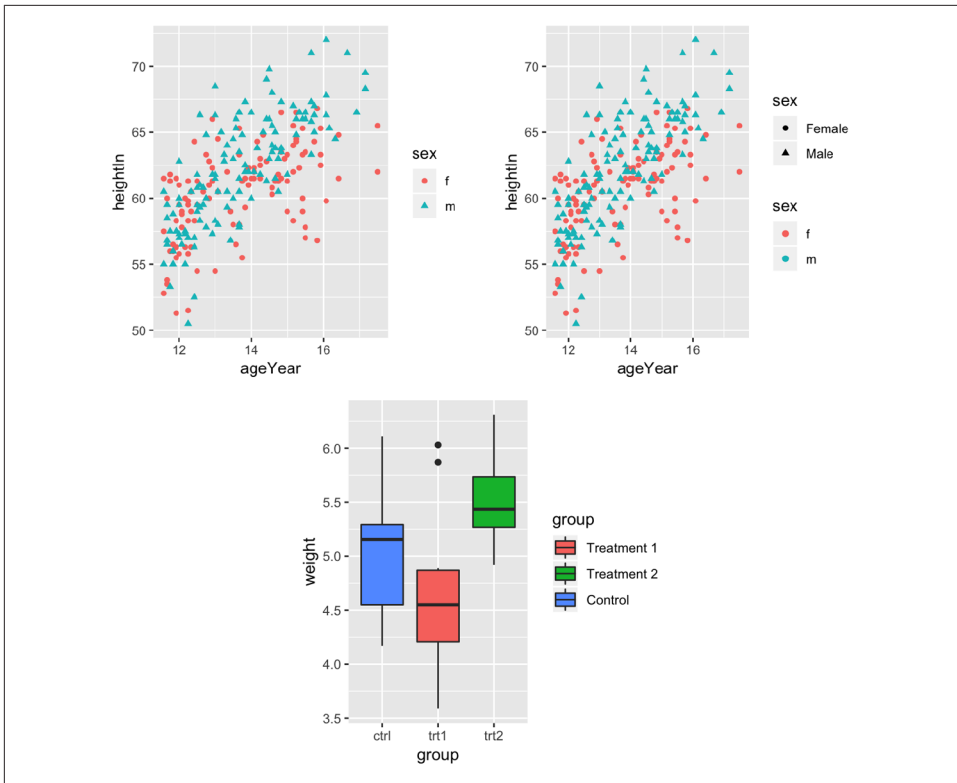


Figure 10-15. A variable mapped to shape and colour (top left); With new labels for shape (top right); With new labels combining both shape and colour (bottom)

Other commonly used scales with legends include:

- `scale_fill_discrete()`
- `scale_fill_hue()`
- `scale_fill_manual()`
- `scale_fill_grey()`
- `scale_fill_brewer()`
- `scale_colour_discrete()`
- `scale_colour_hue()`
- `scale_colour_manual()`
- `scale_colour_grey()`
- `scale_color_viridis_c()`

- `scale_color_viridis_d()`
- `scale_shape_manual()`
- `scale_linetype()`

By default, using `scale_fill_discrete()` is equivalent to using `scale_fill_hue()`; the same is true for color scales.

10.9 Changing the Appearance of Legend Labels

Problem

You want to change the appearance of labels in a legend.

Solution

Use `theme(legend.text=element_text())` (Figure 10-16):

```
# Create the base plot
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight, fill = group)) +
  geom_boxplot()

# Change the legend label appearance
pg_plot +
  theme(legend.text = element_text(
    face = "italic",
    family = "Times",
    colour = "red",
    size = 14)
  )
```

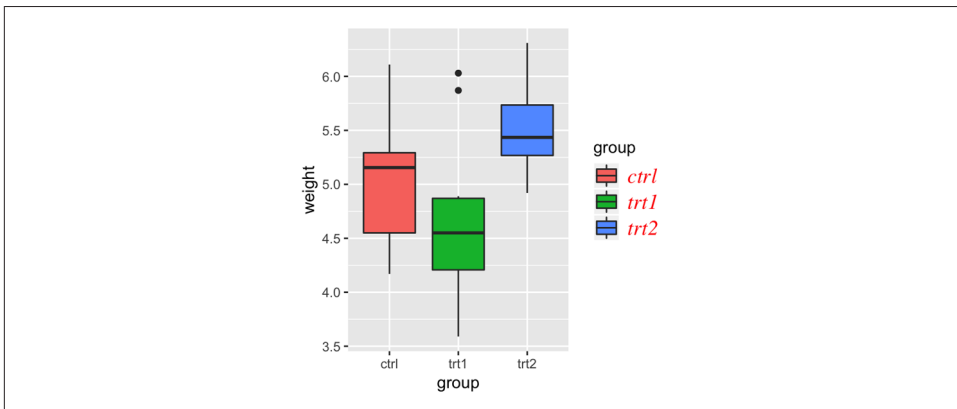


Figure 10-16. Customized legend label appearance

Discussion

It's also possible to specify the legend label appearance via `guides()`, although this method is a bit unwieldy. This has the same effect as the previous code:

```
# Changes the legend title text for the fill legend
pg_plot +
  guides(fill = guide_legend(title.theme = element_text(
    face = "italic",
    family = "times",
    colour = "red",
    size = 14))
  )
```

See Also

See [Recipe 9.2](#) for more on controlling the appearance of text.

10.10 Using Labels with Multiple Lines of Text

Problem

You want to use legend labels that have more than one line of text.

Solution

Set the labels in the scale, using `\n` to represent a newline. In this example, we'll use `scale_fill_discrete()` to control the legend for the `fill` scale ([Figure 10-17](#), left):

```
pg_plot <- ggplot(PlantGrowth, aes(x = group, y = weight, fill = group)) +
  geom_boxplot()

# Labels that have more than one line
pg_plot +
  scale_fill_discrete(labels = c("Control", "Type 1\n\treatment",
                                "Type 2\n\treatment"))
```

Discussion

As you can see in the version on the left in [Figure 10-17](#), with the default settings the lines of text will run into each other when you use labels that have more than one line. To deal with this problem, you can increase the height of the legend keys and decrease the spacing between lines, using `theme()` ([Figure 10-17](#), right). To do this, you will need to specify the height using the `unit()` function from the `grid` package:

```
library(grid)

pg_plot +
  scale_fill_discrete(labels = c("Control", "Type 1\n\treatment",
```

```

      "Type 2\\ntreatment")) +
theme(legend.text = element_text(lineheight = .8),
      legend.key.height = unit(1, "cm"))

```

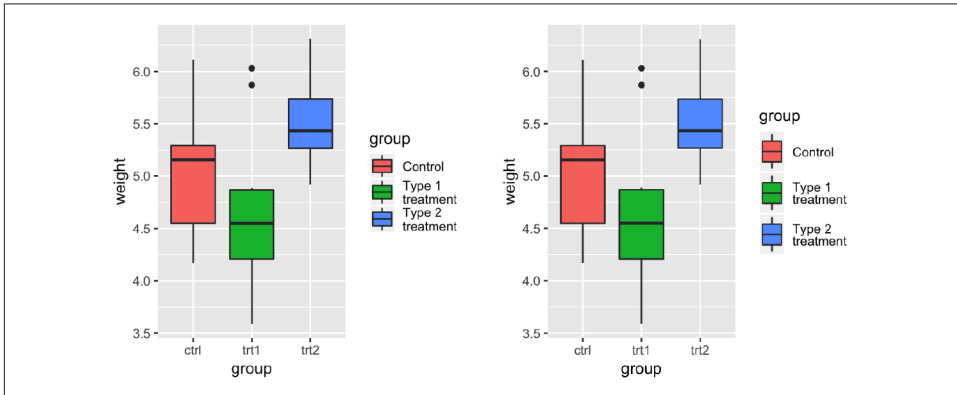


Figure 10-17. Multiline legend labels (left); With increased key height and reduced line spacing (right)

One of the most useful techniques in data visualization is rendering groups of data alongside each other, making it easy to compare the groups. With `ggplot2`, one way to do this is by mapping a discrete variable to an aesthetic, like *x* position, colour, or shape. Another way of doing this is to create a subplot for each group and draw the subplots side by side.

These kinds of plots are known as *Trellis* displays. They're implemented in the `lattice` package as well as in the `ggplot2` package. In `ggplot2`, they're called *facets*. In this chapter I'll explain how to use them.

11.1 Splitting Data into Subplots with Facets

Problem

You want to plot subsets of your data in separate panels.

Solution

Use `facet_grid()` or `facet_wrap()`, and specify the variables on which to split.

With `facet_grid()`, you can specify a variable to split the data into vertical subpanels, and another variable to split it into horizontal subpanels ([Figure 11-1](#)):

```
# Create the base plot
mpg_plot <- ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point()

# Faceted by drv, in vertically arranged subpanels
mpg_plot +
  facet_grid(drv ~ .)
```

```
# Faceted by cyl, in horizontally arranged subpanels
mpg_plot +
  facet_grid(. ~ cyl)

# Split by drv (vertical) and cyl (horizontal)
mpg_plot +
  facet_grid(drv ~ cyl)
```

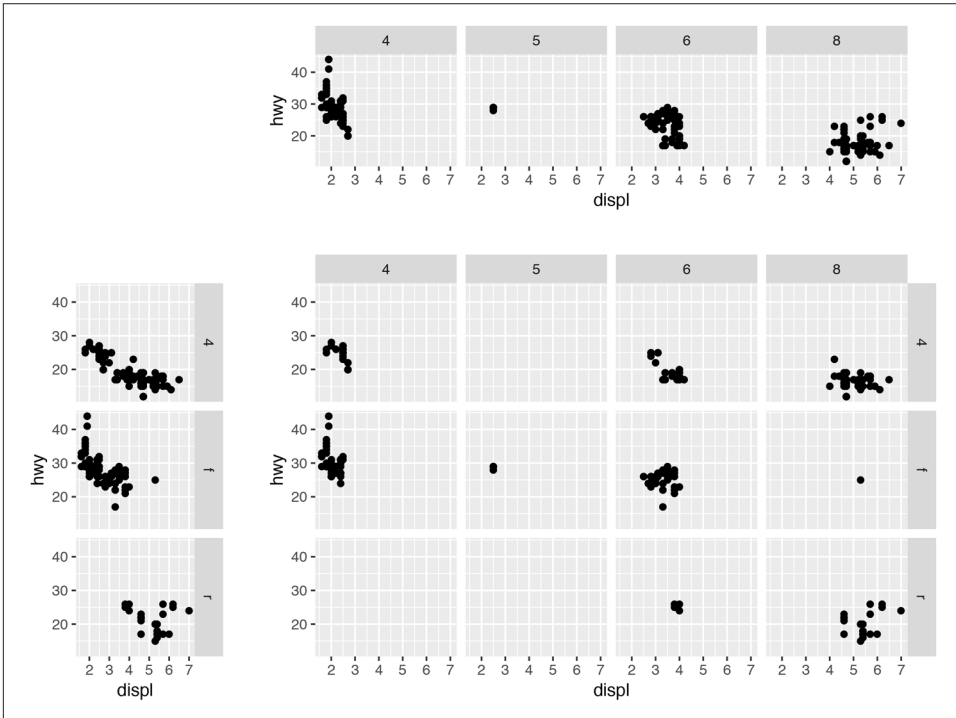


Figure 11-1. Faceting horizontally by *cyl* (top); faceting vertically by *drv* (left); faceting in both directions, with both variables (bottom right)

With `facet_wrap()`, the subplots are laid out horizontally and wrap around, like words on a page, as in [Figure 11-2](#):

```
# Facet on class
# Note that there is nothing before the tilde
mpg_plot +
  facet_wrap(~ class)
```

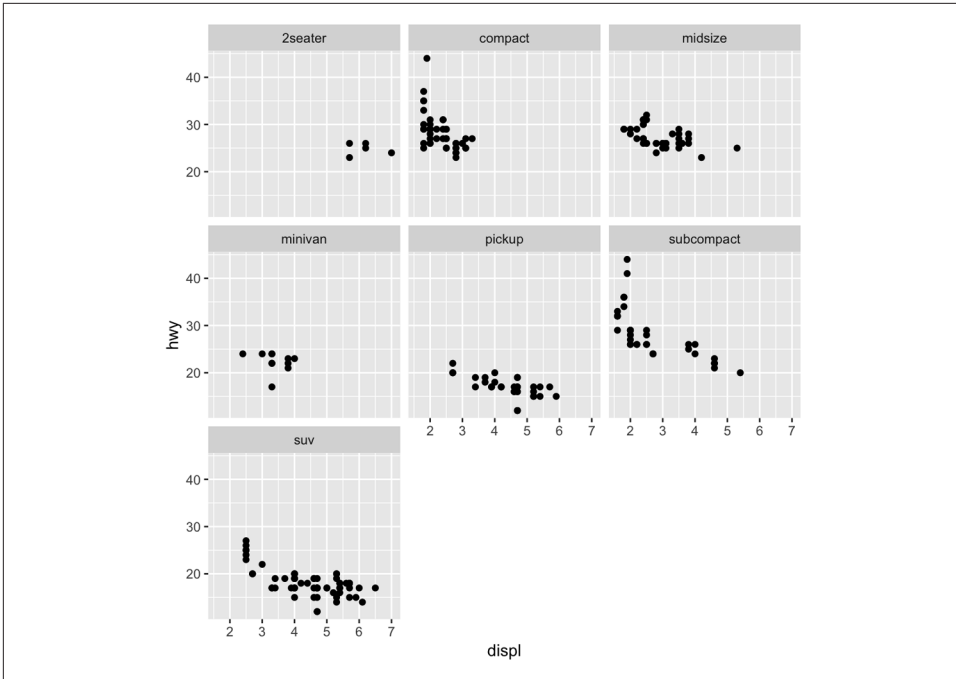


Figure 11-2. A scatter plot with `facet_wrap()` on `class`

Discussion

With `facet_wrap()`, the default is to use the same number of rows and columns. In **Figure 11-2**, there were seven facets, and they fit into a 3×3 square. To change this, you can pass a value for `nrow` or `ncol`:

```
# These will have the same result: 2 rows and 4 cols
mpg_plot +
  facet_wrap( ~ class, nrow = 2)

mpg_plot +
  facet_wrap( ~ class, ncol = 4)
```

The choice of faceting direction depends on the kind of comparison you would like to encourage. For example, if you want to compare heights of bars, it's useful to make the facets go horizontally. If, on the other hand, you want to compare the horizontal distribution of histograms, it makes sense to make the facets go vertically.

Sometimes both kinds of comparisons are important—there may not be a clear answer as to which faceting direction is best. It may turn out that displaying the groups in a single plot by mapping the grouping variable to an aesthetic like color works better than using facets. In these situations, you'll have to rely on your judgment.

11.2 Using Facets with Different Axes

Problem

You want subplots with different ranges or items on their axes.

Solution

Set the scales to "free_x", "free_y", or "free" (Figure 11-3):

```
# Create the base plot
mpg_plot <- ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point()

# With free y scales
mpg_plot +
  facet_grid(drv ~ cyl, scales = "free_y")

# With free x and y scales
mpg_plot +
  facet_grid(drv ~ cyl, scales = "free")
```

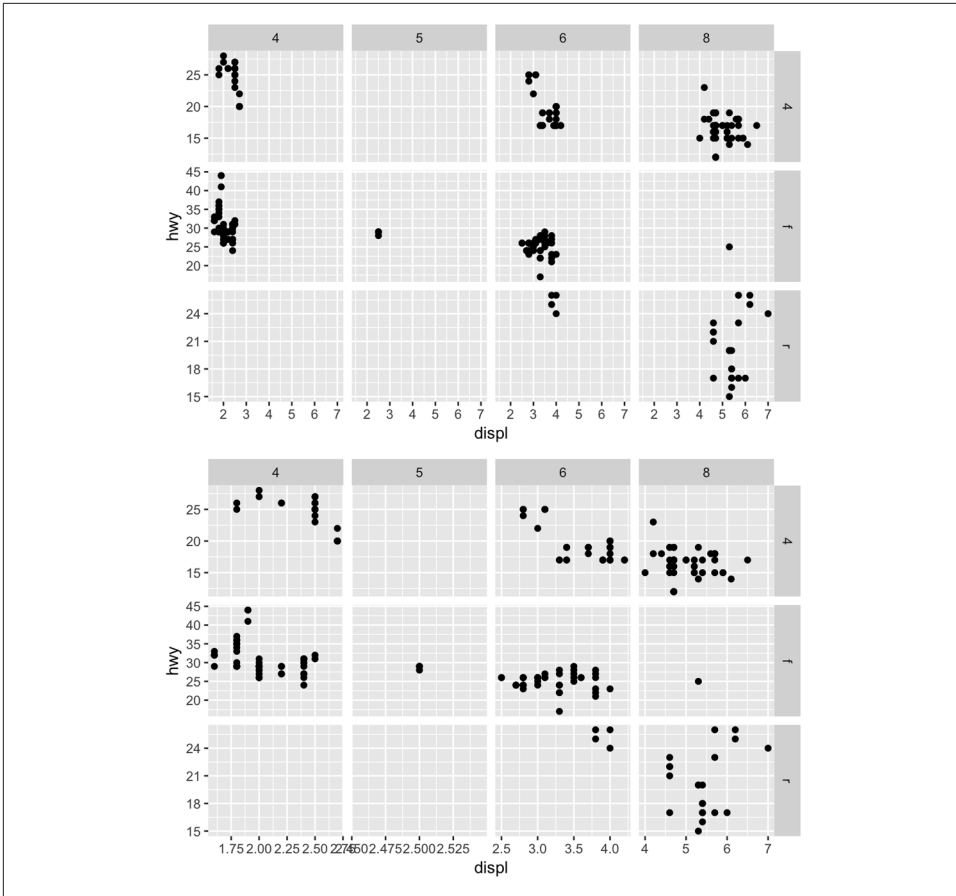



Figure 11-3. With free y scales (top); With free x and y scales (bottom)

Discussion

Each row of subplots has its own y range when free y scales are used; the same applies to columns when free x scales are used.

It's not possible to directly set the range of each row or column, but you can control the ranges by dropping unwanted data (to reduce the ranges), or by adding `geom_blank()` (to expand the ranges).

See Also

See [Recipe 3.10](#) for an example of faceting with free scales and a discrete axis.

11.3 Changing the Text of Facet Labels

Problem

You want to change the text of facet labels.

Solution

Change the names of the factor levels (Figure 11-4):

```
library(dplyr)

# Make a modified copy of the original data
mpg_mod <- mpg %>%
  # Rename 4 to 4wd, f to Front, r to Rear
  mutate(drv = recode(drv, "4" = "4wd", "f" = "Front", "r" = "Rear"))

# Plot the new data
ggplot(mpg_mod, aes(x = displ, y = hwy)) +
  geom_point() +
  facet_grid(drv ~ .)
```

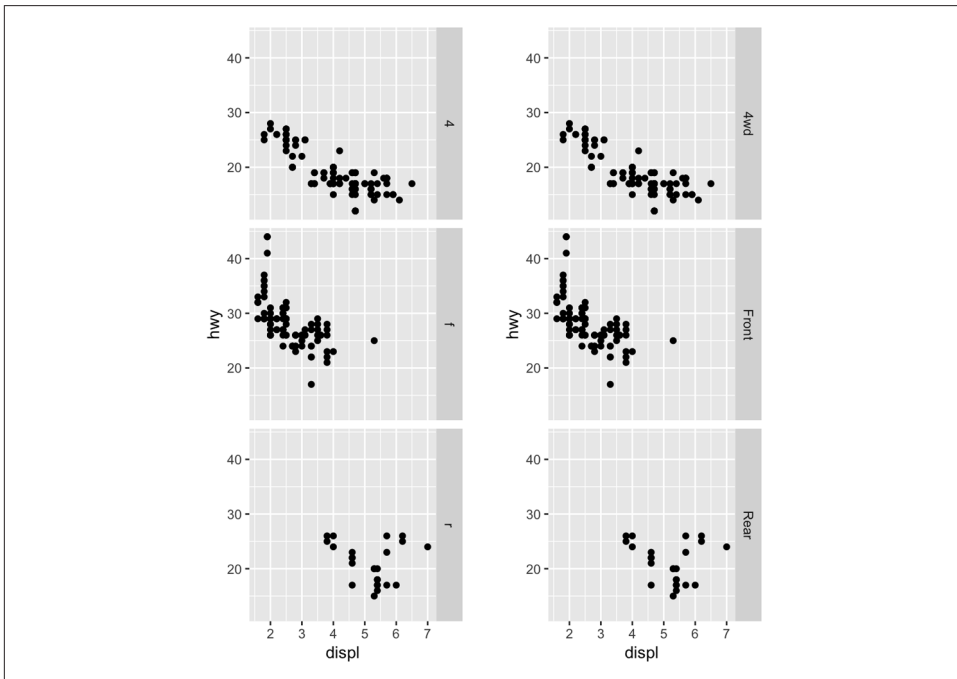


Figure 11-4. Default facet labels (left); Modified facet labels (right)

Discussion

Unlike with scales where you can set the labels, to set facet labels you must change the data values. Also, at the time of this writing, there is no way to show the name of the faceting variable as a header for the facets, so it can be useful to use descriptive facet labels.

With `facet_grid()` and `facet_wrap()`, it's possible to use a labeller function to set the labels. The labeller function `label_both()` will print out both the name of the variable and the value of the variable in each facet (Figure 11-5, left):

```
ggplot(mpg_mod, aes(x = displ, y = hwy)) +  
  geom_point() +  
  facet_grid(drv ~ ., labeller = label_both)
```

Another useful labeller is `label_parsed()`, which takes strings and treats them as R math expressions (Figure 11-5, right):

```
# Make a modified copy of the original data  
mpg_mod <- mpg %>%  
  mutate(drv = recode(drv,  
    "4" = "4^{wd}",  
    "f" = "- Front %.% e^{pi * i}",  
    "r" = "4^{wd} - Front"  
  ))  
  
ggplot(mpg_mod, aes(x = displ, y = hwy)) +  
  geom_point() +  
  facet_grid(drv ~ ., labeller = label_parsed)
```

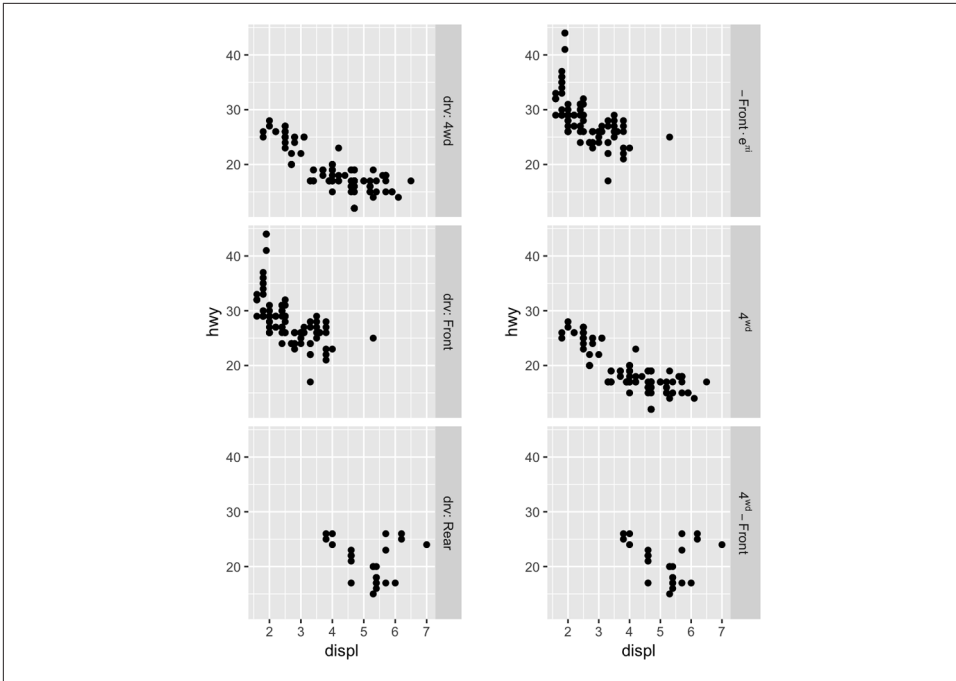


Figure 11-5. With `label_both()` (left); With `label_parsed()` for mathematical expressions (right)

See Also

See [Recipe 15.10](#) for more on renaming factor levels. If the faceting variable is not a factor but a character vector, changing the values is somewhat different. See [Recipe 15.12](#) for information on renaming items in character vectors.

11.4 Changing the Appearance of Facet Labels and Headers

Problem

You want to change the appearance of facet labels and headers.

Solution

With the theming system, set `strip.text` to control the text appearance and `strip.background` to control the background appearance ([Figure 11-6](#)):

```
library(gcookbook) # Load gcookbook for the cabbage_exp data set

ggplot(cabbage_exp, aes(x = Cultivar, y = Weight)) +
  geom_col() +
  facet_grid(. ~ Date) +
  theme(
    strip.text = element_text(face = "bold", size = rel(1.5)),
    strip.background = element_rect(fill = "lightblue", colour = "black",
                                     size = 1)
  )
)
```

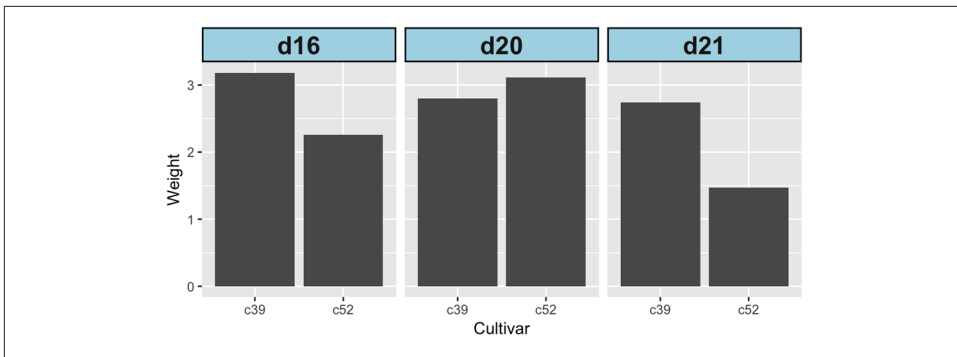


Figure 11-6. Customized appearance for facet labels

Discussion

Using `rel(1.5)` makes the label text 1.5 times the size of the base text size for the theme. Using `size = 1` for the background makes the outline of the facets 1 mm thick.

See Also

For more on how the theme system works, see Recipes [9.3](#) and [9.4](#).

Using Colors in Plots

In ggplot2's implementation of the grammar of graphics, `colour` is an aesthetic, just like x position, y position, and `size`. If color is just another aesthetic, why does it deserve its own chapter? The reason is that color is a more complicated aesthetic than the others. Instead of simply moving geoms left and right or making them larger and smaller, when you use color, there are many degrees of freedom and many more choices to make. What palette should you use for discrete values? Should you use a gradient with several different hues? How do you choose colors that can be interpreted accurately by those with color-vision deficiencies? In this chapter, I'll address these issues.

12.1 Setting the Colors of Objects

Problem

You want to set the color of some geoms in your graph.

Solution

In the call to the geom, set the values of `colour` or `fill` (Figure 12-1):

```
library(MASS) # Load MASS for the birthwt data set

ggplot(birthwt, aes(x = bwt)) +
  geom_histogram(fill = "red", colour = "black")

ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point(colour = "red")
```

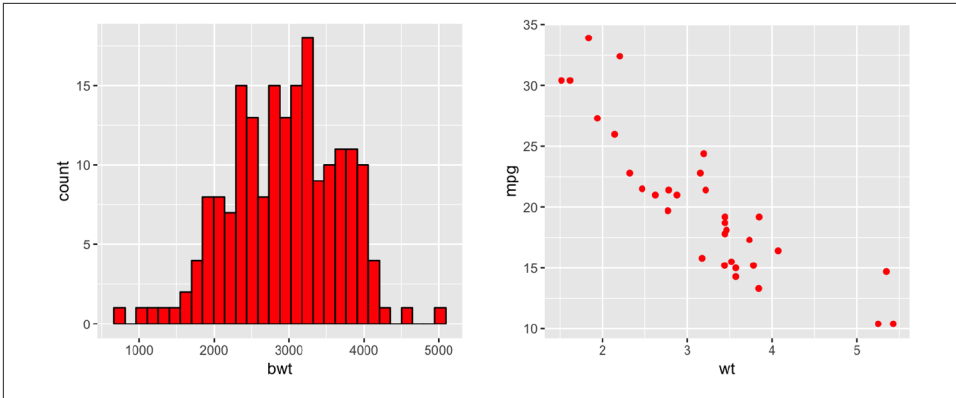


Figure 12-1. Setting fill and colour (left); Setting colour for points (right)

Discussion

In ggplot2, there's an important difference between *setting* and *mapping* aesthetic properties. In the preceding example, we set the color of the objects to "red".

Generally speaking, `colour` controls the color of lines and of the outlines of polygons, while `fill` controls the color of the fill area of polygons. However, point shapes are sometimes a little different. For most point shapes, the color of the entire point is controlled by `colour`, not `fill`. The exception is the point shapes (21–25) that have both a fill and an outline.

You can use `colour` or `color` interchangeably with ggplot2. In this book, I've used `colour`, in keeping with the form used in the official ggplot2 documentation.

See Also

For more information about point shapes, see [Recipe 4.5](#).

See [Recipe 12.5](#) for more on specifying colors.

12.2 Representing Variables with Colors

Problem

You want to use a variable (column from a data frame) to control the color of geoms.

Solution

In the call to the geom, inside of `aes()`, set the value of `colour` or `fill` to the name of one of the columns in the data ([Figure 12-2](#)):


```
library(gcookbook) # Load gcookbook for the cabbage_exp data set

# These both have the same effect
ggplot(cabbage_exp, aes(x = Date, y = Weight, fill = Cultivar)) +
  geom_col(colour = "black", position = "dodge")

ggplot(cabbage_exp, aes(x = Date, y = Weight)) +
  geom_col(aes(fill = Cultivar), colour = "black", position = "dodge")

# These both have the same effect
ggplot(mtcars, aes(x = wt, y = mpg, colour = cyl)) +
  geom_point()

ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point(aes(colour = cyl))
```

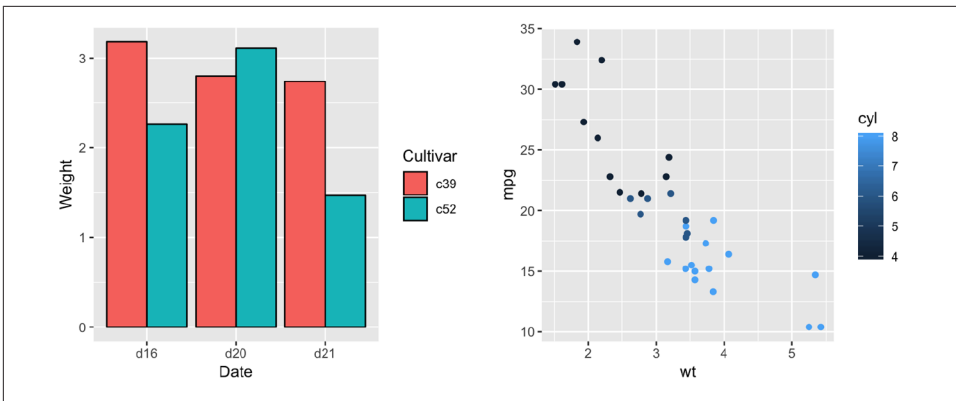


Figure 12-2. Mapping a variable to fill (left); Mapping a variable to colour for points (right)

When the mapping is specified in `ggplot()` it is used as the default mapping, which is inherited by all the geoms. Within a geom, the default mappings can be overridden.

Discussion

In the `cabbage_exp` example, the variable `Cultivar` is mapped to `fill`. The `Cultivar` column in `cabbage_exp` is a factor, so `ggplot` treats it as a categorical variable. You can check the type using `str()`:

```
str(cabbage_exp)
#> 'data.frame': 6 obs. of 6 variables:
#> $ Cultivar: Factor w/ 2 levels "c39","c52": 1 1 1 2 2 2
#> $ Date : Factor w/ 3 levels "d16","d20","d21": 1 2 3 1 2 3
#> $ Weight : num 3.18 2.8 2.74 2.26 3.11 1.47
#> $ sd : num 0.957 0.279 0.983 0.445 0.791 ...
#> $ n : int 10 10 10 10 10 10
#> $ se : num 0.3025 0.0882 0.311 0.1408 0.2501 ...
```

In the `mtcars` example, `cyl` is numeric, so it is treated as a continuous variable. Because of this, even though the actual values of `cyl` include only 4, 6, and 8, the legend has entries for the intermediate values 5 and 7. To make `ggplot` treat `cyl` as a categorical variable, you can convert it to a factor in the call to `ggplot()` (Figure 12-3, left), or you can modify the data so that the column is a character vector or factor (Figure 12-3, right):

```
# Convert to factor in call to ggplot()
ggplot(mtcars, aes(x = wt, y = mpg, colour = factor(cyl))) +
  geom_point()

# Another method: Convert to factor in the data
library(dplyr)
mtcars_mod <- mtcars %>%
  mutate(cyl = as.factor(cyl)) # Convert cyl to a factor

ggplot(mtcars_mod, aes(x = wt, y = mpg, colour = cyl)) +
  geom_point()
```

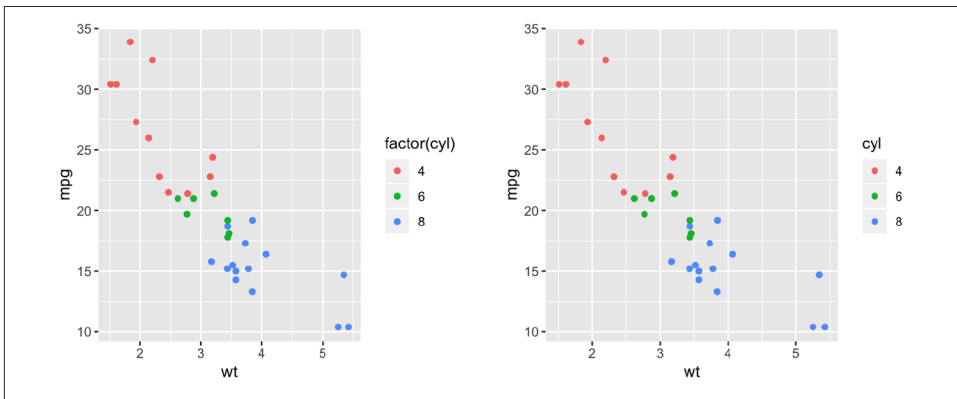


Figure 12-3. Converting `cyl` to a factor, within the call to `ggplot` (left); By modifying the data frame (right)

See Also

You may also want to change the colors that are used in the scale. For continuous data, see [Recipe 12.6](#). For discrete data, see [Recipes 12.4](#) and [12.5](#).

12.3 Using a Colorblind-Friendly Palette

Problem

You want to select a color palette that can also be distinguished by colorblind viewers.

Solution

Use the color scales in the viridis package.

The viridis package contains a set of beautiful color scales that are each designed to span as wide a palette as possible, making it easier to see differences in your data. These scales are also designed to be perceptually uniform, printable in greyscale, and easier to read by those with colorblindness.

Here is an example from the [introduction page to viridis](#) (Figure 12-4):

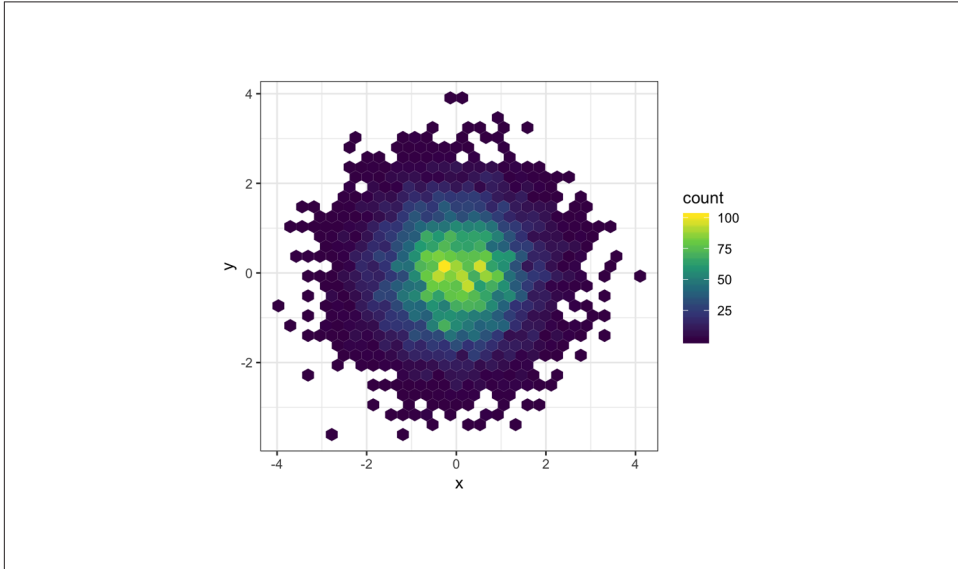


Figure 12-4. Example of viridis color palette

The viridis color scales can be implemented for data that is both continuous and discrete in nature. You will need to add `scale_fill_viridis_c()` to your plot if your data is continuous. If your data is discrete you will need to use `scale_fill_viridis_d()` instead, as in Figure 12-5:

```
library(gcookbook) # Load gcookbook for the uspopage data set

# Create the base plot
uspopage_plot <- ggplot(uspopage, aes(x = Year, y = Thousands,
                                     fill = AgeGroup)) +

  geom_area()

# Add the viridis color scale
uspopage_plot +
  scale_fill_viridis_d()
```

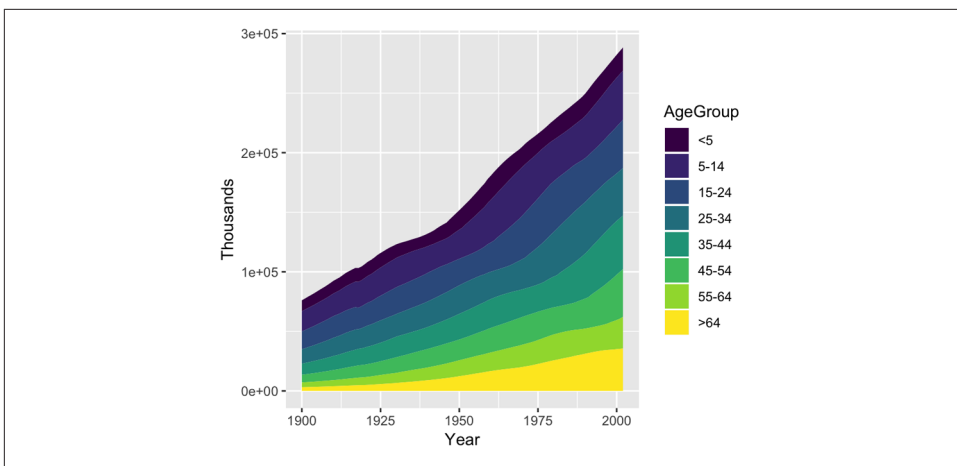


Figure 12-5. A plot with the colorblind-friendly viridis palette

Discussion

About 8 percent of males and 0.5 percent of females have some form of color-vision deficiency, so there's a good chance that someone in your audience will be among them. There are many different forms of color blindness—the palettes that are mentioned in this book are designed to enable people with any of the most common forms of color-vision deficiency to distinguish the colors. (Monochromacy, or total colorblindness, is rare. Those who have it can only see differences in brightness.)

The viridis color scales come with the current version of ggplot2 (3.0.0). There are also other color palettes that are friendly to users with color blindness, such as those in the cetcolor package.

See Also

To see more on the different viridis palettes, see `?scales::viridis_pal`.

The cetcolor scales: <https://github.com/coatless/cetcolor>.

The **Color Oracle** program can simulate how things on your screen appear to someone with color-vision deficiency, but keep in mind that the simulation isn't perfect. In my informal testing, I viewed an image with simulated red-green deficiency, and I could distinguish the colors just fine—but others with actual red-green deficiency viewed the same image and couldn't tell the colors apart!

12.4 Using a Different Palette for a Discrete Variable

Problem

You want to use different colors for a discrete mapped variable.

Solution

Use one of the scales listed in [Table 12-1](#).

Table 12-1. Discrete fill and color scales

Fill scale	Color scale	Description
<code>scale_fill_discrete()</code>	<code>scale_colour_discrete()</code>	Colors evenly spaced around the color wheel (same as hue)
<code>scale_fill_hue()</code>	<code>scale_colour_hue()</code>	Colors evenly spaced around the color wheel (same as discrete)
<code>scale_fill_grey()</code>	<code>scale_colour_grey()</code>	Greyscale palette
<code>scale_fill_viridis_d()</code>	<code>scale_colour_viridis_d()</code>	Viridis palettes
<code>scale_fill_brewer()</code>	<code>scale_colour_brewer()</code>	ColorBrewer palettes
<code>scale_fill_manual()</code>	<code>scale_colour_manual()</code>	Manually specified colors

In the example here we'll use the default palette (hue), a viridis palette, and a ColorBrewer palette ([Figure 12-6](#)):

```
library(gcookbook) # Load gcookbook for the uspopage data set
library(viridis) # Load viridis for the viridis palette

# Create the base plot
uspopage_plot <- ggplot(uspopage, aes(x = Year, y = Thousands,
                                       fill = AgeGroup)) +
  geom_area()

# These four specifications all have the same effect
uspopage_plot
# uspopage_plot + scale_fill_discrete()
# uspopage_plot + scale_fill_hue()
# uspopage_plot + scale_color_viridis()

# Viridis palette
uspopage_plot +
  scale_fill_viridis(discrete = TRUE)

# ColorBrewer palette
uspopage_plot +
  scale_fill_brewer()
```

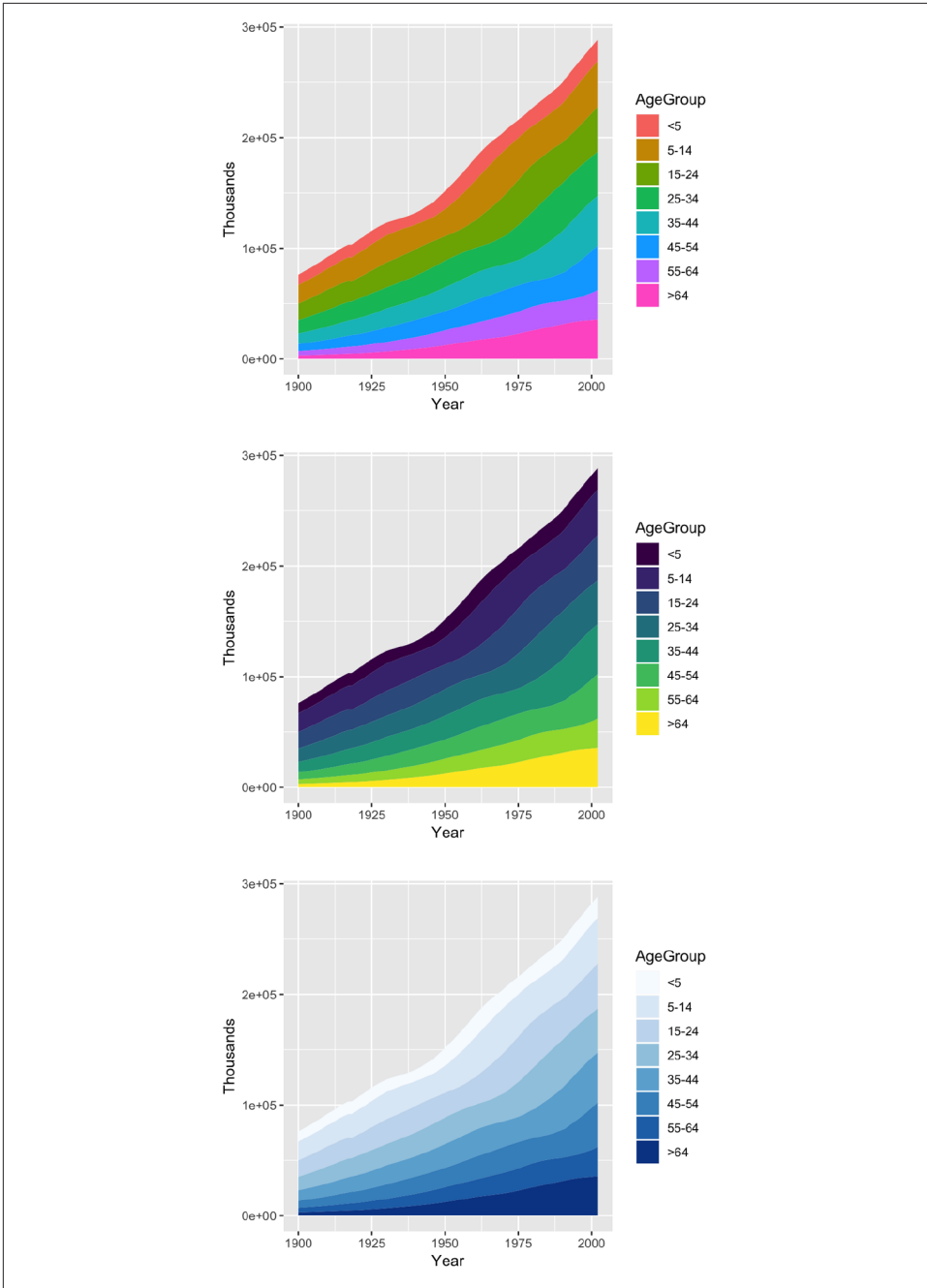


Figure 12-6. Default palette (using hue; top); A viridis palette (middle); A ColorBrewer palette (bottom)

Discussion

Changing a palette is a modification of the color (or fill) scale: it involves a change in the mapping from numeric or categorical values to aesthetic attributes. There are two types of scales that use colors: *fill* scales and *color* scales.

With `scale_fill_hue()`, the colors are taken from around the color wheel in the HCL (hue-chroma-lightness) color space. The default lightness value is 65 on a scale from 0–100. This is good for filled areas, but it's a bit light for points and lines. To make the colors darker for points and lines, as in [Figure 12-7](#) (right), set the value of `l` (luminance/lightness):

```
# Create the base scatter plot
hw_splot <- ggplot(heightweight, aes(x = ageYear, y = heightIn, colour = sex)) +
  geom_point()

# Default lightness = 65
hw_splot

# Slightly darker, set lightness = 45
hw_splot +
  scale_colour_hue(l = 45)
```

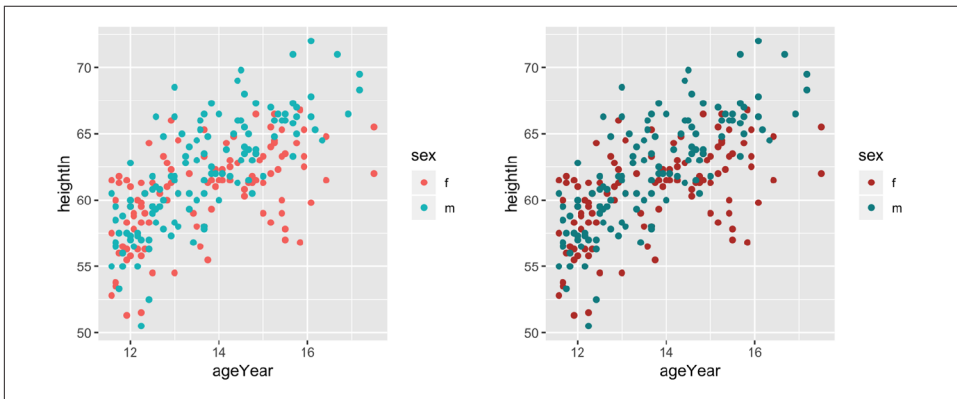


Figure 12-7. Points with default lightness (left); With lightness set to 45 (right)

The `viridis` package provides a number of color scales that make it easy to see differences across your data. See [Recipe 12.3](#) for more details and examples.

The `ColorBrewer` package provides a number of palettes. You can generate a graphic showing all of them, as shown in [Figure 12-8](#):

```
library(RColorBrewer)
display.brewer.all()
```

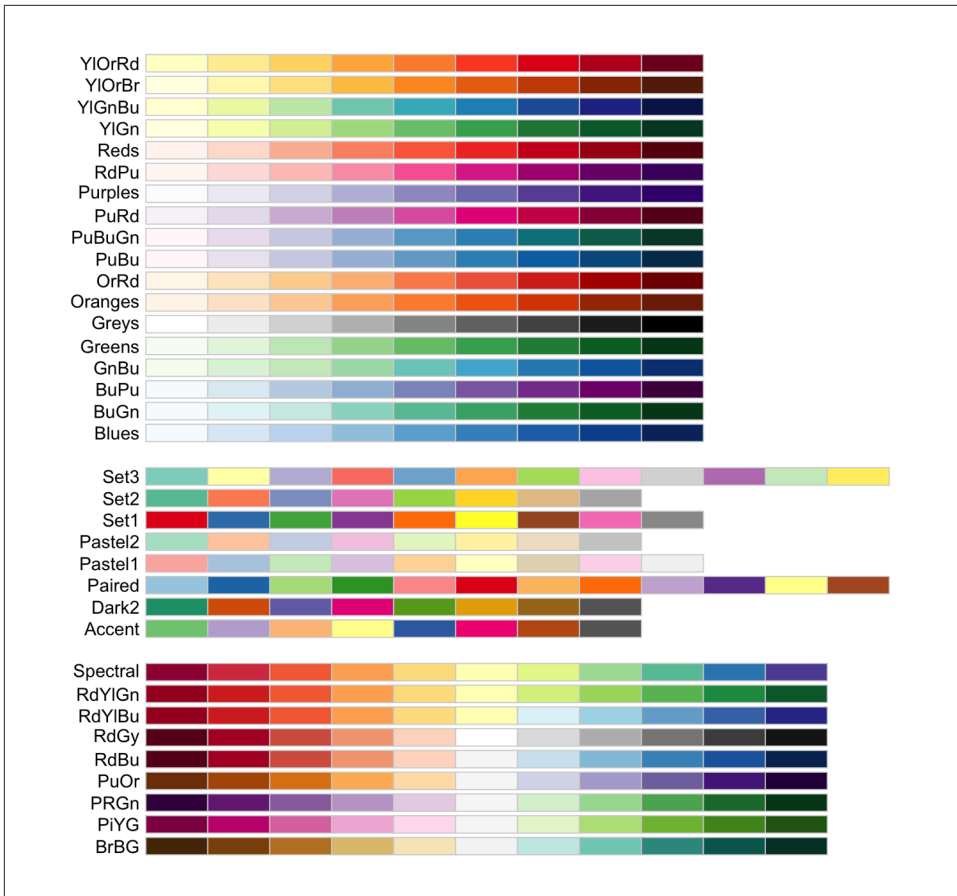


Figure 12-8. All the ColorBrewer palettes

The ColorBrewer palettes can be selected by name. For example, this will use the “Oranges” palette (Figure 12-9):

```
hw_plot +
  scale_colour_brewer(palette = "Oranges") +
  theme_bw()
```

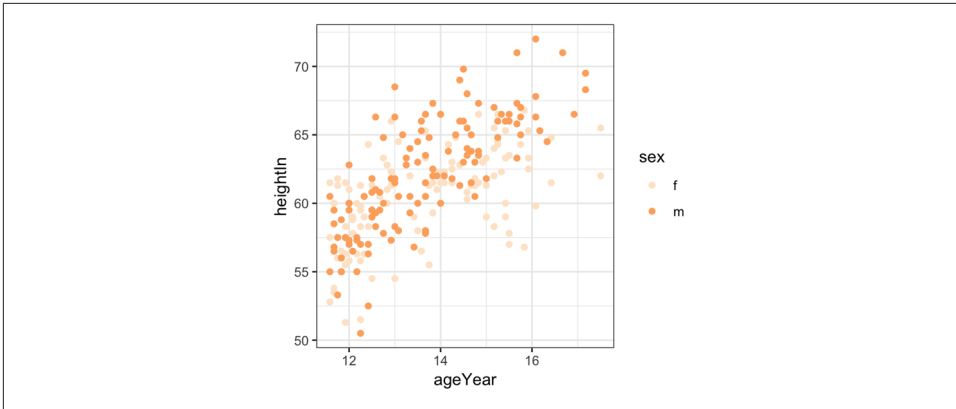



Figure 12-9. Using a named ColorBrewer palette

You can also use a palette of greys. This is useful for print when the output is in black and white. The default is to start at 0.2 and end at 0.8, on a scale from 0 (black) to 1 (white), but you can change the range, as shown in Figure 12-10:

```
hw_splot +
  scale_colour_grey()

# Reverse the direction and use a different range of greys
hw_splot +
  scale_colour_grey(start = 0.7, end = 0)
```

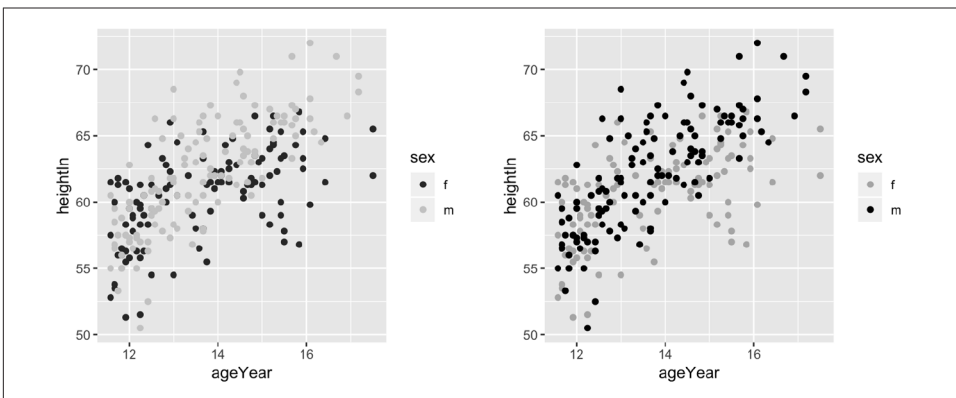


Figure 12-10. Using the default grey palette (left); A different grey palette (right)

See Also

See [Recipe 10.4](#) for more information about reversing the legend.

To select colors manually, see [Recipe 12.5](#).

For more about viridis, see <https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html>. For more about ColorBrewer, see <http://colorbrewer2.org>.

12.5 Using a Manually Defined Palette for a Discrete Variable

Problem

You want to use different colors for a discrete mapped variable.

Solution

In the example here, we'll manually define colors by specifying values with `scale_colour_manual()` (Figure 12-11). The colors can be named, or they can be specified with RGB values:

```
library(gcookbook) # Load gcookbook for the heightweight data set

# Create the base plot
hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn, colour = sex)) +
  geom_point()

# Using color names
hw_plot +
  scale_colour_manual(values = c("red", "blue"))

# Using RGB values
hw_plot +
  scale_colour_manual(values = c("#CC6666", "#7777DD"))

# Using RGB values based on the viridis color scale
hw_plot +
  scale_colour_manual(values = c("#440154FF", "#FDE725FF")) +
  theme_bw()
```

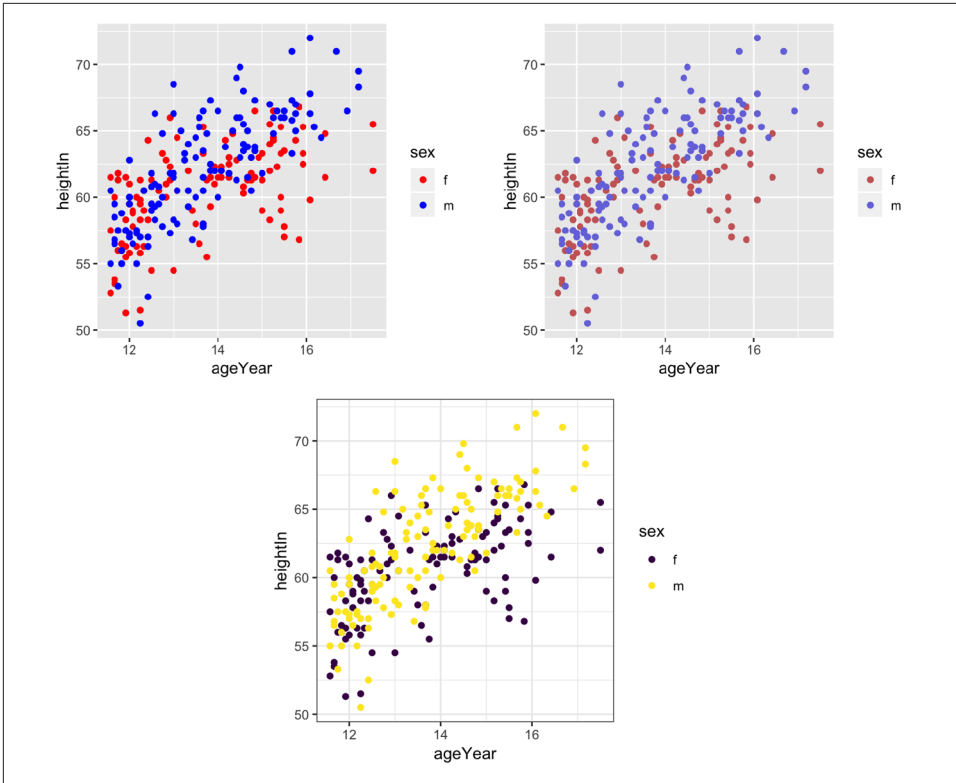


Figure 12-11. Scatter plot with named colors (top left); With slightly different RGB colors (top right); With colors from the viridis color scale (bottom)

For fill scales, use `scale_fill_manual()` instead.

Discussion

The order of the items in the values vector matches the order of the factor levels for the discrete scale. In the preceding example, the order of sex is f, then m, so the first item in values goes with f and the second goes with m. Here's how to see the order of factor levels:

```
levels(heightweight$sex)
#> [1] "f" "m"
```

If the variable is a character vector, not a factor, it will automatically be converted to a factor, and by default the levels will appear in alphabetical order.

It's possible to specify the colors in a different order by using a named vector:

```
hw_plot +
  scale_colour_manual(values = c(m = "blue", f = "red"))
```

There is a large set of named colors in R, which you can see by running `color()`. Some basic color names are useful: "white", "black", "grey80", "red", "blue", "darkred", and so on. There are many other named colors, but their names are generally not very informative (I certainly have no idea what "thistle3" and "seashell" look like), so it's often easier to use numeric RGB values for specifying colors.

RGB colors are specified as six-digit hexadecimal (base-16) numbers of the form #RRGGBB. In hexadecimal, the digits go from 0 to 9, and then continue with A (10 in base 10) to F (15 in base 10). Each color is represented by two digits and can range from 00 to FF (255 in base 10). So, for example, the color #FF0099 has a value of 255 for red, 0 for green, and 153 for blue, resulting in a shade of magenta. The hexadecimal numbers for each color channel often repeat the same digit because it makes them a little easier to read, and because the precise value of the second digit has a relatively insignificant effect on appearance.

Here are some rules of thumb for specifying and adjusting RGB colors:

- In general, higher numbers are brighter and lower numbers are darker.
- To get a shade of grey, set all the channels to the same value.
- The opposites of RGB are CMY: Cyan, Magenta, and Yellow. Higher values for the red channel make it more red, and lower values make it more cyan. The same is true for the pairs green and magenta, and blue and yellow.

You may want to manually select colors based on the color scales in the `viridis` package, as described in [Recipe 12.4](#). You can do so by calling `viridis()` and passing it the number of discrete categories you have. This will generate the RGB hexadecimal values. You can similarly generate the RGB values for the other color scales in the `viridis` package: "magma", "plasma", "inferno", and "cividis":

```
library(viridis)
viridis(2) ## Specifying 2 discrete categories with the viridis color scale
#> [1] "#440154FF" "#FDE725FF"
inferno(5) ## Specifying 5 discrete categories with the inferno color scale
#> [1] "#000004FF" "#56106EFF" "#BB3754FF" "#F98C0AFF" "#FCFFA4FF"
```

See Also

A chart of RGB color codes: <http://html-color-codes.com>.

12.6 Using a Manually Defined Palette for a Continuous Variable

Problem

You want to use different colors for a continuous variable.

Solution

In the example here, we'll specify the colors for a continuous variable using various gradient scales ([Figure 12-12](#)). The colors can be named, or they can be specified with RGB values:

```
library(gcookbook) # Load gcookbook for the heightweight data set

# Create the base plot
hw_plot <- ggplot(heightweight, aes(x = ageYear, y = heightIn,
                                     colour = weightLb)) +
  geom_point(size = 3)

hw_plot

# A gradient with a white midpoint
library(scales)
hw_plot +
  scale_colour_gradient2(
    low = muted("red"),
    mid = "white",
    high = muted("blue"),
    midpoint = 110
  )

# With a gradient between two colors (black and white)
hw_plot +
  scale_colour_gradient(low = "black", high = "white")

# A gradient of n colors
hw_plot +
  scale_colour_gradientn(colours = c("darkred", "orange", "yellow", "white"))
```

For fill scales, use `scale_fill_XXX()` versions instead, where `XXX` is one of `gradient`, `gradient2`, or `gradientn`.

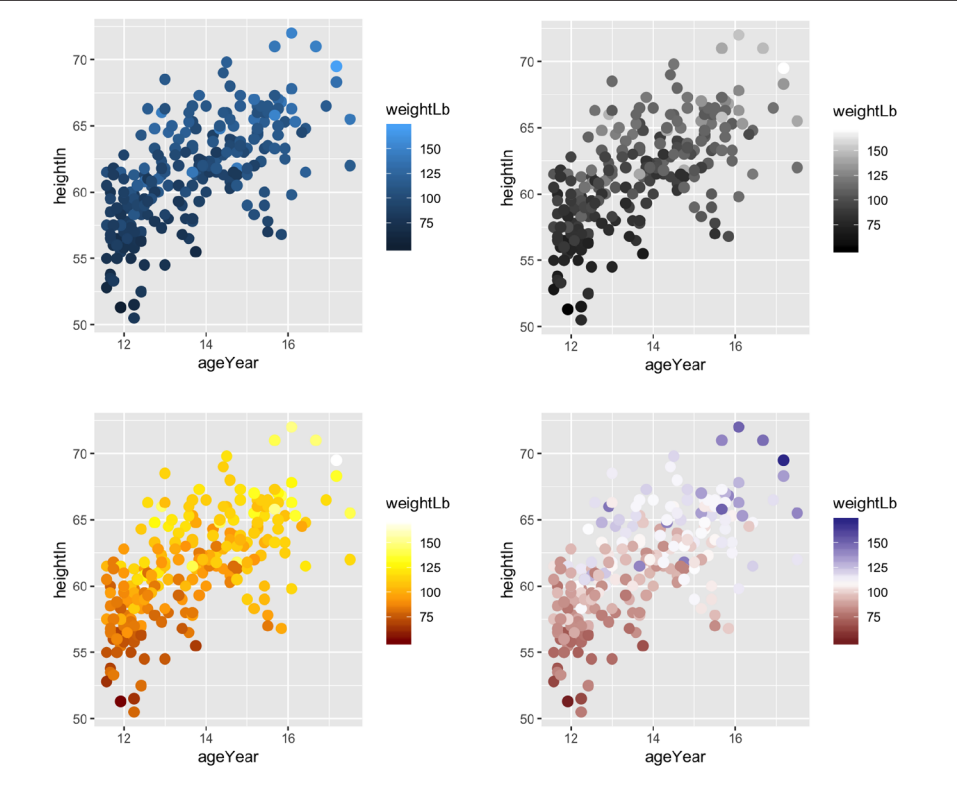


Figure 12-12. Clockwise from top left: default colors; two-color gradient (black and white) with `scale_colour_gradient()`; three-color gradient with midpoint with `scale_colour_gradient2()`; four-color gradient with `scale_colour_gradientn()`

Discussion

Mapping continuous values to a color scale requires a continuously changing the palette of colors. [Table 12-2](#) lists the continuous color and fill scales.

Table 12-2. Continuous fill and color scales

Fill scale	Color scale	Description
<code>scale_fill_gradient()</code>	<code>scale_colour_gradient()</code>	Two-color gradient
<code>scale_fill_gradient2()</code>	<code>scale_colour_gradient2()</code>	Gradient with a middle color and two colors that diverge from it
<code>scale_fill_gradientn()</code>	<code>scale_colour_gradientn()</code>	Gradient with n colors, equally spaced
<code>scale_fill_viridis_c()</code>	<code>scale_colour_viridis_c()</code>	Viridis palettes

Notice that we used the `muted()` function in the examples. This is a function from the `scales` package that returns an RGB value that is a less-saturated version of the color chosen.

See Also

If you want use a discrete (categorical) scale instead of a continuous one, you can recode your data into categorical values. See [Recipe 15.14](#).

12.7 Coloring a Shaded Region Based on Value

Problem

You want to set the color of a shaded region based on the y value.

Solution

Add a column that categorizes the y values, then map that column to fill. In this example, we'll first categorize the values as positive or negative:

```
library(gcookbook) # Load gcookbook for the climate data set
library(dplyr)

climate_mod <- climate %>%
  filter(Source == "Berkeley") %>%
  mutate(valence = if_else(Anomaly10y >= 0, "pos", "neg"))

climate_mod
#>   Source Year Anomaly1y Anomaly5y Anomaly10y Unc10y valence
#> 1 Berkeley 1800      NA      NA      -0.435  0.505    neg
#> 2 Berkeley 1801      NA      NA      -0.453  0.493    neg
#> 3 Berkeley 1802      NA      NA      -0.460  0.486    neg
#> ...<199 more rows>...
#> 203 Berkeley 2002      NA      NA       0.856  0.028    pos
#> 204 Berkeley 2003      NA      NA       0.869  0.028    pos
#> 205 Berkeley 2004      NA      NA       0.884  0.029    pos
```

Once we've categorized the values as positive or negative, we can make the plot, mapping `valence` to the fill color, as shown in [Figure 12-13](#):

```
ggplot(climate_mod, aes(x = Year, y = Anomaly10y)) +
  geom_area(aes(fill = valence)) +
  geom_line() +
  geom_hline(yintercept = 0)
```

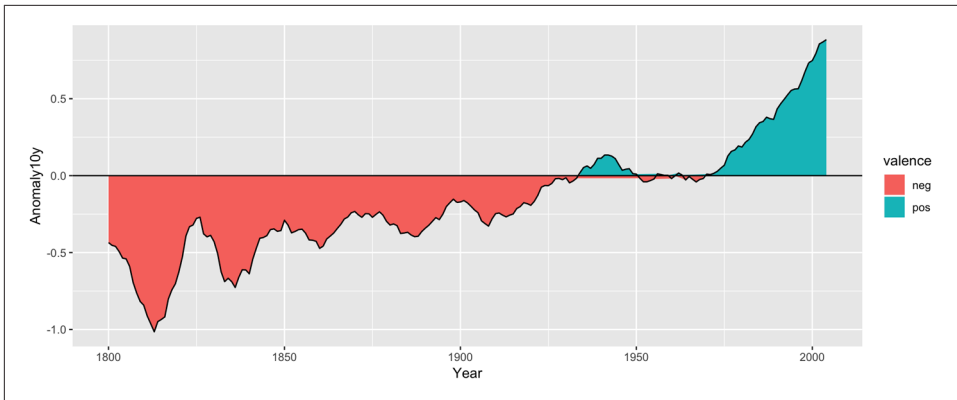


Figure 12-13. Mapping valence to fill color—notice the red area under the zero line around 1950

Discussion

If you look closely at the figure, you'll notice that there are some stray shaded areas near the zero line. This is because each of the two colored areas is a single polygon bounded by the data points, and the data points are not actually at zero. To solve this problem, we can interpolate the data to 1,000 points by using `approx()`:

```
# approx() returns a list with x and y vectors
interp <- approx(climate_mod$Year, climate_mod$Anomaly10y, n = 1000)

# Put in a data frame and recalculate valence
cbi <- data.frame(Year = interp$x, Anomaly10y = interp$y) %>%
  mutate(valence = if_else(Anomaly10y >= 0, "pos", "neg"))
```

It would be more precise (and more complicated) to interpolate exactly where the line crosses zero, but `approx()` works fine for the purposes here.

Now we can plot the interpolated data (Figure 12-14). This time we'll make a few adjustments—we'll make the shaded regions partially transparent, change the colors, remove the legend, and remove the padding on the left and right sides:

```
ggplot(cbi, aes(x = Year, y = Anomaly10y)) +
  geom_area(aes(fill = valence), alpha = .4) +
  geom_line() +
  geom_hline(yintercept = 0) +
  scale_fill_manual(values = c("#CCEEFF", "#FFDDDD"), guide = FALSE) +
  scale_x_continuous(expand = c(0, 0))
```

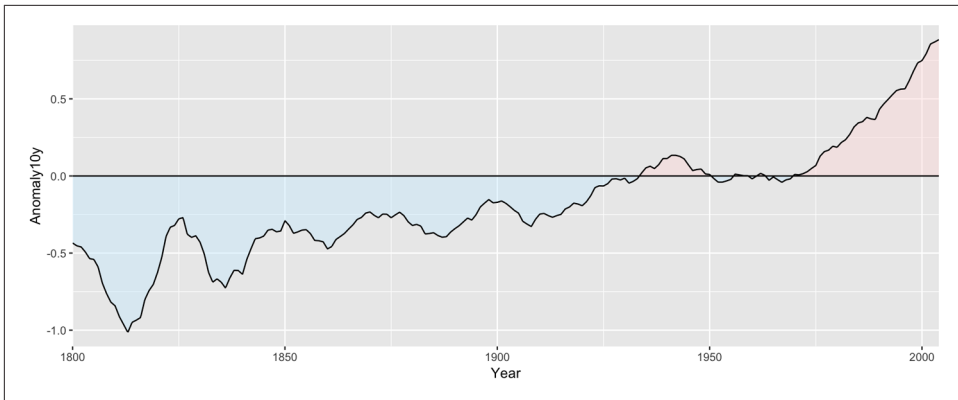



Figure 12-14. Shaded regions with interpolated data

Miscellaneous Graphs

There are many, many ways of visualizing data, and sometimes things don't fit into nice, tidy categories. This chapter shows how to make some of these other visualizations.

13.1 Making a Correlation Matrix

Problem

You want to make a graphical correlation matrix.

Solution

We'll look at the `mtcars` data set:

```
mtcars
#>           mpg cyl disp  hp drat   wt  qsec vs am gear carb
#> Mazda RX4      21.0   6  160 110 3.90 2.620 16.46 0  1   4   4
#> Mazda RX4 Wag  21.0   6  160 110 3.90 2.875 17.02 0  1   4   4
#> Datsun 710      22.8   4  108  93 3.85 2.320 18.61 1  1   4   1
#> ...<26 more rows>...
#> Ferrari Dino   19.7   6  145 175 3.62 2.770 15.50 0  1   5   6
#> Maserati Bora  15.0   8  301 335 3.54 3.570 14.60 0  1   5   8
#> Volvo 142E     21.4   4  121 109 4.11 2.780 18.60 1  1   4   2
```

First, generate the numerical correlation matrix using `cor`. This will generate correlation coefficients for each pair of columns:

```
mcor <- cor(mtcars)
# Print mcor and round to 2 digits
round(mcor, digits = 2)
#>           mpg   cyl  disp    hp  drat    wt  qsec    vs  am  gear  carb
#> mpg      1.00 -0.85 -0.85 -0.78  0.68 -0.87  0.42  0.66  0.60  0.48 -0.55
```

```
#> cyl  -0.85  1.00  0.90  0.83 -0.70  0.78 -0.59 -0.81 -0.52 -0.49  0.53
#> disp -0.85  0.90  1.00  0.79 -0.71  0.89 -0.43 -0.71 -0.59 -0.56  0.39
#> ...<5 more rows>...
#> am    0.60 -0.52 -0.59 -0.24  0.71 -0.69 -0.23  0.17  1.00  0.79  0.06
#> gear  0.48 -0.49 -0.56 -0.13  0.70 -0.58 -0.21  0.21  0.79  1.00  0.27
#> carb -0.55  0.53  0.39  0.75 -0.09  0.43 -0.66 -0.57  0.06  0.27  1.00
```

If there are any columns that you don't want used for correlations (for example, a column of names), you should exclude them. If there are any NA cells in the original data, the resulting correlation matrix will have NA values. To deal with this, you will probably want to use the argument `use="complete.obs"` or `use="pairwise.complete.obs"`.

To plot the correlation matrix (Figure 13-1), we'll use the `corrplot` package:

```
# If needed, first install with install.packages("corrplot")
library(corrplot)

corrplot(mcor)
```

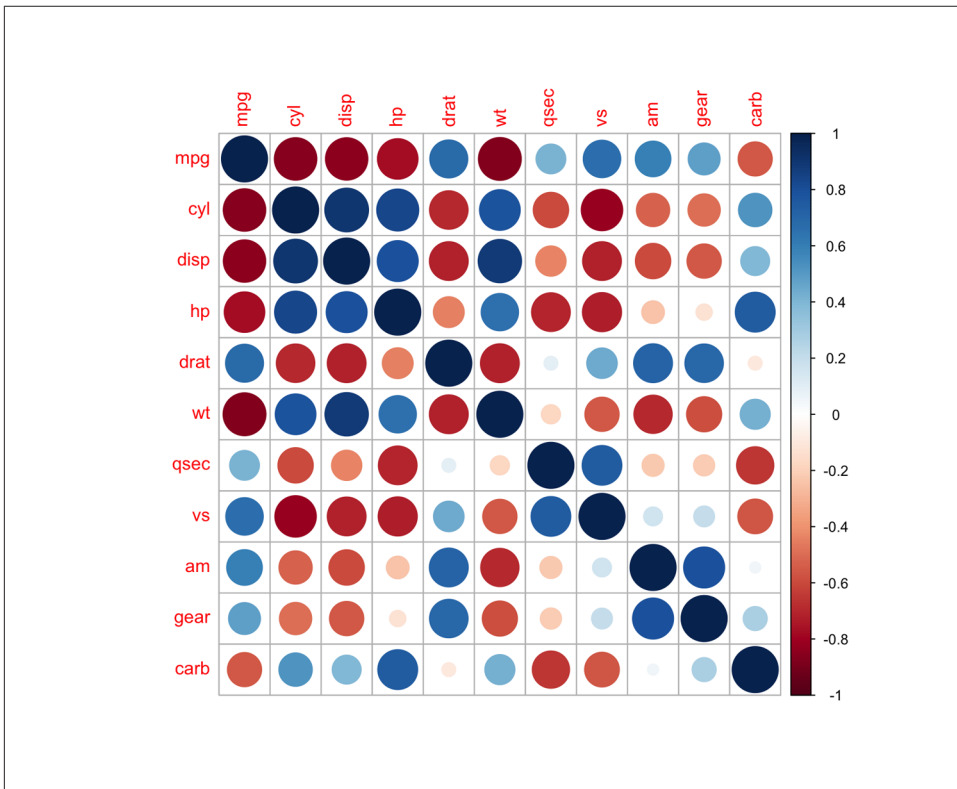


Figure 13-1. A correlation matrix

Discussion

The `corrplot()` function has many, many options. Here is an example of how to make a correlation matrix with colored squares and black labels, rotated 45 degrees along the top ([Figure 13-2](#)):

```
corrplot(mcor, method = "shade", shade.col = NA, tl.col = "black", tl.srt = 45)
```

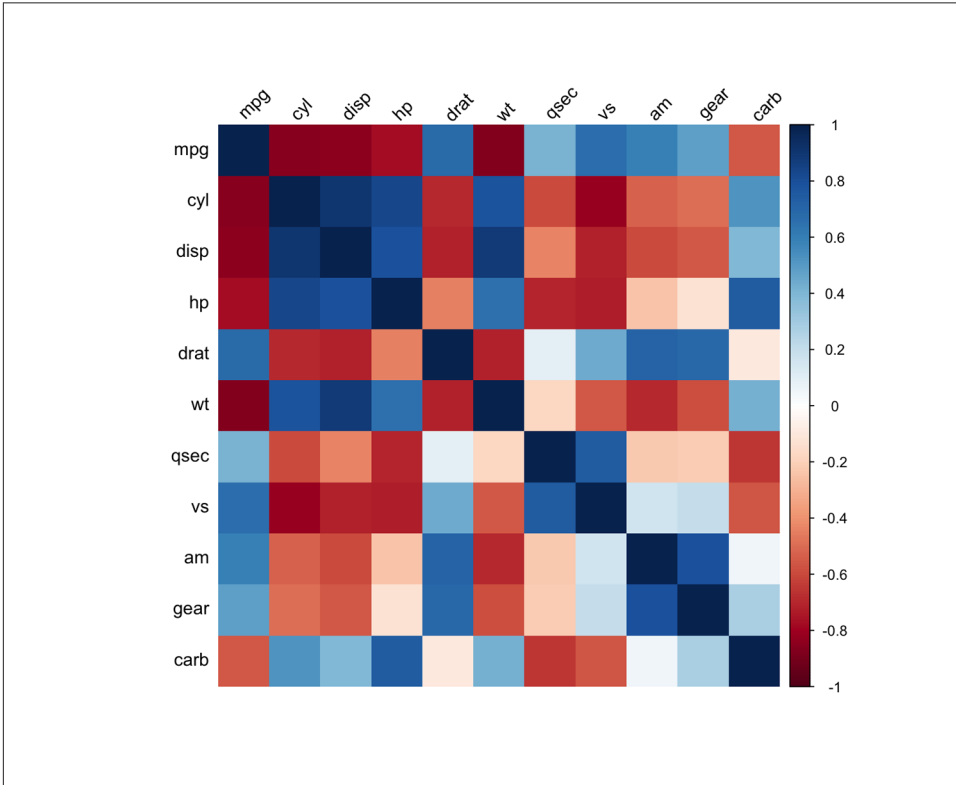


Figure 13-2. Correlation matrix with colored squares and black, rotated labels

It may also be helpful to display labels representing the correlation coefficient on each square in the matrix. In this example we'll make a lighter palette so that the text is readable, and we'll remove the color legend, since it's redundant. We'll also order the items so that correlated items are closer together, using the `order = "AOE"` (angular order of eigenvectors) option. The result is shown in [Figure 13-3](#):

```
# Generate a lighter palette
col <- colorRampPalette(c("#BB4444", "#EE9988", "#FFFFFF", "#77AADD", "#4477AA"))

corrplot(mcor, method = "shade", shade.col = NA, tl.col = "black", tl.srt = 45,
         col = col(200), addCoef.col = "black", cl.pos = "n", order = "AOE")
```

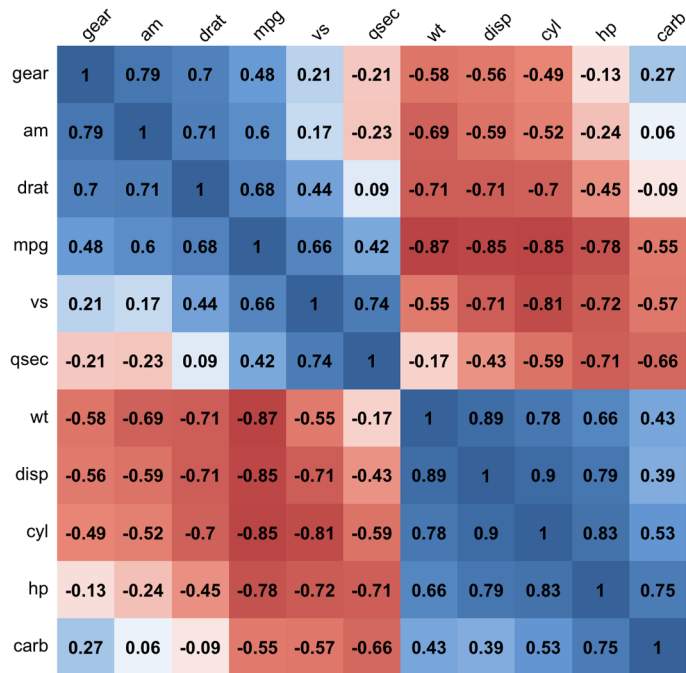


Figure 13-3. Correlation matrix with correlation coefficients and no legend

Like many other standalone plotting functions, `corrplot()` has its own menagerie of options, which can't all be illustrated here. Table 13-1 lists some useful options.

Table 13-1. Options for `corrplot()`

Option	Description
<code>type={"lower" "upper"}</code>	Only use the lower or upper triangle
<code>diag=FALSE</code>	Don't show values on the diagonal
<code>addshade="all"</code>	Add lines indicating the direction of the correlation
<code>shade.col=NA</code>	Hide correlation direction lines
<code>method="shade"</code>	Use colored squares
<code>method="ellipse"</code>	Use ellipses
<code>addCoef.col="*color*"</code>	Add correlation coefficients, in <i>color</i>
<code>tl.srt="*number*"</code>	Specify the rotation angle for top labels
<code>tl.col="*color*"</code>	Specify the label color

Option	Description
<code>order={"AOE" "FPC" "hclust"}</code>	Sort labels using angular order of eigenvectors, first principal component, or hierarchical clustering

See Also

To create a scatter plot matrix, see [Recipe 5.13](#).

For more on subsetting data, see [Recipe 15.7](#).

13.2 Plotting a Function

Problem

You want to plot a function.

Solution

Use `stat_function()`. It's also necessary to give `ggplot` a dummy data frame so that it will get the proper x range. In this example we'll use `dnorm()`, which gives the density of the normal distribution ([Figure 13-4](#), left):

```
# The data frame is only used for setting the range
p <- ggplot(data.frame(x = c(-3, 3)), aes(x = x))

p + stat_function(fun = dnorm)
```

Discussion

Some functions take additional arguments. For example, `dt()`, the function for the density of the t -distribution, takes a parameter for degrees of freedom ([Figure 13-4](#), right). These additional arguments can be passed to the function by putting them in a list and giving the list to `args`:

```
p + stat_function(fun = dt, args = list(df = 2))
```

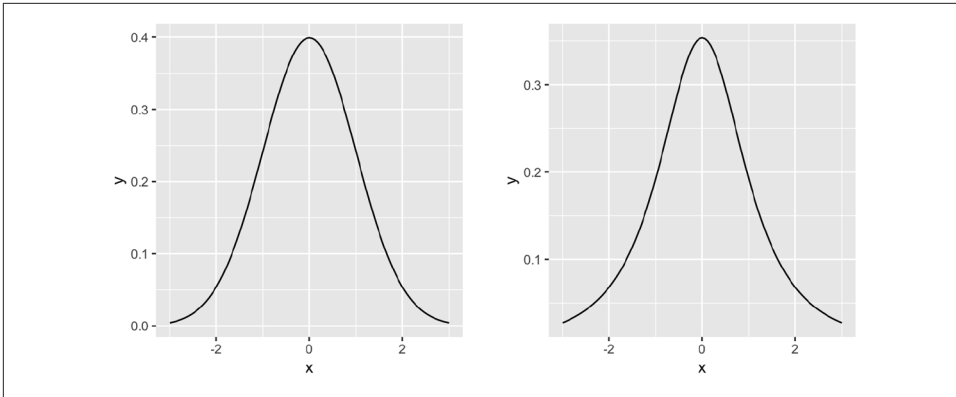


Figure 13-4. The normal distribution (left); The t-distribution with $df=2$ (right)

It's also possible to define your own functions. It should take an x value for its first argument, and it should return a y value. In this example, we'll define a sigmoid function (Figure 13-5):

```
myfun <- function(xvar) {  
  1 / (1 + exp(-xvar + 10))  
}  
  
ggplot(data.frame(x = c(0, 20)), aes(x = x)) +  
  stat_function(fun = myfun)
```

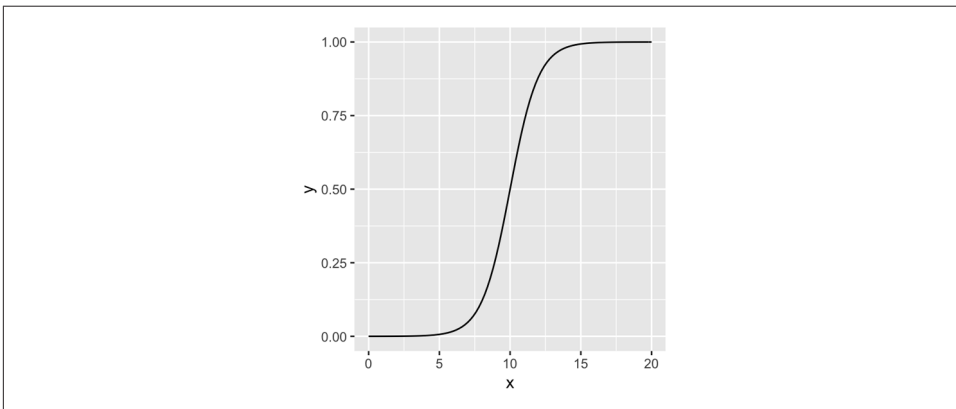


Figure 13-5. A user-defined function

By default, the function is calculated at 101 points along the x range. If you have a rapidly fluctuating function, you may be able to see the individual segments. To smooth out the curve, pass a larger value of n to `stat_function()`, as in `stat_function(fun=myfun, n=200)`.

See Also

For plotting predicted values from model objects (such as `lm` and `glm`), see [Recipe 5.7](#).

13.3 Shading a Subregion Under a Function Curve

Problem

You want to shade part of the area under a function curve.

Solution

Define a new wrapper function around your curve function, and replace out-of-range values with NA, as shown in [Figure 13-6](#):

```
# Return dnorm(x) for 0 < x < 2, and NA for all other x
dnorm_limit <- function(x) {
  y <- dnorm(x)
  y[x < 0 | x > 2] <- NA
  return(y)
}

# ggplot() with dummy data
p <- ggplot(data.frame(x = c(-3, 3)), aes(x = x))

p +
  stat_function(fun = dnorm_limit, geom = "area", fill = "blue", alpha = 0.2) +
  stat_function(fun = dnorm)
```

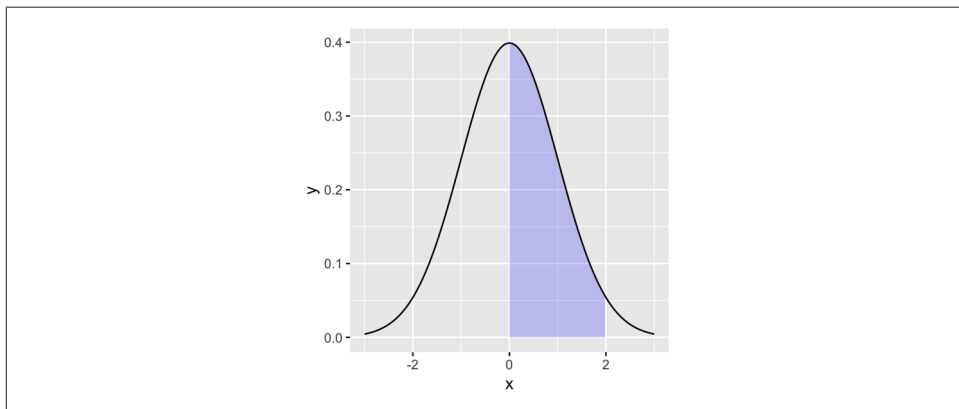


Figure 13-6. Function curve with a shaded region

Remember that what gets passed to this function is a vector, not individual values. If this function operated on single elements at a time, it might make sense to use an

if-else statement to decide what to return, conditional on the value of *x*. But that won't work here, since *x* is a vector with many values.

Discussion

R has first-class functions, and we can write a function that returns a *closure*—that is, we can program a function to program another function.

This function will allow you to pass in a function, a minimum value, and a maximum value. Values outside the range will again be returned with NA:

```
limitRange <- function(fun, min, max) {  
  function(x) {  
    y <- fun(x)  
    y[x < min | x > max] <- NA  
    return(y)  
  }  
}
```

Now we can call this function to create another function—one that is effectively the same as the `dnorm_limit()` function used earlier:

```
# This returns a function  
dlimit <- limitRange(dnorm, 0, 2)  
# Now we'll try out the new function -- it only returns values for inputs  
# between 0 and 2  
dlimit(-2:4)  
#> [1]      NA      NA 0.39894228 0.24197072 0.05399097      NA      NA
```

We can use `limitRange()` to create a function that is passed to `stat_function()`:

```
p +  
  stat_function(fun = dnorm) +  
  stat_function(fun = limitRange(dnorm, 0, 2), geom = "area", fill = "blue",  
               alpha = 0.2)
```

The `limitRange()` function can be used with any function, not just `dnorm()`, to create a range-limited version of that function. The result of all this is that instead of having to write functions with different hard-coded values for each situation that arises, we can write one function and simply pass it different arguments depending on the situation.

If you look very, very closely at the graph in [Figure 13-6](#), you may see that the shaded region does not align exactly with the range we specified. This is because `ggplot2` does a numeric approximation by calculating values at fixed intervals, and these intervals may not fall exactly within the specified range. As in [Recipe 13.2](#), we can improve the approximation by increasing the number of interpolated values with `stat_function(n = 200)`.

13.4 Creating a Network Graph

Problem

You want to create a network graph.

Solution

Use the igraph package. To create a graph, pass a vector containing pairs of items to `graph()`, then plot the resulting object (Figure 13-7):

```
# May need to install first, with install.packages("igraph")
library(igraph)

# Specify edges for a directed graph
gd <- graph(c(1,2, 2,3, 2,4, 1,4, 5,5, 3,6))
plot(gd)

# For an undirected graph
gu <- graph(c(1,2, 2,3, 2,4, 1,4, 5,5, 3,6), directed = FALSE)
# No labels
plot(gu, vertex.label = NA)
```

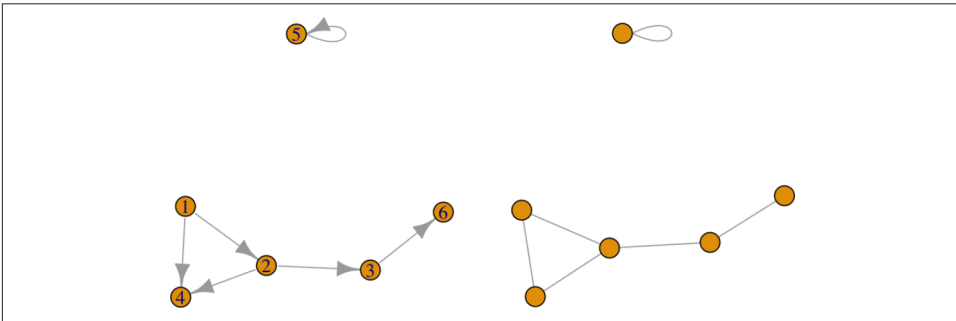


Figure 13-7. A directed graph (left); An undirected graph, with no vertex labels (right)

This is the structure of each of the graph objects:

```
gd
#> IGRAPH ea9b58e D--- 6 6 --
#> + edges from ea9b58e:
#> [1] 1->2 2->3 2->4 1->4 5->5 3->6
gu
#> IGRAPH a3410b5 U--- 6 6 --
#> + edges from a3410b5:
#> [1] 1--2 2--3 2--4 1--4 5--5 3--6
```

Discussion

In a network graph, the position of the nodes is unspecified by the data, and they're placed randomly. To make the output repeatable, you can set the random seed before making the plot. You can try different random numbers until you get a result that you like:

```
set.seed(229)
plot(gu)
```

It's also possible to create a graph from a data frame. The first two columns of the data frame are used, and each row specifies a connection between two nodes. In the next example (Figure 13-8), we'll use the `madmen2` data set, which has this structure. We'll also use the Fruchterman-Reingold layout algorithm. The idea is that all the nodes have a magnetic repulsion from one another, but the edges between nodes act as springs, pulling the nodes together:

```
library(gcookbook) # For the data set
madmen2
#>
#> 1           Abe Drexler      Peggy Olson
#> 2           Allison        Don Draper
#> 3           Arthur Case     Betty Draper
#> ...<81 more rows>...
#> 85           Vicky Roger Sterling
#> 86           Waitress        Don Draper
#> 87 Woman at the Clio's party Don Draper
# Create a graph object from the data set
g <- graph.data.frame(madmen2, directed=TRUE)
# Remove unnecessary margins
par(mar = c(0, 0, 0, 0))
plot(g, layout = layout.fruchterman.reingold, vertex.size = 8,
      edge.arrow.size = 0.5, vertex.label = NA)
```

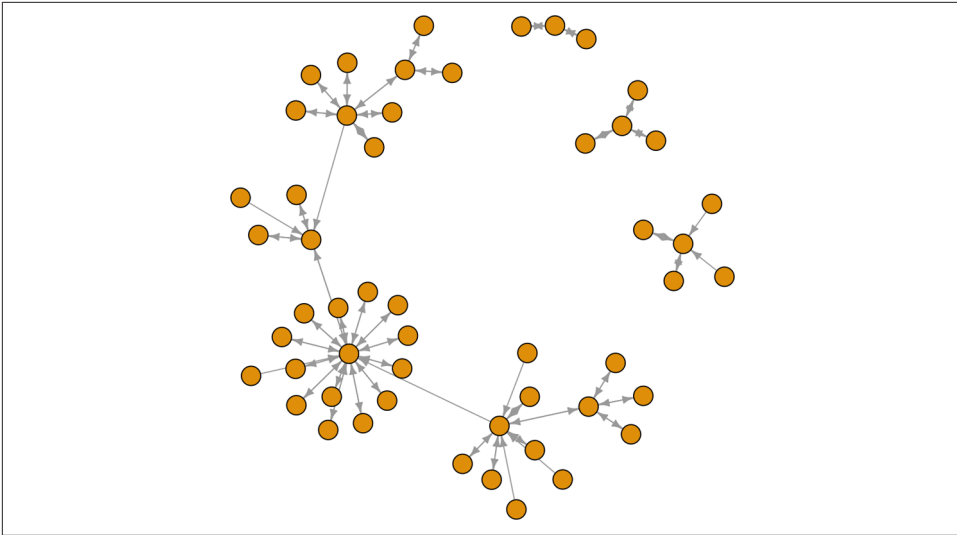


Figure 13-8. A directed graph from a data frame, with the Fruchterman-Reingold algorithm

It's also possible to make a directed graph from a data frame. The `madmen` data set has only one row for each pairing, since direction doesn't matter for an undirected graph. This time we'll use a circle layout (Figure 13-9):

```
g <- graph.data.frame(madmen, directed = FALSE)
par(mar = c(0, 0, 0, 0)) # Remove unnecessary margins
plot(g, layout = layout.circle, vertex.size = 8, vertex.label = NA)
```

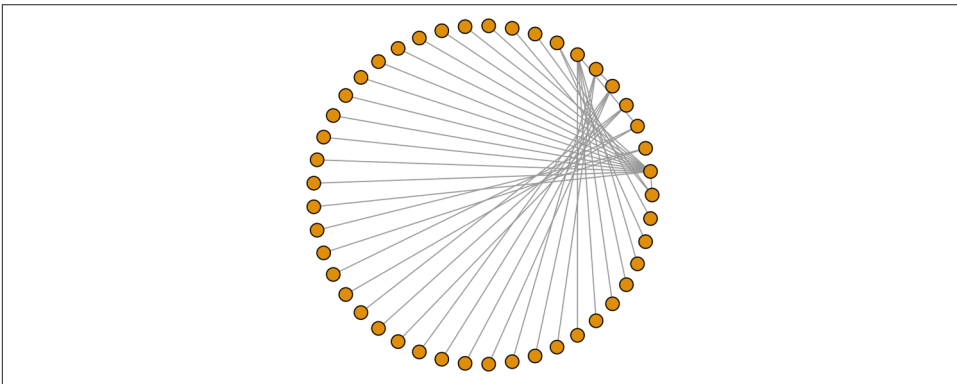


Figure 13-9. A circular undirected graph from a data frame

See Also

For more information about the available output options, see `?plot.igraph`. Also see `?igraph::layout` for layout options.

An alternative to `igraph` is `Rgraphviz`, which is a frontend for `Graphviz`, an open source library for visualizing graphs. It works better with labels and makes it easier to create graphs with a controlled layout, but it can be a bit challenging to install. `Rgraphviz` is available through the Bioconductor repository system.

13.5 Using Text Labels in a Network Graph

Problem

You want to use text labels in a network graph.

Solution

The vertices/nodes may have names, but these names are not used as labels by default. To set the labels, pass in a vector of names to `vertex.label` (Figure 13-10):

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

# Copy madmen and drop every other row
m <- madmen[1:nrow(madmen) %% 2 == 1, ]

g <- graph.data.frame(m, directed=FALSE) # Print out the names of each vertex

V(g)$name
#> [1] "Betty Draper"      "Don Draper"      "Harry Crane"
#> [4] "Joan Holloway"    "Lane Pryce"      "Peggy Olson"
#> [7] "Pete Campbell"    "Roger Sterling"  "Sal Romano"
#> [10] "Henry Francis"    "Allison"         "Candace"
#> [13] "Faye Miller"      "Megan Calvet"    "Rachel Menken"
#> [16] "Suzanne Farrell"  "Hildy"           "Franklin"
#> [19] "Rebecca Pryce"    "Abe Drexler"     "Duck Phillips"
#> [22] "Playtex bra model" "Ida Blankenship" "Mirabelle Ames"
#> [25] "Vicky"            "Kitty Romano"

plot(g, layout=layout.fruchterman.reingold,
     vertex.size = 4, # Smaller nodes
     vertex.label = V(g)$name, # Set the labels
     vertex.label.cex = 0.8, # Slightly smaller font
     vertex.label.dist = 0.4, # Offset the labels
     vertex.label.color = "black")
```

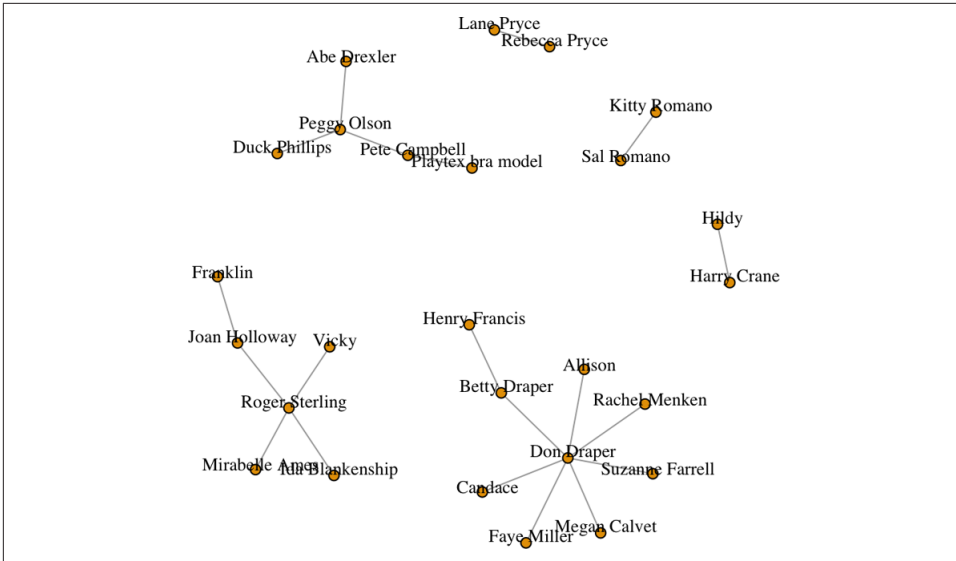


Figure 13-10. A network graph with labels

Discussion

Another way to achieve the same effect is to modify the plot object, instead of passing in the values as arguments to `plot()`. To do this, use `V(g)$xxx<-` instead of passing a value to a `vertex.xxx` argument. For example, this will result in the same output as the previous code:

```
# This is equivalent to the preceding code
V(g)$size      <- 4
V(g)$label     <- V(g)$name
V(g)$label.cex <- 0.8
V(g)$label.dist <- 0.4
V(g)$label.color <- "black"

# Set a property of the entire graph
g$layout <- layout.fruchterman.reingold

plot(g)
```

The properties of the edges can also be set, either with the `E()` function or by passing values to `edge.xxx` arguments (Figure 13-11):

```
# View the edges
E(g)
#> + 20/20 edges from b4d1a80 (vertex names):
#> [1] Betty Draper --Henry Francis    Don Draper    --Allison
#> [3] Betty Draper --Don Draper      Don Draper    --Candace
#> [5] Don Draper   --Faye Miller      Don Draper    --Megan Calvet
```

```

#> [7] Don Draper    --Rachel Menken    Don Draper    --Suzanne Farrell
#> [9] Harry Crane   --Hildy             Joan Holloway  --Franklin
#> [11] Joan Holloway --Roger Sterling   Lane Pryce    --Rebecca Pryce
#> [13] Peggy Olson   --Abe Drexler     Peggy Olson   --Duck Phillips
#> [15] Peggy Olson   --Pete Campbell  Pete Campbell --Playtex bra model
#> [17] Roger Sterling--Ida Blankenship Roger Sterling--Mirabelle Ames
#> [19] Roger Sterling--Vicky          Sal Romano    --Kitty Romano

# Set some of the labels to "M"
E(g)[c(2,11,19)]$label <- "M"

# Set color of all to grey, and then color a few red
E(g)$color <- "grey70"
E(g)[c(2,11,19)]$color <- "red"

plot(g)

```

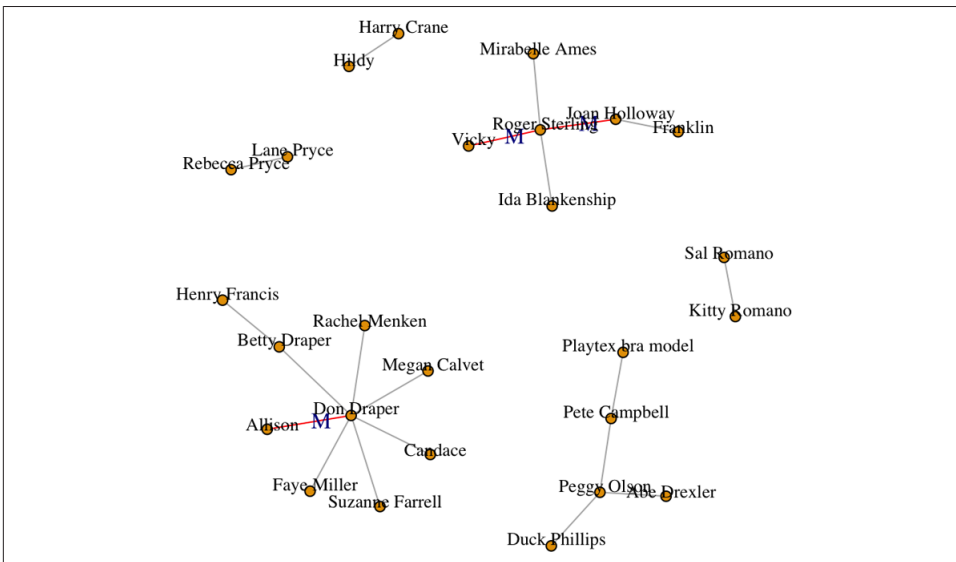


Figure 13-11. A network graph with labeled and colored edges

See Also

See `?igraph.plotting` for more information about graphical parameters in `igraph`.

13.6 Creating a Heat Map

Problem

You want to make a heat map.

Solution

Use `geom_tile()` or `geom_raster()` and map a continuous variable to fill. We'll use the `presidents` data set, which is a time series object rather than a data frame:

```
presidents
#>      Qtr1 Qtr2 Qtr3 Qtr4
#> 1945   NA  87  82   75
#> 1946   63  50  43   32
#> 1947   35  60  54   55
#> ...
#> 1972   49  61  NA   NA
#> 1973   68  44  40   27
#> 1974   28  25  24   24
str(presidents)
#> Time-Series [1:120] from 1945 to 1975: NA 87 82 75 63 50 43 32 35 60 ...
```

We'll first convert it to a format that is usable by `ggplot`—a data frame with columns that are numeric:

```
pres_rating <- data.frame(
  rating = as.numeric(presidents),
  year = as.numeric(floor(time(presidents))),
  quarter = as.numeric(cycle(presidents))
)
pres_rating
#>      rating year quarter
#> 1      NA 1945      1
#> 2      87 1945      2
#> 3      82 1945      3
#> ...<114 more rows>...
#> 118     25 1974      2
#> 119     24 1974      3
#> 120     24 1974      4
```

Now we can make the plot using `geom_tile()` or `geom_raster()` (Figure 13-12). Simply map one variable to `x`, one to `y`, and one to `fill`:

```
# Base plot
p <- ggplot(pres_rating, aes(x = year, y = quarter, fill = rating))

# Using geom_tile()
p + geom_tile()

# Using geom_raster() - looks the same, but a little more efficient
p + geom_raster()
```

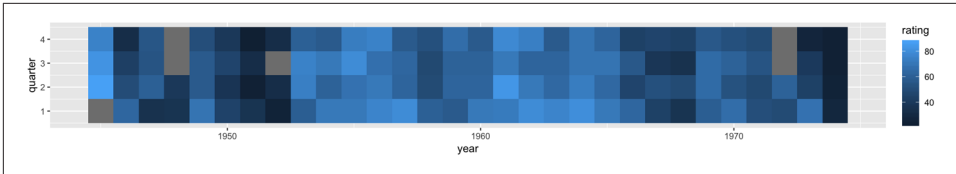


Figure 13-12. A heat map—the grey squares represent NAs in the data



The results with `geom_tile()` and `geom_raster()` *should* look the same, but in practice they might appear different. See [Recipe 6.12](#) for more information about this issue.

Discussion

To better convey useful information, you may want to customize the appearance of the heat map. With this example, we'll reverse the y-axis so that it progresses from top to bottom, and we'll add tick marks every four years along the x-axis, to correspond with each presidential term. For the *x* and *y* scales, we remove the padding by using `expand=c(0, 0)`. We'll also change the color scale using `scale_fill_gradient2()`, which lets you specify a midpoint color and the two colors at the low and high ends ([Figure 13-13](#)):

```
p +
  geom_tile() +
  scale_x_continuous(breaks = seq(1940, 1976, by = 4), expand = c(0, 0)) +
  scale_y_reverse(expand = c(0, 0)) +
  scale_fill_gradient2(midpoint = 50, mid = "grey70", limits = c(0, 100))
```

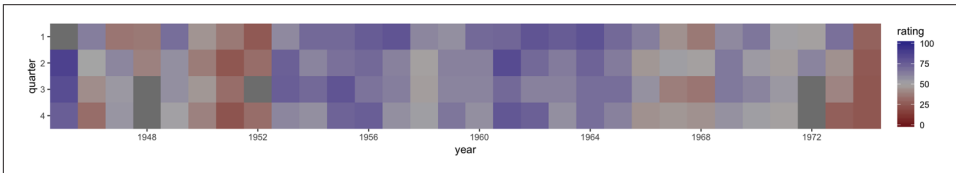


Figure 13-13. A heat map with customized appearance

See Also

If you want to use a different color palette, see [Recipe 12.6](#).

13.7 Creating a Three-Dimensional Scatter Plot

Problem

You want to create a three-dimensional scatter plot.

Solution

We'll use the `rgl` package, which provides an interface to the OpenGL graphics library for 3D graphics. To create a 3D scatter plot, as in **Figure 13-14**, use `plot3d()` and pass in a data frame where the first three columns represent x, y, and z coordinates, or pass in three vectors representing the x, y, and z coordinates:

```
# You may need to install first, with install.packages("rgl")
library(rgl)
plot3d(mtcars$wt, mtcars$displ, mtcars$mpg, type = "s", size = 0.75, lit = FALSE)
```

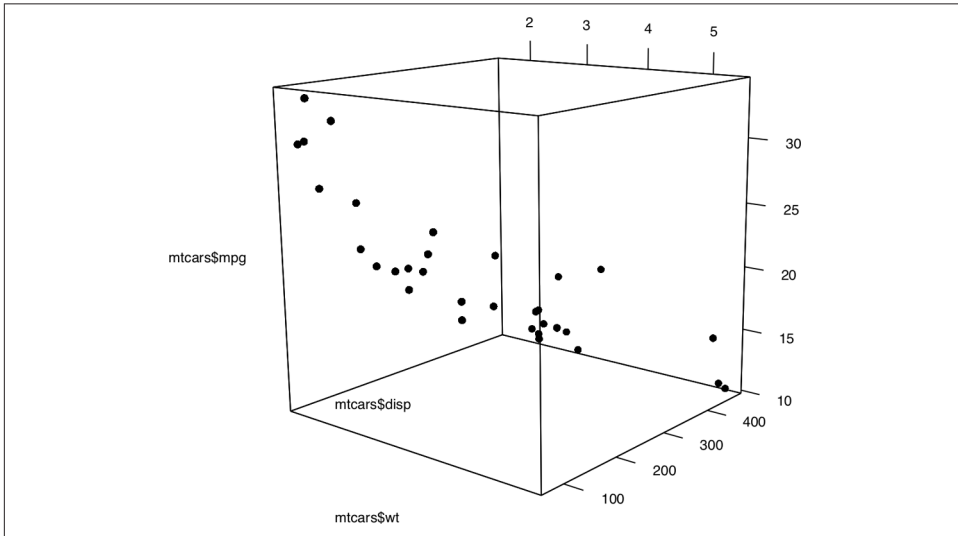


Figure 13-14. A 3D scatter plot



If you are using a Mac, you will need to install XQuartz before you can use `rgl`. You can download it from <https://www.xquartz.org/>.

Viewers can rotate the image by clicking and dragging with the mouse, and zoom in and out with the scroll wheel.



By default, `plot3d()` uses square points, which do not appear properly when saving to a PDF. For improved appearance, the preceding example uses `type="s"` for spherical points, made them smaller with `size=0.75`, and turned off the 3D lighting with `lit=FALSE` (otherwise they look like shiny spheres).

Discussion

Three-dimensional scatter plots can be difficult to interpret, so it's often better to use a two-dimensional representation of the data. That said, there are things that can help make a 3D scatter plot easier to understand.

In [Figure 13-15](#), we'll add vertical segments to help give a sense of the spatial positions of the points:

```
# Function to interleave the elements of two vectors
interleave <- function(v1, v2) as.vector(rbind(v1,v2))

# Plot the points
plot3d(mtcars$wt, mtcars$disp, mtcars$mpg,
       xlab = "Weight", ylab = "Displacement", zlab = "MPG",
       size = .75, type = "s", lit = FALSE)

# Add the segments
segments3d(interleave(mtcars$wt, mtcars$wt),
           interleave(mtcars$disp, mtcars$disp),
           interleave(mtcars$mpg, min(mtcars$mpg))),
          alpha = 0.4, col = "blue")
```

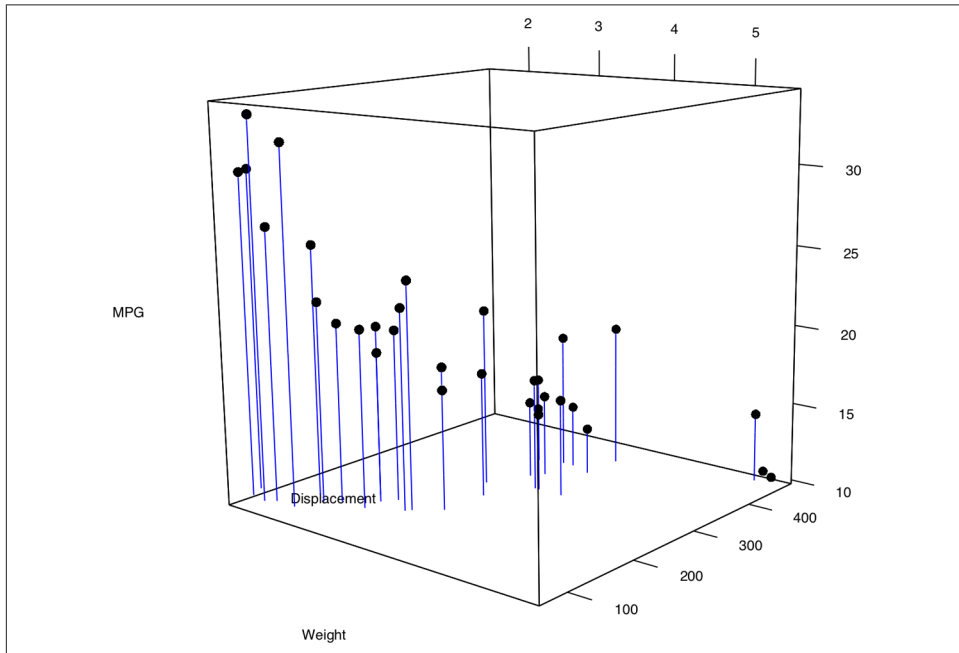


Figure 13-15. A 3D scatter plot with vertical lines for each point

It's possible to tweak the appearance of the background and the axes. In [Figure 13-16](#), we change the number of tick marks and add tick marks and axis labels to the specified sides:

```
# Make plot without axis ticks or labels
plot3d(mtcars$wt, mtcars$disp, mtcars$mpg,
       xlab = "", ylab = "", zlab = "",
       axes = FALSE,
       size = .75, type = "s", lit = FALSE)

segments3d(interleave(mtcars$wt, mtcars$wt),
           interleave(mtcars$disp, mtcars$disp),
           interleave(mtcars$mpg, min(mtcars$mpg)),
           alpha = 0.4, col = "blue")

# Draw the box.
rgl.bbox(color = "grey50",          # grey60 surface and black text
         emission = "grey50",      # emission color is grey50
         xlen = 0, ylen = 0, zlen = 0) # Don't add tick marks

# Set default color of future objects to black
rgl.material(color = "black")

# Add axes to specific sides. Possible values are "x--", "x-+", "x+-", and "x++".
axes3d(edges = c("x--", "y+-", "z--"),
       ntick = 6,                      # Attempt 6 tick marks on each side
       cex = .75)                     # Smaller font

# Add axis labels. 'line' specifies how far to set the label from the axis.
mtext3d("Weight", edge = "x--", line = 2)
mtext3d("Displacement", edge = "y+-", line = 3)
mtext3d("MPG", edge = "z--", line = 3)
```

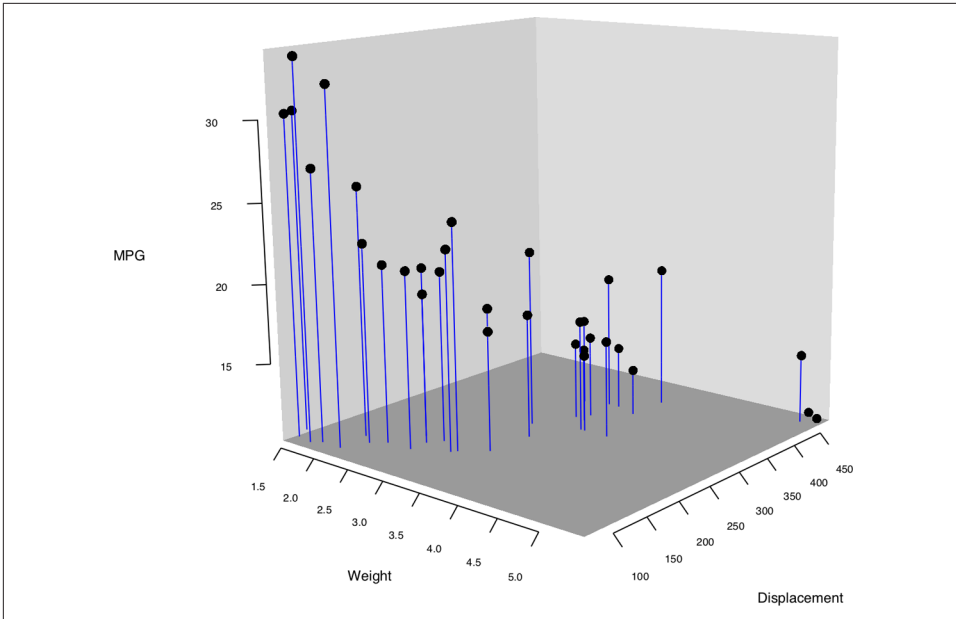


Figure 13-16. 3D scatter plot with axis ticks and labels repositioned

See Also

See `?plot3d` for more options for controlling the output.

13.8 Adding a Prediction Surface to a Three-Dimensional Plot

Problem

You want to add a surface of predicted value to a three-dimensional scatter plot.

Solution

First, we need to define some utility functions for generating the predicted values from a model object:

```
# Given a model, predict zvar from xvar and yvar
# Defaults to range of x and y variables, and a 16x16 grid
predictgrid <- function(model, xvar, yvar, zvar, res = 16, type = NULL) {
  # Find the range of the predictor variable. This works for lm and glm
  # and some others, but may require customization for others.
  xrange <- range(model$model[[xvar]])
  yrange <- range(model$model[[yvar]])
```

```

newdata <- expand.grid(x = seq(xrange[1], xrange[2], length.out = res),
                      y = seq(yrange[1], yrange[2], length.out = res))
names(newdata) <- c(xvar, yvar)
newdata[[zvar]] <- predict(model, newdata = newdata, type = type)
newdata
}

# Convert long-style data frame with x, y, and z vars into a list
# with x and y as row/column values, and z as a matrix.
df2mat <- function(p, xvar = NULL, yvar = NULL, zvar = NULL) {
  if (is.null(xvar)) xvar <- names(p)[1]
  if (is.null(yvar)) yvar <- names(p)[2]
  if (is.null(zvar)) zvar <- names(p)[3]

  x <- unique(p[[xvar]])
  y <- unique(p[[yvar]])
  z <- matrix(p[[zvar]], nrow = length(y), ncol = length(x))

  m <- list(x, y, z)
  names(m) <- c(xvar, yvar, zvar)
  m
}

# Function to interleave the elements of two vectors
interleave <- function(v1, v2) as.vector(rbind(v1,v2))

```

With these utility functions defined, we can make a linear model from the data and plot it as a mesh along with the data, using the `surface3d()` function, as shown in [Figure 13-17](#):

```

library(rgl)

# Make a copy of the data set
m <- mtcars

# Generate a linear model
mod <- lm(mpg ~ wt + disp + wt:disp, data = m)

# Get predicted values of mpg from wt and disp
m$pred_mpg <- predict(mod)

# Get predicted mpg from a grid of wt and disp
mpgrid_df <- predictgrid(mod, "wt", "disp", "mpg")
mpgrid_list <- df2mat(mpgrid_df)

# Make the plot with the data points
plot3d(m$wt, m$disp, m$mpg, type = "s", size = 0.5, lit = FALSE)

# Add the corresponding predicted points (smaller)
spheres3d(m$wt, m$disp, m$pred_mpg, alpha = 0.4, type = "s", size = 0.5,
          lit = FALSE)

```

```
# Add line segments showing the error
segments3d(interleave(m$wt, m$wt),
           interleave(m$disp, m$disp),
           interleave(m$mpg, m$pred_mpg),
           alpha = 0.4, col = "red")

# Add the mesh of predicted values
surface3d(mpgrid_list$wt, mpgrid_list$disp, mpgrid_list$mpg,
          alpha = 0.4, front = "lines", back = "lines")
```

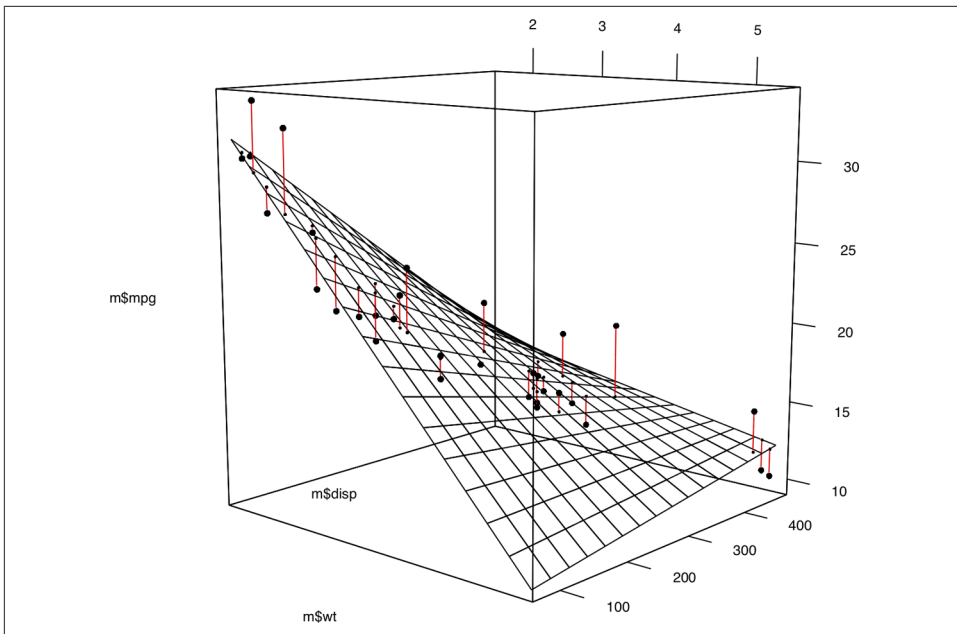


Figure 13-17. A 3D scatter plot with a prediction surface

Discussion

We can tweak the appearance of the graph, as shown in Figure 13-18. We'll add each of the components of the graph separately:

```
plot3d(mtcars$wt, mtcars$disp, mtcars$mpg,
       xlab = "", ylab = "", zlab = "",
       axes = FALSE,
       size = .5, type = "s", lit = FALSE)

# Add the corresponding predicted points (smaller)
spheres3d(m$wt, m$disp, m$pred_mpg, alpha = 0.4, type = "s", size = 0.5,
          lit = FALSE)

# Add line segments showing the error
segments3d(interleave(m$wt, m$wt),
```



```

interleave(m$disp, m$disp),
interleave(m$mpg, m$pred_mpg),
alpha = 0.4, col = "red")

# Add the mesh of predicted values
surface3d(mpgrid_list$wt, mpgrid_list$disp, mpgrid_list$mpg,
          alpha = 0.4, front = "lines", back = "lines")

# Draw the box
rgl.bbox(color = "grey50",          # grey60 surface and black text
          emission = "grey50",      # emission color is grey50
          xlen = 0, ylen = 0, zlen = 0) # Don't add tick marks

# Set default color of future objects to black
rgl.material(color = "black")

# Add axes to specific sides. Possible values are "x--", "x-+", "x+-", and "x++".
axes3d(edges = c("x--", "y+-", "z--"),
        ntick = 6,                      # Attempt 6 tick marks on each side
        cex = .75)                     # Smaller font

# Add axis labels. 'line' specifies how far to set the label from the axis.
mtext3d("Weight",      edge = "x--", line = 2)
mtext3d("Displacement", edge = "y+-", line = 3)
mtext3d("MPG",         edge = "z--", line = 3)

```

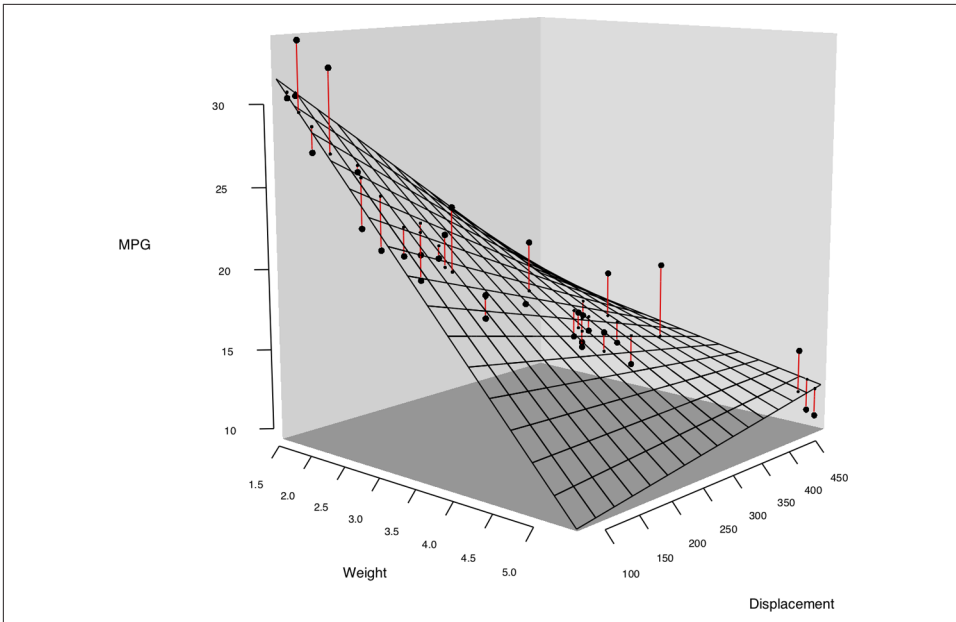


Figure 13-18. Three-dimensional scatter plot with customized appearance

See Also

For more on changing the appearance of the surface, see `?rgl.material`.

13.9 Saving a Three-Dimensional Plot

Problem

You want to save a three-dimensional plot created with the `rgl` package.

Solution

To save a bitmap image of a plot created with `rgl`, use `rgl.snapshot()`. This will capture the exact image that is on the screen:

```
library(rgl)
plot3d(mtcars$wt, mtcars$disp, mtcars$mpg, type = "s", size = 0.75, lit = FALSE)

rgl.snapshot('3dplot.png', fmt = 'png')
```

You can also use `rgl.postscript()` to save a PostScript or PDF file:

```
rgl.postscript('3dplot.pdf', fmt = 'pdf')

rgl.postscript('3dplot.ps', fmt = 'ps')
```

PostScript and PDF output do not support many features of the OpenGL library on which `rgl` is based. For example, they do not support transparency, and the sizes of objects such as points and lines may not be the same as what appears on the screen.

Discussion

To make the output more repeatable, you can save your current viewpoint and restore it later:

```
# Save the current viewpoint
view <- par3d("userMatrix")

# Restore the saved viewpoint
par3d(userMatrix = view)
```

To save view in a script, you can use `dput()`, then copy and paste the output into your script:

```
dput(view)
```

Once you have the text representation of the `userMatrix`, add the following to your script:

```
view <- structure(c(0.907931625843048, 0.267511069774628, -0.322642296552658,
0, -0.410978674888611, 0.417272746562958, -0.810543060302734,
```

```
0, -0.0821993798017502, 0.868516683578491, 0.488796472549438,
0, 0, 0, 0, 1), .Dim = c(4L, 4L))

par3d(userMatrix = view)
```

13.10 Animating a Three-Dimensional Plot

Problem

You want to animate a three-dimensional plot by moving the viewpoint around the plot.

Solution

Rotating a 3D plot can provide a more complete view of the data. To animate a 3D plot, use `play3d()` with `spin3d()`:

```
library(rgl)
plot3d(mtcars$wt, mtcars$disp, mtcars$mpg, type = "s", size = 0.75, lit = FALSE)

play3d(spin3d())
```

Discussion

By default, the graph will be rotated on the z (vertical) axis, until you send a break command to R.

You can change the rotation axis, rotation speed, and duration:

```
# Spin on x-axis, at 4 rpm, for 20 seconds
play3d(spin3d(axis = c(1,0,0), rpm = 4), duration = 20)
```

To save the movie, use the `movie3d()` function in the same way as `play3d()`. It will generate a series of `.png` files, one for each frame, and then attempt to combine them into a single animated `.gif` file using the `convert` program from the ImageMagick image utility.

This will spin the plot once in 15 seconds, at 50 frames per second:

```
# Spin on z axis, at 4 rpm, for 15 seconds
movie3d(spin3d(axis = c(0,0,1), rpm = 4), duration = 15, fps = 50)
```

The output file will be saved in a temporary directory, and the name will be printed on the R console.

If you don't want to use ImageMagick to convert the output to a `.gif`, you can specify `convert=FALSE` and then convert the series of `.png` files to a movie using some other utility.

13.11 Creating a Dendrogram

Problem

You want to make a dendrogram to show how items are clustered.

Solution

Use `hclust()` and plot the output from it. This can require a fair bit of data preprocessing. For this example, we'll first take a subset of the `countries` data set from the year 2009. For simplicity, we'll also drop all rows that contain an NA, and then select a random 25 of the remaining rows:

```
library(dplyr)
library(tidyr)      # For drop_na function
library(gcookbook) # For the data set

# Set random seed to make random operation below repeatable
set.seed(392)

c2 <- countries %>%
  filter(Year == 2009) %>% # Get data from year 2009
  drop_na() %>%           # Drop rows that have any NA values
  sample_n(25)            # Select 25 random rows

c2
#>           Name Code Year      GDP laborrate  healthexp infmortality
#> 111  Liberia  LBR 2009  229.2703      71.1   29.35613      77.6
#> 86   Hungary  HUN 2009 12847.3031     50.1  937.98617       5.7
#> 194    Togo   TGO 2009   534.8508     74.4  28.93053      67.1
#> ...<19 more rows>...
#> 19    Belgium BEL 2009 43640.1962     53.5 5104.01899       3.6
#> 53    Denmark DNK 2009 55933.3545     65.4 6272.72868       3.4
#> 199 Turkmenistan TKM 2009 3710.4536     68.0  77.06955      48.0
```

Notice that the row names (the first column) are essentially random numbers, since the rows were selected randomly. We need to do a few more things to the data before making a dendrogram from it. First, we need to set the *row names*—right now there's a column called `Name`, but the row names are those random numbers (we don't often use row names, but for the `hclust()` function they're essential). Next, we'll need to drop all the columns that aren't values used for clustering. These columns are `Name`, `Code`, and `Year`:

```
rownames(c2) <- c2$Name
c2 <- c2[, 4:7]
c2
#>           GDP laborrate  healthexp infmortality
#> Liberia      229.2703      71.1   29.35613      77.6
#> Hungary     12847.3031     50.1  937.98617       5.7
```

```
#> Togo          534.8508      74.4  28.93053      67.1
#> ...<19 more rows>...
#> Belgium    43640.1962     53.5 5104.01899      3.6
#> Denmark    55933.3545     65.4 6272.72868      3.4
#> Turkmenistan 3710.4536     68.0  77.06955     48.0
```

The values for GDP are several orders of magnitude larger than the values for, say, infmortality. Because of this, the effect of infmortality on the clustering will be negligible compared to the effect of GDP. This probably isn't what we want. To address this issue, we'll scale the data:

```
c3 <- scale(c2)
c3
#>          GDP laborrate healthexp infmortality
#> Liberia   -0.70164181  1.0118324 -0.5949323   1.8611870
#> Hungary   -0.06608287 -1.2419310 -0.1825555  -0.8070883
#> Togo      -0.68624999  1.3659953 -0.5951254   1.4715223
#> ...<19 more rows>...
#> Belgium    1.48492740 -0.8770359  1.7081758  -0.8850213
#> Denmark    2.10412271  0.4000967  2.2385884  -0.8924435
#> Turkmenistan -0.52629774  0.6791340 -0.5732778   0.7627037
```

By default the `scale()` function scales each column relative to its standard deviation, but other methods may be used.

Finally, we're ready to make the dendrogram, as shown in [Figure 13-19](#):

```
hc <- hclust(dist(c3))

# Make the dendrogram
plot(hc)

# With text aligned
plot(hc, hang = -1)
```

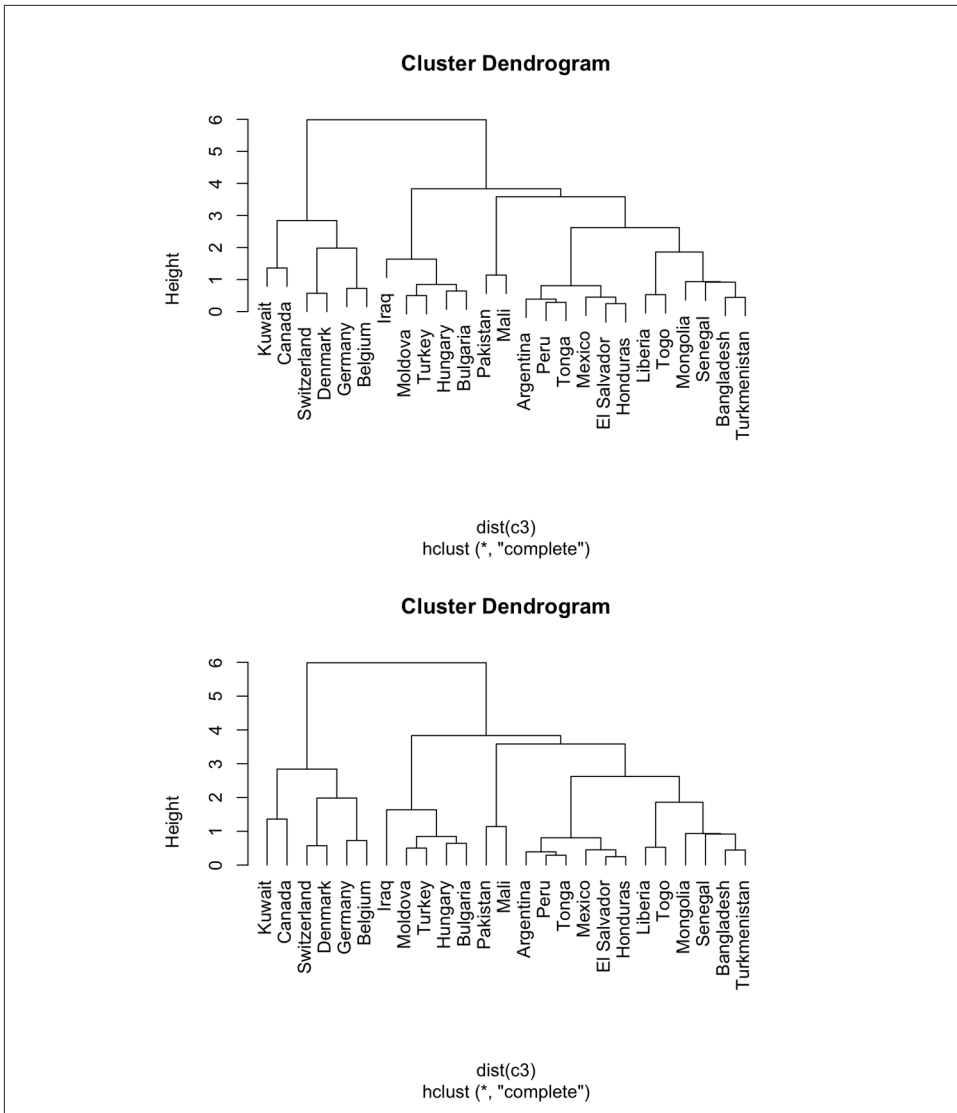


Figure 13-19. A dendrogram (top); With text aligned (bottom)

Discussion

A cluster analysis is simply a way of assigning points to groups in an n -dimensional space (four dimensions, in this example). A hierarchical cluster analysis divides each group into two smaller groups, and can be represented with the dendrograms in this recipe. There are many different parameters you can control in the hierarchical cluster analysis process, and there may not be a single “right” way to do it for your data.

First, we normalized the data using `scale()` with its default settings. You can scale your data differently, or not at all. (With this data set, *not* scaling the data will lead to GDP overwhelming the other variables, as shown in [Figure 13-20](#).)

```
hc_unscaled <- hclust(dist(c2))
plot(hc_unscaled)
```

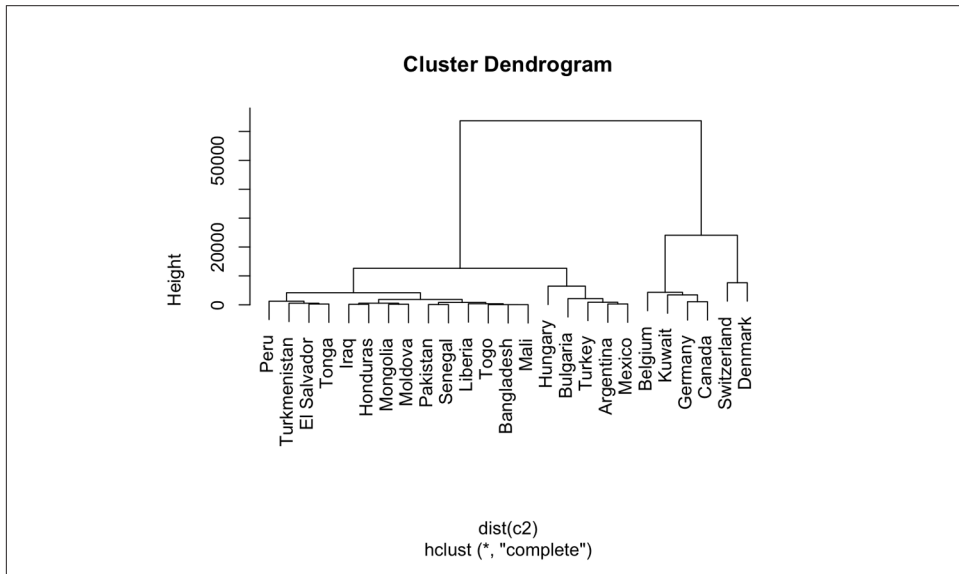


Figure 13-20. Dendrogram with unscaled data—notice the much larger Height values, which are largely due to the unscaled GDP values

For the distance calculation, we used the default method, "euclidean", which calculates the Euclidean distance between the points. The other possible methods are "maximum", "manhattan", "canberra", "binary", and "minkowski".

The `hclust()` function provides several methods for performing the cluster analysis. The default is "complete"; the other possible methods are "ward", "single", "average", "mcquitty", "median", and "centroid".

See Also

See `?hclust` for more information about the different clustering methods.

13.12 Creating a Vector Field

Problem

You want to make a vector field.

Solution

Use `geom_segment()`. For this example, we'll use the `isabel` data set:

```
library(gcookbook) # For the isabel data set
isabel
#>      x      y      z      vx      vy      vz      t      speed
#> 1 -83.000 41.700 0.035      NA      NA      NA      NA      NA
#> 2 -83.000 41.555 0.035      NA      NA      NA      NA      NA
#> 3 -83.000 41.411 0.035      NA      NA      NA      NA      NA
#> ...<156,244 more rows>...
#> 156248 -62.126 24.096 18.035 -11.397 -5.3151 0.009657 -66.995 12.575
#> 156249 -62.126 23.952 18.035 -11.379 -5.2750 0.040921 -67.000 12.542
#> 156250 -62.126 23.808 18.035 -12.166 -5.4358 0.030216 -66.980 13.325
```

`x` and `y` are the longitude and latitude, respectively, and `z` is the height in kilometers. The `vx`, `vy`, and `vz` values are the wind speed components in each of these directions, in meters per second, and `speed` is the wind speed.

The height (`z`) ranges from 0.035 km to 18.035 km. For this example, we'll just use the lowest slice of data.

To draw the vectors (Figure 13-21), we'll use `geom_segment()`. Each segment has a starting point and an ending point. We'll use the `x` and `y` values as the starting points for each segment, then add a fraction of the `vx` and `vy` values to get the end points for each segment. If we didn't scale down these values, the lines would be much too long:

```
islice <- filter(isabel, z == min(z))

ggplot(islice, aes(x = x, y = y)) +
  geom_segment(aes(xend = x + vx/50, yend = y + vy/50),
    size = 0.25) # Make the line segments 0.25 mm thick
```

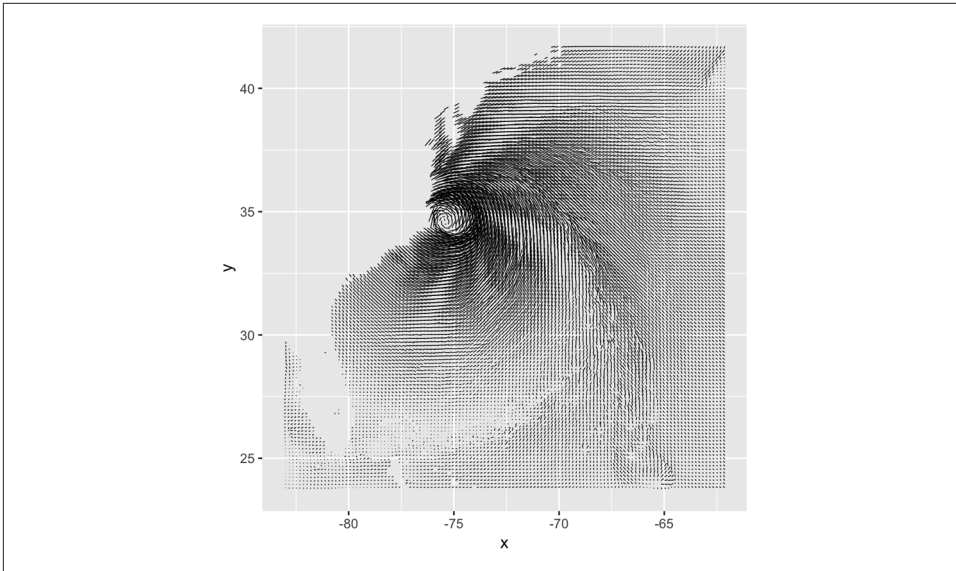



Figure 13-21. First attempt at a vector field. The resolution of the data is too high, but it does hint at some interesting patterns not visible in graphs with a lower data resolution.

This vector field has two problems: the data is at too high a resolution to read, and the segments do not have arrows indicating the direction of the flow. To reduce the resolution of the data, we'll define a function `every_n()` that keeps one out of every `n` values in the data and drops the rest:

```
# Take a slice where z is equal to the minimum value of z
islice <- filter(isabel, z == min(z))

# Keep 1 out of every 'by' values in vector x
every_n <- function(x, by = 2) {
  x <- sort(x)
  x[seq(1, length(x), by = by)]
}

# Keep 1 of every 4 values in x and y
keepx <- every_n(unique(isabel$x), by = 4)
keepy <- every_n(unique(isabel$y), by = 4)

# Keep only those rows where x value is in keepx and y value is in keepy
islicesub <- filter(islice, x %in% keepx & y %in% keepy)
```

Now that we've taken a subset of the data, we can plot it, with arrowheads, as shown in Figure 13-22:

```
# Need to load grid for arrow() function
library(grid)
```

```
# Make the plot with the subset, and use an arrowhead 0.1 cm long
ggplot(islicesub, aes(x = x, y = y)) +
  geom_segment(aes(xend = x+vx/50, yend = y+vy/50),
    arrow = arrow(length = unit(0.1, "cm")), size = 0.25)
```

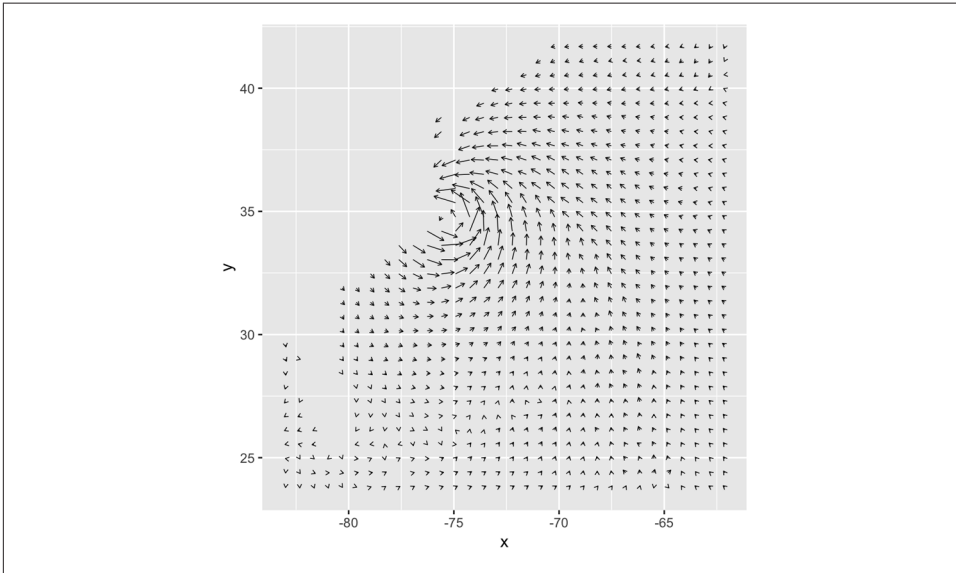


Figure 13-22. Vector field with arrowheads

Discussion

One effect of arrowheads is that short vectors appear with more ink than is proportional to their length. This could somewhat distort the interpretation of the data. To mitigate this effect, it may also be useful to map the speed to other properties, like size (line thickness), alpha, or colour. Here, we'll map speed to alpha (Figure 13-23, left):

```
# The existing 'speed' column includes the z component. We'll calculate
# speedxy, the horizontal speed.
islicesub$speedxy <- sqrt(islicesub$vx^2 + islicesub$vy^2)

# Map speed to alpha
ggplot(islicesub, aes(x = x, y = y)) +
  geom_segment(aes(xend = x+vx/50, yend = y+vy/50, alpha = speed),
    arrow = arrow(length = unit(0.1, "cm")), size = 0.6)
```

Next, we'll map speed to colour. We'll also add a map of the United States and zoom in on the area of interest, as shown in the graph on the right in Figure 13-23, using `coord_cartesian()` (without this, the entire USA will be displayed):

```
# Get USA map data
usa <- map_data("usa")
```

```
# Map speed to colour, and set go from "grey80" to "darkred"
ggplot(islicesub, aes(x = x, y = y)) +
  geom_segment(aes(xend = x+vx/50, yend = y+vy/50, colour = speed),
    arrow = arrow(length = unit(0.1,"cm")), size = 0.6) +
  scale_colour_continuous(low = "grey80", high = "darkred") +
  geom_path(aes(x = long, y = lat, group = group), data = usa) +
  coord_cartesian(xlim = range(islicesub$x), ylim = range(islicesub$y))
```

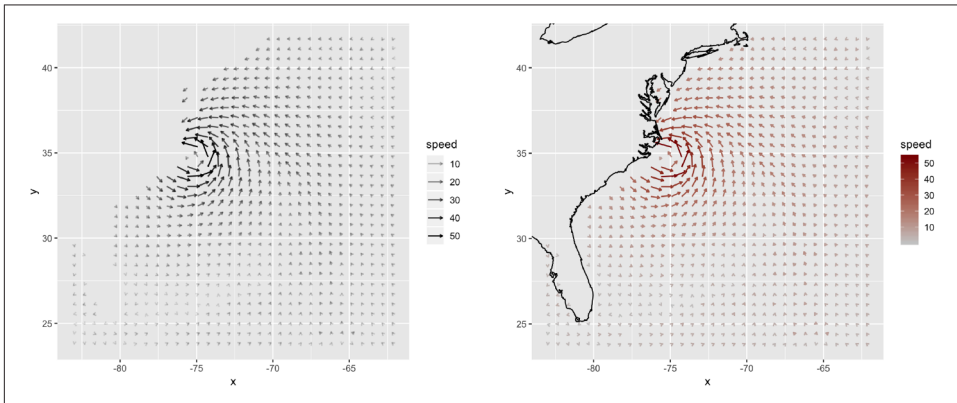


Figure 13-23. Vector field with speed mapped to alpha (left); With speed mapped to colour (right)

The isabel data set has three-dimensional data, so we can also make a faceted graph of the data, as shown in Figure 13-24. Because each facet is small, we will use a sparser subset than before:

```
# Keep 1 out of every 5 values in x and y, and 1 in 2 values in z
keepx <- every_n(unique(isabel$x), by = 5)
keepy <- every_n(unique(isabel$y), by = 5)
keepz <- every_n(unique(isabel$z), by = 2)

isub <- filter(isabel, x %in% keepx & y %in% keepy & z %in% keepz)

ggplot(isub, aes(x = x, y = y)) +
  geom_segment(aes(xend = x+vx/50, yend = y+vy/50, colour = speed),
    arrow = arrow(length = unit(0.1,"cm")), size = 0.5) +
  scale_colour_continuous(low = "grey80", high = "darkred") +
  facet_wrap(~ z)
```

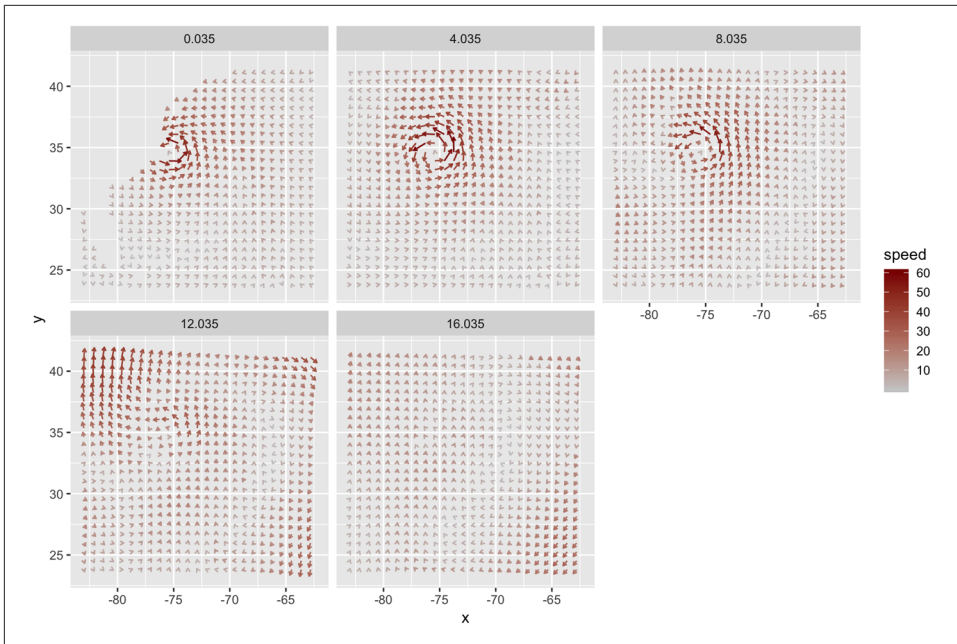


Figure 13-24. Vector field of wind speeds, faceted on z

See Also

If you want to use a different color palette, see [Recipe 12.6](#).

See [Recipe 8.2](#) for more information about zooming in on part of a graph.

13.13 Creating a QQ Plot

Problem

You want to make a quantile-quantile (QQ) plot to compare an empirical distribution to a theoretical distribution.

Solution

Use `geom_qq()` and `geom_qq_line()` to compare to a normal distribution ([Figure 13-25](#)):

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

ggplot(heightweight, aes(sample = heightIn)) +
  geom_qq() +
  geom_qq_line()
```

```
ggplot(heightweight, aes(sample = ageYear)) +
  geom_qq() +
  geom_qq_line()
```

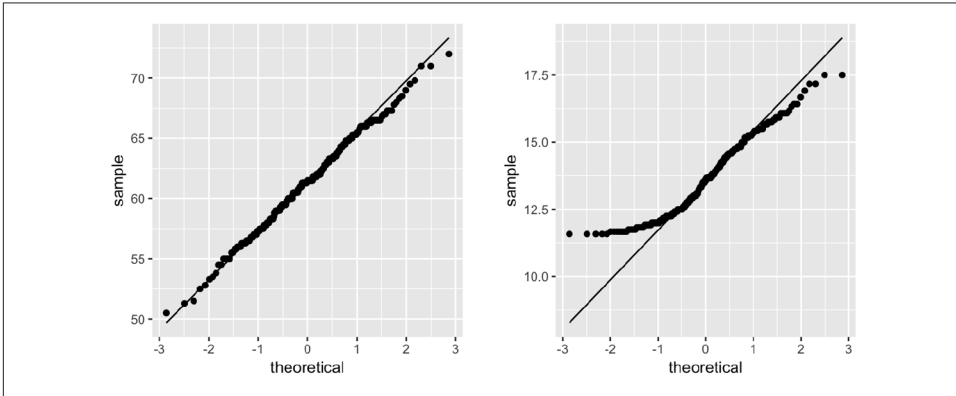


Figure 13-25. QQ plot of height, which is close to normally distributed (left); QQ plot of age, which is not normally distributed (right)

Discussion

The points for heightIn are close to the line, which means that the distribution is close to normal. In contrast, the points for ageYear veer far away from the line, especially on the left, indicating that the distribution is skewed. A histogram may also be useful for exploring how the data is distributed.

See Also

See `?stat_qq` for information on comparing data to theoretical distributions other than the normal distribution.

13.14 Creating a Graph of an Empirical Cumulative Distribution Function

Problem

You want to graph the empirical cumulative distribution function (ECDF) of a data set.

Solution

Use `stat_ecdf()` (Figure 13-26):

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

```
# ecdf of heightIn
ggplot(heightweight, aes(x = heightIn)) +
  stat_ecdf()

# ecdf of ageYear
ggplot(heightweight, aes(x = ageYear)) +
  stat_ecdf()
```

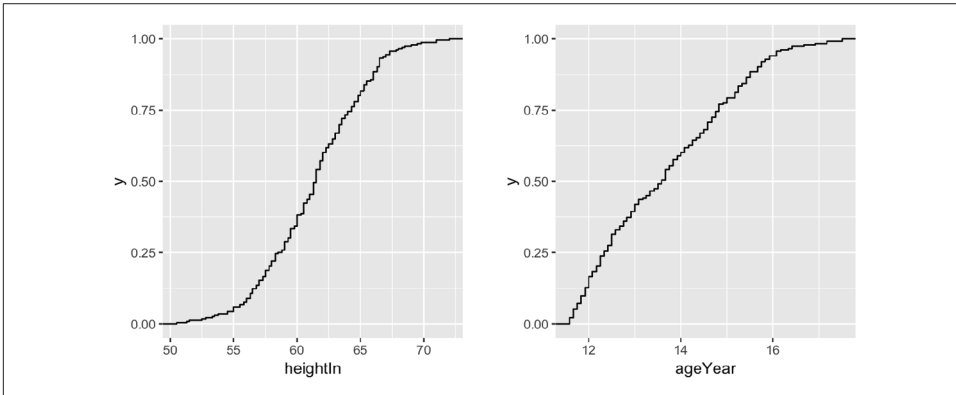


Figure 13-26. ECDF of height (left); ECDF of age (right)

Discussion

The ECDF shows what proportion of observations are at or below the given x value. Because it is *empirical*, the line takes a step up at each x value where there are one or more observations.

13.15 Creating a Mosaic Plot

Problem

You want to make a mosaic plot to visualize a contingency table.

Solution

Use the `mosaic()` function from the `vcd` package. For this example we'll use the `UCBAdmissions` data set, which is a contingency table with three dimensions. We'll first take a look at the data in a few different ways:

```
UCBAdmissions
#> , , Dept = A
#>
#>      Gender
#> Admit   Male Female
#> Admitted  512     89
#> Rejected  313     19
```

```

#>
#> , , Dept = B
#>
#>      Gender
#> Admit      Male Female
#>  Admitted  353     17
#>  Rejected  207      8
#>
#> ... with 41 more lines of text

# Print a "flat" contingency table
ftable(UCBAdmissions)
#>      Dept      A      B      C      D      E      F
#> Admit  Gender
#> Admitted Male      512 353 120 138  53  22
#>      Female      89  17 202 131  94  24
#> Rejected Male      313 207 205 279 138 351
#>      Female      19   8 391 244 299 317

dimnames(UCBAdmissions)
#> $Admit
#> [1] "Admitted" "Rejected"
#>
#> $Gender
#> [1] "Male"      "Female"
#>
#> $Dept
#> [1] "A" "B" "C" "D" "E" "F"

```

The three dimensions are Admit, Gender, and Dept. To visualize the relationships between the variables (Figure 13-27), use `mosaic()` and pass it a formula with the variables that will be used to split up the data:

```

# You may need to install first, with install.packages("vcd")
library(vcd)
# Split by Admit, then Gender, then Dept
mosaic( ~ Admit + Gender + Dept, data = UCBAdmissions)

```

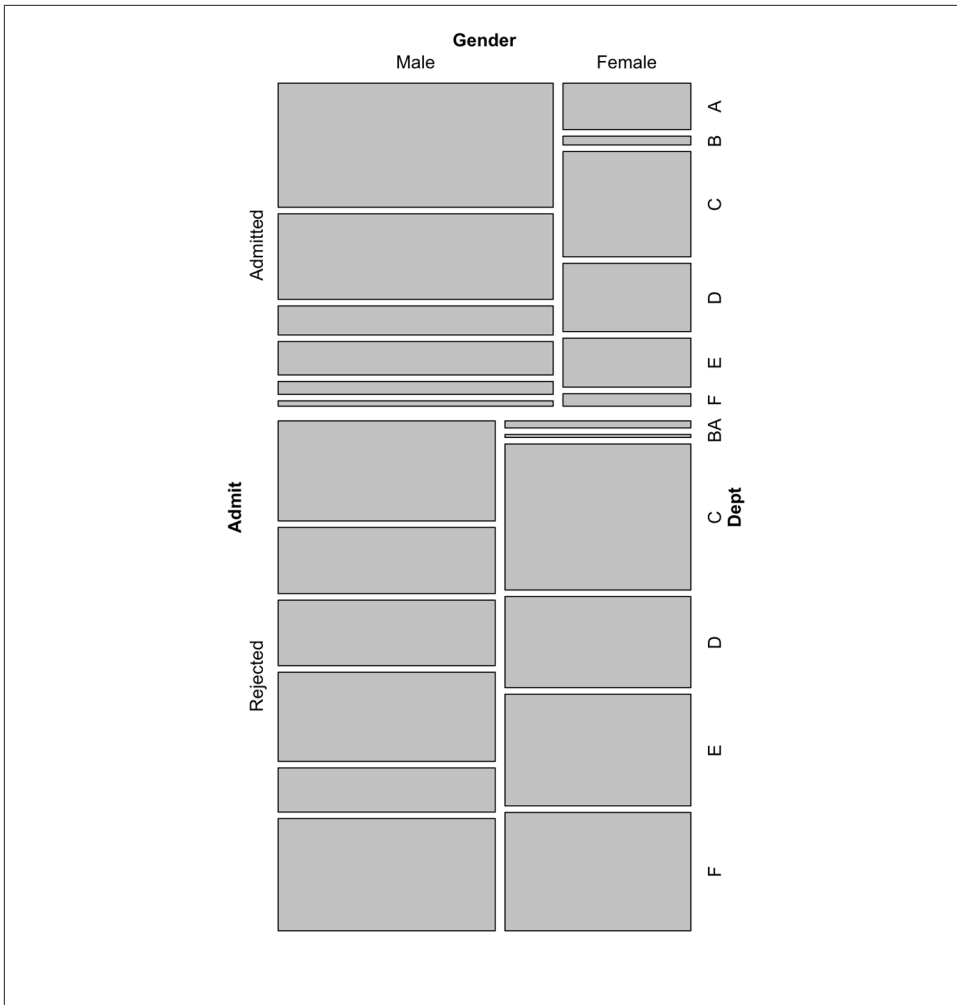


Figure 13-27. Mosaic plot of UC-Berkeley admissions data—the area of each rectangle is proportional to the number of cases in that cell

Notice that `mosaic()` splits the data in the order in which the variables are provided: first on admission status, then gender, then department. The resulting plot order makes it very clear that more applicants were rejected than admitted. It is also clear that within the admitted group there were many more men than women, while in the rejected group there were approximately the same number of men and women. It is difficult to make comparisons within each department, though. A different variable splitting order may reveal some other interesting information.

Another way of looking at the data is to split first by department, then gender, then admission status, as in [Figure 13-28](#). This makes the admission status the last variable

that is partitioned, so that *after* partitioning by department and gender, the admitted and rejected cells for each group are right next to each other:

```
mosaic( ~ Dept + Gender + Admit, data = UCBAmissions,
  highlighting = "Admit", highlighting_fill = c("lightblue", "pink"),
  direction = c("v", "h", "v"))
```

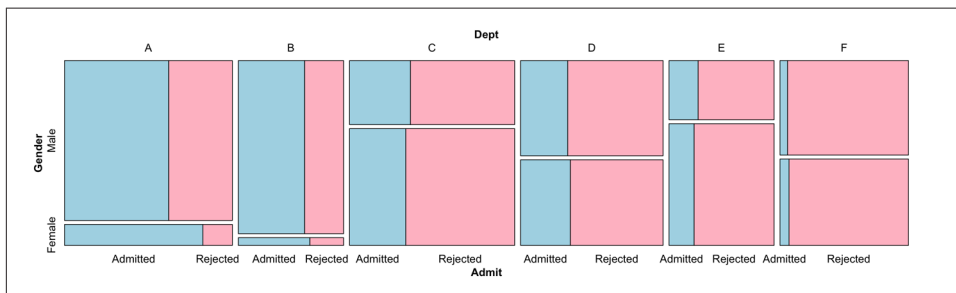


Figure 13-28. Mosaic plot with a different variable splitting order: first department, then gender, then admission status

We also specified a variable to highlight (Admit), and which colors to use in the highlighting.

Discussion

In the preceding example we also specified the *direction* in which each variable will be split. The first variable, Dept, is split vertically; the second variable, Gender, is split horizontally; and the third variable, Admit, is split vertically. The reason that we chose these directions is because, in this particular example, it makes it easy to compare the male and female groups within each department.

We can also use different splitting directions, as shown in Figures 13-29 and 13-30:

```
# Another possible set of splitting directions
mosaic( ~ Dept + Gender + Admit, data = UCBAmissions,
  highlighting = "Admit", highlighting_fill = c("lightblue", "pink"),
  direction = c("v", "v", "h"))
```

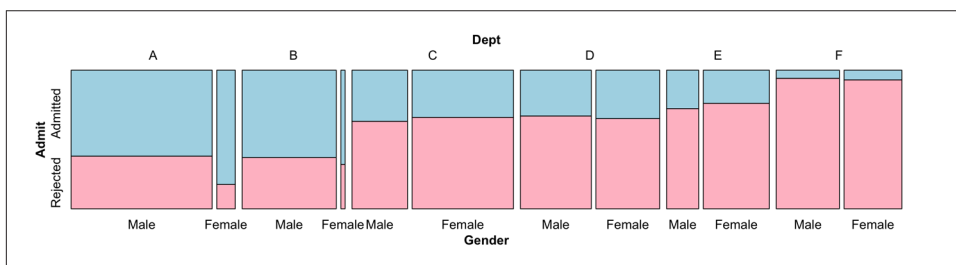


Figure 13-29. Splitting Dept vertically, Gender vertically, and Admit horizontally

```
# This order makes it difficult to compare male and female
mosaic( ~ Dept + Gender + Admit, data = UCBAAdmissions,
  highlighting = "Admit", highlighting_fill = c("lightblue", "pink"),
  direction = c("v", "h", "h"))
```



Figure 13-30. Splitting Dept vertically, Gender horizontally, and Admit horizontally

The example here illustrates a classic case of Simpson's paradox, in which a relationship between variables within subgroups can change (or reverse!) when the groups are combined. The UC Berkeley table contains admissions data from the University of California-Berkeley in 1973. Overall, men were admitted at a higher rate than women, and because of this, the university was sued for gender bias. But when each department was examined separately, it was found that they each had approximately equal admission rates for men and women. The difference in overall admission rates was because women were more likely to apply to competitive departments with lower admission rates.

In Figures 13-28 and 13-29, you can see that within each department, admission rates were approximately equal between men and women. You can also see that departments with higher admission rates (A and B) were very imbalanced in the gender ratio of applicants: far more men applied to these departments than did women. As you can see, partitioning the data in different orders and directions can bring out different aspects of the data. In Figure 13-29, as in Figure 13-28, it's easy to compare male and female admission rates within each department and across departments. Splitting Dept vertically, Gender horizontally, and Admit horizontally, as in Figure 13-30, makes it difficult to compare male and female admission rates within each department, but it is easy to compare male and female application rates across departments.

See Also

See `?mosaicplot` for another function that can create mosaic plots.

P.J. Bickel, E.A. Hammel, and J.W. O'Connell, "Sex Bias in Graduate Admissions: Data from Berkeley," *Science* 187 (1975): 398–404.

13.16 Creating a Pie Chart

Problem

You want to make a pie chart.

Solution

Use the `pie()` function. In this example (Figure 13-31), we'll use the survey data set from the MASS library:

```
library(MASS) # For the data set
# Get a table of how many cases are in each level of fold
fold <- table(survey$Fold)
fold

# Reduce margins so there's less blank space around the plot
par(mar = c(1, 1, 1, 1))
# Make the pie chart
pie(fold)
```

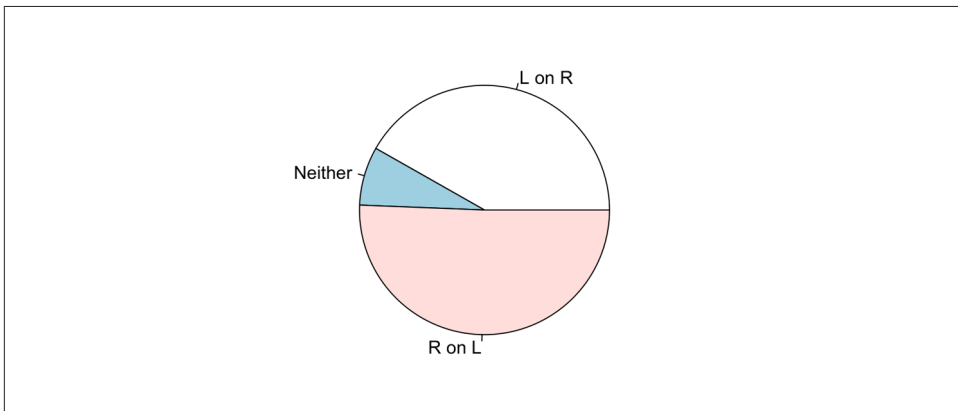


Figure 13-31. A pie chart

We passed `pie()` an object of class `table`. We could have instead given it a named vector, or a vector of values and a vector of labels like this, with the same result:

```
pie(c(99, 18, 120), labels = c("L on R", "Neither", "R on L"))
```

Discussion

The lowly pie chart is the subject of frequent abuse from data visualization experts. If you're thinking of using a pie chart, consider whether a bar graph (or stacked bar graph) would convey the information more effectively. Despite their faults, pie charts do have one important virtue: everyone knows how to read them.

13.17 Creating a Map

Problem

You want to create a geographical map.

Solution

Retrieve map data from the `maps` package and draw it with `geom_polygon()` (which can have a color fill) or `geom_path()` (which can't have a fill). By default, the latitude and longitude will be drawn on a Cartesian coordinate plane, but you can use `coord_map()` and specify a projection. The default projection is "mercator", which, unlike the Cartesian plane, has a progressively changing spacing for latitude lines (Figure 13-32):

```
library(maps) # For map data
# Get map data for USA
states_map <- map_data("state") # ggplot2 must be loaded to use map_data()

ggplot(states_map, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill = "white", colour = "black")

# geom_path (no fill) and Mercator projection
ggplot(states_map, aes(x = long, y = lat, group = group)) +
  geom_path() + coord_map("mercator")
```

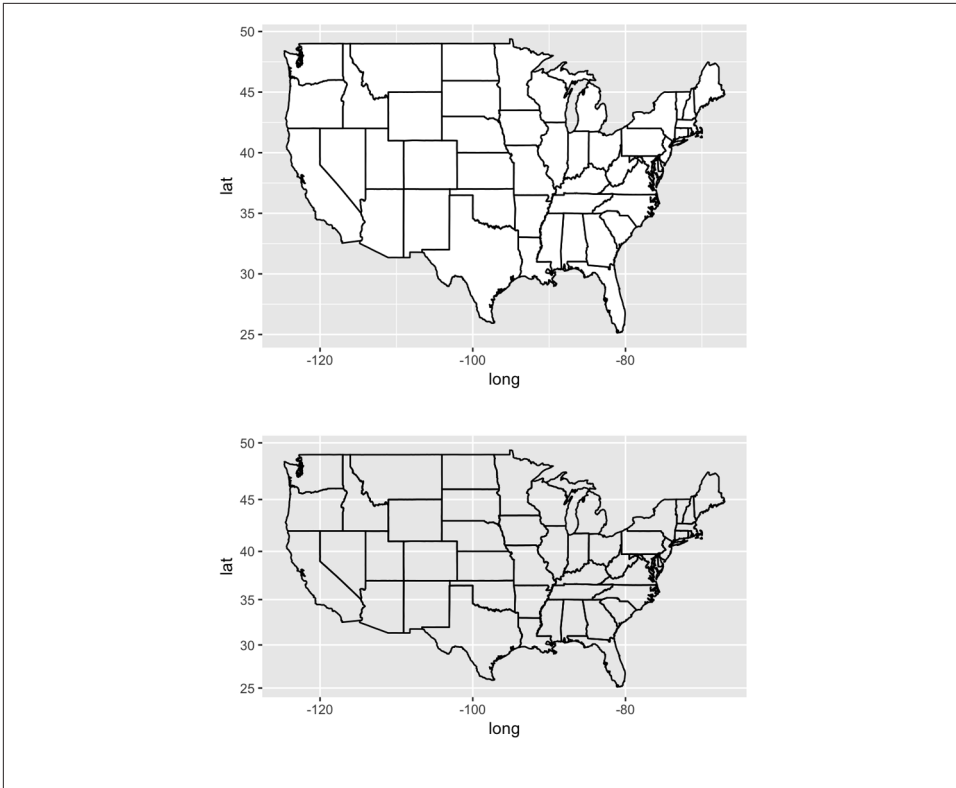


Figure 13-32. Top: a basic map with fill; Bottom: with no fill, and Mercator projection

Discussion

The `map_data()` function returns a data frame with the following columns:

`long`

Longitude.

`lat`

Latitude.

`group`

This is a grouping variable for each polygon. A region or subregion might have multiple polygons, for example, if it includes islands.

`order`

The order to connect each point within a group.

region

Roughly, the names of countries, although some other objects are present (such as some lakes).

subregion

The names of subregions within a region, which can contain multiple groups. For example, the Alaska subregion includes many islands, each with its own group.

There are a number of different maps available, including `world`, `nz`, `france`, `italy`, `usa` (outline of the United States), `state` (each state in the USA), and `county` (each county in the USA). For example, to get map data for the world:

```
# Get map data for world
world_map <- map_data("world")
world_map
#>           long      lat group  order  region subregion
#> 1    -69.89912 12.45200     1     1   Aruba      <NA>
#> 2    -69.89571 12.42300     1     2   Aruba      <NA>
#> 3    -69.94219 12.43853     1     3   Aruba      <NA>
#> ...<99,332 more rows>...
#> 100962 12.42754 41.90073 1627 100962 Vatican  enclave
#> 100963 12.43057 41.89756 1627 100963 Vatican  enclave
#> 100964 12.43916 41.89839 1627 100964 Vatican  enclave
```

If you want to draw a map of a region in the world map for which there isn't a separate map, you can first look for the region name, like so:

```
sort(unique(world_map$region))
#> [1] "Afghanistan"           "Albania"
#> [3] "Algeria"                "American Samoa"
#> [5] "Andorra"                "Angola"
#> ...
#> [247] "Virgin Islands"         "Wallis and Futuna"
#> [249] "Western Sahara"         "Yemen"
#> [251] "Zambia"                 "Zimbabwe"
```

It's possible to get data for specific regions from a particular map (Figure 13-33):

```
east_asia <- map_data("world", region = c("Japan", "China", "North Korea",
                                           "South Korea"))

# Map region to fill color
ggplot(east_asia, aes(x = long, y = lat, group = group, fill = region)) +
  geom_polygon(colour = "black") +
  scale_fill_brewer(palette = "Set2")
```

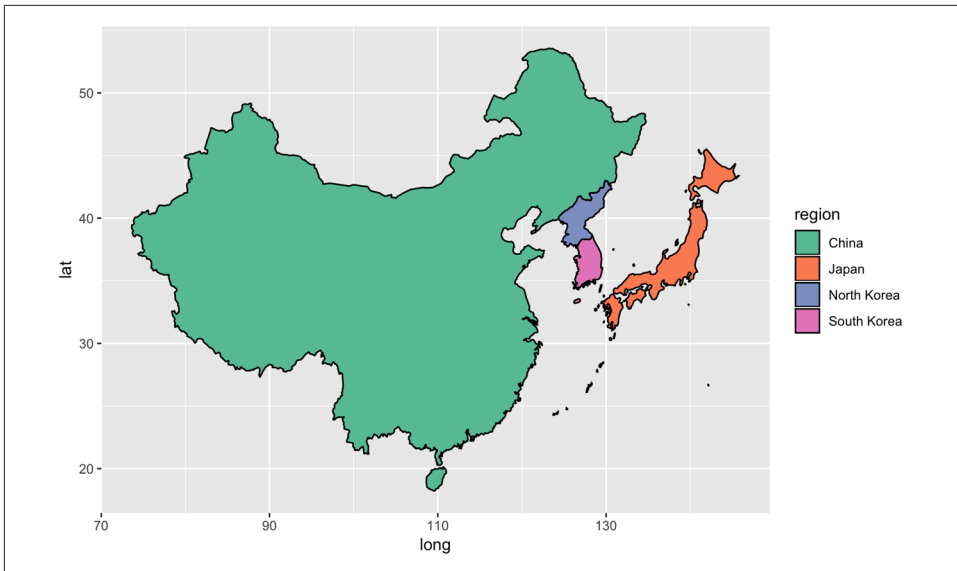


Figure 13-33. Specific regions from the world map

If there is a separate map available for a region, such as `nz` (New Zealand), that map data will be at a higher resolution than if you were to extract it from the world map, as shown in Figure 13-34:

```
# Get New Zealand data from world map
nz1 <- map_data("world", region = "New Zealand") %>%
  filter(long > 0 & lat > -48) # Trim off islands

ggplot(nz1, aes(x = long, y = lat, group = group)) +
  geom_path()

# Get New Zealand data from the nz map
nz2 <- map_data("nz")
ggplot(nz2, aes(x = long, y = lat, group = group)) +
  geom_path()
```

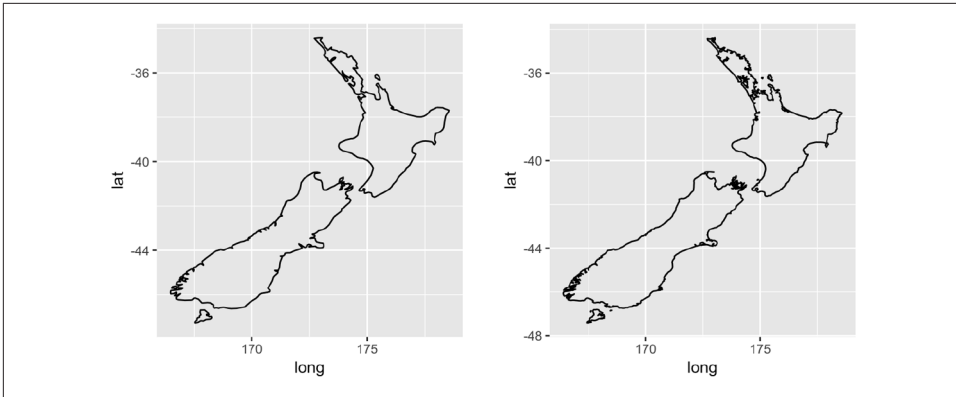


Figure 13-34. New Zealand data taken from world map (left); Data from nz map (right)

See Also

See the `mapdata` package for more map data sets. It includes maps of China and Japan, as well as a high-resolution world map, `worldHires`.

See the `map()` function for quickly generating maps.

See `?mapproject` for a list of available map projections.

13.18 Creating a Choropleth Map

Problem

You want to create a map with regions that are colored according to variable values.

Solution

Merge the value data with the map data, then map a variable to fill:

```
# Transform the USArrests data set to the correct format
crimes <- data.frame(state = tolower(rownames(USArrests)), USArrests)
crimes
#>           state Murder Assault UrbanPop Rape
#> Alabama      alabama  13.2    236     58 21.2
#> Alaska       alaska   10.0    263     48 44.5
#> Arizona      arizona    8.1    294     80 31.0
#> ...<44 more rows>...
#> West Virginia west virginia  5.7     81     39  9.3
#> Wisconsin    wisconsin    2.6     53     66 10.8
#> Wyoming      wyoming    6.8    161     60 15.6

library(maps) # For map data
states_map <- map_data("state")
```



```

# Merge the data sets together
crime_map <- merge(states_map, crimes, by.x = "region", by.y = "state")
# After merging, the order has changed, which would lead to polygons drawn in
# the incorrect order. So, we'll sort the data.
crime_map
#>      region    long  lat group order subregion Murder Assault UrbanPop Rape
#> 1  alabama -87.5 30.4     1     1    <NA>    13.2    236      58 21.2
#> 2  alabama -87.5 30.4     1     2    <NA>    13.2    236      58 21.2
#> 3  alabama -88.0 30.2     1    13    <NA>    13.2    236      58 21.2
#> ...<15,521 more rows>...
#> 15525 wyoming -107.9 41.0    63 15597    <NA>     6.8    161      60 15.6
#> 15526 wyoming -109.1 41.0    63 15598    <NA>     6.8    161      60 15.6
#> 15527 wyoming -109.1 41.0    63 15599    <NA>     6.8    161      60 15.6

library(dplyr) # For arrange() function
# Sort by group, then order
crime_map <- arrange(crime_map, group, order)
crime_map
#>      region    long  lat group order subregion Murder Assault UrbanPop Rape
#> 1  alabama -87.5 30.4     1     1    <NA>    13.2    236      58 21.2
#> 2  alabama -87.5 30.4     1     2    <NA>    13.2    236      58 21.2
#> 3  alabama -87.5 30.4     1     3    <NA>    13.2    236      58 21.2
#> ...<15,521 more rows>...
#> 15525 wyoming -107.9 41.0    63 15597    <NA>     6.8    161      60 15.6
#> 15526 wyoming -109.1 41.0    63 15598    <NA>     6.8    161      60 15.6
#> 15527 wyoming -109.1 41.0    63 15599    <NA>     6.8    161      60 15.6

```

Once the data is in the correct format, it can be plotted (Figure 13-35), mapping one of the columns with data values to fill:

```

ggplot(crime_map, aes(x = long, y = lat, group = group, fill = Assault)) +
  geom_polygon(colour = "black") +
  coord_map("polyconic")

```

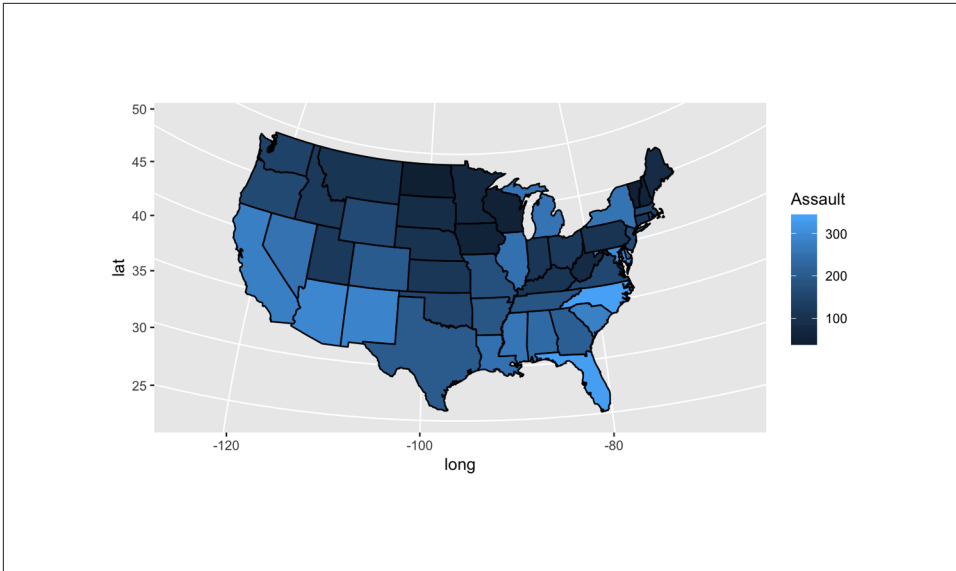


Figure 13-35. A map with a variable mapped to fill

Discussion

The preceding example used the default color scale, which goes from dark to light blue. If you want to show how the values diverge from some middle value, you can use `scale_fill_gradient2()` or `scale_fill_viridis_c()` as shown in [Figure 13-36](#):

```
# Create a base plot
crime_p <- ggplot(crimes, aes(map_id = state, fill = Assault)) +
  geom_map(map = states_map, colour = "black") +
  expand_limits(x = states_map$long, y = states_map$lat) +
  coord_map("polyconic")

crime_p +
  scale_fill_gradient2(low = "#559999", mid = "grey90", high = "#BB650B",
                      midpoint = median(crimes$Assault))

crime_p +
  scale_fill_viridis_c()
```

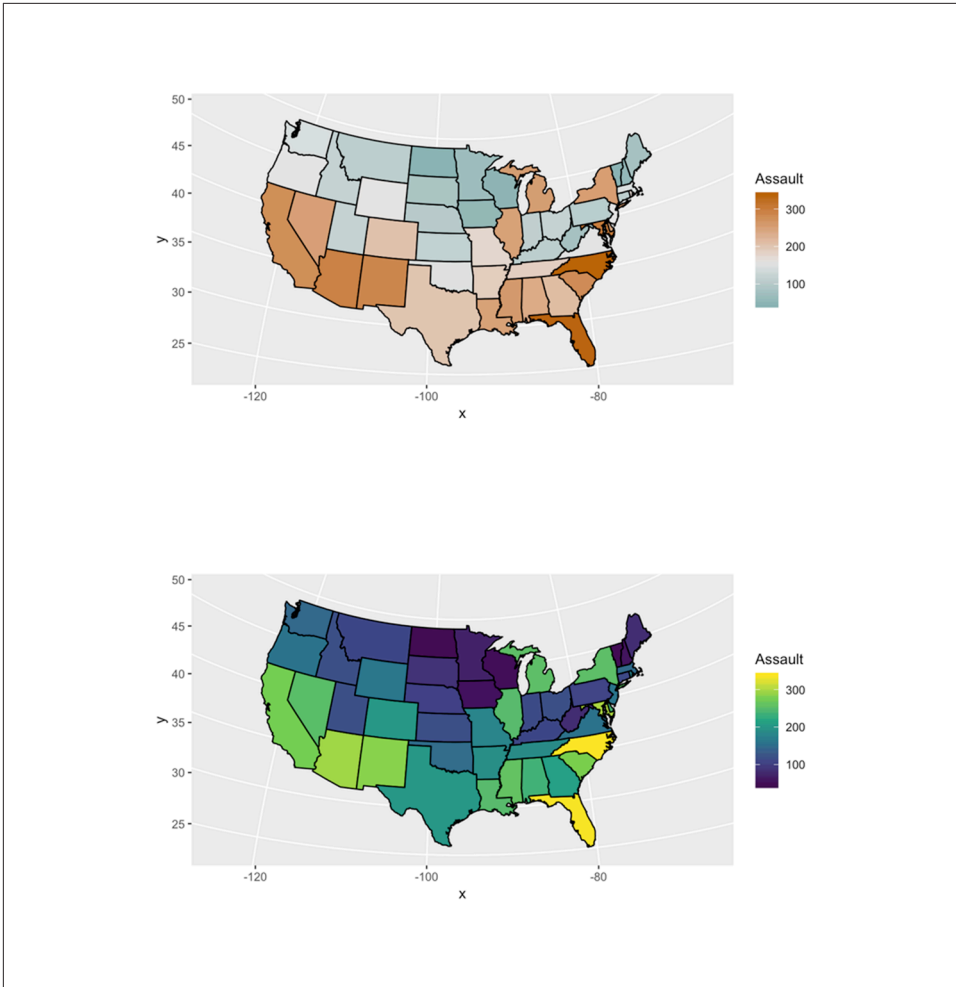


Figure 13-36. With a diverging color scale: `scale_fill_gradient2()` (top); `scale_fill_viridis_c()` (bottom)

The previous example mapped continuous values to fill, but we could just as well use discrete values. It's sometimes easier to interpret the data if the values are discretized. For example, we can categorize the values into quantiles and show those quantiles, as in [Figure 13-37](#):

```
# Find the quantile bounds
qa <- quantile(crimes$Assault, c(0, 0.2, 0.4, 0.6, 0.8, 1.0))
qa
#>    0%    20%    40%    60%    80%   100%
#> 45.0  98.8 135.0 188.8 254.2 337.0
```

```
# Add a column of the quantile category
crimes$Assault_q <- cut(crimes$Assault, qa,
  labels = c("0-20%", "20-40%", "40-60%", "60-80%", "80-100%"),
  include.lowest = TRUE)

crimes
#>      state Murder Assault UrbanPop Rape Assault_q
#> Alabama    alabama    13.2    236     58 21.2    60-80%
#> Alaska      alaska    10.0    263     48 44.5    80-100%
#> Arizona     arizona     8.1    294     80 31.0    80-100%
#> ...<44 more rows>...
#> West Virginia west virginia    5.7     81     39  9.3     0-20%
#> Wisconsin   wisconsin     2.6     53     66 10.8     0-20%
#> Wyoming     wyoming     6.8    161     60 15.6    40-60%
# Generate a discrete color palette with 5 values
pal <- colorRampPalette(c("#559999", "grey80", "#BB650B"))(5)
pal
#> [1] "#559999" "#90B2B2" "#CCCCC" "#C3986B" "#BB650B"

ggplot(crimes, aes(map_id = state, fill = Assault_q)) +
  geom_map(map = states_map, colour = "black") +
  scale_fill_manual(values = pal) +
  expand_limits(x = states_map$long, y = states_map$lat) +
  coord_map("polyconic") +
  labs(fill = "Assault Rate\nPercentile")
```

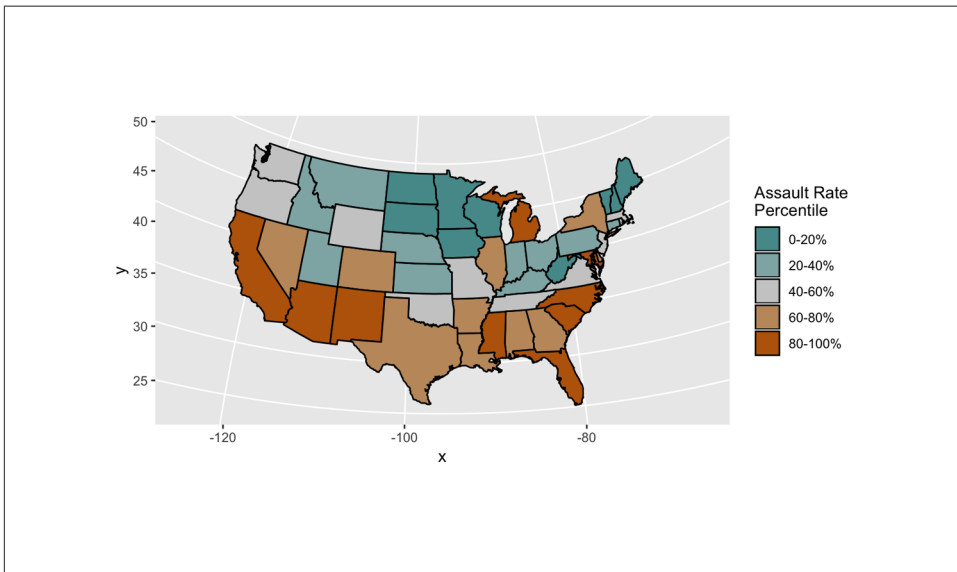


Figure 13-37. Choropleth map with discretized data

Another way to make a choropleth, but without needing to merge the map data with the value data, is to use `geom_map()`. As of this writing, this will render maps faster than the method just described.

For this method, the map data frame must have columns named `lat`, `long`, and `region`. In the value data frame, there must be a column that is matched to the `region` column in the map data frame, and this column is specified by mapping it to the `map_id` aesthetic. For example, this code will have the same output as the first example (Figure 13-35):

```
# The 'state' column in the crimes data is to be matched to the 'region' column
# in the states_map data
ggplot(crimes, aes(map_id = state, fill = Assault)) +
  geom_map(map = states_map) +
  expand_limits(x = states_map$long, y = states_map$lat) +
  coord_map("polyconic")
```

Notice that we also needed to use `expand_limits()`. This is because unlike most geoms, `geom_map()` doesn't automatically set the x and y limits; the use of `expand_limits()` makes it include those x and y values. (Another way to accomplish the same result is to use `ylim()` and `xlim()`.)

See Also

For an example of data overlaid on a map, see [Recipe 13.12](#).

For more on using continuous colors, see [Recipe 12.6](#).

13.19 Making a Map with a Clean Background

Problem

You want to remove background elements from a map.

Solution

Use `theme_void()` (Figure 13-38). In this example, we'll add it to one of the choropleths we created in [Recipe 13.18](#):

```
ggplot(crimes, aes(map_id = state, fill = Assault_q)) +
  geom_map(map = states_map, colour = "black") +
  scale_fill_manual(values = pal) +
  expand_limits(x = states_map$long, y = states_map$lat) +
  coord_map("polyconic") +
  labs(fill = "Assault Rate\nPercentile") +
  theme_void()
```

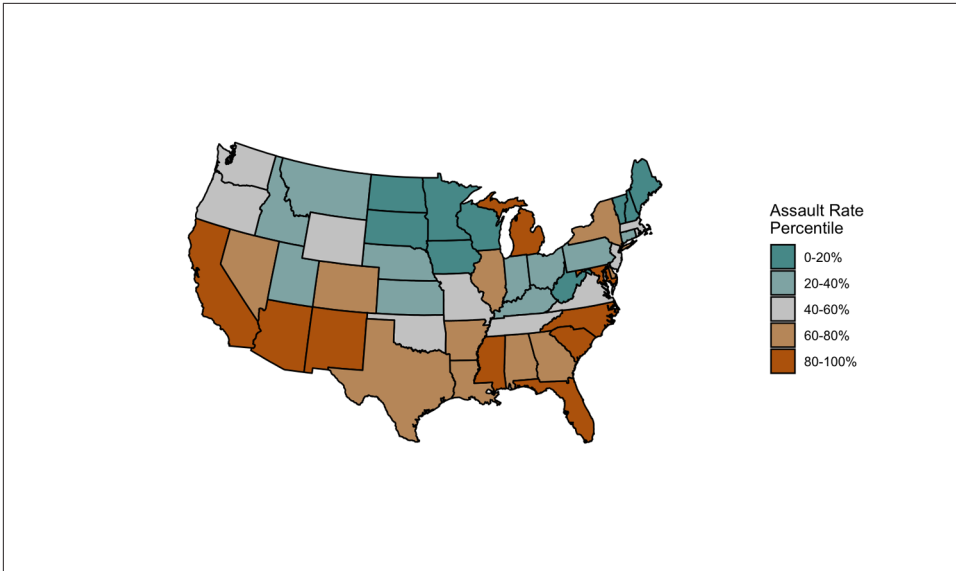


Figure 13-38. A map with a clean background

Discussion

In some maps, it's important to include contextual information such as the latitude and longitude. In others, this information is unimportant and distracts from the information that's being conveyed. In [Figure 13-38](#), it's unlikely that viewers will care about the latitude and longitude of the states. They can probably identify the states by shape and relative position, and even if they can't, having the latitude and longitude isn't really helpful.

13.20 Creating a Map from a Shapefile

Problem

You want to create a geographical map from an Esri shapefile.

Solution

Load the shapefile using `st_read()` from the `sf` package, then plot it with `geom_sf()` ([Figure 13-39](#)):

```
library(sf)

# Load the shapefile
taiwan_shp <- st_read("fig/TWN_adm/TWN_adm2.shp")
#> Reading layer `TWN_adm2' from data source `fig/TWN_adm/TWN_adm2.shp' using
```

```
#> driver `ESRI Shapefile'
#> Simple feature collection with 22 features and 11 fields
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: 116.71 ymin: 20.6975 xmax: 122.1085 ymax: 25.63431
#> epsg (SRID):    4326
#> proj4string:     +proj=longlat +datum=WGS84 +no_defs

ggplot(taiwan_shp) +
  geom_sf()
```

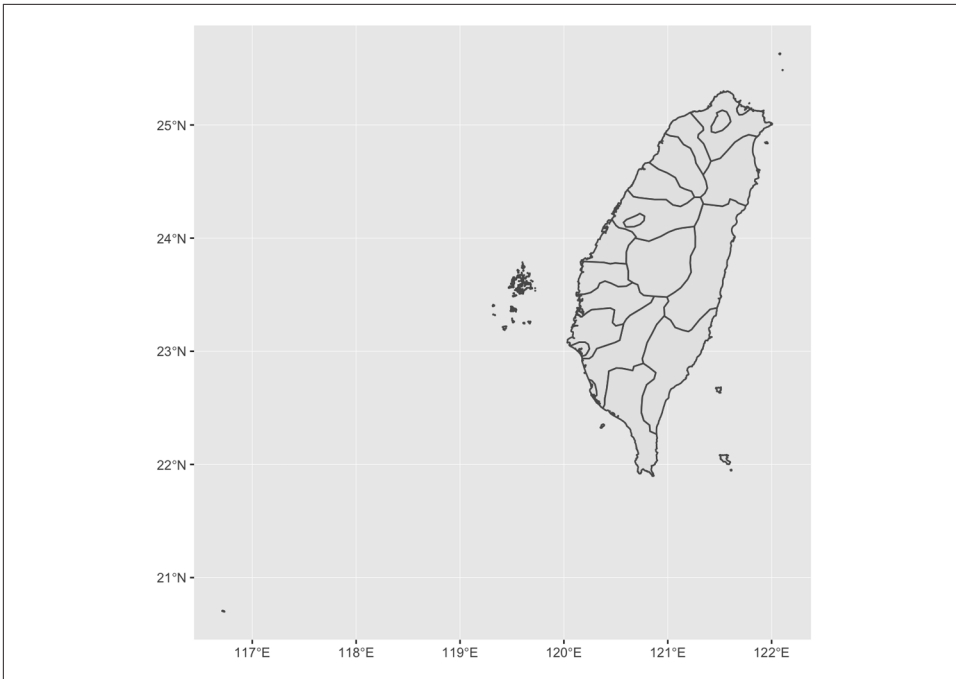


Figure 13-39. A map created from a shapefile

Discussion

Esri shapefiles are a common format for map data. The `st_read()` function reads a shapefile and returns an `sf` object, which will also have other useful columns. Here's a look at the contents of the object:

```
taiwan_shp
#> Simple feature collection with 22 features and 11 fields
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: 116.71 ymin: 20.6975 xmax: 122.1085 ymax: 25.63431
#> epsg (SRID):    4326
#> proj4string:     +proj=longlat +datum=WGS84 +no_defs
#> First 6 features:
```

```

#>   ID_0 ISO NAME_0 ID_1      NAME_1 ID_2      NAME_2 NL_NAME_2
#> 1  223 TWN Taiwan   1   Kaohsiung  1 Kaohsiung City   <NA>
#> 2  223 TWN Taiwan   2 Pratas Islands  2      <NA>      <NA>
#> 3  223 TWN Taiwan   3   Taipei  3   Taipei City   <NA>
#> 4  223 TWN Taiwan   4   Taiwan  4   Changhwa   <NA>
#> 5  223 TWN Taiwan   4   Taiwan  5   Chiayi   <NA>
#> 6  223 TWN Taiwan   4   Taiwan  6   Hsinchu   <NA>
#>      VARNAME_2      TYPE_2      ENGTYPE_2
#> 1   Gaoxiong Shi   Chuan-shih Special Municipality
#> 2      <NA>      <NA>      <NA>
#> 3   Taipei Shi   Chuan-shih Special Municipality
#> 4 Zhonghua/Changhua District/Hsien      County
#> 5   Jiayi/Chiai District/Hsien      County
#> 6   Xinzhu District/Hsien      County
#>      geometry
#> 1 MULTIPOLYGON (((120.239 22....
#> 2 MULTIPOLYGON (((116.7172 20...
#> 3 MULTIPOLYGON (((121.525 25....
#> 4 MULTIPOLYGON (((120.4176 24...
#> 5 MULTIPOLYGON (((120.1526 23...
#> 6 MULTIPOLYGON (((120.9146 24...

```

The `sf` object is a special kind of data frame, with 22 rows and 12 columns. Each row corresponds to one feature, and each column has some data about each feature. One of the columns is the geometry for each feature. This is a list-column—a special type of column where each of the 22 elements contains one or more matrices of numbers representing the shape of the feature.

Columns in the data can be mapped to aesthetics like `fill`. For example, we can map the `ENGTYPE_2` column to `fill`, as shown in [Figure 13-40](#):

```

# Remove rows for which ENGTYPE_2 is NA; otherwise NA will show in the legend
taiwan_shp_mod <- taiwan_shp
taiwan_shp_mod <- taiwan_shp[!is.na(taiwan_shp$ENGTYPE_2), ]

ggplot(taiwan_shp_mod) +
  geom_sf(aes(fill = ENGTYPE_2))

```

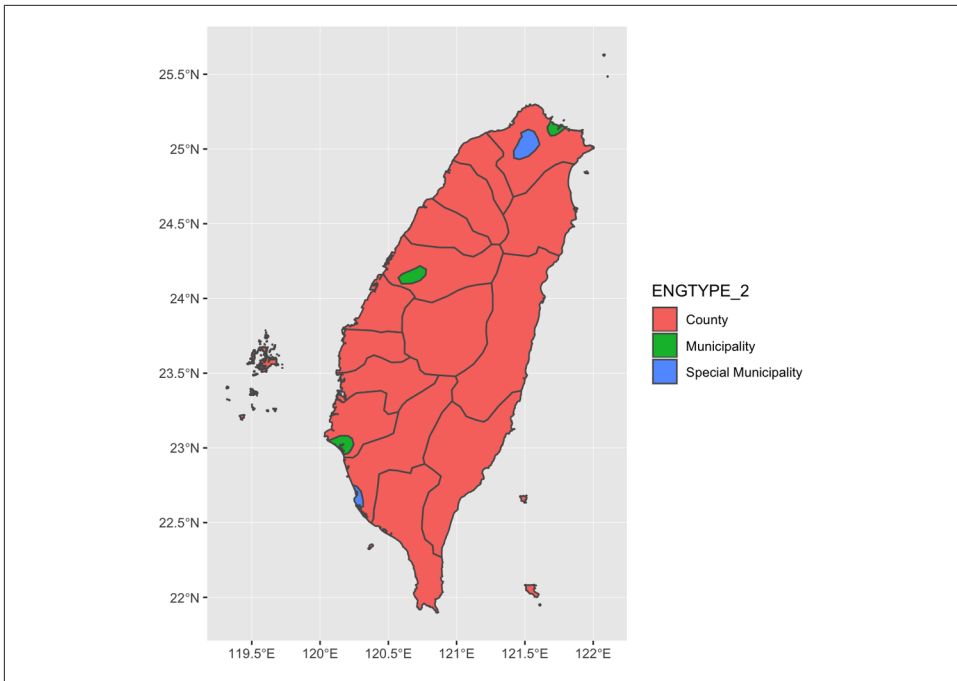



Figure 13-40. With a column mapped to fill

See Also

The shapefile used in this example is not included in the gcookbook package. It and many other shapefiles are available for download at <http://www.gadm.org>.

Output for Presentation

Broadly speaking, visualizations of data serve two purposes: discovery and communication. In the discovery phase, you'll create exploratory graphics, and when you do this, it's important to be able try out different things quickly. In the communication phase, you'll present your graphics to others. When you do that, you'll need to tweak the appearance of the graphics (which I've written about in previous chapters), and you'll usually need to put them somewhere other than on your computer screen. This chapter is about that last part: *saving* your graphics so that they can be presented in documents.

14.1 Outputting to PDF Vector Files

Problem

You want to create a PDF of your plot.

Solution

There are two ways to output to PDF files. One method is to open the PDF graphics device with `pdf()`, make the plots, then close the device with `dev.off()`. This method works for most graphics in R, including base graphics and grid-based graphics like those created by `ggplot2` and `lattice`:

```
# width and height are in inches
pdf("myplot.pdf", width = 4, height = 4)

# Make plots
plot(mtcars$wt, mtcars$mpg)
print(ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point())

dev.off()
```

If you make more than one plot, each one will go on a separate page in the PDF output. Notice that we called `print()` on the `ggplot` object to make sure that it will generate graphical output even when this code is in a script.

The width and height are in inches, so to specify the dimensions in centimeters, you must do the conversion manually:

```
# 8x8 cm
pdf("myplot.pdf", width = 8/2.54, height = 8/2.54)
```

If you are creating plots from a script and it throws an error while creating one, R might not reach the call to `dev.off()`, and could be left in a state where the PDF device is still open. When this happens, the PDF file won't open properly until you manually call `dev.off()`.

If you are creating a plot with `ggplot2`, using `ggsave()` can be a little simpler. You can store the `ggplot` object in a variable, and then call `ggsave()` on it:

```
plot1 <- ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point()

# Default is inches, but you can specify unit
ggsave("myplot.pdf", plot1, width = 8, height = 8, units = "cm")
```

Another way of using it is to skip the variable, and just call `ggsave()` after calling `ggplot()`. It will save the last `ggplot` object:

```
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point()

ggsave("myplot.pdf", width = 8, height = 8, units = "cm")
```

With `ggsave()`, you don't need to `print()` the `ggplot` object, and if there is an error while creating or saving the plot, there's no need to manually close the graphic device. `ggsave()` can't be used to make multipage plots, though.

Discussion

PDF files are usually the best option when your goal is to output to printed documents. They work easily with LaTeX and can be used in presentations with Apple's Keynote, but Microsoft programs may have trouble importing them. (See [Recipe 14.3](#) for details on creating vector images that can be imported into Microsoft programs.)

PDF files are also generally smaller than bitmap files such as portable network graphics (PNG) files, because they contain a set of drawing instructions, such as "Draw a line from here to there," instead of information about the color of each pixel. However, there are cases where bitmap files are smaller. For example, if you have a scatter plot that is heavily overplotted, a PDF file can end up much larger than a PNG—even though most of the points are obscured, the PDF file will still contain instructions for

drawing each and every point, whereas a bitmap file will not contain the redundant information. See [Recipe 5.5](#) for an example.

See Also

If you want to manually edit the PDF or SVG file, see [Recipe 14.4](#).

14.2 Outputting to SVG Vector Files

Problem

You want to create a scalable vector graphics (SVG) image of your plot.

Solution

Use `svglite()` from the `svglite` package:

```
library(svglite)
svglite("myplot.svg", width = 4, height = 4)
plot(...)
dev.off()

# With ggsave()
ggsave("myplot.svg", width = 8, height = 8, units = "cm")
```

Discussion

Although R has a built-in `svg()` function that can generate SVG output, the `svglite` package provides more standards-compliant output.

When it comes to importing images, some programs may handle SVG files better than PDFs, and vice versa. For example, web browsers tend to have better SVG support, while document-creation programs like LaTeX tend to have better PDF support.

14.3 Outputting to WMF Vector Files

Problem

You want to create a Windows metafile (WMF) image of your plot.

Solution

WMF files can be created and used in much the same way as PDF files—but they can only be created on Windows:

```
win.metafile("myplot.wmf", width = 4, height = 4)
plot(...)
dev.off()

# With ggsave()
ggsave("myplot.wmf", width = 8, height = 8, units = "cm")
```

Discussion

Windows programs such as Microsoft Word and PowerPoint have poor support for importing PDF files, but they natively support WMF. One drawback is that WMF files do not support transparency (alpha).

14.4 Editing a Vector Output File

Problem

You want to open a vector output file for final editing.

Solution

Sometimes you need to make final tweaks to the appearance of a graph for presentation. You can open PDF and SVG files with the excellent free program Inkscape, or with the commercial program Adobe Illustrator.

Discussion

Font support can be a problem when you open a PDF file with Inkscape. Normally, point objects drawn with the PDF device will be written as symbols from the Zapf Dingbats font. This can be problematic if you want to open the file in an editor like Illustrator or Inkscape; for example, points may appear as the letter *q*, as in [Figure 14-1](#), because that is the corresponding letter for a solid bullet in Zapf Dingbats.

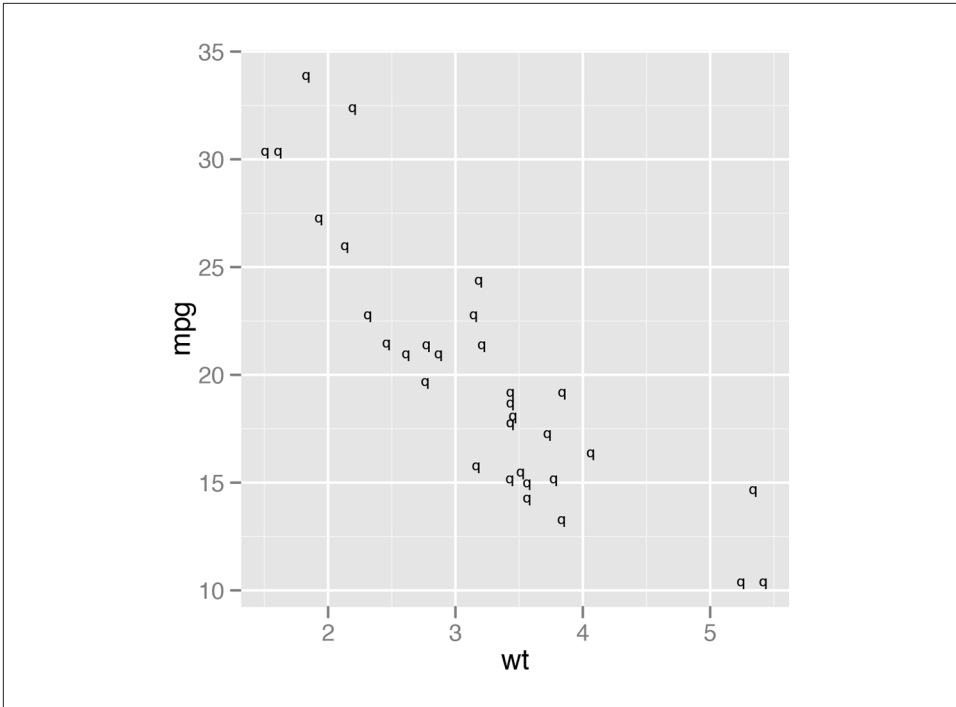


Figure 14-1. Bad conversion of point symbols after opening in Inkscape—also notice that the spacing of the fonts is slightly off

To avoid this problem, set `useDingbats = FALSE`. This will make the circles be drawn as circles instead of as font characters:

```
pdf("myplot.pdf", width = 4, height = 4, useDingbats = FALSE)

# or
ggsave("myplot.pdf", width = 4, height = 4, useDingbats = FALSE)
```



Inkscape might have some issues with fonts as well. You may have noticed that the fonts in [Figure 14-1](#) don't look quite right. This is because Inkscape (version 0.48) couldn't find Helvetica, and substituted the font Bitstream Vera Sans instead. A workaround is to copy the Helvetica font file to your personal font library. For example, on macOS, run `cp System/Library/Fonts/Helvetica.dfont ~/Library/Fonts/` from a Terminal window to do this, then, when it says there is a font conflict, click "Ignore Conflict." After this, Inkscape should properly display the Helvetica font.

14.5 Outputting to Bitmap (PNG/TIFF) Files

Problem

You want to create a bitmap of your plot, writing to a PNG file.

Solution

There are two ways to output to PNG bitmap files. One method is to open the PNG graphics device with `png()`, make the plots, then close the device with `dev.off()`. This method works for most graphics in R, including base graphics and grid-based graphics like those created by `ggplot2` and `lattice`:

```
# width and height are in pixels
png("myplot.png", width = 400, height = 400)

# Make plot
plot(mtcars$wt, mtcars$mpg)

dev.off()
```

For outputting multiple plots, put `%d` in the filename. This will be replaced with 1, 2, 3, and so on, for each subsequent plot:

```
# width and height are in pixels
png("myplot-%d.png", width = 400, height = 400)

plot(mtcars$wt, mtcars$mpg)
print(ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point())

dev.off()
```

Notice that we called `print()` on the `ggplot` object to make sure that it will be output even when this code is in a script.

The width and height are in pixels, and the default is to output at 72 pixels per inch (ppi). This resolution is suitable for displaying on a screen, but will look pixelated and jagged in print.

For high-quality print output, use at least 300 ppi. [Figure 14-2](#) shows portions of the same plot at different resolutions. In this example, we'll use 300 ppi and create a 4×4-inch PNG file:

```
ppi <- 300
# Calculate the height and width (in pixels) for a 4x4-inch image at 300 ppi
png("myplot.png", width = 4*ppi, height = 4*ppi, res = ppi)
plot(mtcars$wt, mtcars$mpg)
dev.off()
```

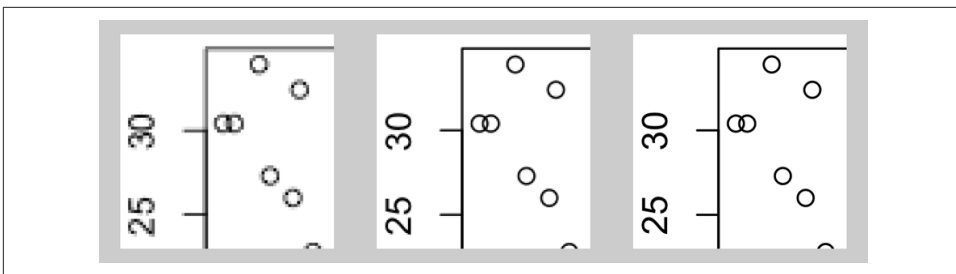



Figure 14-2. From left to right: PNG output at 72, 150, and 300 ppi (actual size)

If you are creating plots from a script and it throws an error while creating one, R might not reach the call to `dev.off()`, and could be left in a state where the PNG device is still open. When this happens, the PNG file won't open properly in a viewing program until you manually call `dev.off()`.

If you are creating a plot with `ggplot`, using `ggsave()` can be a little simpler. It simply saves the last plot created with `ggplot()`. You specify the width and height in inches, not pixels, and tell it how many pixels per inch to use:

```
ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point()

# Default dimensions are in inches, but you can specify the unit
ggsave("myplot.png", width = 8, height = 8, unit = "cm", dpi = 300)
```

With `ggsave()`, you don't need to print the `ggplot` object, and if there is an error while creating or saving the plot there's no need to manually close the graphics device.



Although the argument name is `dpi`, it really controls the *pixels* per inch (ppi), not the *dots* per inch. When a grey pixel is rendered in print, it is output with many smaller dots of black ink—and so print output has more dots per inch than pixels per inch.

Discussion

R supports other bitmap formats, like BMP, TIFF, and JPEG, but there's really not much reason to use them instead of PNG.

The exact appearance of the resulting bitmaps varies from platform to platform. Unlike R's PDF output device, which renders consistently across platforms, the bitmap output devices may render the same plot differently on Windows, Linux, and macOS. There can even be variation within each of these operating systems.

Different platforms will render fonts differently, some platforms will antialias (smooth) lines while others will not, and some platforms support alpha (transpar-

ency) while others do not. If your platform lacks support for features like antialiasing and alpha, you can use `CairoPNG()`, from the Cairo package:

```
install.packages("Cairo") # One-time installation
CairoPNG("myplot.png")
plot(...)
dev.off()
```

While `CairoPNG()` does not guarantee identical rendering across platforms (fonts may not be exactly the same), it does support features like antialiasing and alpha.

Changing the resolution affects the size (in pixels) of graphical objects like text, lines, and points. For example, a 6×6-inch image at 75 ppi has the same pixel dimensions as a 3×3-inch image at 150 ppi, but the appearance will be different, as shown in [Figure 14-3](#). Both of these images are 450×450 pixels. When displayed on a computer screen, they may display at approximately the same size, as they do here.

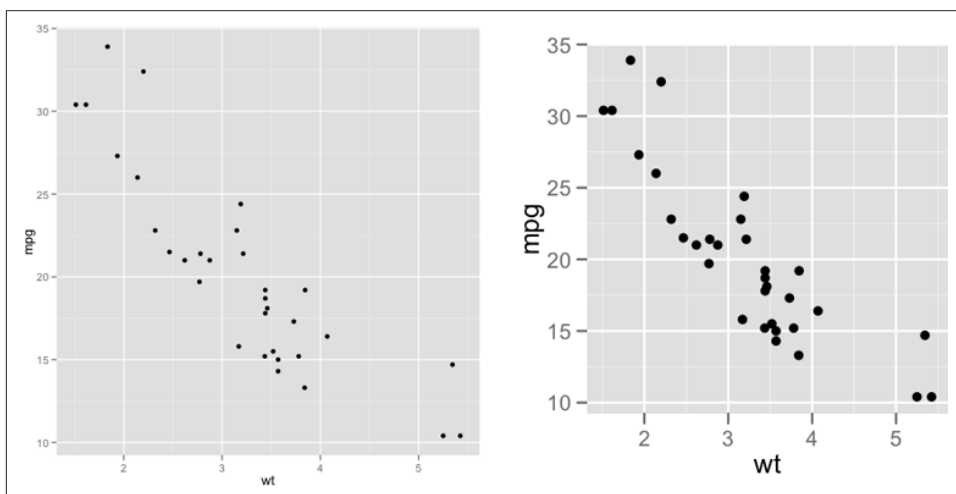


Figure 14-3. 6×6-inch image at 75 ppi (left); 3×3-inch image at 150 ppi (right)

14.6 Using Fonts in PDF Files

Problem

You want to use fonts other than the basic ones provided by R in a PDF file.

Solution

The `extrafont` package can be used to create PDF files with different fonts.

There are a number of steps involved, beginning with some one-time setup. Download and install [Ghostscript](#), then run the following in R:

```
install.packages("extrafont")
library(extrafont)

# Find and save information about fonts installed on your system
font_import()

# List the fonts
fonts()
```

After the one-time setup is done, there are tasks you need to do in each R session:

```
library(extrafont)
# Register the fonts with R
loadfonts()

# On Windows, you may need to tell it where Ghostscript is installed
# (adjust the path to match your installation of Ghostscript)
Sys.setenv(R_GSCMD = "C:/Program Files/gs/gs9.05/bin/gswin32c.exe")
```

Finally, you can create a PDF file and embed fonts into it, as in [Figure 14-4](#):

```
library(ggplot2)

ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point() +
  ggtitle("Title text goes here") +
  theme(text = element_text(size = 16, family = "Impact"))

ggsave("myplot.pdf", width = 4, height = 4)

embed_fonts("myplot.pdf")
```

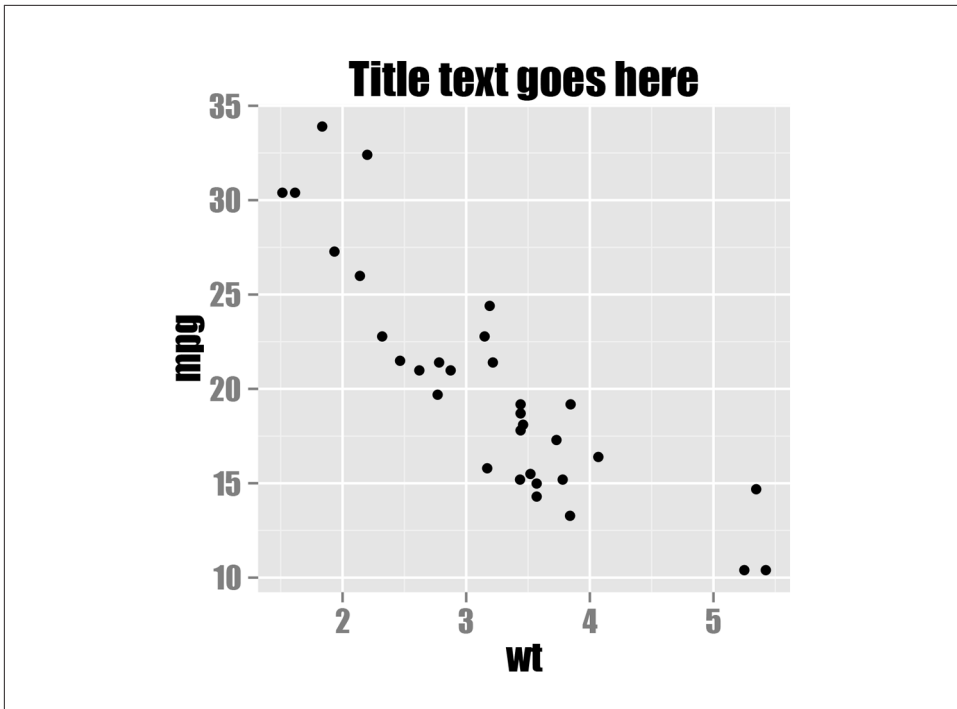


Figure 14-4. PDF output with embedded font *Impact*

Discussion

Fonts can be difficult to work with in R. Some output devices, such as the on-screen quartz device on macOS, can display any font installed on the computer. Other output devices, such as the default PNG device on Windows, aren't able to display system fonts.

On top of this, PDF files have their own quirks when it comes to fonts. The PDF specification has 14 “core” fonts. These are fonts that every PDF renderer has, and they include standards such as Times, Helvetica, and Courier. If you create a PDF with these fonts, any PDF renderer should display it properly.

If you want to use a font that is *not* one of these core fonts, though, there's no guarantee that the PDF renderer on a given device will have that font, so you can't be sure that the font will display properly on another computer or printer. To solve this problem, noncore fonts can be *embedded* into the PDF; in other words, the PDF file can itself contain a copy of the font you want to use.

If you are putting multiple PDF figures in a PDF document, you may want to embed the fonts in the finished document instead of in each figure. This will make the final

document smaller, since it will only have the font embedded once, instead of once for each figure.

Embedding fonts with R can be a tricky process, but the `extrafont` package handles many of the ugly details for you.



As of this writing, `extrafont` will only import TrueType (`.tff`) fonts, but it may support other common formats, such as OpenType (`.otf`), in the future.

See Also

`Showtext` is another package for using different fonts in R graphics. It supports TrueType, OpenType, and PostScript Type 1 fonts, and makes it easy to download web fonts. However, it currently does not work correctly with the RStudio viewer pane. See <https://cran.r-project.org/web/packages/showtext/vignettes/introduction.html>.

For more on controlling text appearance, see [Recipe 9.2](#).

14.7 Using Fonts in Windows Bitmap or Screen Output

Problem

You are using Windows and want to use fonts other than the basic ones provided by R for bitmap or screen output.

Solution

The `extrafont` package can be used to create bitmap or screen output. The procedure is similar to using `extrafont` with PDF files ([Recipe 14.6](#)). The one-time setup is almost the same, except that Ghostscript is not required:

```
install.packages("extrafont")
library(extrafont)

# Find and save information about fonts installed on your system
font_import()

# List the fonts
fonts()
```

After the one-time setup is done, there are tasks you need to do in each R session:

```
library(extrafont)
```

```
# Register the fonts for Windows
```

```
loadfonts("win")
```

Finally, you can create each output file or display graphs on screen, as in [Figure 14-5](#):

```
library(ggplot2)
```

```
ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point() +
```

```
  ggtitle("Title text goes here") +
```

```
  theme(text = element_text(size = 16, family = "Georgia", face = "italic"))
```

```
ggsave("myplot.png", width = 4, height = 4, dpi = 300)
```

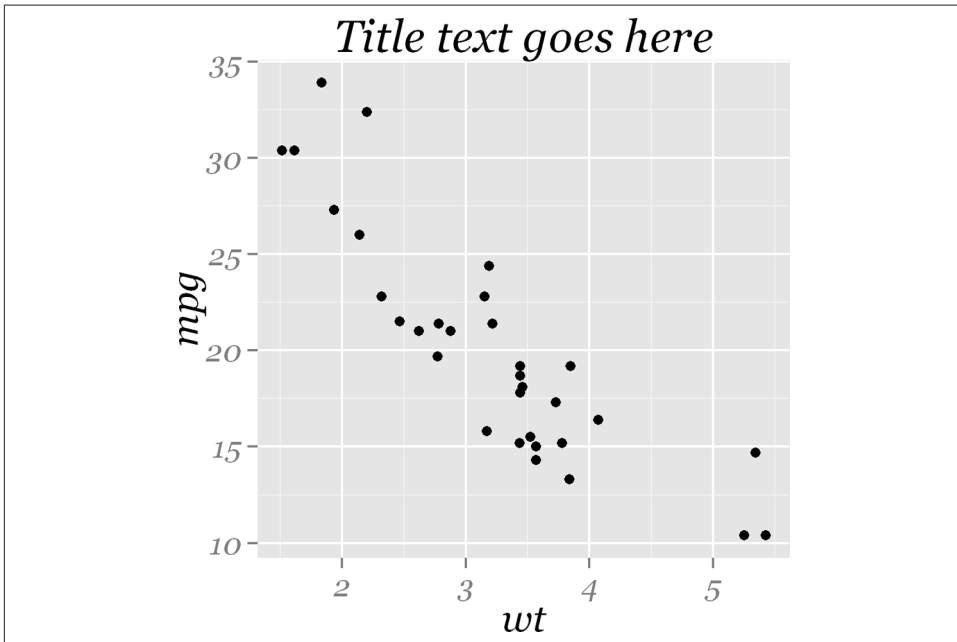


Figure 14-5. PNG output with font Georgia Italic

Discussion

Fonts are handled in a completely different way for bitmaps than they are for PDF files.

On Windows, for bitmap output it is necessary to register each font manually with R (extrafont makes this much easier). On macOS and Linux, the fonts should already be available for bitmap output; it isn't necessary to register them manually.

14.8 Combining Several Plots into the Same Graphic

Problem

You want to combine several plots into one graphic output.

Solution

The patchwork package can be used to combine several ggplot2 plots into one graphic.

As of this writing, the patchwork package is not available on CRAN, so you need to install it using the devtools package. (The latest installation instructions can be found on the project's [GitHub page](#).)

```
# install.packages("devtools")
# devtools::install_github("thomasp85/patchwork")
```

Once you have installed patchwork, you can start stitching plots together (Figure 14-6).

```
library(patchwork)

plot1 <- ggplot(PlantGrowth, aes(x = weight)) +
  geom_histogram(bins = 12)

plot2 <- ggplot(PlantGrowth, aes(x = group, y = weight, group = group)) +
  geom_boxplot()

plot1 + plot2
```

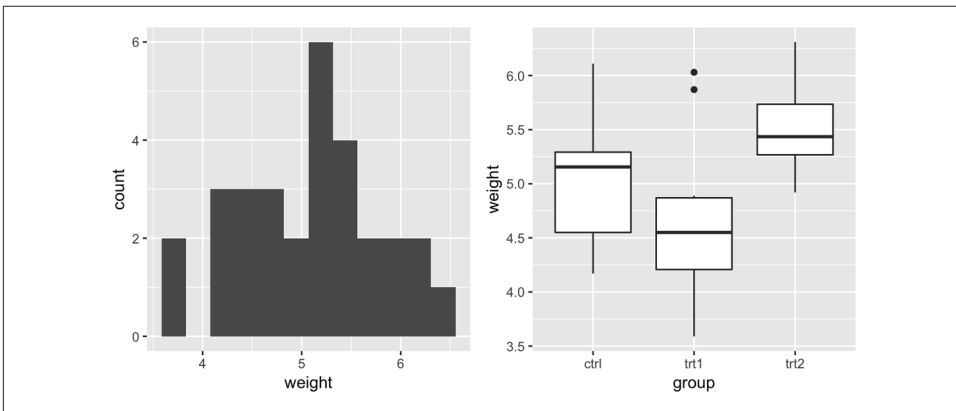


Figure 14-6. Combining two plots together using patchwork()

Discussion

Patchwork also allows you to determine how you want to lay out the ggplots in relation to each other, by adding a `plot_layout()` call. You can use this call to determine the number of columns you want the plots to be arranged in (Figure 14-7), and the size of the plots (Figure 14-8):

```
plot3 <- ggplot(PlantGrowth, aes(x = weight, fill = group)) +  
  geom_density(alpha = 0.25)  
  
plot1 + plot2 + plot3 +  
  plot_layout(ncol = 2)
```

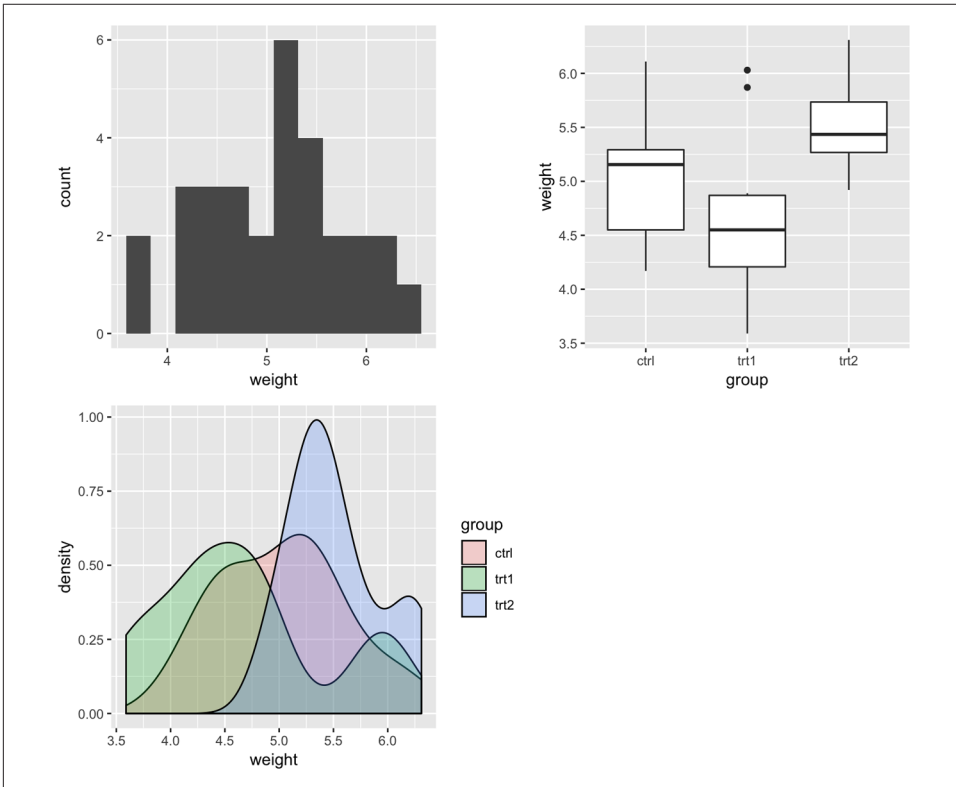


Figure 14-7. Using `plot_layout()` to specify that the plots should be arranged in two columns

```
plot1 + plot2 +  
  plot_layout(ncol = 1, heights = c(1, 4))
```

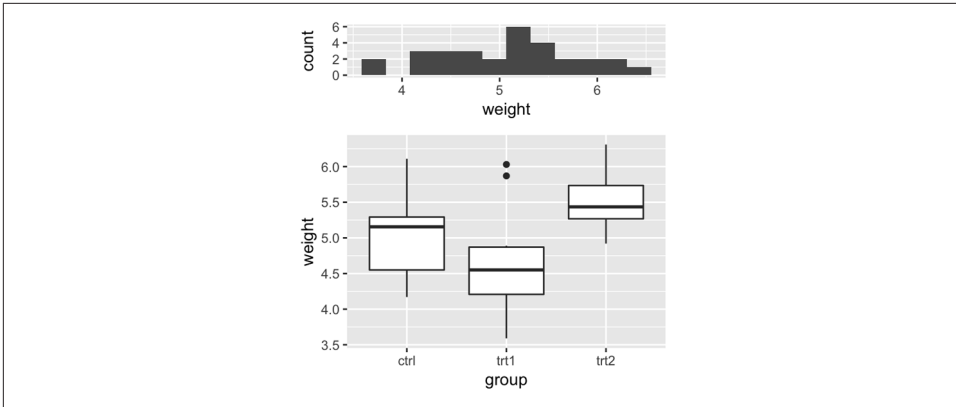



Figure 14-8. Using `plot_layout()` to specify the heights of each plot

In the future, `patchwork` may be available on CRAN, in which case it can be installed using the usual `install.packages()`.

See Also

For more on additional `patchwork` features, see <https://github.com/thomasp85/patchwork>.

Getting Your Data into Shape

When it comes to making data graphics, half the battle occurs before you call any plotting commands. Before you pass your data to the plotting functions, it must first be read in and given the correct structure. The data sets provided with R are ready to use, but when dealing with real-world data, this usually isn't the case: you'll have to clean up and restructure the data before you can visualize it.

The recipes in this chapter will often use packages from the *tidyverse*. For a little background about the tidyverse, see the introduction section of [Chapter 1](#). I will also show how to do many of the same tasks using base R, because in some situations it is important to minimize the number of packages you use, and because it is useful to be able to understand code written for base R.



The `%>%` symbol, also known as the pipe operator, is used extensively in this chapter. If you are not familiar with it, see [Recipe 1.7](#).

Most of the tidyverse functions used in this chapter are from the `dplyr` package, and in this chapter, I'll assume that `dplyr` is already loaded. You can load it with either `library(tidyverse)` as shown above, or, if you want to keep things more streamlined, you can load `dplyr` directly:

```
library(dplyr)
```

Data sets in R are most often stored in data frames. They're typically used as two-dimensional data structures, with each row representing one case and each column representing one variable. Data frames are essentially lists of vectors and factors, all of the same length, where each vector or factor represents one column.

Here's the heightweight data set:

```
library(gcookbook) # Load gcookbook for the heightweight data set
heightweight
#>      sex ageYear ageMonth heightIn weightLb
#> 1    f   11.92     143     56.3     85.0
#> 2    f   12.92     155     62.3    105.0
#> ...<232 more rows>...
#> 236  m   13.92     167     62.0    107.5
#> 237  m   12.58     151     59.3     87.0
```

It consists of five columns, with each row representing one case: a set of information about a single person. We can get a clearer idea of how it's structured by using the `str()` function:

```
str(heightweight)
#> 'data.frame':    236 obs. of  5 variables:
#> $ sex      : Factor w/ 2 levels "f","m": 1 1 1 1 1 1 1 1 1 1 ...
#> $ ageYear  : num  11.9 12.9 12.8 13.4 15.9 ...
#> $ ageMonth: int   143 155 153 161 191 171 185 142 160 140 ...
#> $ heightIn: num   56.3 62.3 63.3 59 62.5 62.5 59 56.5 62 53.8 ...
#> $ weightLb: num   85 105 108 92 112 ...
```

The first column, `sex`, is a factor with two levels, "f" and "m", and the other four columns are vectors of numbers (one of them, `ageMonth`, is specifically a vector of integers, but for the purposes here, it behaves the same as any other numeric vector).

Factors and character vectors behave similarly in `ggplot`—the main difference is that with character vectors, items will be displayed in lexicographical order, but with factors, items will be displayed in the same order as the factor levels, which you can control.

15.1 Creating a Data Frame

Problem

You want to create a data frame from vectors.

Solution

You can put vectors together in a data frame with `data.frame()`:

```
# Two starting vectors
g <- c("A", "B", "C")
x <- 1:3
dat <- data.frame(g, x)
dat
#>      g x
#> 1 A 1
```

```
#> 2 B 2  
#> 3 C 3
```

Discussion

A data frame is essentially a list of vectors and factors. Each vector or factor can be thought of as a column in the data frame.

If your vectors are in a list, you can convert the list to a data frame with the `as.data.frame()` function:

```
lst <- list(group = g, value = x)    # A list of vectors  
  
dat <- as.data.frame(lst)
```

The tidyverse way of creating a data frame is to use `data_frame()` or `as_data_frame()` (note the underscores instead of periods). This returns a special kind of data frame—a *tibble*—which behaves like a regular data frame in most contexts, but prints out more nicely and is specifically designed to play well with the tidyverse functions:

```
data_frame(g, x)  
#> # A tibble: 3 x 2  
#>   g         x  
#>   <chr> <int>  
#> 1 A         1  
#> 2 B         2  
#> 3 C         3  
  
# Convert the list of vectors to a tibble  
as_data_frame(lst)
```

A regular data frame can be converted to a tibble using `as_tibble()`:

```
as_tibble(dat)  
#> # A tibble: 3 x 2  
#>   group value  
#>   <fct> <int>  
#> 1 A         1  
#> 2 B         2  
#> 3 C         3
```

15.2 Getting Information About a Data Structure

Problem

You want to find out information about an object or data structure.

Solution

Use the `str()` function:

```
str(ToothGrowth)
#> 'data.frame': 60 obs. of 3 variables:
#> $ len : num 4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
#> $ supp: Factor w/ 2 levels "OJ", "VC": 2 2 2 2 2 2 2 2 2 ...
#> $ dose: num 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

This tells us that `ToothGrowth` is a data frame with three columns, `len`, `supp`, and `dose`. `len` and `dose` contain numeric values, while `supp` is a factor with two levels.

Another useful function is the `summary()` function:

```
summary(ToothGrowth)
#>      len      supp      dose
#> Min.   : 4.20   OJ:30   Min.   :0.500
#> 1st Qu.:13.07   VC:30   1st Qu.:0.500
#> Median :19.25           Median :1.000
#> Mean   :18.81           Mean   :1.167
#> 3rd Qu.:25.27           3rd Qu.:2.000
#> Max.   :33.90           Max.   :2.000
```

Instead of showing you the first few values of each column as `str()` does, `summary()` provides basic descriptive statistics (the minimum, maximum, median, mean, and first and third quartile values) for numeric variables, and tells you the number of values corresponding to each character value or factor level if it is a character or factor variable.

Discussion

The `str()` function is very useful for finding out more about data structures. One common source of problems is a data frame where one of the columns is a character vector instead of a factor, or vice versa. This can cause puzzling issues with analyses or graphs.

When you print out a data frame the normal way, by just typing the name at the prompt and pressing Enter, factor and character columns appear exactly the same. The difference will be revealed only when you run `str()` on the data frame, or print out the column by itself:

```
tg <- ToothGrowth
tg$supp <- as.character(tg$supp)
str(tg)
#> 'data.frame': 60 obs. of 3 variables:
#> $ len : num 4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
#> $ supp: chr "VC" "VC" "VC" "VC" ...
#> $ dose: num 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

[illegible]

15.3 Adding a Column to a Data Frame

Problem

You want to add a column to a data frame.

Solution

Use `mutate()` from `dplyr` to add a new column and assign values to it. This returns a new data frame, which you'll typically want to save over the original.

If you assign a single value to the new column, the entire column will be filled with that value. This adds a column named `newcol`, filled with `NA`:

```
library(dplyr)

ToothGrowth %>%
  mutate(newcol = NA)

#>      len supp dose newcol
#> 1    4.2   VC   0.5     NA
#> 2   11.5   VC   0.5     NA
#> ...<56 more rows>...
#> 59 29.4   OJ   2.0     NA
#> 60 23.0   OJ   2.0     NA
```

You can also assign a vector to the new column:

```
# Since ToothGrowth has 60 rows, we must create a new vector that has 60 rows
vec <- rep(c(1, 2), 30)

ToothGrowth %>%
  mutate(newcol = vec)
```

Note that the vector being added to the data frame must either have one element, or the same number of elements as the data frame has rows. In the preceding example

we created a new vector that had 60 rows by repeating the values `c(1, 2)` thirty times.

Discussion

Each column of a data frame is a vector. R handles columns in data frames slightly differently from standalone vectors because all the columns in a data frame must have the same length.

To add a column using base R, you can simply assign values into the new column like so:

```
# Make a copy of ToothGrowth for this example
ToothGrowth2 <- ToothGrowth

# Assign NA's for the whole column
ToothGrowth2$newcol <- NA

# Assign 1 and 2, automatically repeating to fill
ToothGrowth2$newcol <- c(1, 2)
```

With base R, the vector being assigned into the data frame will automatically be repeated to fill the number of rows in the data frame.

15.4 Deleting a Column from a Data Frame

Problem

You want to delete a column from a data frame. This returns a new data frame, which you'll typically want save over the original.

Solution

Use `select()` from `dplyr` and specify the columns you want to drop by using `-` (a minus sign):

```
# Remove the len column
ToothGrowth %>%
  select(-len)
```

Discussion

You can list multiple columns that you want to drop at the same time, or conversely specify only the columns that you want to keep. The following two pieces of code are thus equivalent:

```
# Remove both len and supp from ToothGrowth
ToothGrowth %>%
  select(-len, -supp)
```



```
# This keeps just dose, which has the same effect for this data set
ToothGrowth %>%
  select(dose)
```

To remove a column using base R, you can simply assign NULL to that column:

```
ToothGrowth$len <- NULL
```

See Also

[Recipe 15.7](#) for more on getting a subset of a data frame.

See `?select` for more ways to drop and keep columns.

15.5 Renaming Columns in a Data Frame

Problem

You want to rename the columns in a data frame.

Solution

Use `rename()` from `dplyr`. This returns a new data frame:

```
tg_mod <- ToothGrowth %>%
  rename(length = len)
```

Discussion

You can rename multiple columns within the same call to `rename()`:

```
ToothGrowth %>%
  rename(
    length = len,
    supplement_type = supp
  )
#>   length supplement_type dose
#> 1     4.2              VC  0.5
#> 2    11.5              VC  0.5
#> ...<56 more rows>...
#> 59    29.4             OJ  2.0
#> 60    23.0             OJ  2.0
```

Renaming a column using base R is a bit more verbose. It uses the `names()` function on the left side of the `<-` operator:

```
# Make a copy of ToothGrowth for this example
ToothGrowth2 <- ToothGrowth

names(ToothGrowth2) # Print the names of the columns
```

```
#> [1] "len" "supp" "dose"

# Rename "len" to "length"
names(ToothGrowth2)[names(ToothGrowth2) == "len"] <- "length"

names(ToothGrowth)
#> [1] "len" "supp" "dose"
```

See Also

See `?select` for more ways to rename columns within a data frame.

15.6 Reordering Columns in a Data Frame

Problem

You want to change the order of columns in a data frame.

Solution

Use `select()` from `dplyr`:

```
ToothGrowth %>%
  select(dose, len, supp)
#>   dose len supp
#> 1  0.5  4.2  VC
#> 2  0.5 11.5  VC
#> ...<56 more rows>...
#> 59 2.0 29.4  OJ
#> 60 2.0 23.0  OJ
```

The new data frame will contain the columns you specified in `select()`, in the order you specified. Note that `select()` returns a new data frame, so if you want to change the original variable, you'll need to save the new result over it.

Discussion

If you are only reordering a few variables and want to keep the rest of the variables in order, you can use `everything()` as a placeholder:

```
ToothGrowth %>%
  select(dose, everything())
#>   dose len supp
#> 1  0.5  4.2  VC
#> 2  0.5 11.5  VC
#> ...<56 more rows>...
#> 59 2.0 29.4  OJ
#> 60 2.0 23.0  OJ
```

See `?select_helpers` for other ways to select columns. You can, for example, select columns by matching parts of the name.

Using base R, you can also reorder columns by their name or numeric position. This returns a new data frame, which can be saved over the original:

```
ToothGrowth[c("dose", "len", "supp")]
```

```
ToothGrowth[c(3, 1, 2)]
```

In these examples, I used list-style indexing. A data frame is essentially a list of vectors, and indexing into it as a list will return another data frame. You can get the same effect with matrix-style indexing:

```
ToothGrowth[c("dose", "len", "supp")] # List-style indexing
```

```
ToothGrowth[, c("dose", "len", "supp")] # Matrix-style indexing
```

In this case, both methods return the same result, a data frame. However, when retrieving a single column, list-style indexing will return a data frame, while matrix-style indexing will return a vector:

```
ToothGrowth["dose"]
#>      dose
#> 1    0.5
#> 2    0.5
#> ...<56 more rows>...
#> 59   2.0
#> 60   2.0
ToothGrowth[, "dose"]
#> [1] 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
#> [19] 1.0 1.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 0.5 0.5 0.5 0.5 0.5 0.5
#> [37] 0.5 0.5 0.5 0.5 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 2.0 2.0
#> [55] 2.0 2.0 2.0 2.0 2.0 2.0
```

You can use `drop=FALSE` to ensure that it returns a data frame:

```
ToothGrowth[, "dose", drop=FALSE]
#>      dose
#> 1    0.5
#> 2    0.5
#> ...<56 more rows>...
#> 59   2.0
#> 60   2.0
```

15.7 Getting a Subset of a Data Frame

Problem

You want to get a subset of a data frame.

Solution

Use `filter()` to get the rows, and `select()` to get the columns you want. These operations can be chained together using the `%>%` operator. These functions return a new data frame, so if you want to change the original variable, you'll need to save the new result over it.

We'll use the `climate` data set for the examples here:

```
library(gcookbook) # Load gcookbook for the climate data set
climate
#>      Source Year Anomaly1y Anomaly5y Anomaly10y Unc10y
#> 1 Berkeley 1800      NA      NA      -0.435  0.505
#> 2 Berkeley 1801      NA      NA      -0.453  0.493
#> ...<495 more rows>...
#> 498 CRUTEM3 2010  0.8023      NA      NA      NA
#> 499 CRUTEM3 2011  0.6193      NA      NA      NA
```

Let's say that you only want to keep rows where `Source` is "Berkeley" and where the year is inclusive of and between 1900 and 2000. You can do so with the `filter()` function:

```
climate %>%
  filter(Source == "Berkeley" & Year >= 1900 & Year <= 2000)
```

If you want only the `Year` and `Anomaly10y` columns, use `select()`, as we did in [Recipe 15.4](#):

```
climate %>%
  select(Year, Anomaly10y)
#>      Year Anomaly10y
#> 1  1800      -0.435
#> 2  1801      -0.453
#> ...<495 more rows>...
#> 498 2010          NA
#> 499 2011          NA
```

These operations can be chained together using the `%>%` operator:

```
climate %>%
  filter(Source == "Berkeley" & Year >= 1900 & Year <= 2000) %>%
  select(Year, Anomaly10y)
#>      Year Anomaly10y
#> 1  1900      -0.171
#> 2  1901      -0.162
#> ...<97 more rows>...
#> 100 1999      0.734
#> 101 2000      0.748
```

Discussion

The `filter()` function picks out rows based on a condition. If you want to pick out rows based on their numeric position, use the `slice()` function:

```
slice(climate, 1:100)
```

I generally recommend indexing using names rather than numbers when possible. It makes the code easier to understand when you're collaborating with others or when you come back to it months or years after writing it, and it makes the code less likely to break when there are changes to the data, such as when columns are added or removed.

With base R, you can get a subset of rows like this:

```
climate[climate$Source == "Berkeley" &
        climate$Year >= 1900 &
        climate$Year <= 2000, ]
#>      Source Year Anomaly1y Anomaly5y Anomaly10y Unc10y
#> 101 Berkeley 1900      NA      NA      -0.171  0.108
#> 102 Berkeley 1901      NA      NA      -0.162  0.109
#> ...<97 more rows>...
#> 200 Berkeley 1999      NA      NA       0.734  0.025
#> 201 Berkeley 2000      NA      NA       0.748  0.026
```

Notice that we needed to prefix each column name with `climate$`, and that there's a comma after the selection criteria. This indicates that we're getting rows, not columns.

This row filtering can also be combined with the column selection from [Recipe 15.4](#):

```
climate[climate$Source == "Berkeley" &
        climate$Year >= 1900 &
        climate$Year <= 2000,
        c("Year", "Anomaly10y")]
#>      Year Anomaly10y
#> 101 1900     -0.171
#> 102 1901     -0.162
#> ...<97 more rows>...
#> 200 1999       0.734
#> 201 2000       0.748
```

15.8 Changing the Order of Factor Levels

Problem

You want to change the order of levels in a factor.

Solution

Pass the factor to `factor()`, and give it the levels in the order you want. This returns a new factor, so if you want to change the original variable, you'll need to save the new result over it:

```
# By default, levels are ordered alphabetically
sizes <- factor(c("small", "large", "large", "small", "medium"))
sizes
#> [1] small large large small medium
#> Levels: large medium small

factor(sizes, levels = c("small", "medium", "large"))
#> [1] small large large small medium
#> Levels: small medium large
```

The order can also be specified with `levels` when the factor is first created:

```
factor(c("small", "large", "large", "small", "medium"),
      levels = c("small", "medium", "large"))
```

Discussion

There are two kinds of factors in R: ordered factors and regular factors. (In practice, ordered levels are not commonly used.) In both types, the levels are arranged in *some* order; the difference is that the order is meaningful for an ordered factor, but it is arbitrary for a regular factor—it simply reflects how the data is stored. For plotting data, the distinction between ordered and regular factors is generally unimportant, and they can be treated the same.

The order of factor levels affects graphical output. When a factor variable is mapped to an aesthetic property in ggplot, the aesthetic adopts the ordering of the factor levels. If a factor is mapped to the x-axis, the ticks on the axis will be in the order of the factor levels, and if a factor is mapped to color, the items in the legend will be in the order of the factor levels.

To reverse the level order, you can use `rev(levels())`:

```
factor(sizes, levels = rev(levels(sizes)))
```

The tidyverse function for reordering factors is `fct_relevel()` from the `forcats` package. It has a syntax similar to the `factor()` function from base R:

```
# Change the order of levels
library(forcats)
fct_relevel(sizes, "small", "medium", "large")
#> [1] small large large small medium
#> Levels: small medium large
```

See Also

To reorder a factor based on the value of another variable, see [Recipe 15.9](#).

Reordering factor levels is useful for controlling the order of axes and legends. See [Recipes 8.4](#) and [10.3](#) for more information.

15.9 Changing the Order of Factor Levels Based on Data Values

Problem

You want to change the order of levels in a factor based on values in the data.

Solution

Use `reorder()` with the factor that has levels to reorder, the values to base the reordering on, and a function that aggregates the values:

```
# Make a copy of the InsectSprays data set since we're modifying it
iss <- InsectSprays
iss$spray
#> [1] A A A A A A A A A A A B B B B B B B B B B C C C C C C C C C C C
#> [37] D D D D D D D D D D D E E E E E E E E E E F F F F F F F F F F F
#> Levels: A B C D E F

iss$spray <- reorder(iss$spray, iss$count, FUN = mean)
iss$spray
#> [1] A A A A A A A A A A A B B B B B B B B B B C C C C C C C C C C C
#> [37] D D D D D D D D D D D E E E E E E E E E E F F F F F F F F F F F
#> attr("scores")
#>      A          B          C          D          E          F
#> 14.500000 15.333333  2.083333  4.916667  3.500000 16.666667
#> Levels: C E D A B F
```

Notice that the original levels were ABCDEF, while the reordered levels are CEDABF. What we've done is reorder the levels of `spray` based on the mean value of `count` for each level of `spray`.

Discussion

The usefulness of `reorder()` might not be obvious from just looking at the raw output. [Figure 15-1](#) shows three plots made with `reorder()`. In these plots, the order in which the items appear is determined by their values.

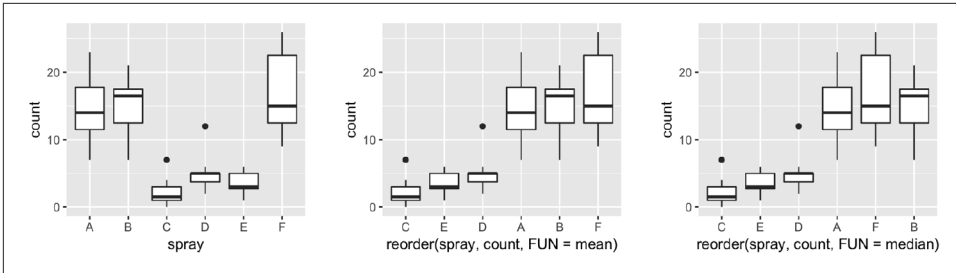


Figure 15-1. Original data (left); Reordered by the mean of each group (middle); Reordered by the median of each group (right)

In the middle plot in [Figure 15-1](#), the boxes are sorted by the mean. The horizontal line that runs across each box represents the *median* of the data. Notice that these values do not increase strictly from left to right. That's because with this particular data set, sorting by the mean gives a different order than sorting by the median. To make the median lines increase from left to right, as in the plot on the right in [Figure 15-1](#), we used the `median()` function in `reorder()`.

The tidyverse function for reordering factors is `fct_reorder()`, and it is used the same way as `reorder()`. These do the same thing:

```
reorder(iss$spray, iss$count, FUN = mean)
fct_reorder(iss$spray, iss$count, .fun = mean)
```

See Also

Reordering factor levels is also useful for controlling the order of axes and legends. See [Recipes 8.4](#) and [10.3](#) for more information.

15.10 Changing the Names of Factor Levels

Problem

You want to change the names of levels in a factor.

Solution

Use `fct_recode()` from the `forcats` package:

```
sizes <- factor(c( "small", "large", "large", "small", "medium"))
sizes
#> [1] small large large small medium
#> Levels: large medium small

# Pass it a named vector with the mappings
fct_recode(sizes, S = "small", M = "medium", L = "large")
```



```
#> [1] S L L S M
#> Levels: L M S
```

Discussion

If you want to use two vectors, one with the original levels and one with the new ones, use `do.call()` with `fct_recode()`:

```
old <- c("small", "medium", "large")
new <- c("S", "M", "L")

# Create a named vector that has the mappings between old and new
mappings <- setNames(old, new)
mappings
#>      S      M      L
#> "small" "medium" "large"

# Create a list of the arguments to pass to fct_recode
args <- c(list(sizes), mappings)

# Look at the structure of the list
str(args)
#> List of 4
#> $ : Factor w/ 3 levels "large","medium",...: 3 1 1 3 2
#> $ S: chr "small"
#> $ M: chr "medium"
#> $ L: chr "large"

# Use do.call to call fct_recode with the arguments
do.call(fct_recode, args)
#> [1] S L L S M
#> Levels: L M S
```

Or, more concisely, we can do all of that in one go:

```
do.call(
  fct_recode,
  c(list(sizes), setNames(c("small", "medium", "large"), c("S", "M", "L"))))
#> [1] S L L S M
#> Levels: L M S
```

For a more traditional (and clunky) base R method for renaming factor levels, use the `levels()<-` function:

```
sizes <- factor(c("small", "large", "large", "small", "medium"))

# Index into the levels and rename each one
levels(sizes)[levels(sizes) == "large"] <- "L"
levels(sizes)[levels(sizes) == "medium"] <- "M"
levels(sizes)[levels(sizes) == "small"] <- "S"
sizes
```

```
#> [1] S L L S M
#> Levels: L M S
```

If you are renaming *all* your factor levels, there is a simpler method. You can pass a list to `levels()`:-:

```
sizes <- factor(c("small", "large", "large", "small", "medium"))
levels(sizes) <- list(S = "small", M = "medium", L = "large")
sizes
#> [1] S L L S M
#> Levels: S M L
```

With this method, all factor levels must be specified in the list; if any are missing, they will be replaced with NA.

It's also possible to rename factor levels by position, but this is somewhat inelegant:

```
sizes <- factor(c("small", "large", "large", "small", "medium"))
levels(sizes)[1] <- "L"
sizes
#> [1] small L      L      small medium
#> Levels: L medium small

# Rename all levels at once
levels(sizes) <- c("L", "M", "S")
sizes
#> [1] S L L S M
#> Levels: L M S
```

It's safer to rename factor levels by name rather than by position, since you will be less likely to make a mistake (and mistakes here may be hard to detect). Also, if your input data set changes to have more or fewer levels, the numeric positions of the existing levels could change, which could cause serious but nonobvious problems for your analysis.

See Also

If, instead of a factor, you have a character vector with items to rename, see [Recipe 15.12](#).

15.11 Removing Unused Levels from a Factor

Problem

You want to remove unused levels from a factor.

Solution

Sometimes, after processing your data, you will have a factor that contains levels that are no longer used. Here's an example:

```
sizes <- factor(c("small", "large", "large", "small", "medium"))
sizes <- sizes[1:3]
sizes
#> [1] small large large
#> Levels: large medium small
```

To remove them, use `droplevels()`:

```
droplevels(sizes)
#> [1] small large large
#> Levels: large small
```

Discussion

The `droplevels()` function preserves the order of factor levels. You can use the `except` parameter to keep particular levels.

To do it the tidyverse way, use `fct_drop()` from the `forcats` package:

```
fct_drop(sizes)
#> [1] small large large
#> Levels: large small
```

15.12 Changing the Names of Items in a Character Vector

Problem

You want to change the names of items in a character vector.

Solution

Use `recode()` from the `dplyr` package:

```
library(dplyr)

sizes <- c("small", "large", "large", "small", "medium")
sizes
#> [1] "small" "large" "large" "small" "medium"

# With recode(), pass it a named vector with the mappings
recode(sizes, small = "S", medium = "M", large = "L")
#> [1] "S" "L" "L" "S" "M"

# Can also use quotes -- useful if there are spaces or other strange characters
recode(sizes, "small" = "S", "medium" = "M", "large" = "L")
#> [1] "S" "L" "L" "S" "M"
```

Discussion

If you want to use two vectors, one with the original levels and one with the new ones, use `do.call()` with `fct_recode()`:

```
old <- c("small", "medium", "large")
new <- c("S", "M", "L")
# Create a named vector that has the mappings between old and new
mappings <- setNames(new, old)
mappings
#> small medium large
#> "S" "M" "L"

# Create a list of the arguments to pass to fct_recode
args <- c(list(sizes), mappings)
# Look at the structure of the list
str(args)
#> List of 4
#> $ : chr [1:5] "small" "large" "large" "small" ...
#> $ small : chr "S"
#> $ medium: chr "M"
#> $ large : chr "L"
# Use do.call to call fct_recode with the arguments
do.call(recode, args)
#> [1] "S" "L" "L" "S" "M"
```

Or, more concisely, we can do all of that in one go:

```
do.call(
  recode,
  c(list(sizes), setNames(c("S", "M", "L"), c("small", "medium", "large"))))
#> [1] "S" "L" "L" "S" "M"
```

Note that for `recode()`, the name and value of the arguments is reversed, compared to the `fct_recode()` function from the `forcats` package. With `recode()`, you would use `small="S"`, whereas for `fct_recode()`, you would use `S="small"`.

A more traditional R method is to use square-bracket indexing to select the items and rename them:

```
sizes <- c("small", "large", "large", "small", "medium")
sizes[sizes == "small"] <- "S"
sizes[sizes == "medium"] <- "M"
sizes[sizes == "large"] <- "L"
sizes
#> [1] "S" "L" "L" "S" "M"
```

See Also

If, instead of a character vector, you have a factor with levels to rename, see [Recipe 15.10](#).

15.13 Recoding a Categorical Variable to Another Categorical Variable

Problem

You want to recode a categorical variable to another variable.

Solution

For the examples here, we'll use a subset of the `PlantGrowth` data set:

```
# Work on a subset of the PlantGrowth data set
pg <- PlantGrowth[c(1,2,11,21,22), ]
pg
#>   weight group
#> 1    4.17  ctrl
#> 2    5.58  ctrl
#> 11   4.81 trt1
#> 21   6.31 trt2
#> 22   5.12 trt2
```

In this example, we'll recode the categorical variable `group` into another categorical variable, `treatment`. If the old value was `"ctrl"`, the new value will be `"No"`, and if the old value was `"trt1"` or `"trt2"`, the new value will be `"Yes"`.

This can be done with the `recode()` function from the `dplyr` package:

```
library(dplyr)

recode(pg$group, ctrl = "No", trt1 = "Yes", trt2 = "Yes")
#> [1] No  No  Yes Yes Yes
#> Levels: No Yes
```

You can assign it as a new column in the data frame:

```
pg$treatment <- recode(pg$group, ctrl = "No", trt1 = "Yes", trt2 = "Yes")
```

Note that since the input was a factor, it returns a factor. If you want to get a character vector instead, use `as.character()`:

```
recode(as.character(pg$group), ctrl = "No", trt1 = "Yes", trt2 = "Yes")
#> [1] "No"  "No"  "Yes" "Yes" "Yes"
```

Discussion

You can also use the `fct_recode()` function from the `forcats` package. It works the same, except the names and values are swapped, which may be a little more intuitive:

```
library(forcats)
fct_recode(pg$group, No = "ctrl", Yes = "trt1", Yes = "trt2")
```

```
#> [1] No  No  Yes Yes Yes
#> Levels: No Yes
```

Another difference is that `fct_recode()` will always return a factor, whereas `recode()` will return a character vector if it is given a character vector, and will return a factor if it is given a factor. (Although `dplyr` does have a `recode_factor()` function that also always returns a factor.)

Using base R, recoding can be done with the `match()` function:

```
oldvals <- c("ctrl", "trt1", "trt2")
newvals <- factor(c("No", "Yes", "Yes"))

newvals[ match(pg$group, oldvals) ]
#> [1] No  No  Yes Yes Yes
#> Levels: No Yes
```

It can also be done by indexing in the vectors:

```
pg$treatment[pg$group == "ctrl"] <- "No"
pg$treatment[pg$group == "trt1"] <- "Yes"
pg$treatment[pg$group == "trt2"] <- "Yes"

# Convert to a factor
pg$treatment <- factor(pg$treatment)
pg
#>   weight group treatment
#> 1    4.17  ctrl       No
#> 2    5.58  ctrl       No
#> 11   4.81 trt1       Yes
#> 21   6.31 trt2       Yes
#> 22   5.12 trt2       Yes
```

Here, we combined two of the factor levels and put the result into a new column. If you simply want to rename the levels of a factor, see [Recipe 15.10](#).

The coding criteria can also be based on values in multiple columns, by using the `&` and `|` operators:

```
pg$newcol[pg$group == "ctrl" & pg$weight < 5] <- "no_small"
pg$newcol[pg$group == "ctrl" & pg$weight >= 5] <- "no_large"
pg$newcol[pg$group == "trt1"] <- "yes"
pg$newcol[pg$group == "trt2"] <- "yes"
pg$newcol <- factor(pg$newcol)
pg
#>   weight group newcol
#> 1    4.17  ctrl no_small
#> 2    5.58  ctrl no_large
#> 11   4.81 trt1     yes
#> 21   6.31 trt2     yes
#> 22   5.12 trt2     yes
```

It's also possible to combine two columns into one using the `interaction()` function, which appends the values with a `.` in between. This combines the `weight` and `group` columns into a new column, `weightgroup`:

```
pg$weightgroup <- interaction(pg$weight, pg$group)
pg
#>   weight group weightgroup
#> 1   4.17  ctrl   4.17.ctrl
#> 2   5.58  ctrl   5.58.ctrl
#> 11  4.81 trt1    4.81.trt1
#> 21  6.31 trt2    6.31.trt2
#> 22  5.12 trt2    5.12.trt2
```

See Also

For more on renaming factor levels, see [Recipe 15.10](#).

See [Recipe 15.14](#) for recoding continuous values to categorical values.

15.14 Recoding a Continuous Variable to a Categorical Variable

Problem

You want to recode a continuous variable to another variable.

Solution

Use the `cut()` function. In this example, we'll use the `PlantGrowth` data set and recode the continuous variable `weight` into a categorical variable, `wtclass`, using the `cut()` function:

```
pg <- PlantGrowth
pg$wtclass <- cut(pg$weight, breaks = c(0, 5, 6, Inf))
pg
#>   weight group wtclass
#> 1   4.17  ctrl  (0,5]
#> 2   5.58  ctrl  (5,6]
#> ...<26 more rows>...
#> 29  5.80 trt2  (5,6]
#> 30  5.26 trt2  (5,6]
```

Discussion

For three categories we specify four bounds, which can include `Inf` and `-Inf`. If a data value falls outside of the specified bounds, it's categorized as `NA`. The result of

cut() is a factor, and you can see from the example that the factor levels are named after the bounds.

To change the names of the levels, set the labels:

```
pg$wtclass <- cut(pg$weight, breaks = c(0, 5, 6, Inf),
                  labels = c("small", "medium", "large"))

pg
#>   weight group wtclass
#> 1    4.17  ctrl  small
#> 2    5.58  ctrl  medium
#> ...<26 more rows>...
#> 29    5.80 trt2  medium
#> 30    5.26 trt2  medium
```

As indicated by the factor levels, the bounds are by default *open* on the left and *closed* on the right. In other words, they don't include the lowest value, but they do include the highest value. For the smallest category, you can have it include both the lower and upper values by setting include.lowest=TRUE. In this example, this would result in 0 values going into the small category; otherwise, 0 would be coded as NA.

If you want the categories to be closed on the left and open on the right, set right = FALSE:

```
cut(pg$weight, breaks = c(0, 5, 6, Inf), right = FALSE)
#> [1] [0,5) [5,6) [5,6) [6,Inf) [0,5) [0,5) [5,6) [0,5) [5,6)
#> [10] [5,6) [0,5) [0,5) [0,5) [0,5) [5,6) [0,5) [6,Inf) [0,5)
#> [19] [0,5) [0,5) [6,Inf) [5,6) [5,6) [5,6) [5,6) [5,6) [0,5)
#> [28] [6,Inf) [5,6) [5,6)
#> Levels: [0,5) [5,6) [6,Inf)
```

See Also

To recode a categorical variable to another categorical variable, see [Recipe 15.13](#).

15.15 Calculating New Columns from Existing Columns

Problem

You want to calculate a new column of values in a data frame.

Solution

Use mutate() from the dplyr package:

```
library(gcookbook) # Load gcookbook for the heightweight data set
heightweight
#>   sex ageYear ageMonth heightIn weightLb
#> 1   f   11.92    143    56.3    85.0
#> 2   f   12.92    155    62.3   105.0
```



```
#> ...<232 more rows>...
#> 236 m 13.92 167 62.0 107.5
#> 237 m 12.58 151 59.3 87.0
```

This will convert heightIn to centimeters and store it in a new column, heightCm:

```
heightweight %>%
  mutate(heightCm = heightIn * 2.54)
#>   sex ageYear ageMonth heightIn weightLb heightCm
#> 1   f   11.92     143     56.3    85.0  143.002
#> 2   f   12.92     155     62.3   105.0  158.242
#> ...<232 more rows>...
#> 235 m   13.92     167     62.0   107.5  157.480
#> 236 m   12.58     151     59.3    87.0  150.622
```

This returns a new data frame, so if you want to replace the original variable, you will need to save the result over it.

Discussion

You can use mutate() to transform multiple columns at once:

```
heightweight %>%
  mutate(
    heightCm = heightIn * 2.54,
    weightKg = weightLb / 2.204
  )
#>   sex ageYear ageMonth heightIn weightLb heightCm weightKg
#> 1   f   11.92     143     56.3    85.0  143.002  38.56624
#> 2   f   12.92     155     62.3   105.0  158.242  47.64065
#> ...<232 more rows>...
#> 235 m   13.92     167     62.0   107.5  157.480  48.77495
#> 236 m   12.58     151     59.3    87.0  150.622  39.47368
```

It is also possible to calculate a new column based on multiple columns:

```
heightweight %>%
  mutate(bmi = weightKg / (heightCm / 100)^2)
```

With mutate(), the columns are added sequentially. That means that we can reference a newly created column when calculating a new column:

```
heightweight %>%
  mutate(
    heightCm = heightIn * 2.54,
    weightKg = weightLb / 2.204,
    bmi = weightKg / (heightCm / 100)^2
  )
#>   sex ageYear ageMonth heightIn weightLb heightCm weightKg    bmi
#> 1   f   11.92     143     56.3    85.0  143.002  38.56624 18.85919
#> 2   f   12.92     155     62.3   105.0  158.242  47.64065 19.02542
#> ...<232 more rows>...
#> 235 m   13.92     167     62.0   107.5  157.480  48.77495 19.66736
#> 236 m   12.58     151     59.3    87.0  150.622  39.47368 17.39926
```

With base R, calculating a new column can be done by referencing the new column with the `$` operator and assigning some values to it:

```
heightweight$heightCm <- heightweight$heightIn * 2.54
```

See Also

See [Recipe 15.16](#) for how to perform group-wise transformations on data.

15.16 Calculating New Columns by Groups

Problem

You want to create new columns that are the result of calculations performed on groups of data, as specified by a grouping column.

Solution

Use `group_by()` from the `dplyr` package to specify the grouping variable, and then specify the operations in `mutate()`:

```
library(MASS) # Load MASS for the cabbages data set
library(dplyr)
```

```
cabbages %>%
  group_by(Cult) %>%
  mutate(DevWt = HeadWt - mean(HeadWt))

#> # A tibble: 60 x 5
#> # Groups:   Cult [2]
#>   Cult Date HeadWt VitC DevWt
#>   <fct> <fct>   <dbl> <int> <dbl>
#> 1 c39 d16      2.5    51 -0.407
#> 2 c39 d16      2.2    55 -0.707
#> 3 c39 d16      3.1    45  0.193
#> 4 c39 d16      4.3    42  1.39
#> 5 c39 d16      2.5    53 -0.407
#> 6 c39 d16      4.3    50  1.39
#> # ... with 54 more rows
```

This returns a new data frame, so if you want to replace the original variable, you will need to save the result over it.

Discussion

Let's take a closer look at the `cabbages` data set. It has two grouping variables (factors): `Cult`, which has levels `c39` and `c52`, and `Date`, which has levels `d16`, `d20`, and `d21`. It also has two measured numeric variables, `HeadWt` and `VitC`:

```
cabbages
#>   Cult Date HeadWt VitC
#> 1  c39 d16    2.5  51
#> 2  c39 d16    2.2  55
#> ...<56 more rows>...
#> 59 c52 d21    1.5  66
#> 60 c52 d21    1.6  72
```

Suppose we want to find, for each case, the deviation of HeadWt from the overall mean. All we have to do is take the overall mean and subtract it from the observed value for each case:

```
mutate(cabbages, DevWt = HeadWt - mean(HeadWt))
#>   Cult Date HeadWt VitC      DevWt
#> 1  c39 d16    2.5  51 -0.09333333
#> 2  c39 d16    2.2  55 -0.39333333
#> ...<56 more rows>...
#> 59 c52 d21    1.5  66 -1.09333333
#> 60 c52 d21    1.6  72 -0.99333333
```

You'll often want to do separate operations like this for each group, where the groups are specified by one or more grouping variables. Suppose, for example, we want to normalize the data within each group by finding the deviation of each case from the mean *within the group*, where the groups are specified by Cult. In these cases, we can use `group_by()` and `mutate()` together:

```
cb <- cabbages %>%
  group_by(Cult) %>%
  mutate(DevWt = HeadWt - mean(HeadWt))
```

First, it groups cabbages based on the value of Cult. There are two levels of Cult, c39 and c52. It then applies the `mutate()` function to each data frame.

The before and after results are shown in [Figure 15-2](#):

```
# The data before normalizing
ggplot(cb, aes(x = Cult, y = HeadWt)) +
  geom_boxplot()

# After normalizing
ggplot(cb, aes(x = Cult, y = DevWt)) +
  geom_boxplot()
```

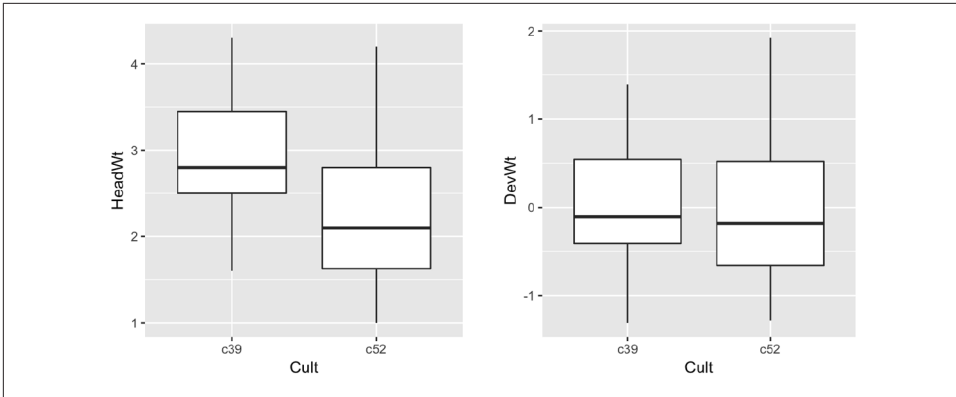


Figure 15-2. Before normalizing (left); After normalizing (right)

You can also group the data frame on multiple variables and perform operations on multiple variables. The following code groups the data by `Cult` and `Date`, forming a group for each distinct combination of the two variables. After forming these groups, the code will calculate the deviation of `HeadWt` and `VitC` from the mean of each group:

```
cabbages %>%
  group_by(Cult, Date) %>%
  mutate(
    DevWt = HeadWt - mean(HeadWt),
    DevVitC = VitC - mean(VitC)
  )
#> # A tibble: 60 x 6
#> # Groups:   Cult, Date [6]
#>   Cult Date HeadWt VitC DevWt DevVitC
#>   <fct> <fct>   <dbl> <int> <dbl>   <dbl>
#> 1 c39 d16     2.5    51 -0.68    0.7
#> 2 c39 d16     2.2    55 -0.98    4.7
#> 3 c39 d16     3.1    45 -0.08   -5.30
#> 4 c39 d16     4.3    42  1.12   -8.30
#> 5 c39 d16     2.5    53 -0.68    2.7
#> 6 c39 d16     4.3    50  1.12   -0.300
#> # ... with 54 more rows
```

See Also

To summarize data by groups, see [Recipe 15.17](#).

15.17 Summarizing Data by Groups

Problem

You want to summarize your data, based on one or more grouping variables.

Solution

Use `group_by()` and `summarise()` from the `dplyr` package, and specify the operations to do:

```
library(MASS) # Load MASS for the cabbages data set
library(dplyr)

cabbages %>%
  group_by(Cult, Date) %>%
  summarise(
    Weight = mean(HeadWt),
    VitC = mean(VitC)
  )
#> # A tibble: 6 x 4
#> # Groups:   Cult [?]
#>   Cult Date Weight VitC
#>   <fct> <fct>   <dbl> <dbl>
#> 1 c39  d16     3.18  50.3
#> 2 c39  d20     2.8   49.4
#> 3 c39  d21     2.74  54.8
#> 4 c52  d16     2.26  62.5
#> 5 c52  d20     3.11  58.9
#> 6 c52  d21     1.47  71.8
```

Discussion

There are a few things going on here that may be unfamiliar if you're new to `dplyr` and the tidyverse in general.

First, let's take a closer look at the `cabbages` data set. It has two factors that can be used as grouping variables: `Cult`, which has levels `c39` and `c52`, and `Date`, which has levels `d16`, `d20`, and `d21`. It also has two numeric variables, `HeadWt` and `VitC`:

```
cabbages
#>   Cult Date HeadWt VitC
#> 1  c39  d16    2.5   51
#> 2  c39  d16    2.2   55
#> ...<56 more rows>...
#> 59 c52  d21    1.5   66
#> 60 c52  d21    1.6   72
```

Finding the overall mean of `HeadWt` is simple. We could just use the `mean()` function on that column, but for reasons that will soon become clear, we'll use the `summarise()` function instead:

```
library(dplyr)
summarise(cabbages, Weight = mean(HeadWt))
#>   Weight
#> 1 2.593333
```

The result is a data frame with one row and one column, named `Weight`.

Often we want to find information about each subset of the data, as specified by a grouping variable. For example, suppose we want to find the mean of each `Cult` group. To do this, we can use `summarise()` with `group_by()`:

```
tmp <- group_by(cabbages, Cult)
summarise(tmp, Weight = mean(HeadWt))
#> # A tibble: 2 x 2
#>   Cult Weight
#>   <fct>   <dbl>
#> 1 c39     2.91
#> 2 c52     2.28
```

The command first groups the data frame `cabbages` based on the value of `Cult`. There are two levels of `Cult`, `c39` and `c52`, so there are two groups. It then applies the `summarise()` function to each of these data frames; it calculates `Weight` by taking the `mean()` of the `HeadWt` column in each of the subdata frames. The resulting summaries for each group are assembled into a data frame, which is returned.

You can imagine that the `cabbages` data is split up into two separate data frames, then `summarise()` is called on each data frame (returning a one-row data frame for each), and then those results are combined together into a final data frame. This is actually how things worked in `dplyr`'s predecessor, `plyr`, with the `ddply()` function.

The syntax of the previous code used a temporary variable to store results. That's a little verbose, so instead, we can use `%>`, also known as the pipe operator, to chain the function calls together. The pipe operator simply takes what's on its left and substitutes it as the first argument of the function call on the right. The following two lines of code are equivalent:

```
group_by(cabbages, Cult)
# The pipe operator moves `cabbages` to the first argument position of group_by()
cabbages %>% group_by(Cult)
```

The reason it's called a pipe operator is that it lets you connect function calls together in sequence to form a pipeline of operations. Another common term for this is a different metaphor: *chaining*.

So the first argument of the function call is in a different place. So what? The advantages become apparent when chaining is involved. Here's what it would look like if

you wanted to call `group_by()` and then `summarise()` without making use of a temporary variable. Instead of proceeding left to right, the computation occurs from the inside out:

```
summarise(group_by(cabbages, Cult), Weight = mean(HeadWt))
```

Using a temporary variable, as we did earlier, makes it more readable, but a more elegant solution is to use the pipe operator:

```
cabbages %>%  
  group_by(Cult) %>%  
  summarise(Weight = mean(HeadWt))
```

Back to summarizing data. Summarizing the data frame by grouping using more variables (or columns) is simple: just give it the names of the additional variables. It's also possible to get more than one summary value by specifying more calculated columns. Here we'll summarize each `Cult` and `Date` group, getting the average of `HeadWt` and `VitC`:

```
cabbages %>%  
  group_by(Cult, Date) %>%  
  summarise(  
    Weight = mean(HeadWt),  
    Vitc = mean(VitC)  
  )  
#> # A tibble: 6 x 4  
#> # Groups:   Cult [?]  
#>   Cult Date Weight Vitc  
#>   <fct> <fct>   <dbl> <dbl>  
#> 1 c39  d16     3.18  50.3  
#> 2 c39  d20     2.8   49.4  
#> 3 c39  d21     2.74  54.8  
#> 4 c52  d16     2.26  62.5  
#> 5 c52  d20     3.11  58.9  
#> 6 c52  d21     1.47  71.8
```



You might have noticed that it says that the result is grouped by `Cult`, but not `Date`. This is because the `summarise()` function removes one level of grouping. This is typically what you want when the input has one grouping variable. When there are multiple grouping variables, this may or may not be what you want. To remove all grouping, use `ungroup()`, and to add back the original grouping, use `group_by()` again.

It's possible to do more than take the mean. You may, for example, want to compute the standard deviation and count of each group. To get the standard deviation, use `sd()`, and to get a count of rows in each group, use `n()`:

```

cabbages %>%
  group_by(Cult, Date) %>%
  summarise(
    Weight = mean(HeadWt),
    sd = sd(HeadWt),
    n = n()
  )
#> # A tibble: 6 x 5
#> # Groups:   Cult [?]
#>   Cult Date Weight    sd    n
#>   <fct> <fct>   <dbl> <dbl> <int>
#> 1 c39  d16     3.18 0.957   10
#> 2 c39  d20     2.8  0.279   10
#> 3 c39  d21     2.74 0.983   10
#> 4 c52  d16     2.26 0.445   10
#> 5 c52  d20     3.11 0.791   10
#> 6 c52  d21     1.47 0.211   10

```

Other useful functions for generating summary statistics include `min()`, `max()`, and `median()`. The `n()` function is a special function that works only inside of the dplyr functions `summarise()`, `mutate()`, and `filter()`. See `?summarise` for more useful functions.

The `n()` function gets a count of rows, but if you want to have it *not* count NA values from a column, you need to use a different technique. For example, if you want it to ignore any NAs in the `HeadWt` column, use `sum(!is.na(Headwt))`.

Dealing with NAs

One potential pitfall is that NAs in the data will lead to NAs in the output. Let's see what happens if we sprinkle a few NAs into `HeadWt`:

```

c1 <- cabbages # Make a copy
c1$HeadWt[c(1, 20, 45)] <- NA # Set some values to NA

c1 %>%
  group_by(Cult) %>%
  summarise(
    Weight = mean(HeadWt),
    sd = sd(HeadWt),
    n = n()
  )
#> # A tibble: 2 x 4
#>   Cult Weight    sd    n
#>   <fct>   <dbl> <dbl> <int>
#> 1 c39      NA    NA    30
#> 2 c52      NA    NA    30

```

The problem is that `mean()` and `sd()` simply return NA if any of the input values are NA. Fortunately, these functions have an option to deal with this very issue: setting `na.rm=TRUE` will tell them to ignore the NAs:


```

c1 %>%
  group_by(Cult) %>%
  summarise(
    Weight = mean(HeadWt, na.rm = TRUE),
    sd = sd(HeadWt, na.rm = TRUE),
    n = n()
  )
#> # A tibble: 2 x 4
#>   Cult Weight    sd    n
#>   <fct> <dbl> <dbl> <int>
#> 1 c39    2.9  0.822   30
#> 2 c52    2.23 0.828   30

```

Missing combinations

If there are any empty combinations of the grouping variables, they will not appear in the summarized data frame. These missing combinations can cause problems when making graphs. To illustrate, we'll remove all entries that have levels c52 and d21. The graph on the left in [Figure 15-3](#) shows what happens when there's a missing combination in a bar graph:

```

# Copy cabbages and remove all rows with both c52 and d21
c2 <- filter(cabbages, !( Cult == "c52" & Date == "d21" ))
c2a <- c2 %>%
  group_by(Cult, Date) %>%
  summarise(Weight = mean(HeadWt))

ggplot(c2a, aes(x = Date, fill = Cult, y = Weight)) +
  geom_col(position = "dodge")

```

To fill in the missing combination ([Figure 15-3](#), right), use the `complete()` function from the `tidyr` package—which is also part of the `tidyverse`. Also, the grouping for `c2a` must be removed, with `ungroup()`; otherwise it will return too many rows:

```

library(tidyr)
c2b <- c2a %>%
  ungroup() %>%
  complete(Cult, Date)

ggplot(c2b, aes(x = Date, fill = Cult, y = Weight)) +
  geom_col(position = "dodge")

# Copy cabbages and remove all rows with both c52 and d21
c2 <- filter(cabbages, !( Cult == "c52" & Date == "d21" ))
c2a <- c2 %>%
  group_by(Cult, Date) %>%
  summarise(Weight = mean(HeadWt))

ggplot(c2a, aes(x = Date, fill = Cult, y = Weight)) +
  geom_col(position = "dodge")
library(tidyr)
c2b <- c2a %>%

```

```

ungroup() %>%
complete(Cult, Date)

ggplot(c2b, aes(x = Date, fill = Cult, y = Weight)) +
  geom_col(position = "dodge")

```

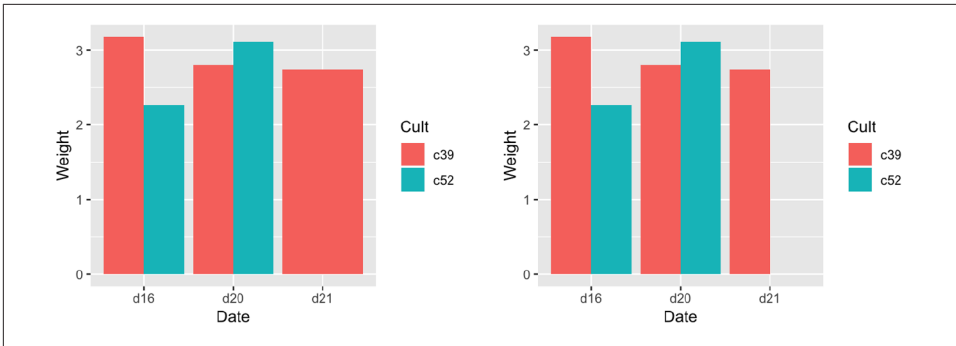


Figure 15-3. Bar graph with a missing combination (left); With missing combination filled (right)

When we used `complete()`, it filled in the missing combinations with NA. It's possible to fill with a different value, with the `fill` parameter. See `?complete` for more information.

See Also

If you want to calculate standard errors and confidence intervals, see [Recipe 15.18](#).

See [Recipe 6.8](#) for an example of using `stat_summary()` to calculate means and overlay them on a graph.

To perform transformations on data by groups, see [Recipe 15.16](#).

15.18 Summarizing Data with Standard Errors and Confidence Intervals

Problem

You want to summarize your data with the standard error of the mean and/or confidence intervals.

Solution

Getting the standard error of the mean involves two steps: first get the standard deviation and count for each group, then use those values to calculate the standard error.

The standard error for each group is just the standard deviation divided by the square root of the sample size:

```
library(MASS) # Load MASS for the cabbages data set
library(dplyr)

ca <- cabbages %>%
  group_by(Cult, Date) %>%
  summarise(
    Weight = mean(HeadWt),
    sd = sd(HeadWt),
    n = n(),
    se = sd / sqrt(n)
  )

Ca
#> # A tibble: 6 x 6
#> # Groups:   Cult [?]
#>   Cult Date Weight    sd     n     se
#>   <fct> <fct> <dbl> <dbl> <int> <dbl>
#> 1 c39 d16    3.18 0.957    10 0.303
#> 2 c39 d20    2.8  0.279    10 0.0882
#> 3 c39 d21    2.74 0.983    10 0.311
#> 4 c52 d16    2.26 0.445    10 0.141
#> 5 c52 d20    3.11 0.791    10 0.250
#> 6 c52 d21    1.47 0.211    10 0.0667
```

Discussion

The `summarise()` function computes the columns in order, so you can refer to previously newly created columns. That's why `se` can use the `sd` and `n` columns.

The `n()` function gets a count of rows, but if you want to have it *not* count NA values from a column, you need to use a different technique. For example, if you want it to ignore any NAs in the `HeadWt` column, use `sum(!is.na(HeadWt))`.

Confidence intervals

Confidence intervals are calculated using the standard error of the mean and the degrees of freedom. To calculate a confidence interval, use the `qt()` function to get the quantile, then multiply that by the standard error. The `qt()` function will give quantiles of the *t*-distribution when given a probability level and degrees of freedom. For a 95% confidence interval, use a probability level of .975; for the bell-shaped *t*-distribution, this will in essence cut off 2.5% of the area under the curve at either end. The degrees of freedom equal the sample size minus one.

This will calculate the multiplier for each group. There are six groups and each has the same number of observations (10), so they will all have the same multiplier:

```
ciMult <- qt(.975, ca$n - 1)
ciMult
#> [1] 2.262157 2.262157 2.262157 2.262157 2.262157 2.262157
```

Now we can multiply that vector by the standard error to get the 95% confidence interval:

```
ca$ci95 <- ca$se * ciMult
ca
#> # A tibble: 6 x 7
#> # Groups:   Cult [?]
#>   Cult Date Weight sd n se ci95
#>   <fct> <fct> <dbl> <dbl> <int> <dbl> <dbl>
#> 1 c39 d16 3.18 0.957 10 0.303 0.684
#> 2 c39 d20 2.8 0.279 10 0.0882 0.200
#> 3 c39 d21 2.74 0.983 10 0.311 0.703
#> 4 c52 d16 2.26 0.445 10 0.141 0.318
#> 5 c52 d20 3.11 0.791 10 0.250 0.566
#> 6 c52 d21 1.47 0.211 10 0.0667 0.151
```

This could be done in one line, like this:

```
ca$ci95 <- ca$se * qt(.975, ca$n - 1)
```

For a 99% confidence interval, use .995.

Error bars that represent the standard error of the mean and confidence intervals serve the same general purpose: to give the viewer an idea of how good the estimate of the population mean is. The standard error is the standard deviation of the sampling distribution. Confidence intervals are a little easier to interpret. Very roughly, a 95% confidence interval means that there's a 95% chance that the true population mean is within the interval (actually, it doesn't mean this at all, but this seemingly simple topic is way too complicated to cover here; if you want to know more, read up on Bayesian statistics).

This function will perform all the steps of calculating the standard deviation, count, standard error, and confidence intervals. It can also handle NAs and missing combinations, with the `na.rm` option. By default, it provides a 95% confidence interval, but this can be set with the `conf.interval` argument:

```
summarySE <- function(data = NULL, measurevar, groupvars = NULL, na.rm = FALSE,
  conf.interval = .95) {

  # New version of length which can handle NA's: if na.rm=T, don't count them
  length2 <- function(x, na.rm = FALSE) {
    if (na.rm) sum(!is.na(x))
    else      length(x)
  }

  groupvars <- rlang::syms(groupvars)
  measurevar <- rlang::sym(measurevar)
```

```

datac <- data %>%
  dplyr::group_by(!!!groupvars) %>%
  dplyr::summarise(
    N          = length2(!!measurevar, na.rm = na.rm),
    sd         = sd      (!!measurevar, na.rm = na.rm),
    !!measurevar := mean  (!!measurevar, na.rm = na.rm),
    se         = sd / sqrt(N),
    # Confidence interval multiplier for standard error
    # Calculate t-statistic for confidence interval:
    # e.g., if conf.interval is .95, use .975 (above/below), and use df=N-1
    ci         = se * qt(conf.interval/2 + .5, N - 1)
  ) %>%
  dplyr::ungroup() %>%
  # Rearrange the columns so that sd, se, ci are last
  dplyr::select(seq_len(ncol(.) - 4), ncol(.) - 2, sd, se, ci)

datac
}

```

The following usage example has a 99% confidence interval and handles NAs and missing combinations:

```

# Remove all rows with both c52 and d21
c2 <- filter(cabbages, !(Cult == "c52" & Date == "d21" ))
# Set some values to NA
c2$HeadWt[c(1, 20, 45)] <- NA
summarySE(c2, "HeadWt", c("Cult", "Date"),
  conf.interval = .99, na.rm = TRUE, .drop = FALSE)
#> # A tibble: 5 x 7
#>   Cult Date      N HeadWt    sd    se    ci
#>   <fct> <fct> <int>   <dbl> <dbl> <dbl> <dbl>
#> 1 c39 d16      9   3.26 0.982 0.327  1.10
#> 2 c39 d20      9   2.72 0.139 0.0465 0.156
#> 3 c39 d21     10   2.74 0.983 0.311  1.01
#> 4 c52 d16     10   2.26 0.445 0.141  0.458
#> 5 c52 d20      9   3.04 0.809 0.270  0.905

```

See Also

See [Recipe 7.7](#) to use the values calculated here to add error bars to a graph.

15.19 Converting Data from Wide to Long

Problem

You want to convert a data frame from “wide” format to “long” format.

Solution

Use `gather()` from the `tidyr` package. In the `anthoming` data set, for each `angle`, there are two measurements—one column contains measurements in the experimental condition and the other contains measurements in the control condition:

```
library(gcookbook) # For the data set
anthoming
#>   angle expt ctrl
#> 1   -20    1    0
#> 2   -10    7    3
#> 3    0    2    3
#> 4    10    0    3
#> 5    20    0    1
```

We can reshape the data so that all the measurements are in one column. This will put the values from `expt` and `ctrl` into one column, and put the names into a different column:

```
library(tidyr)
gather(anthoming, condition, count, expt, ctrl)
#>   angle condition count
#> 1   -20      expt     1
#> 2   -10      expt     7
#> ...<6 more rows>...
#> 9    10      ctrl     3
#> 10   20      ctrl     1
```

This data frame represents the same information as the original one, but it is structured in a way that is more conducive to some analyses.

Discussion

In the source data, there are *ID* variables and *value* variables. The ID variables are those that specify which values go together. In the source data, the first row holds measurements for when `angle` is `-20`. In the output data frame, the two measurements, for `expt` and `ctrl`, are no longer in the same row, but we can still tell that they belong together because they have the same value of `angle`.

The value variables are by default all the non-ID variables. The names of these variables are put into a new *key* column, which we called `condition`, and the values are put into a new *value* column, which we called `count`.

You can designate the *value* columns from the source data by naming them individually, as we did with `expt` and `ctrl`. `gather()` automatically inferred that the ID variable was the remaining column, `angle`. Another way to tell it which columns are values is to do the reverse; if you exclude the `angle` column, then `gather()` will infer that the value columns are the remaining ones, `expt` and `ctrl`:

```
gather(anthoming, condition, count, expt, ctrl)
# Prepending the column name with a '-' means it is not a value column
gather(anthoming, condition, count, -angle)
```

There are other convenient shortcuts to specify which columns are values. For example, `expt:ctrl` means to select all columns between `expt` and `ctrl` (in this particular case, there are no other columns in between, but for a larger data set you can imagine how this would save typing).

By default, `gather()` will use all of the columns from the source data as either ID columns or value columns. That means that if you want to ignore some columns, you'll need to filter them out first using the `select()` function.

For example, in the `drunk` data set, suppose we want to convert it to long format, keeping `sex` in one column and putting the numeric values in another column. This time, we want the values for only the `0-29` and `30-39` columns, and we want to discard the values for the other age ranges:

```
# Our source data
drunk
#>      sex 0-29 30-39 40-49 50-59 60+
#> 1  male 185   207   260   180   71
#> 2 female    4    13    10     7   10

# Try gather() with just 0-29 and 30-39
drunk %>%
  gather(age, count, "0-29", "30-39")
#>      sex 40-49 50-59 60+ age count
#> 1  male  260   180   71 0-29  185
#> 2 female   10     7  10 0-29    4
#> 3  male  260   180   71 30-39  207
#> 4 female   10     7  10 30-39   13
```

That doesn't look right! We told `gather()` that `0-29` and `30-39` were the value columns we wanted, and it automatically inferred that we wanted to use all of the other columns as ID columns, when we wanted to just keep `sex` and discard the others. The solution is to use `select()` to remove the unwanted columns first, and then `gather()`:

```
library(dplyr) # For the select() function

drunk %>%
  select(sex, "0-29", "30-39") %>%
  gather(age, count, "0-29", "30-39")
#>      sex age count
#> 1  male 0-29  185
#> 2 female 0-29    4
#> 3  male 30-39  207
#> 4 female 30-39   13
```

There are times where you may want to use more than one column as the ID variables:

```
plum_wide
#>   length      time dead alive
#> 1  long  at_once   84   156
#> 2  long in_spring 156    84
#> 3  short at_once  133   107
#> 4  short in_spring 209    31
# Use length and time as the ID variables (by not naming them as value variables)
gather(plum_wide, "survival", "count", dead, alive)
#>   length      time survival count
#> 1  long  at_once      dead    84
#> 2  long in_spring      dead   156
#> ...<4 more rows>...
#> 7  short at_once      alive   107
#> 8  short in_spring      alive    31
```

Some data sets don't come with a column with an ID variable. For example, in the corneas data set, each row represents one pair of measurements, but there is no ID variable. Without an ID variable, you won't be able to tell how the values are meant to be paired together. In these cases, you can add an ID variable before using `melt()`:

```
# Make a copy of the data
co <- corneas
# Add an ID column
co$id <- 1:nrow(co)

gather(co, "eye", "thickness", affected, notaffected)
#>   id      eye thickness
#> 1  1 affected      488
#> 2  2 affected      478
#> ...<12 more rows>...
#> 15 7 notaffected      464
#> 16 8 notaffected      476
```

Having numeric values for the ID variable may be problematic for subsequent analyses, so you may want to convert `id` to a character vector with `as.character()`, or a factor with `factor()`.

See Also

See [Recipe 15.20](#) to do conversions in the other direction, from long to wide.

See the `stack()` function for another way of converting from wide to long.

15.20 Converting Data from Long to Wide

Problem

You want to convert a data frame from “long” format to “wide” format.

Solution

Use the `spread()` function from the `tidyr` package. In this example, we’ll use the `plum` data set, which is in a long format:

```
library(gcookbook) # For the data set
plum
#>   length      time survival count
#> 1   long  at_once    dead    84
#> 2   long in_spring    dead   156
#> ...<4 more rows>...
#> 7   short at_once    alive   107
#> 8   short in_spring    alive    31
```

The conversion to wide format takes each unique value in one column and uses those values as headers for new columns, then uses another column for source values. For example, we can “move” values in the `survival` column to the top and fill them with values from `count`:

```
library(tidyr)
spread(plum, survival, count)
#>   length      time dead alive
#> 1   long  at_once   84   156
#> 2   long in_spring 156    84
#> 3   short at_once  133   107
#> 4   short in_spring 209    31
```

Discussion

The `spread()` function requires you to specify a *key* column that is used for header names, and a *value* column that is used to fill the values in the output data frame. It’s assumed that you want to use all the other columns as ID variables.

In the preceding example, there are two ID columns, `length` and `time`, one key column, `survival`, and one value column, `count`. What if we want to use two of the columns as keys? Suppose, for example, that we want to use `length` and `survival` as keys. This would leave us with `time` as the ID column.

The way to do this is to combine the `length` and `survival` columns together and put them in a new column, then use that new column as a key:

```

# Create a new column, length_survival, from length and survival.
plum %>%
  unite(length_survival, length, survival)
#>   length_survival   time count
#> 1      long_dead at_once    84
#> 2      long_dead in_spring 156
#> ...<4 more rows>...
#> 7      short_alive at_once  107
#> 8      short_alive in_spring  31

# Now pass it to spread() and use length_survival as a key
plum %>%
  unite(length_survival, length, survival) %>%
  spread(length_survival, count)
#>      time long_alive long_dead short_alive short_dead
#> 1  at_once      156        84        107        133
#> 2 in_spring       84       156         31        209

```

See Also

See [Recipe 15.19](#) to do conversions in the other direction, from wide to long.

See the `unstack()` function for another way of converting from long to wide.

15.21 Converting a Time Series Object to Times and Values

Problem

You have a time series object that you wish to convert to numeric vectors representing the time and values at each time.

Solution

Use the `time()` function to get the time for each observation, then convert the times and values to numeric vectors with `as.numeric()`:

```

# Look at nhtemp Time Series object
nhtemp
#> Time Series:
#> Start = 1912
#> End = 1971
#> ...
#> [43] 52.0 52.0 50.9 52.6 50.2 52.6 51.6 51.9 50.5 50.9 51.7 51.4 51.7 50.8
#> [57] 51.9 51.8 51.9 53.0

# Get times for each observation
as.numeric(time(nhtemp))
#> [1] 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925
#> [15] 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939
#> [29] 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953

```

```
#> [43] 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967
#> [57] 1968 1969 1970 1971

# Get value of each observation
as.numeric(nhtemp)
#> [1] 49.9 52.3 49.4 51.1 49.4 47.9 49.8 50.9 49.3 51.9 50.8 49.6 49.3 50.6
#> [15] 48.4 50.7 50.9 50.6 51.5 52.8 51.8 51.1 49.8 50.2 50.4 51.6 51.8 50.9
#> [29] 48.8 51.7 51.0 50.6 51.7 51.5 52.1 51.3 51.0 54.0 51.4 52.7 53.1 54.6
#> [43] 52.0 52.0 50.9 52.6 50.2 52.6 51.6 51.9 50.5 50.9 51.7 51.4 51.7 50.8
#> [57] 51.9 51.8 51.9 53.0
# Put them in a data frame
nht <- data.frame(year = as.numeric(time(nhtemp)), temp = as.numeric(nhtemp))
nht
#>   year temp
#> 1 1912 49.9
#> 2 1913 52.3
#> ...<56 more rows>...
#> 59 1970 51.9
#> 60 1971 53.0
```

Discussion

Time series objects efficiently store information when there are observations at regular time intervals, but for use with ggplot, they need to be converted to a format that separately represents times and values for each observation.

Some time series objects are cyclical. The `presidents` data set, for example, contains four observations per year, one for each quarter:

```
presidents
#>      Qtr1 Qtr2 Qtr3 Qtr4
#> 1945   NA  87   82   75
#> 1946   63   50   43   32
#> ...
#> 1973   68   44   40   27
#> 1974   28   25   24   24
```

To convert it to a two-column data frame with one column representing the year with fractional values, we can do the same as before:

```
pres_rating <- data.frame(
  year = as.numeric(time(presidents)),
  rating = as.numeric(presidents)
)
pres_rating
#>      year rating
#> 1 1945.00     NA
#> 2 1945.25     87
#> ...<116 more rows>...
#> 119 1974.50     24
#> 120 1974.75     24
```

It is also possible to store the year and quarter in separate columns, which may be useful in some visualizations:

```
pres_rating2 <- data.frame(
  year = as.numeric(floor(time(presidents))),
  quarter = as.numeric(cycle(presidents)),
  rating = as.numeric(presidents)
)
pres_rating2
#>   year quarter rating
#> 1  1945      1    NA
#> 2  1945      2    87
#> ...<116 more rows>...
#> 119 1974      3    24
#> 120 1974      4    24
```

See Also

The zoo package is also useful for working with time series objects.

Understanding ggplot2

Most of the recipes in this book involve the `ggplot2` package, which was originally created by Hadley Wickham. It is not a part of “base” R, but it has attracted many users in the R community because of its versatility, clear and consistent interface, and beautiful output.

`ggplot2` takes a different approach to graphics than other plotting packages in R. It gets its name from Leland Wilkinson’s *grammar of graphics*, which provides a formal, structured perspective on how to describe data graphics.

Even though this book deals largely with `ggplot2`, I don’t mean to say that it’s the be-all and end-all of graphics in R. For example, I sometimes find it faster and easier to inspect and explore data with R’s base graphics, especially when the data isn’t already structured properly for use with `ggplot2`. There are some things that `ggplot2` can’t do, or can’t do as well as other plotting packages. There are other things that `ggplot2` can do, but that specialized packages are better suited to handling. For most purposes, though, I believe that `ggplot2` gives the best return on time invested, and it provides beautiful, publication-ready results.

Another excellent package for general-purpose plots is `lattice`, by Deepayan Sarkar, which is an implementation of *trellis* graphics. It is included as part of the base installation of R.

If you want a deeper understanding of `ggplot2`, read on!

Background

In a data graphic, there is a mapping (or correspondence) from properties of the data to visual properties in the graphic. The data properties are typically numerical or categorical values, while the visual properties include the x and y positions of points,

colors of lines, heights of bars, and so on. A data visualization that didn't map the data to visual properties wouldn't be a data visualization. On the surface, representing a number with an x coordinate may seem very different from representing a number with a color of a point, but at an abstract level, they are the same. Everyone who has made data graphics has at least an implicit understanding of this. For most of us, that's where our understanding remains.

In the grammar of graphics, this deep similarity is not just recognized, but made central. In R's base graphics functions, each mapping of data properties to visual properties is its own special case, and changing the mappings may require restructuring your data, issuing completely different plotting commands, or both.

To illustrate, I'll show a graph made from the `simplifiedat` data set from the `gcookbook` package:

```
# Install gcookbook if you don't already have it installed.  
# install.packages("gcookbook")  
  
library(gcookbook) # Load gcookbook for the simplifiedat data set  
simplifiedat  
#>      A1 A2 A3  
#> B1 10  7 12  
#> B2  9 11  6
```

The following will make a simple grouped bar plot, with the As going along the x-axis and the bars grouped by the Bs (Figure A-1):

```
barplot(simplifiedat, beside = TRUE)
```

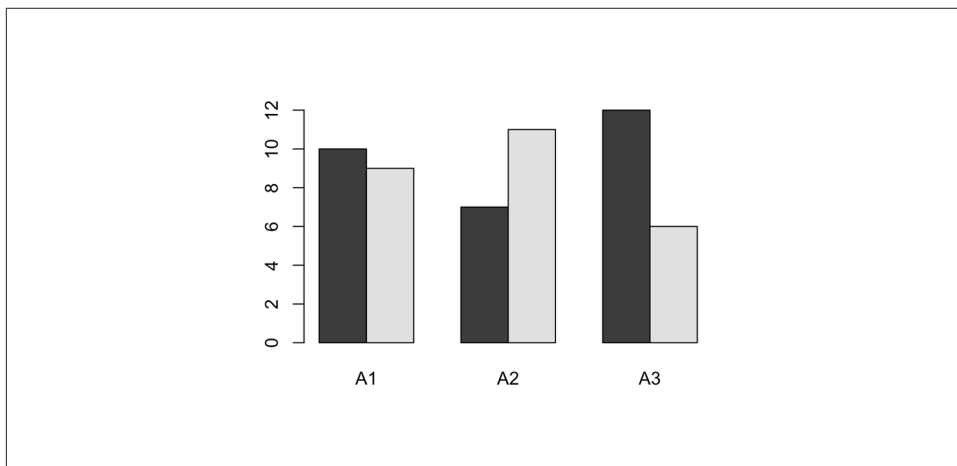


Figure A-1. A bar plot made with `barplot()`

One thing we might want to do is switch things up so the Bs go along the x-axis and the As are used for grouping. To do this, we need to restructure the data by transposing the matrix:

```
t(simpledat)
#>   B1 B2
#> A1 10  9
#> A2  7 11
#> A3 12  6
```

With the restructured data, we can create the plot the same way as before (Figure A-2):

```
barplot(t(simpledat), beside=TRUE)
```

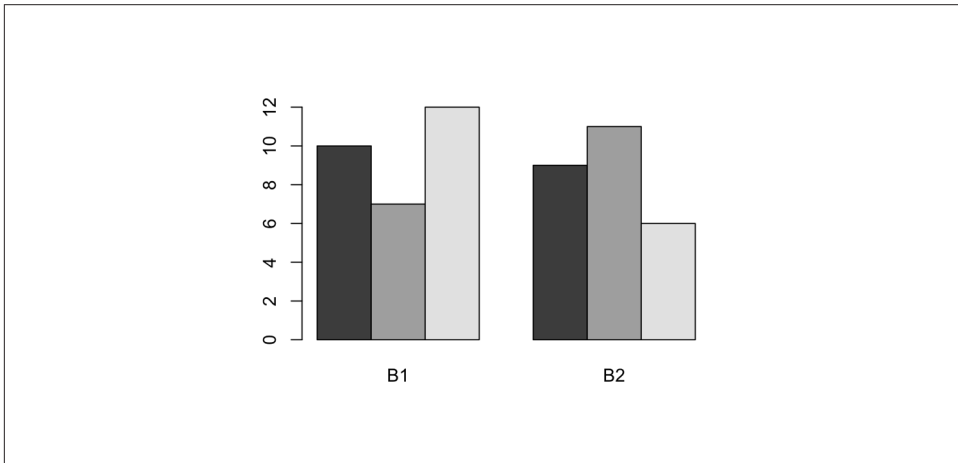


Figure A-2. A bar plot with transposed data

Another thing we might want to do is to represent the data with lines instead of bars, as shown in Figure A-3. To do this with base graphics, we need to use a completely different set of commands. First, we call `plot()`, which tells R to create a new plot and draw a line for one row of data. Then we tell it to draw a second row with `lines()`:

```
plot(simpledat[1,], type="l")
lines(simpledat[2,], type="l", col="blue")
```

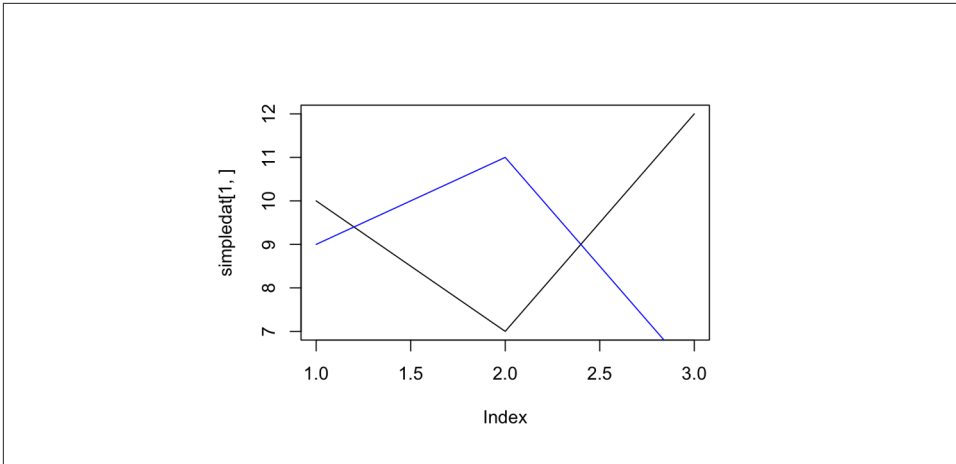


Figure A-3. A line graph made with `plot()` and `lines()`

The resulting plot has a few quirks. The second (blue) line runs below the visible range, because the y range was set only for the first line, when the `plot()` function was called. Additionally, the x -axis is numbered instead of categorical.

Now let's take a look at the corresponding code and plots with `ggplot2`. With `ggplot2`, the structure of the data is always the same: it requires a data frame in “long” format, as opposed to the “wide” format used previously. When the data is in long format, each row represents one item. Instead of having their groups determined by their *positions* in the matrix, the items have their groups specified in a separate column. Here is `simplifiedat`, converted to long format:

```
simplifiedat_long
#>   Aval Bval value
#> 1  A1  B1    10
#> 2  A1  B2     9
#> 3  A2  B1     7
#> 4  A2  B2    11
#> 5  A3  B1    12
#> 6  A3  B2     6
```

This represents the same information, but with a different structure. Another term for it is *tidy data*, where each row represents one observation. There are advantages and disadvantages to this format, but on the whole, it makes things simpler when dealing with complicated data sets. See Recipes 15.19 and 15.20 for information about converting between wide and long data formats.

To make the first grouped bar plot (Figure A-4), we first have to load the `ggplot2` package. Then we tell it to map `Aval` to the x position, with `x = Aval`, and `Bval` to the fill color, with `fill = Bval`. This will make the `As` run along the x -axis and the `Bs` determine the grouping. We also tell it to map `value` to the y position, or height, of the

bars, with `y = value`. Finally, we tell it to draw bars with `geom_col()` (don't worry about the other details yet; we'll get to those later):

```
library(ggplot2)
ggplot(simpledat_long, aes(x = Aval, y = value, fill = Bval)) +
  geom_col(position = "dodge")
```

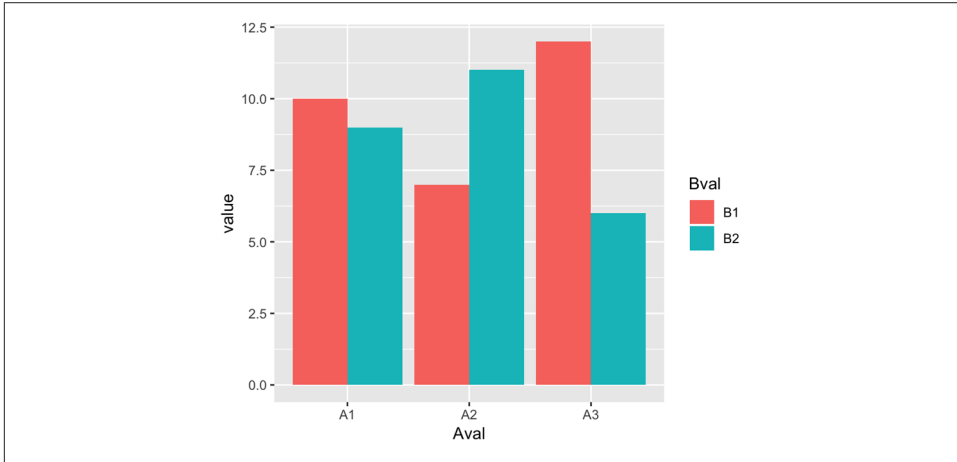


Figure A-4. A bar graph made with `ggplot()` and `geom_col()`

To switch things so that the Bs go along the x-axis and the As determine the grouping (Figure A-5), we simply swap the mapping specification, with `x = Bval` and `fill = Aval`. Unlike with base graphics, we don't have to change the data; we just change the commands for making the plot:

```
ggplot(simpledat_long, aes(x = Bval, y = value, fill = Aval)) +
  geom_col(position = "dodge")
```

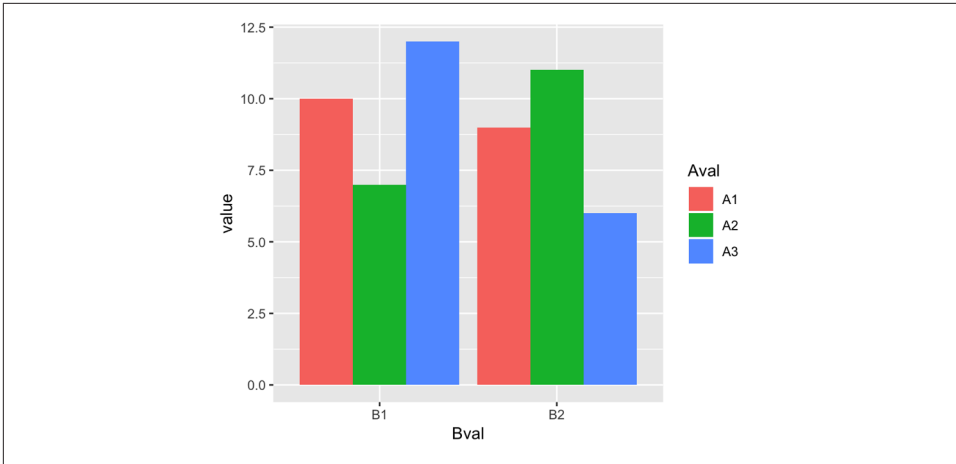


Figure A-5. Bar plot of the same data, but with x and fill mappings switched



You may have noticed that with ggplot2, components of the plot are combined with the + operator. You can gradually build up a ggplot object by adding components to it. Then, when you're all done, you can tell it to print.

To change it to a line plot (Figure A-6), we'll change `geom_col()` to `geom_line()`. We'll also map Bval to the line color, with `colour` (note the British spelling—the author of ggplot2 is a Kiwi), instead of the fill color. Again, don't worry about the other details yet:

```
ggplot(simpledat_long, aes(x = Aval, y = value, colour = Bval, group = Bval)) +  
  geom_line()
```

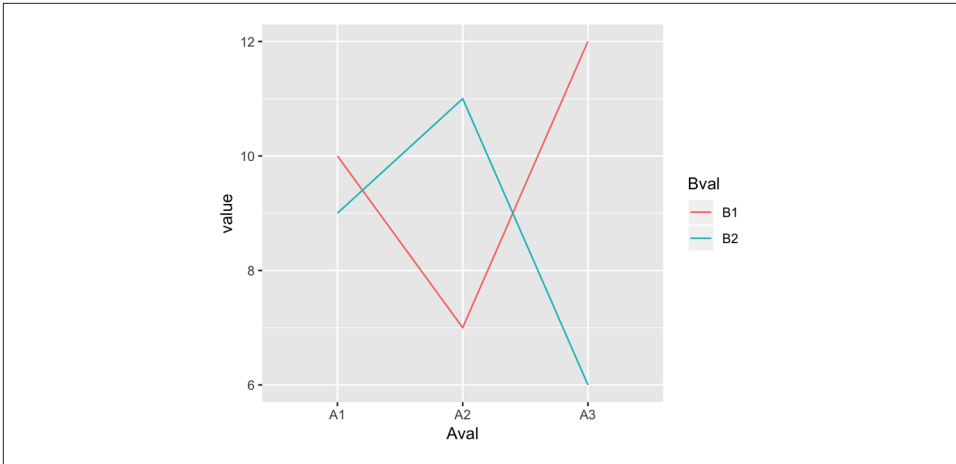


Figure A-6. A line graph made with `ggplot()` and `geom_line()`

With base graphics, we had to use completely different commands to make a line plot instead of a bar plot. With `ggplot2`, we just changed the *geom* from bars to lines. The resulting plot also has important differences from the base graphics version: the *y* range is automatically adjusted to fit all the data because all the lines are drawn together instead of one at a time, and the *x*-axis remains categorical instead of being converted to a numeric axis. The `ggplot2` plots also have automatically generated legends.

Some Terminology and Theory

Before we go any further, it'll be helpful to define some of the terminology used in `ggplot2`:

- The *data* is what we want to visualize. It consists of *variables*, which are stored as columns in a data frame.
- *Geoms* are the geometric objects that are drawn to represent the data, such as bars, lines, and points.
- Aesthetic attributes, or *aesthetics*, are visual properties of geoms, such as *x* and *y* position, line color, point shapes, etc.
- There are *mappings* from data values to aesthetics.
- *Scales* control the mapping from the values in the data space to values in the aesthetic space. A continuous *y* scale maps larger numerical values to vertically higher positions in space.

- *Guides* show the viewer how to map the visual properties back to the data space. The most commonly used guides are the tick marks and labels on an axis.

Here's an example of how a typical mapping works. You have *data*, which is a set of numerical or categorical values. You have *geoms* to represent each observation. You have an *aesthetic*, such as *y* (vertical) position. And you have a *scale*, which defines the mapping from the data space (numeric values) to the aesthetic space (vertical position). A typical linear *y* scale might map the value 0 to the baseline of the graph, 5 to the middle, and 10 to the top. A logarithmic *y* scale would place them differently.

These aren't the only kinds of data and aesthetic spaces possible. In the abstract grammar of graphics, the data and aesthetics could be anything; in the *ggplot2* implementation, there are some predetermined types of data and aesthetics. Commonly used data types include numeric values, categorical values, and text strings. Some commonly used aesthetics include horizontal and vertical position, color, size, and shape.

To interpret the plot, viewers refer to the *guides*. An example of a guide is the *y*-axis, including the tick marks and labels. The viewer refers to this guide to interpret what it means when a point is in the middle of the scale. A *legend* is another type of guide. A legend might show people what it means for a point to be a circle or a triangle, or what it means for a line to be blue or red.

Some aesthetics can only work with categorical variables, such as the shape of a point: triangles, circles, squares, etc. Some aesthetics work with categorical or continuous variables, such as *x* (horizontal) position. For a bar graph, the variable must be categorical—it would make no sense for there to be a continuous variable on the *x*-axis. For a scatter plot, the variable must be numeric. Both of these types of data (categorical and numeric) can be mapped to the aesthetic space of the *x* position, but they require different types of scales.



In *ggplot2* terminology, categorical variables are called *discrete*, and numeric variables are called *continuous*. These terms may not always correspond to how they're used elsewhere. Sometimes a variable that is continuous in the *ggplot2* sense is discrete in the ordinary sense. For example, the number of visible sunspots must be an integer, so it's numeric (*continuous* to *ggplot2*) and discrete (in ordinary language).

Building a Simple Plot

ggplot2 has a simple requirement for data structures: they must be stored in data frames, and each type of variable that is mapped to an aesthetic must be stored in its own column. In the *simplifiedat* examples we looked at earlier, we first mapped one

variable to the `x` aesthetic and another to the `fill` aesthetic; then we changed the mapping specification to change which variable was mapped to which aesthetic.

We'll walk through a simple example here. First, we'll make a data frame of some sample data:

```
dat <- data.frame(  
  xval = 1:4,  
  yval=c(3, 5, 6, 9),  
  group=c("A", "B", "A", "B")  
)
```

```
dat  
#>   xval yval group  
#> 1     1     3     A  
#> 2     2     5     B  
#> 3     3     6     A  
#> 4     4     9     B
```

A basic `ggplot()` specification looks like this:

```
ggplot(dat, aes(x = xval, y = yval))
```

This creates a `ggplot` object using the data frame `dat`. It also specifies default *aesthetic mappings* within `aes()`:

- `x = xval` maps the column `xval` to the x position.
- `y = yval` maps the column `yval` to the y position.

After we've given `ggplot` the data frame and the aesthetic mappings, there's one more critical component: we need to tell it what *geometric objects* to add. At this point, `ggplot2` doesn't know if we want bars, lines, points, or something else to be drawn on the graph. We'll add `geom_point()` to draw points, resulting in a scatter plot (Figure A-7):

```
ggplot(dat, aes(x = xval, y = yval)) +  
  geom_point()
```

If you're going to reuse some of these components, you can store them in variables. We can save the `ggplot` object in `p`, and then add `geom_point()` to it. This has the same effect as the preceding code:

```
p <- ggplot(dat, aes(x = xval, y = yval))  
  
p +  
  geom_point()
```

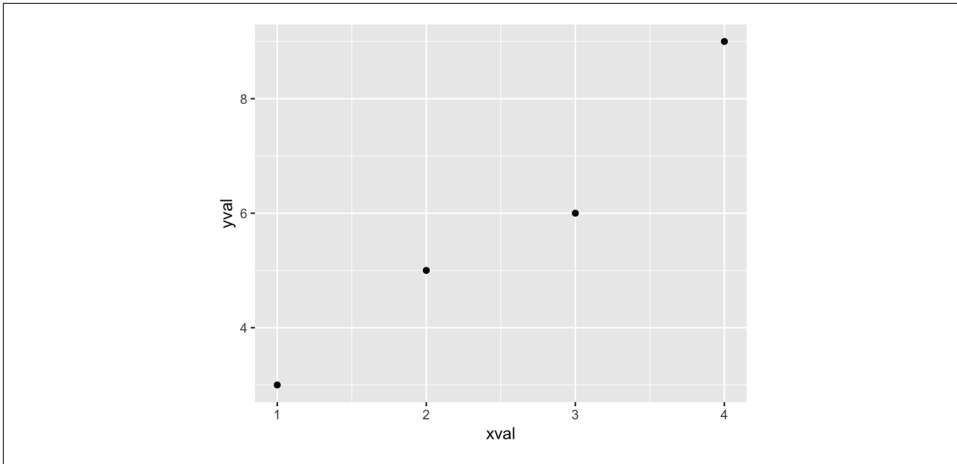


Figure A-7. A basic scatter plot

We can also map the variable `group` to the color of the points, by putting `aes()` inside the call to `geom_point()`, and specifying `colour = group` (Figure A-8):

```
p +  
  geom_point(aes(colour = group))
```

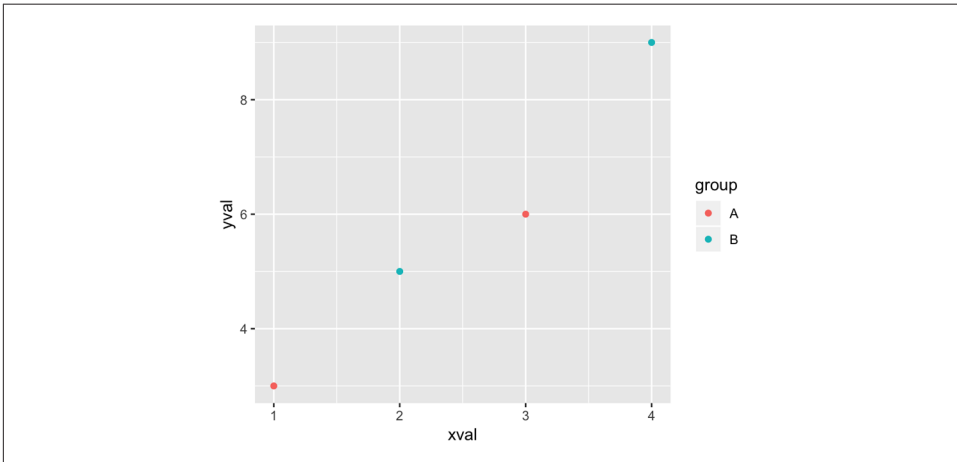


Figure A-8. A scatter plot with a variable mapped to colour

This doesn't alter the *default* aesthetic mappings that we defined previously, inside of `ggplot(...)`. What it does is add an aesthetic mapping for this particular geom, `geom_point()`. If we added other geoms, this mapping would not apply to them.

Contrast this aesthetic *mapping* with aesthetic *setting*. This time, we won't use `aes()`; we'll just set the value of `colour` directly (Figure A-9):

```
p +  
  geom_point(colour = "blue")
```

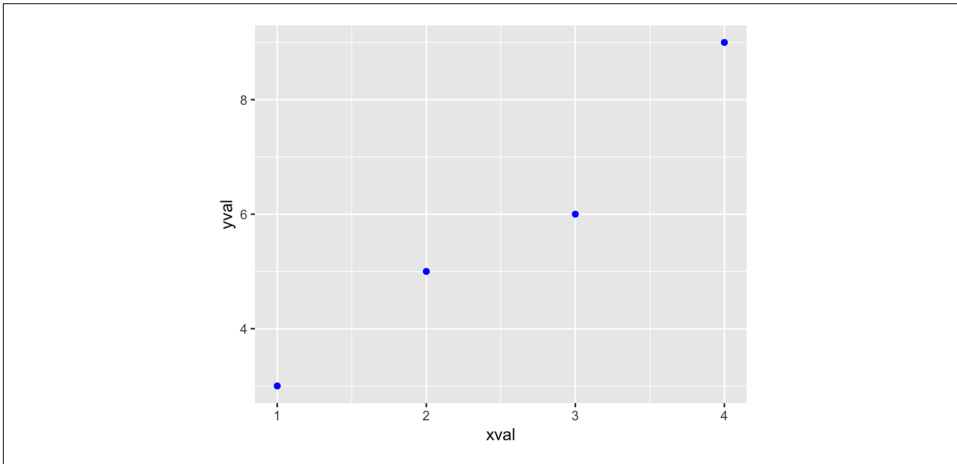


Figure A-9. A scatter plot with colors set instead of mapped

We can also modify the scales; that is, the mappings from data to visual attributes. Here, we'll change the x scale so that it has a larger range (Figure A-10):

```
p +  
  geom_point() +  
  scale_x_continuous(limits = c(0, 8))
```

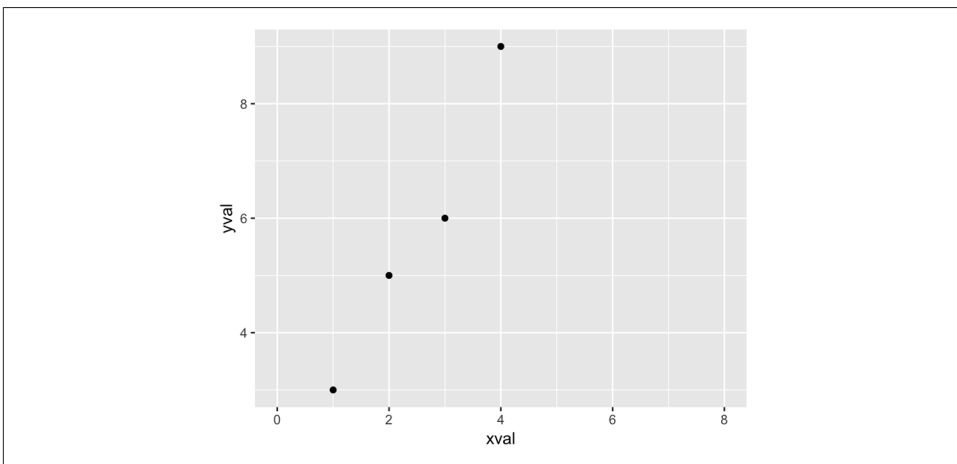


Figure A-10. A scatter plot with increased x range

If we go back to the example with the `colour = group` mapping, we can also modify the color scale (Figure A-11):

```
p +
  geom_point(aes(colour = group)) +
  scale_colour_manual(values = c("orange", "forestgreen"))
```

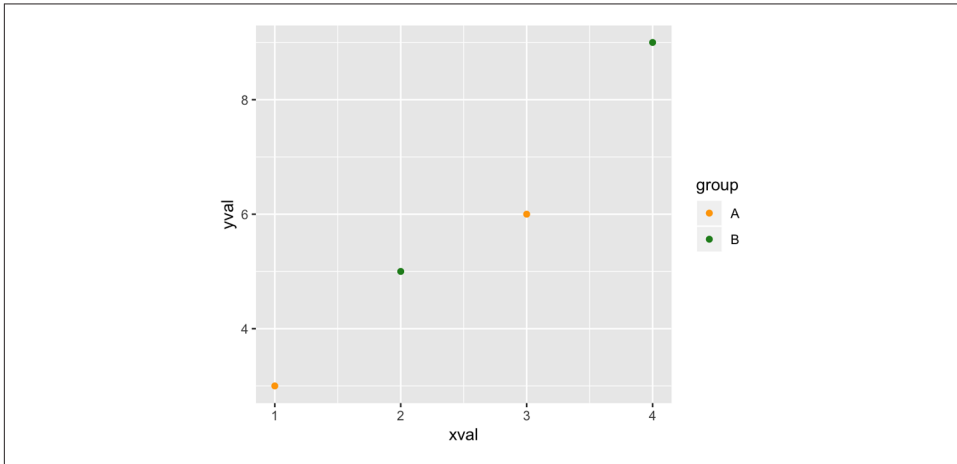


Figure A-11. A scatter plot with modified colors and a different palette

Both times when we modified the scale, the *guide* also changed. With the *x* scale, the guide was the markings along the x-axis. With the color scale, the guide was the legend.

Notice that we've used `+` to join together the pieces. In this last example, we ended a line with `+`, then added more on the next line. If you are going to have multiple lines, you have to put the `+` at the end of each line, instead of at the beginning of the next line. Otherwise, R's parser won't know that there's more stuff coming; it'll think you've finished the expression and evaluate it.

Printing

In R's base graphics, the graphing functions tell R to draw plots to the output device (the screen or a file). `ggplot2` is a little different. The commands don't directly draw to the output device. Instead, the functions build plot *objects*, and the plots aren't drawn until you use the `print()` function, as in `print(object)`. You might be thinking, "But wait, I haven't told R to print anything, yet it's made these plots!" Well, that's not exactly true. In R, when you issue a command at the prompt, it really does two things: first it runs the command, then it calls `print()` with the result returned from that command.

The behavior at the interactive R prompt is different from when you run a script or function. In scripts, commands aren't automatically printed. The same is true for functions, but with a slight catch: the result of the last command in a function is

returned, so if you call the function from the R prompt, the result of that last command will be printed because it's the result of the function.



Some introductions to `ggplot2` make use of a function called `qplot()`, which is intended as a convenient interface for making graphs. It does require a little less typing than using `ggplot()` plus a geom, but I've found it a bit confusing to use because it has a different way of specifying certain parameters. It's simpler to just use `ggplot()`.

Stats

Sometimes your data must be transformed or summarized before it is mapped to an aesthetic. This is true, for example, with a histogram, where the samples are grouped into bins and counted. The counts for each bin are then used to specify the height of a bar. Some geoms, like `geom_histogram()`, automatically do this for you, but sometimes you'll want to do this yourself, using various `stat_xx` functions.

Themes

Some aspects of a plot's appearance fall outside the scope of the grammar of graphics. These include the color of the background and grid lines in the plotting area, the fonts used in the axis labels, and the text in the plot title. These are controlled with the `theme()` function, explored in [Chapter 9](#).

End

Hopefully you now have an understanding of the concepts behind `ggplot2`. The rest of this book shows you how to use it!

Symbols

%+% (add data frame operator), 216
+ (addition operator), for line continuation, 15, 414
& (and operator), 380
: (colon) sequence operator, 191
\$ (dollar sign operator), 384
| (or operator), 380
%>% (pipe operator), 7-9, 370, 388
[] (square brackets), indexing character vectors, 378, 380

A

add data frame operator (%+%), 216
Adobe Illustrator, 348-349
aes() function, 270-272, 285, 411-413
aesthetics
 about, 403, 409-410
 mapping in aes() function, 33, 44, 270-272, 285, 411-413
 setting outside aes() function, 33, 269-270, 412
American or British spellings of functions, 26
and operator (&), 380
annotate() function, 107-109, 111-117, 161-166, 169-170, 171, 225, 226-227
annotations
 for facets, 177-180
 mathematical expressions in, 107-109, 164
 text, 106-107, 161-164
annotation_logticks() function, 209-211
approx() function, 286
arrange() function, 46
arrows on line segments, 170

as.character() function, 379
as.data.frame() function, 363
as.factor() function, 120, 138
as_data_frame() function, 363
as_tibble() function, 363
axes
 circular, 211-216
 dates on, 216-220
 labels for, appearance of, 201-202
 labels for, removing, 199-200
 labels for, text of, 198-199
 lines along, 202-204
 logarithmic, 204-211
 range for, setting, 183-185
 reversing direction of, 182, 186-187, 187-189
 scaling ratio for, 189
 swapping x- and y-axes, 181-183
 tick labels, appearance of, 196-198
 tick labels, format of, 194-196
 tick labels, removing, 192-193
 tick labels, text of, 193-194
 tick marks, logarithmic, 209-211
 tick marks, removing, 192-193
 tick marks, setting, 191-192
 times on, 220-222

B

balloon plot, 118-120
bar graph
 about, 23
 color of bars, 25, 27, 31-35, 39, 41
 continuous x variable, 16, 24
 of counts, 16, 17, 29-31, 44

- creating, 15-18, 23-26
- discrete x variable, 16, 24
- error bars in, 173-176
- grouping bars, 26-29, 36, 45, 406
- labels for, 43-48
- legend for, removing, 34
- legend for, reversing, 38
- negative and positive bars, 33-35
- outline of bars, 25, 27, 33-34, 39, 41
- proportional stacked, 40-43
- spacing between bars, 35-37
- stacked, 37-40, 46
- width of bars, 35-37
- barplot() function, 15, 404
- base R packages, 2, 11, 404-406
- bitmap files
 - fonts in, 355-356
 - outputting to, 350-352
 - saving rgl plots to, 312
 - size of, 90
- box plot, 93
 - creating, 19-21, 142-145
 - mean markers in, 146-147
 - notches in, 145-146
 - overlying on dot plot, 155-156
 - overplotting in, 144
 - of single group, 145
 - whiskers in, 143
 - width of boxes, 144
- boxplot() function, 20, 145
- British or American spellings of functions, 26

C

- Cairo package, 352
- CairoPNG() function, 352
- categorical variables (see discrete variables)
- cetcolor scales, 274
- character vectors
 - converting factors to, 116
 - converting to factors, 120
 - default ordering of, 50, 362
 - renaming, 377-378
- choropleth map, 334-339
- circular plot, 211-216
- Cleveland dot plot, 49-54
- closures, 296
- cluster analysis (see dendrogram)
- colon (:) sequence operator, 191
- color
 - of bars in bar graph, 25, 27, 31-35, 39, 41
 - of bars in histogram, 129
 - gradient scales for, 272, 283-285
 - of graph elements, 269-272
 - highlighting items, 171-173
 - of lines in line graph, 64-66
 - manually-defined palette for, 280-285
 - of map regions, 334-339
 - of points in line graph, 66-68
 - of points in scatter plot, 80
 - of shaded region, 68-73, 285-286
 - variable controlling, 270, 275-285
- color names, 280, 282
- Color Oracle program, 274
- color() function, 282
- colorblind viewers, palette for, 272
- ColorBrewer package, 277
- comma() function, 196
- comma-separated values (CSV) data, 4-5
- complete() function, 391
- confidence region, 73-75, 95, 392-395
- contact information for this book, xii
- contingency table, mosaic plot for, 324-328
- continuous variables
 - about, 410
 - axis tick marks for, 191
 - bar graph using, 16, 24
 - Cleveland dot plots using, 50
 - color palette for, choosing, 283-285
 - line graph using, 55, 56, 61
 - range of axis, setting, 183-185
 - recoding to categorical variable, 381-382
 - reversing direction of axes, 186-187
 - scatter plot using, 84-88
 - viridis palette for, 273
- conventions used in this book, xi
- coord_fixed() function, 189
- coord_flip() function, 181-183
- coord_map() function, 330
- coord_polar() function, 211-216
- corr() function, 289
- correlation matrix, 289-293
- corrplot package, 290
- corrplot() function, 290-293
- CSV (comma-separated values) data, 4-5
- curve() function, 21
- cut() function, 381-382

D

data

- about, [409-410](#)
 - converting time series object to vectors, [400-402](#)
 - loading from delimited text file, [4-5](#)
 - loading from Excel file, [5-6](#)
 - loading from SPSS/SAS/Stata file, [6-7](#)
 - mapping to aesthetics, [33, 44, 270-272, 285, 403, 409-413](#)
 - (see also aesthetics)
 - structure requirements for, [406, 410](#)
 - transforming before mapping, [415](#)
- ### data frames
- about, [43, 361, 409-410](#)
 - character vectors, renaming, [377-378](#)
 - columns, adding, [365-366](#)
 - columns, creating from existing columns, [382-384](#)
 - columns, creating from groups of data, [384](#)
 - columns, deleting, [366-367](#)
 - columns, renaming, [367-368](#)
 - columns, reordering, [368-369](#)
 - converting from long to wide format, [399-400](#)
 - converting from wide to long format, [70, 395-398](#)
 - converting to tibbles, [363](#)
 - creating, [362-363](#)
 - factor levels, removing, [376](#)
 - factor levels, renaming, [374-376](#)
 - factor levels, reordering, [371-374](#)
 - information about, retrieving, [363-365](#)
 - subset of, [369-371](#)
 - summarizing by groups, [387-392](#)
 - summarizing with standard errors, [392-395](#)
- ### data science, [ix](#)
- ### data visualizations, [ix](#)
- (see also plots and graphs)
- ### data.frame() function, [362](#)
- ### data_frame() function, [363](#)
- ### dates, on axes, [216-220](#)
- ### date_format() function, [218-220](#)
- ### delimited text file, [4-5](#)
- ### dendrogram, [314-317](#)
- ### density curve
- creating, [134-138](#)
 - facets in, [139](#)
 - labels in, [140](#)

- multiple, for grouped data, [138-140](#)
 - overlying with histogram, [137, 140](#)
- ### density plot, two-dimensional, [157-160](#)
- ### dev.off() function, [345, 350](#)
- ### discrete (categorical) variables
- about, [410](#)
 - axis tick marks for, [191](#)
 - bar graph using, [16, 24](#)
 - color palette for, choosing, [275-282](#)
 - line graph using, [55, 56, 60-61, 63](#)
 - recoding continuous variables into, [381-382](#)
 - recoding to another variable, [379-381](#)
 - reversing direction of axes, [182, 187-189](#)
 - viridis palette for, [273](#)
- ### do() function, [104-105](#)
- ### do.call() function, [378](#)
- ### dollar sign operator (\$), [384](#)
- ### dollar() function, [196](#)
- ### dot plot
- Cleveland, [49-54](#)
 - multiple, for grouped data, [154-156](#)
 - overlying on box plot, [155-156](#)
 - Wilkinson, creating, [151-154](#)
- ### dplyr package, [x, 1, 361](#)
- ### dput() function, [312](#)
- ### droplevels() function, [377](#)

E

- ### ECDF (empirical cumulative distribution function), [323](#)
- ### element_line() function, [232-235](#)
- ### element_rect() function, [232-235](#)
- ### element_text() function, [197, 201, 225-227, 232-235](#)
- ### empirical distribution
- ECDF graph for, [323](#)
 - QQ plot for, [322](#)
- ### error bars, [173-176](#)
- ### Esri shapefile, map created from, [340-343](#)
- ### everything() function, [368](#)
- ### Excel file, [5-6](#)
- ### expand_limits() function, [57, 185, 339](#)
- ### expression() function, [108](#)
- ### extrafont package, [352-356](#)

F

facets

- about, [259](#)
- annotations for, [177-180](#)

- creating, 259-262
- different axes in, 262-263
- labels for, 264-267
- facet_grid() function, 130-133, 139, 259-262, 265-266
- facet_wrap() function, 259-262, 265-266
- factor levels
 - order of, determining, 281
 - removing, 376
 - renaming, 131, 140, 264, 374-376
 - reordering, 50, 52, 371-374
- factor() function, 17, 24, 56, 372-373
- factors
 - about, 361
 - converting numbers to, 138
 - converting other variables to, 120, 138
 - converting to character vectors, 116
 - default ordering of, 50, 362
- fct_recode() function, 374-376, 378, 379
- fct_revel() function, 372
- fct_reorder() function, 374
- filter() function, 370-371
- first-class functions, 296
- fonts
 - in bitmap files, 351, 355-356
 - in PDF files, 348, 352-355
 - in screen output, 355-356
- forcats package, 379
- foreign package, 6
- frequency polygon, 141-142
 - (see also histogram)
- Fruchterman-Reingold algorithm, 298
- function curve
 - creating, 21-22, 293-295
 - shaded region under, 295-296
- functions, 7
 - (see also specific functions)
 - British or American spellings of, 26
 - chaining, 7-9, 370, 388
 - continuing on multiple lines, 15, 414

G

- gather() function, 396-398
- gcookbook package, x, 1
- geographical map
 - choropleth map, 334-339
 - clean background for, 339
 - creating, 330-334
 - from Esri shapefile, 340-343

- geoms (geometric objects)
 - about, 409-410
 - setting colors in, 269-272
 - specifying, 411
- geom_abline() function, 166-169
- geom_area() function, 69-70
- geom_bar() function, 17, 24, 29-31, 35-37, 44
- geom_boxplot() function, 20, 93, 142-146, 155-156
- geom_col() function, 16, 23-29, 33-34, 38-43, 407
- geom_density() function, 134-140
- geom_dotplot() function, 151-156
- geom_errorbar() function, 173-176
- geom_freqpoly() function, 141-142
- geom_histogram() function, 18, 127-133, 212
- geom_hline() function, 166-169
- geom_label_repel() function, 112
- geom_line() function, 14, 55-58, 60-66, 69, 71, 134-136, 408
- geom_map() function, 338
- geom_path() function, 330-334
- geom_point() function, 13, 49-54, 58-60, 63, 66-68, 77-78, 81-83, 411
- geom_polygon() function, 330-334
- geom_qq() function, 322
- geom_qq_line() function, 322
- geom_raster() function, 303-304
- geom_ribbon() function, 73
- geom_rug() function, 110-111
- geom_segment() function, 52, 318-322
- geom_sf() function, 340-343
- geom_text() function, 43-48, 111-117, 163, 177-180, 226-227
- geom_text_repel() function, 112
- geom_tile() function, 303-304
- geom_violin() function, 147-151
- geom_vline() function, 166-169
- ggplot() function, 12, 23-26, 55-58, 411
- ggplot2 package, x, 1, 11, 403-410
- ggsave() function, 346, 351
- ggtitle() function, 223-225
- gradient color scales, 272, 283-285
- grammar of graphics, 403, 404
- graph() function, 297-300
- graphs (see plots and graphs)
- grid lines
 - controlling, 191, 193
 - hiding, 236-236

grid package, 256
group_by() function, 42, 46, 104-105, 384,
387-392
guides, 410-410, 414
(see also axes; legends)
guides() function, 38, 239-241, 246, 249-250,
256

H

haven package, 6
hclust() function, 314-317
heat map, 302-304
highlighting items, 171-173
hist() function, 18
histogram
 bars, color of, 129
 bins, boundaries of, 129
 bins, number and width of, 128-130
 compared to frequency polygon, 141
 creating, 18-19, 127-130
 facets in, 130-133
 labels in, 131
 multiple, for grouped data, 130-133
 overlying with density curve, 137, 140

I

igraph package, 297
ImageMagick utility, 313
Inkscape, 348-349
install.packages() function, 1-2
interaction() function, 21, 381

K

kernel density curve, 135

L

labels
 for axes, appearance of, 201-202
 for axes, removing, 199-200
 for axes, text of, 198-199
 for axes ticks, appearance of, 196-198
 for axes ticks, format of, 194-196
 for axes ticks, removing, 192-193
 for axes ticks, text of, 193-194
 for bar graphs, 43-48
 for balloon plot circles, 120
 for facets, 264-267
 for scatter plot points, 111-117

 in legend, appearance of, 255
 in legend, changing, 251-255
 in legend, multiline, 256
label_both() function, 265-266
label_parsed() function, 265-266
labs() function, 199, 247-249
lattice package, 259, 403
legend
 background and border for, 243
 labels in, appearance, 255
 labels in, changing, 251-255
 labels in, multiline, 256
 order of items in, 243-245
 position of, 241-243
 removing, 34, 239-241
 reversing order of, 38, 246
 title of, 247-249
 title of, appearance, 249
 title of, removing, 250
levels() function, 281, 375
libraries, 3
 (see also packages)
library() function, 1, 3
line continuation, addition operator (+), 15,
414
line graph, 408
 about, 55
 confidence region in, 73-75
 continuous x variable, 55, 56, 61
 creating, 13-15, 55-58
 discrete x variable, 55-56, 60-61, 63
 error bars in, 173-176
 lines in, appearance of, 64-66
 lines in, multiple, 60
 negative and positive regions, 285-286
 points on, adding, 58-60
 points on, appearance of, 63, 66-68
 proportional stacked areas in, 72-73
 shaded area in, 68-70, 285-286
 stacked areas in, 70-73
lines
 adding to plot, 166-169
 arrows on, 170
 segments, adding to plot, 169-170
lines() function, 14, 405
list, converting to data frame, 363
lm() function, 94, 100-101, 104
LOESS (locally weighted polynomial) curve,
96-99, 101

logarithmic axes, 204-211
logistic regression, 97
LOWESS smoothed line, 123

M

map (see geographical map)
mappings, 403, 409-410
map_data() function, 331
marginal rugs, 109-111
match() function, 380
mathematical expressions in annotations,
107-109, 164
max() function, 390
mean() function, 388
median() function, 374, 390
min() function, 390
missing values, 390-392
mosaic plot, 324-328
mosaic() function, 324-328
movie3d() function, 313
mutate() function, 42, 365-366, 382-384

N

n() function, 389, 393
NA values, 390
names() function, 367
network graph
 creating, 297-300
 labels in, 300-302
numeric x variable (see continuous variable)

O

objects
 geometric (see geoms)
 time series, converting to vectors, 400-402
100% stacked bar graph (see proportional
 stacked bar graph)
online resources
 cetcolor scales, 274
 Color Oracle program, 274
 Dot Plots (Wilkinson), 154
 for this book, xii
 R, x
 RGB color codes, 282
 shapefiles, 343
 Showtext package, 355
 themes, 231
or operator (|), 380

outline of bars, 25, 27, 33-34, 39, 41
output files
 animated .gif files, 313
 bitmap files, 312, 350-352
 editing, 348-349
 PDF vector files, 312, 345-347
 .png files, 313
 PostScript files, 312
 size of, 90
 SVG vector files, 347
 WMF vector files, 347
overplotting, 77, 88-94, 110, 144

P

packages
 about, 1
 installing, x, 1-2
 loading, x, 1, 3
 upgrading, 3
pairs() function, 121-125
par3d() function, 312
parse() function, 108
patchwork package, 357-359
PDF files
 fonts in, 352-355
 outputting to, 345-347
 saving rgl plots to, 312
pdf() function, 345
percent function, 41
percent() function, 196
pie chart, 329
pie() function, 329
pipe operator (%>%), 7-9, 370, 388
platform requirements, x
play3d() function, 313
plot() function, 12-13, 19, 297-302, 405
plot3d() function, 305-308
plots and graphs, ix
 (see also specific plots and graphs)
 aesthetics for (see aesthetics; color)
 annotations for (see annotations)
 axes for (see axes; scales)
 legend for (see legend)
 multiple, combining into one graphic,
 357-359
 objects in (see geoms)
 outputting (see output files)
 overplotting (see overplotting)
 themes for (see themes)

- title for, 223-225
- plot_layout() function, 358
- plus sign
 - %+% (add data frame operator), 216
 - + (addition operator), for line continuation, 15, 414
- PNG bitmap files, 350-352
- png() function, 350
- points on line graph
 - adding, 58-60
 - appearance of, 63, 66-68
- points() function, 14
- polar plot, 211-216
- position_dodge() function, 36-37, 63
- position_jitter() function, 92
- position_stack() function, 39
- PostScript files, saving rgl plots to, 312
- predict() function, 100-101
- predictvals() function, 101-106
- print() function, 346, 350, 414
- printing (see output files)
- proportional stacked area graph, 72-73
- proportional stacked bar graph, 40-43

Q

- qplot() function, 11, 415
- QQ (quantile-quantile) plot, 322
- qt() function, 393

R

- R, ix-x
 - (see also packages)
- random numbers, 298
- read.csv() function, 4
- read_csv() function, 4
- read.dta() function, 7
- read_dta() function, 6
- read_excel() function, 5
- read.octave() function, 7
- read_sas() function, 6
- read_sav() function, 6
- read.spss() function, 7
- read.systat() function, 7
- read.xport() function, 7
- readr package, 4
- readxl package, 5
- recode() function, 377-379
- regression lines, adding to scatter plot, 94-106
- rename() function, 367-368

- reorder() function, 33, 50, 373-374
- rev() function, 46, 372
- RGB color values, 280, 282
- rgl package, 305
- rgl.postscript() function, 312
- rgl.snapshot() function, 312
- Rgraphviz package, 300

S

- scale() function, 315-317
- scales
 - about, 409-410
 - color settings in, 275-285
 - free, 262-263
 - legend settings in, 251-256
 - list of, 240-241, 245
 - scaling ratio for axes, 189
- scales package, 41, 196, 206, 218
- scale_colour_brewer() function, 65, 80, 275
- scale_colour_discrete() function, 275
- scale_colour_gradient() function, 284
- scale_colour_gradient2() function, 284
- scale_colour_gradientn() function, 284
- scale_colour_grey() function, 275, 279
- scale_colour_hue() function, 275
- scale_colour_manual() function, 65, 80, 275, 280-282, 413
- scale_colour_viridis_c() function, 284
- scale_colour_viridis_d() function, 275
- scale_fill_brewer() function, 33, 39, 275
- scale_fill_discrete() function, 240, 243-245, 247, 275
- scale_fill_gradient() function, 86, 91, 284
- scale_fill_gradient2() function, 284, 304
- scale_fill_gradientn() function, 284
- scale_fill_grey() function, 275
- scale_fill_hue() function, 240, 251, 275, 277
- scale_fill_manual() function, 33-34, 83, 172, 275, 281
- scale_fill_viridis_c() function, 273, 284
- scale_fill_viridis_d() function, 273, 275
- scale_shape_manual() function, 80-83
- scale_size_area() function, 86-87, 118-120
- scale_size_continuous() function, 86
- scale_x_continuous() function, 184-185, 207
- scale_x_discrete() function, 182, 187-189, 192
- scale_x_log10() function, 204-208
- scale_x_reverse() function, 186-187

`scale_y_continuous()` function, 41, 184-185, 191, 193-196, 207
`scale_y_discrete()` function, 187-189
`scale_y_log10()` function, 204-208
`scale_y_reverse()` function, 186-187
scatter plot
 annotations for, 106-109
 continuous variable in, mapping, 84-88
 creating, 11-13, 77-78
 fitted lines from existing model(s) in, 99-106
 fitted regression model lines in, 94-99
 lines in, jittering, 110
 marginal rugs in, 109-111
 overplotting in, 77, 88-94, 110
 points in, binning to hexagons, 91
 points in, binning to rectangles, 90
 points in, color of, 80, 84-88
 points in, grouping, 79-81
 points in, jittering, 92
 points in, labeling, 111-117
 points in, semitransparent, 90
 points in, shape of, 78, 80-83
 points in, size of, 78, 84-88
 3D, animating, 313
 3D, creating, 304-308
 3D, prediction surface for, 308-312
 3D, saving to a file, 312
scatter plot matrix, 121-125
`scientific()` function, 196
screen output
 fonts in, 355-356
`sd()` function, 389
`select()` function, 366-371
`seq()` function, 191
sequence operator (:), 191
`sf` package, 340
shaded regions
 below line graph, 68-70, 285-286
 rectangular, 171
shapefile, map created from, 340-343
Showtext package, 355
software requirements, x
`spin3d()` function, 313
`spread()` function, 399-400
SPSS/SAS/Stata file, 6-7
square brackets ([]), indexing character vectors, 378, 380
stacked area graph, 70-73

stacked bar graph, 37-43, 46
standard errors, summarizing data by, 392-395
`stat_bin2d()` function, 91, 159
`stat_binhex()` function, 91
`stat_density2d()` function, 157-160
`stat_ecdf()` function, 323
`stat_function()` function, 22, 293-296
`stat_smooth()` function, 94-99
`stat_summary()` function, 146-147
`stat_xxx` functions, 415
`str()` function, 363-365
strings (see character vectors)
`st_read()` function, 340-343
`summarise()` function, 387-393
`summary()` function, 364
`surface3D()` function, 309-312
SVG vector files, 347
`svg()` function, 347
`svglite` package, 347
`svglite()` function, 347
system requirements, x

T

`table()` function, 16
text, 106
 (see also labels)
 annotations, 106-107, 161-164
 appearance of text geoms, 226-227
 appearance of theme elements, 225-227
`theme()` function, 192, 196, 200-204, 225-227, 232-236, 240, 249, 255-256, 415
themes
 axis settings (see axes)
 creating, 231, 235-236
 default, setting, 231
 facet settings with, 266
 grid lines, hiding, 236-236
 legend settings in, 255
 legend, position of, 241-243
 legend, removing, 240
 list of elements in, 227, 233
 modifying, 231-235
 premade, using, 228-231
 text appearance, 225-227
 title, 223-225
`theme_bw()` function, 228
`theme_classic()` function, 228
`theme_grey()` function, 228
`theme_minimal()` function, 228

theme_set() function, 231
theme_void() function, 229, 339
theoretical distribution
 comparing to empirical distribution, 322
 comparing to observed distribution, 137
3D scatter plot
 animating, 313
 creating, 304-308
 prediction surface for, 308-312
 saving to a file, 312
tibble
 about, 43
 converting data frames to, 363
 creating, 363
tidy data, 406
tidyr package, 396, 399
tidyverse packages, x, 1-2, 361
time series object, converting to numeric vectors, 400-402
time() function, 400-402
times, on axes, 220-222
title of graph, 223-225
trans_format() function, 206
Trellis displays, 259
 (see also facets)
trellis graphics, 403

U

ungroup() function, 389
unit() function, 256
update.packages() function, 3

V

variables
 color controlled with, 270-272, 275-285
 continuous (see continuous variables)
 discrete (categorical) (see discrete (categorical) variables)
vector field, 317-322
vector files
 editing, 348-349
 PDF, 345-347
 size of, 90
 SVG, 347
 WMF, 347
vectors, 361
 (see also character vectors)
violin plot, 147-151
viridis color scales, 272
viridis package, 282

W

Wilkinson dot plot, 151-154
WMF vector files, 347

X

xlab() function, 198-200
xlim() function, 183-185
.xls or .xlsx file, 5-6

Y

ylab() function, 198-200
ylim() function, 57, 183-185, 187

About the Author

Winston Chang is a software engineer at RStudio, where he works on data visualization and software development tools for R. He has a Ph.D. in Psychology from Northwestern University, and created the Cookbook for R website, which contains recipes for common tasks in R.

Colophon

The animal on the cover of *R Graphics Cookbook* is a reindeer (*Rangifer tarandus*), also known as caribou in North America, which is a species of deer native to Arctic and Subarctic regions. Reindeer are ideally designed for life in hostile, cold environments, as their fur, antlers, noses, hooves, and vision have adapted to the low temperatures.

Their fur coat consists of an outer layer of straight, hollow, tubular hairs, which provide insulation from the cold and buoyancy in water, and a woolly undercoat. The coat is such an efficient insulator that when they lay on the snow, the snow does not melt. Reindeer are the only species of deer in which both male and female (and even calves) have antlers, and they have the largest antlers relative to body size among living deer species. Their antlers are shed annually and new antler growth occurs in the spring and summer.

Reindeer hooves adapt to the season: in the summer, when the tundra is soft and wet, the footpads become sponge-like and provide extra traction. In the winter, the pads shrink and tighten, exposing the rim of the hoof, which cuts into the ice and crusted snow to keep the deer from slipping. This also enables them to dig down (an activity known as cratering) through the snow to their favorite food, a lichen known as reindeer moss.

In 2012, researchers at University College London discovered reindeer are the only mammals that can see ultraviolet light. While human vision cuts off at wavelengths around 400 nm, reindeer can see up to 320 nm. This range only covers the part of the spectrum we can see with the help of a black light, but it is still enough to help reindeer see things in the glowing white of the Arctic that they would otherwise miss.

In the Santa Claus tale, Santa Claus's sleigh is pulled by flying reindeer. These were first named in the 1823 poem "A Visit from St. Nicholas," where they are called Dasher, Dancer, Prancer, Vixen, Comet, Cupid, Dunder, and Blixem.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from Shaw's *Zoology*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

The O'Reilly logo is displayed in white, bold, sans-serif capital letters. The background of the entire advertisement is a vibrant red-to-orange gradient, overlaid with several large, semi-transparent, overlapping circles in varying shades of red and orange, creating a dynamic, abstract pattern.

O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning