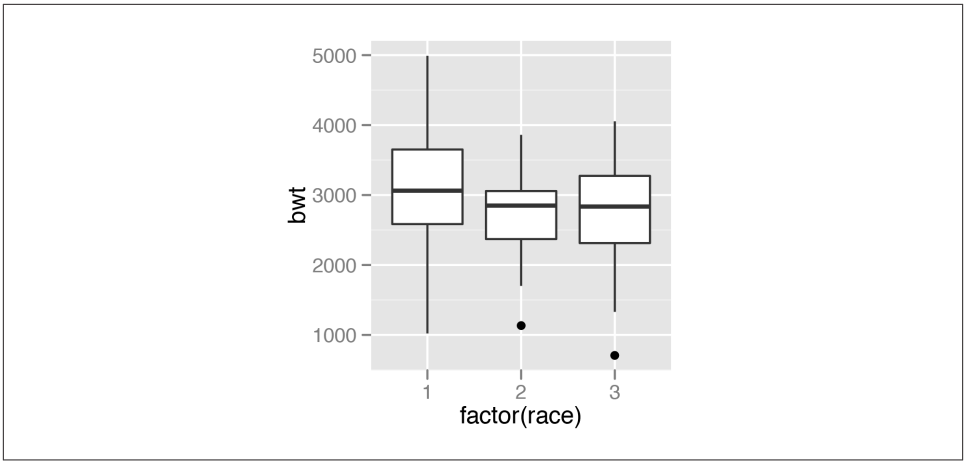

Figure 6-15. A box plot

Discussion

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

```
birthwt

low age lwt race smoke ptl ht ui ftv  bwt
  0  19 182    2     0   0  0  1   0 2523
  0  33 155    3     0   0  0  0   3 2551
  0  20 105    1     1   0  0  0   1 2557
...
```

In Figure 6-15, the data is divided into groups by `race`, and we visualize the distributions of `bwt` for each group. The value of `race` is 1, 2, or 3, but since it's stored as a numeric vector, `ggplot()` doesn't know how to use it as a grouping variable. To make this work, we can modify the data frame by converting `race` to a factor, or tell `ggplot()` to treat it 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 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. If there are any data points that are past the ends of the whiskers, they are considered outliers and displayed with dots. Figure 6-16 shows the relationship between a histogram, a density curve, and a box plot, using a skewed data set.

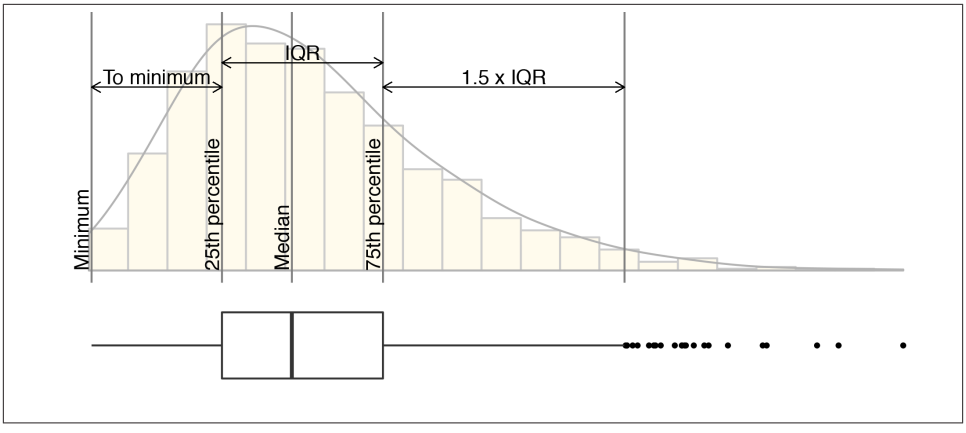

Figure 6-16. Box plot compared to histogram and density curve

To change the width of the boxes, you can set `width` (Figure 6-17, left):

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

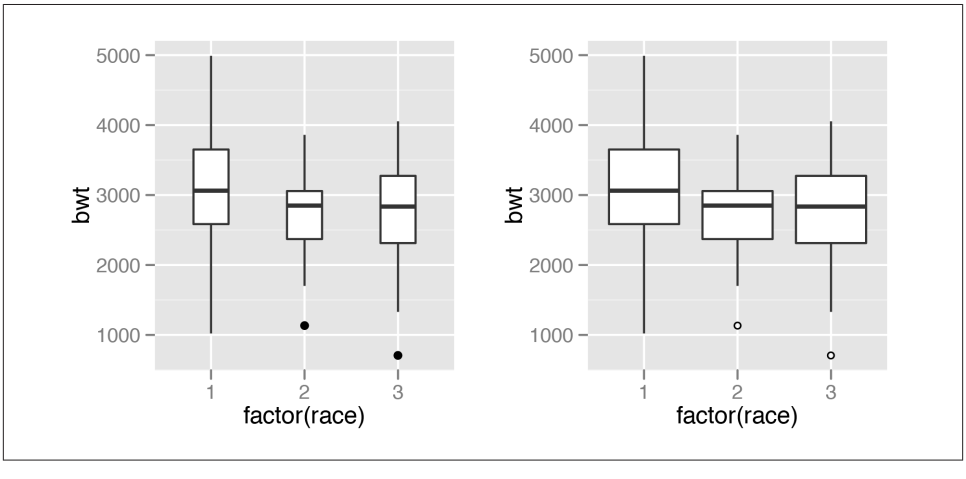

Figure 6-17. Left: box plot with narrower boxes; right: with smaller, hollow outlier points

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 6-17, right):

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

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 6-18):

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

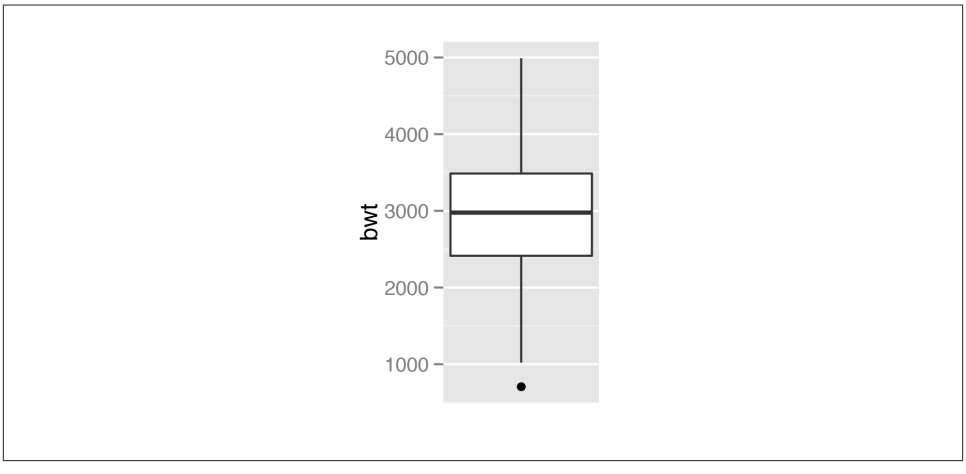

Figure 6-18. Box plot of a single group

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 6-19):

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

ggplot(birthwt, aes(x=factor(race), y=bwt)) + geom_boxplot(notch=TRUE)
```

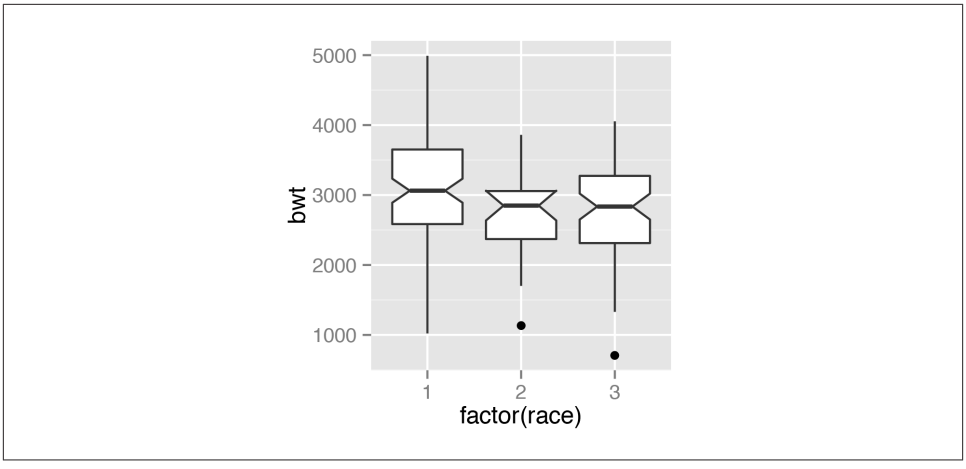

Figure 6-19. A notched box plot

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 6-20):

```
library(MASS) # For the 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")
```

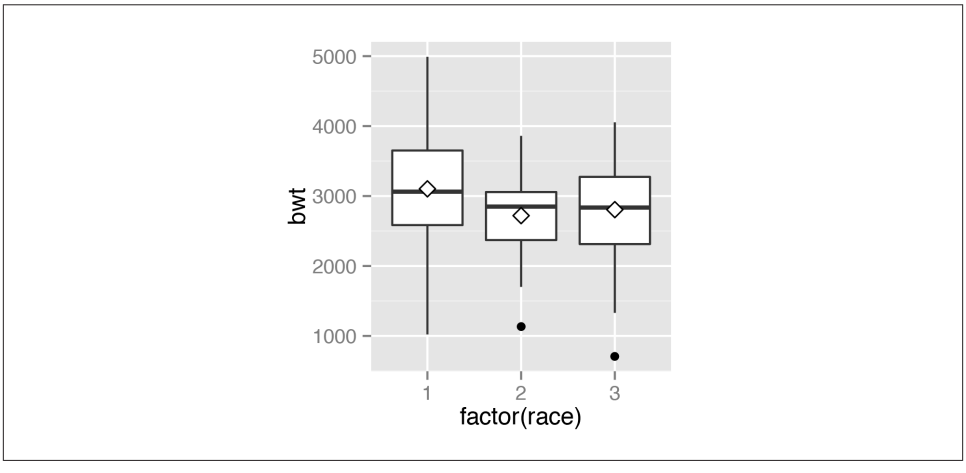

Figure 6-20. Mean markers on a box plot

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 6-21):

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

# Base plot
p <- ggplot(heightweight, aes(x=sex, y=heightIn))

p + geom_violin()
```

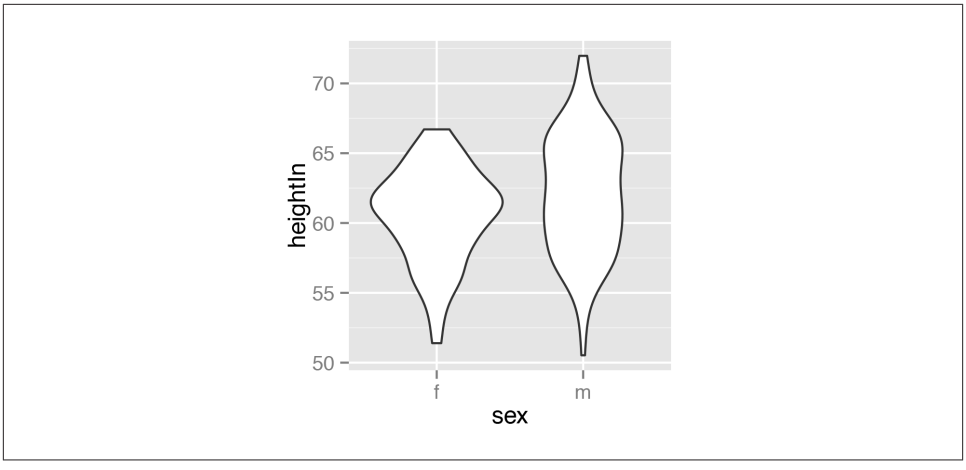

Figure 6-21. A violin plot

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 6-22. Additionally, the box plot outliers are not displayed, which we do by setting `outlier.colour=NA`.:

```
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)
```

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 6-23):

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

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 6-24). In this example, there are slightly fewer females than males, so the `f` violin is slightly narrower:

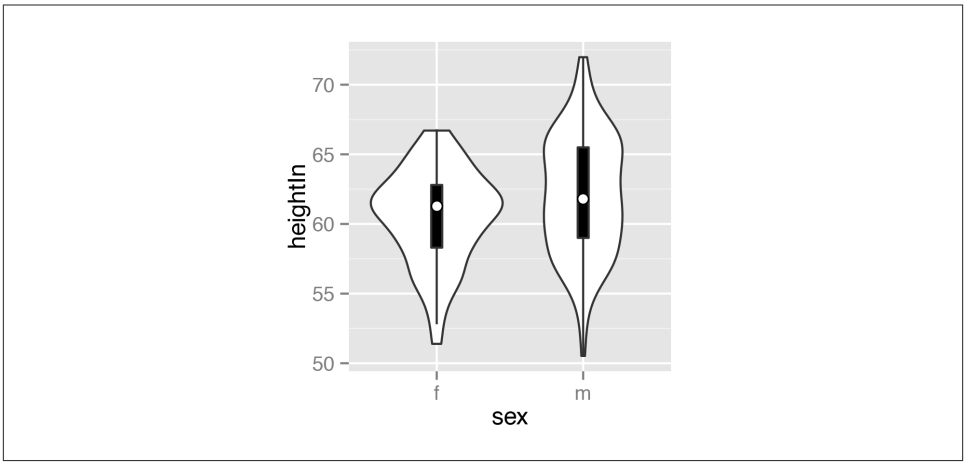

Figure 6-22. A violin plot with box plot overlaid on it

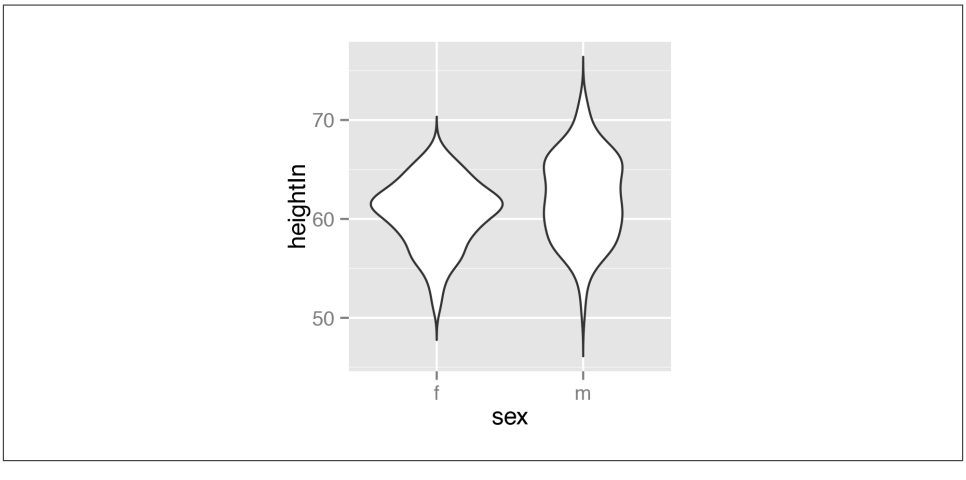

Figure 6-23. A violin plot with tails

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

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 6-25):

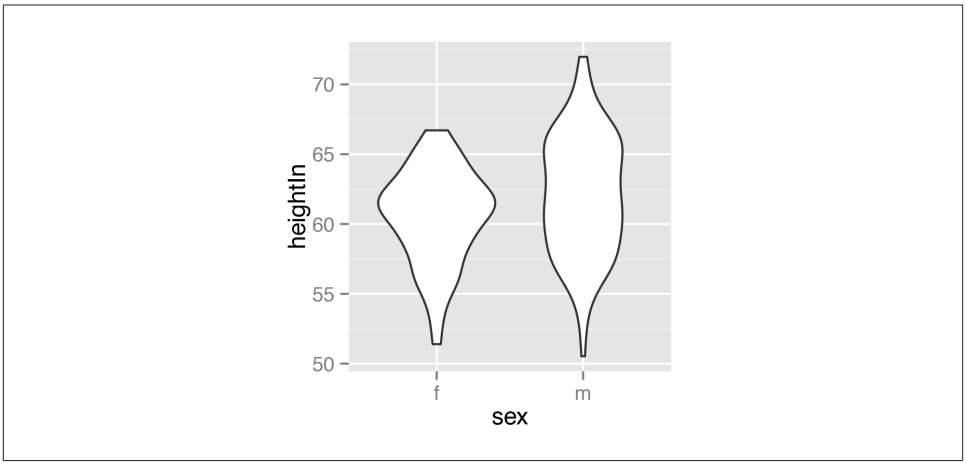

Figure 6-24. Violin plot with area proportional to number of observations

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

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

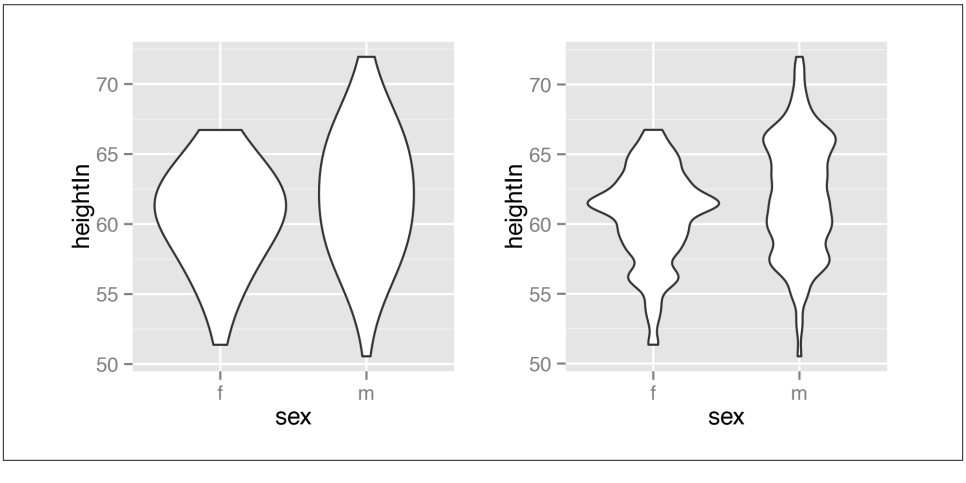

Figure 6-25. Left: violin plot with more smoothing; right: with less smoothing

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 6-26), we'll use a subset of the `countries` data set:

```
library(gcookbook) # For the data set
countries2009 <- subset(countries, Year==2009 & healthexp>2000)

p <- ggplot(countries2009, aes(x=infmortality))

p + geom_dotplot()
```

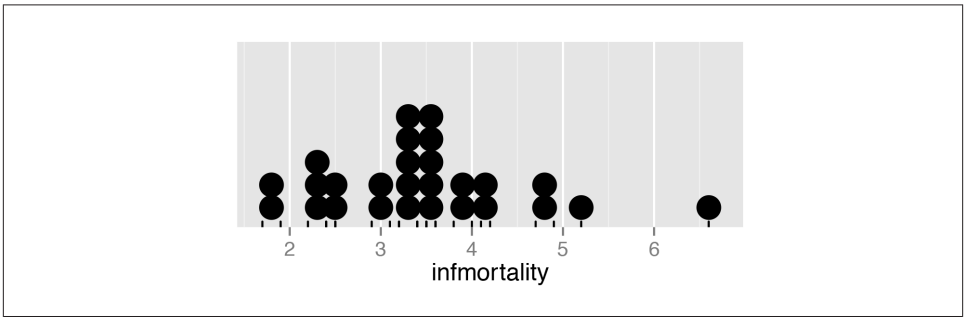

Figure 6-26. A dot plot

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 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 for reasons related 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 6-27):

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

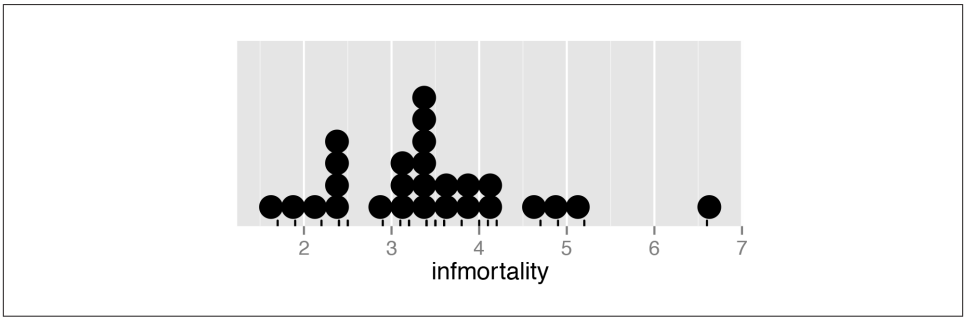

Figure 6-27. Dot plot with no y labels, max bin size of .25, and a rug showing each data point

You may notice that the stacks aren't regularly spaced in the horizontal direction. With the default `dotdensity` 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 6-28, you'll notice that the stacks *aren't* centered above the data:

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

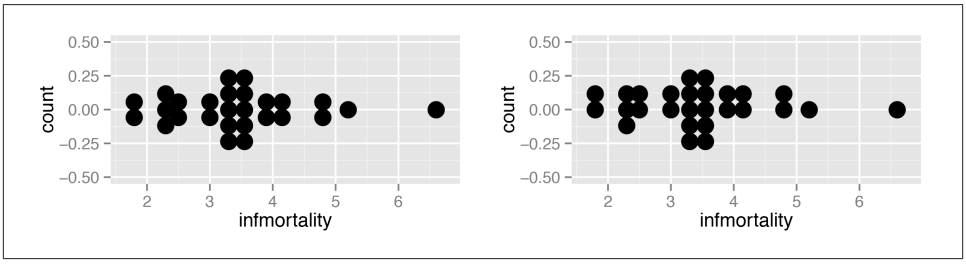

Figure 6-28. Dot plot with histodot (fixed-width) binning

The dots can also be stacked centered, or centered in such a way that stacks with even and odd quantities stay aligned. This can by done by setting `stackdir="center"` or `stackdir="centerwhole"`, as illustrated in Figure 6-29:

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

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

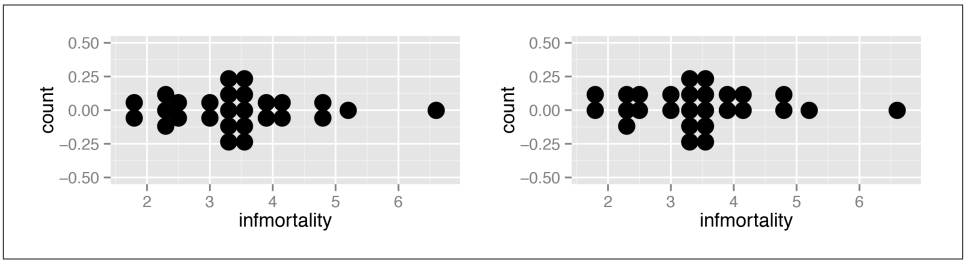

Figure 6-29. Left: dot plot with stackdir="center"; right: with stackdir="centerwhole"

See Also

Leland Wilkinson, "Dot Plots," *The American Statistician* 53 (1999): 276–281, *http://www.cs.uic.edu/~wilkinson/Publications/dots.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 6-30):

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

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

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 be shown as part of the dot plot (Figure 6-31):

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

It's also possible to show the dot plots next to the box plots, as shown in Figure 6-32. This requires using a bit of a hack, by treating the *x* variable as a numeric variable and subtracting or adding a small quantity to shift the box plots and dot plots left and right.

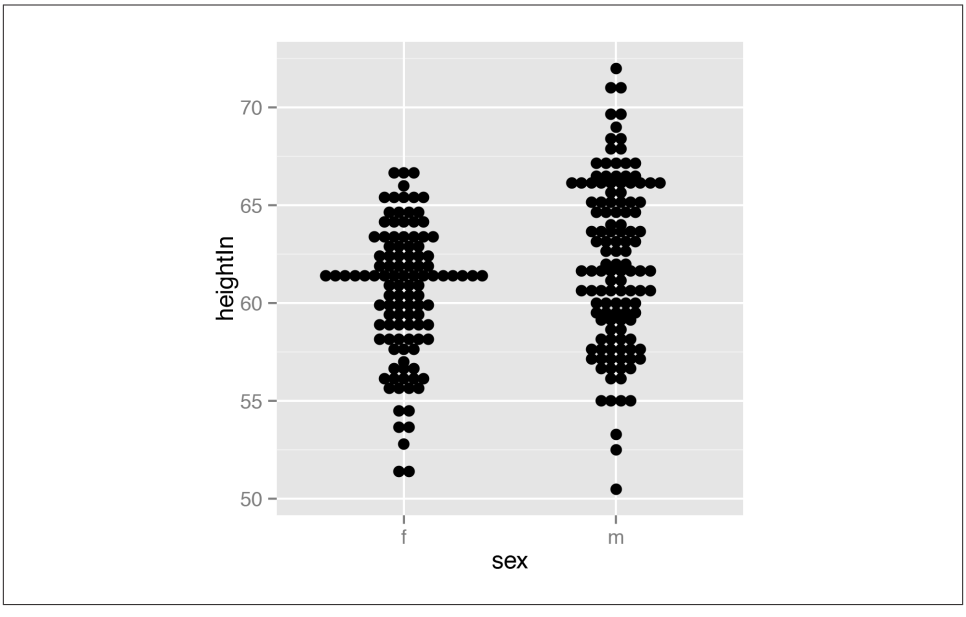

Figure 6-30. Dot plot of multiple groups, binning along the y-axis

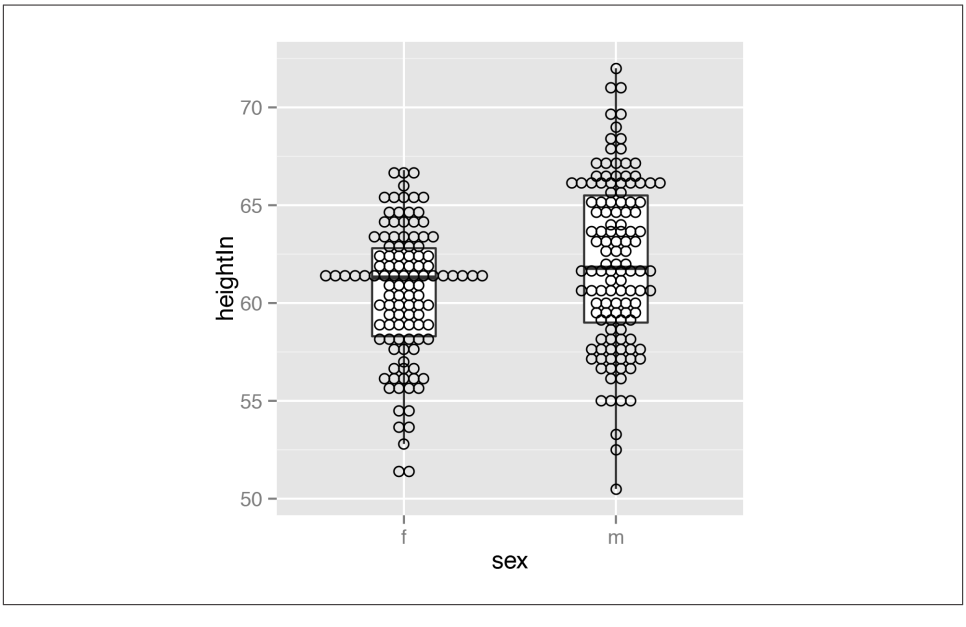

Figure 6-31. Dot plot overlaid on box plot

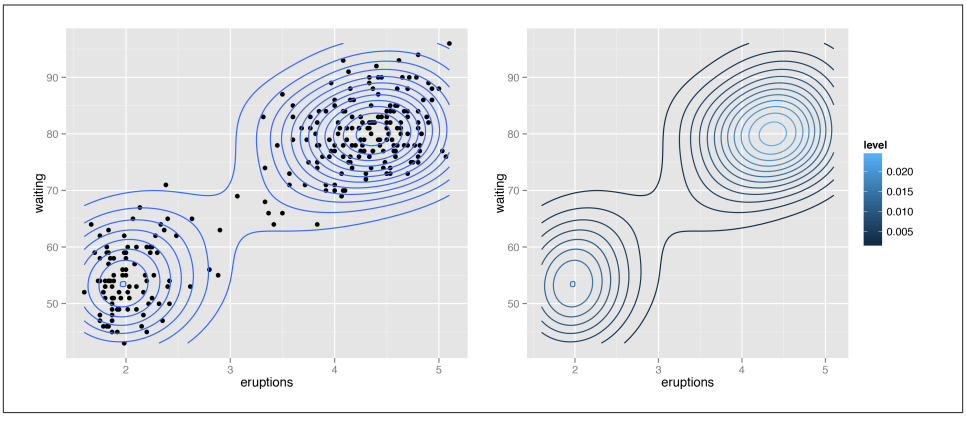

Figure 6-32. Dot plot next to box plot

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))
```

6.12. Making a Density Plot of Two-Dimensional Data

Problem

You want to plot the density of two-dimensional (2D) 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 6-33, left):

```
# The base plot
p <- ggplot(faithful, aes(x=eruptions, y=waiting))

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 6-33, right):

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

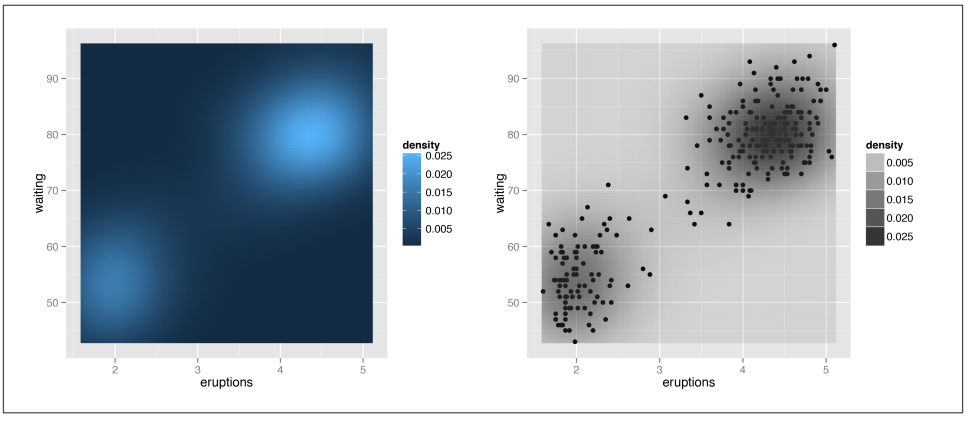

Figure 6-33. Left: points and density contour; right: with ..level.. mapped to color

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 map the density estimate to the fill color, or to the transparency of the tiles, as shown in Figure 6-34:

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

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

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).

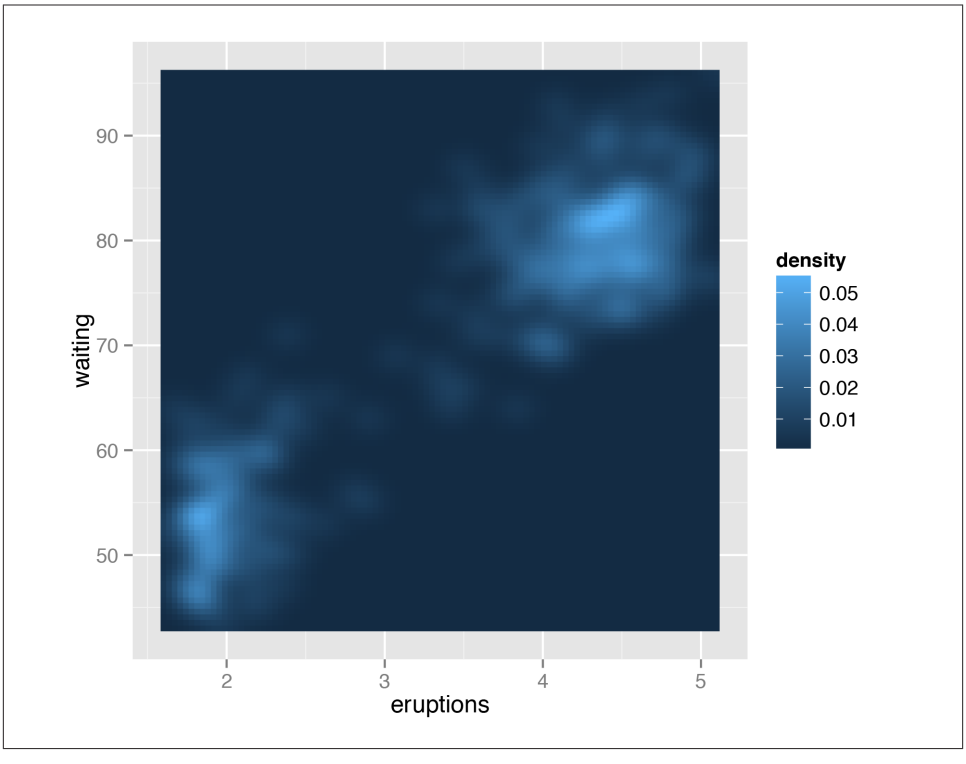

Figure 6-34. Marginal rug added to a scatter plot

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 6-35), 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:

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

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 assumptions, 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.

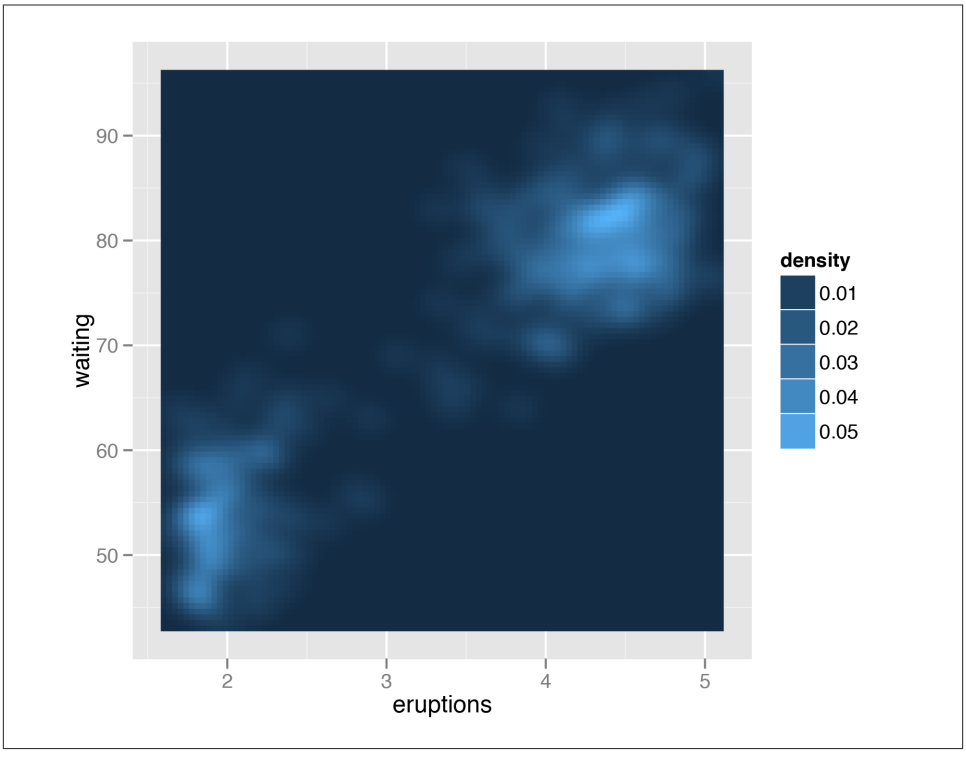

Figure 6-35. Density plot with a smaller bandwidth in the x and y directions

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

CHAPTER 7

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 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")
```

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:

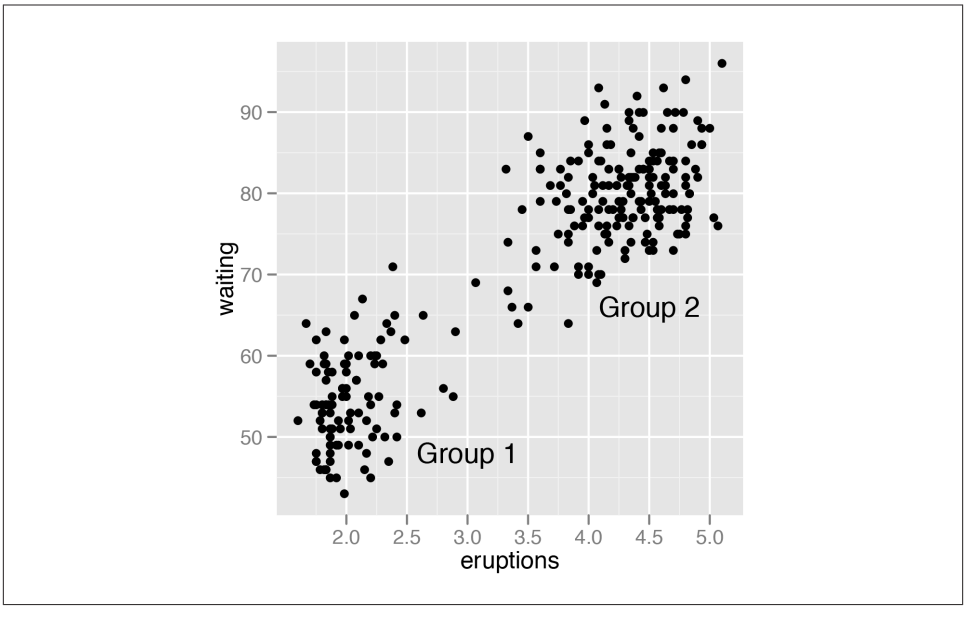

Figure 7-1. Text annotations

```
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)
```

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:

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

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:

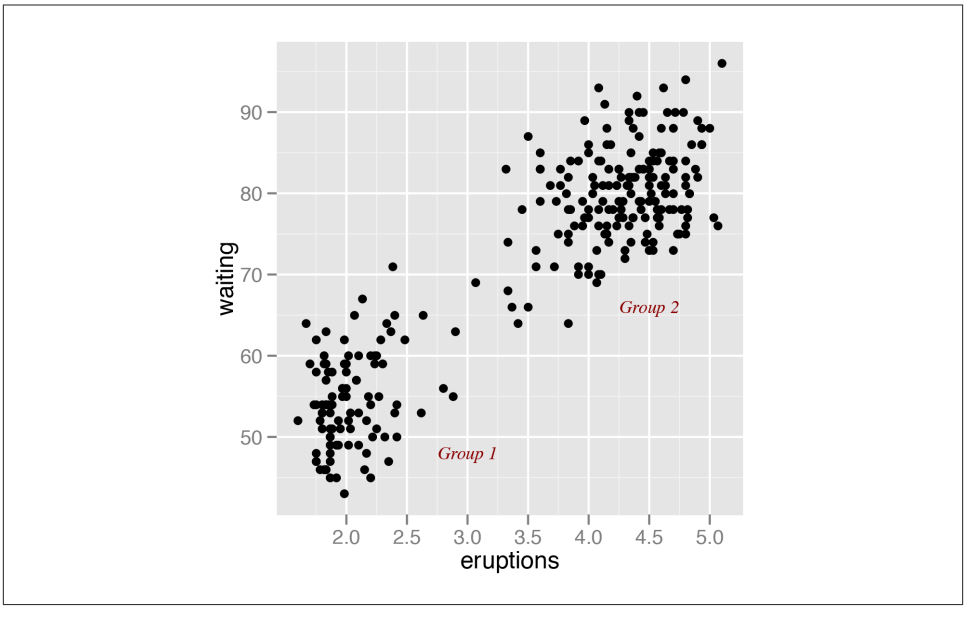

Figure 7-2. Modified text properties

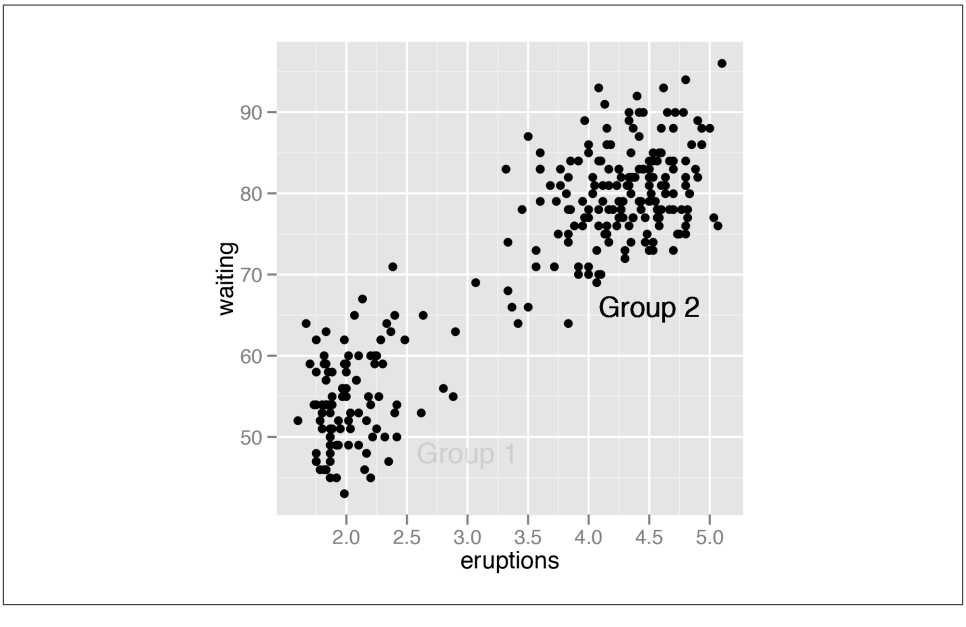

Figure 7-3. Overplotting one of the labels—both should be 90% transparent

```
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")
```

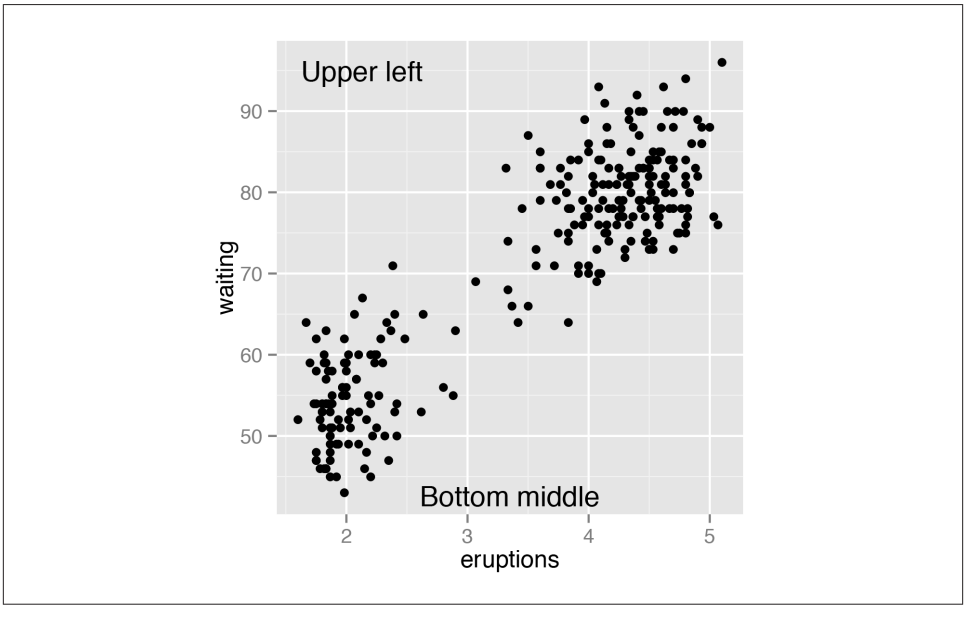

Figure 7-4. Text positioned at the edge of the plotting area

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")` and set `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}")
```

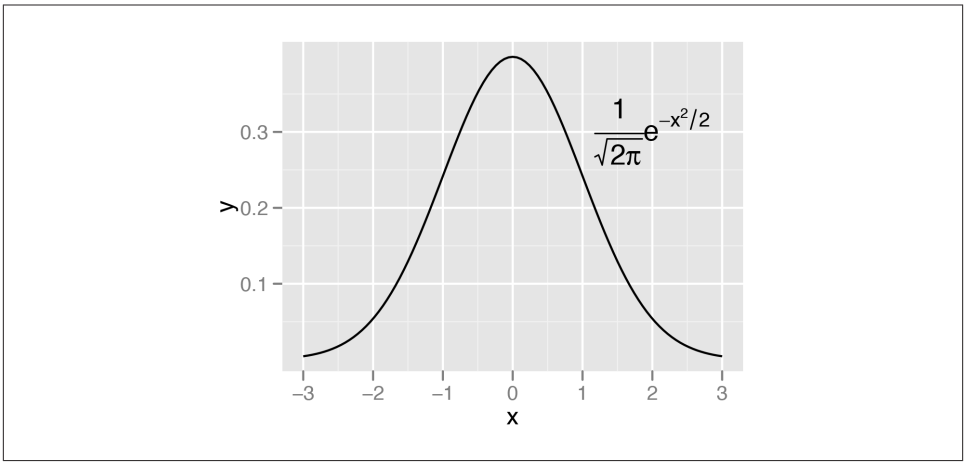

Figure 7-5. Annotation with mathematical expressions

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}")
```

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.

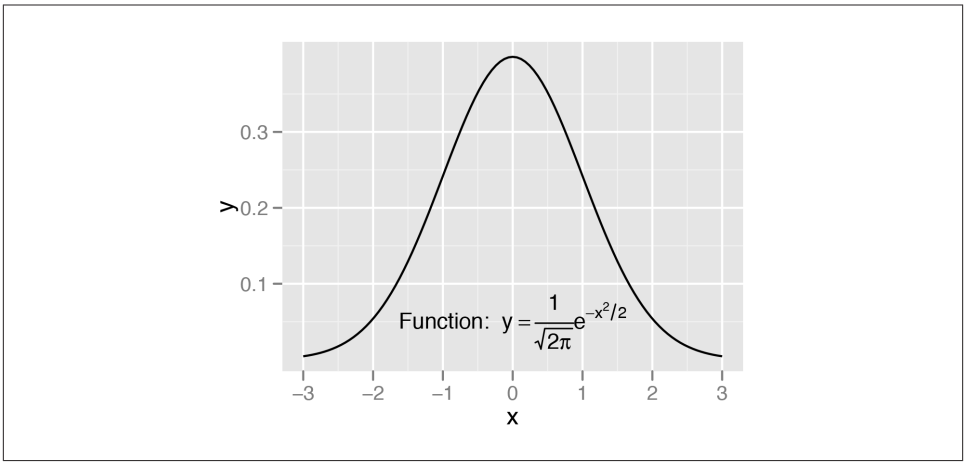

Figure 7-6. Mathematical expression with regular text

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 `heightweight` data set:

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

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

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

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

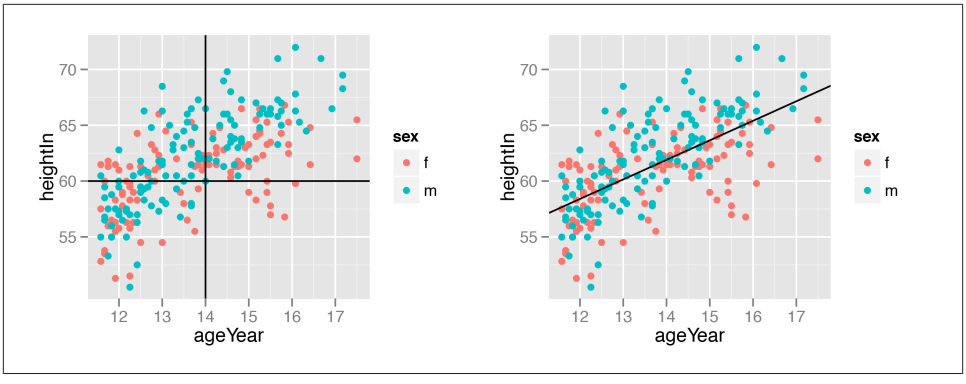

Figure 7-7. Left: horizontal and vertical lines; right: angled line

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(plyr) # For the ddply() function
hw_means <- ddply(heightweight, "sex", summarise, heightIn=mean(heightIn))
hw_means
```

```
 sex heightIn
   f 60.52613
   m 62.06000
```

```
p + geom_hline(aes(yintercept=heightIn, colour=sex), data=hw_means,
               linetype="dashed", size=1)
```

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

pg + geom_vline(xintercept = 2)

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

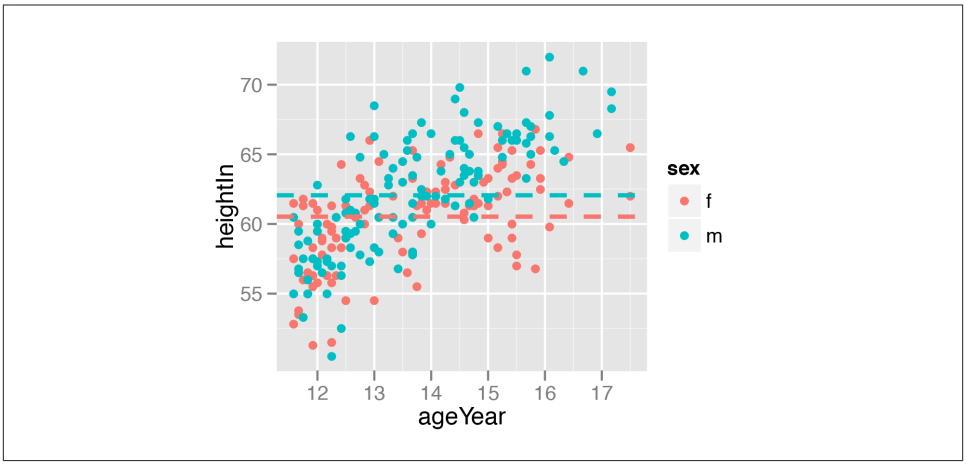

Figure 7-8. Multiple lines, drawn at the mean of each group

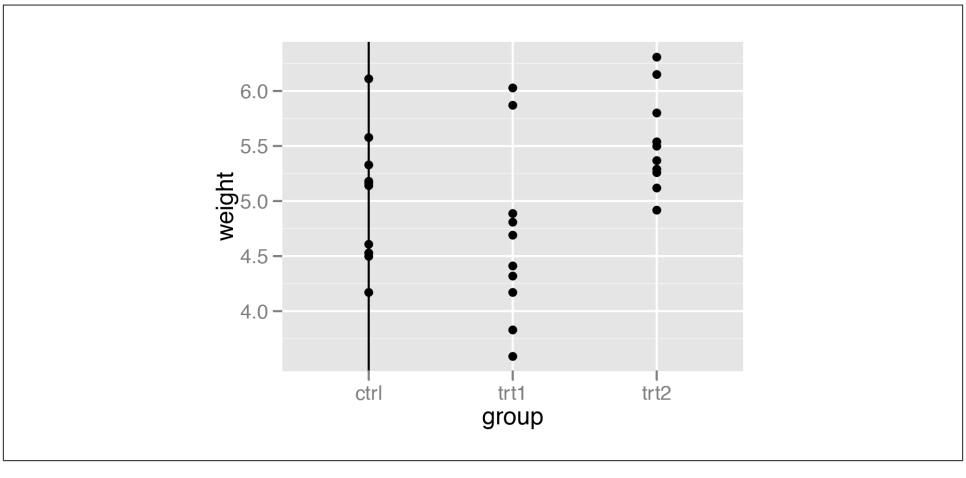

Figure 7-9. Lines with a discrete axis

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 previous 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. In a future version of ggplot2, this will change, and `annotate()` will work with line geoms.

See Also

For adding regression lines, see Recipes Recipe 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) # For the data set

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

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

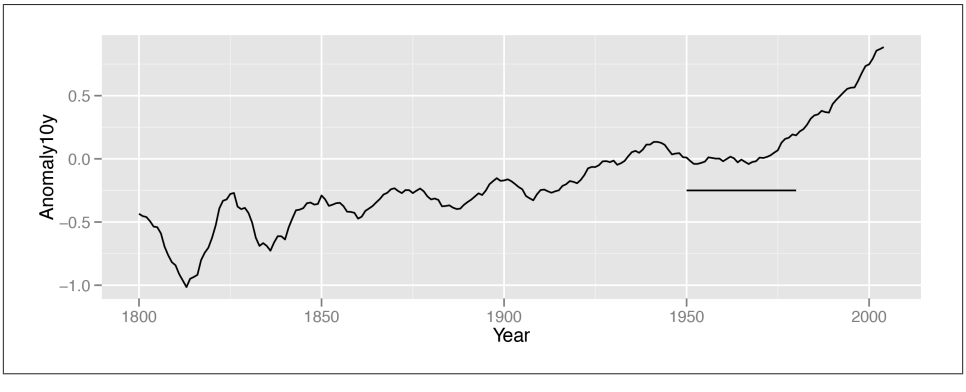

Figure 7-10. Line segment annotation

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")))
```

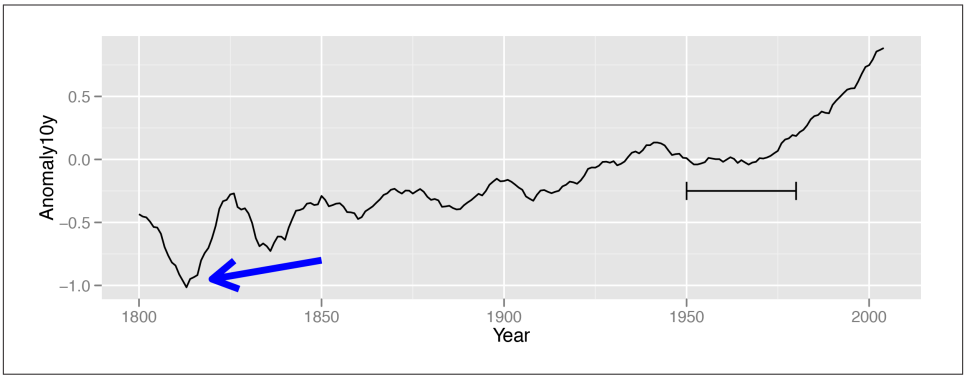

Figure 7-11. Line segments with arrow heads

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) # For the data set

p <- ggplot(subset(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")
```

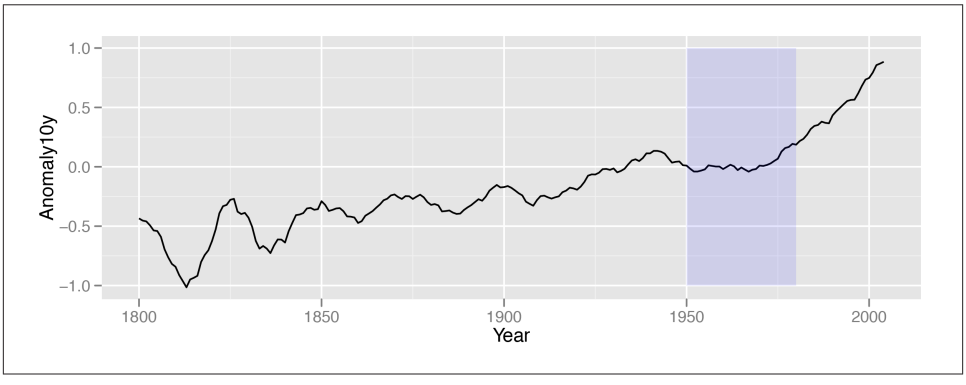

Figure 7-12. A shaded rectangle

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 new column, `hl`, and set its value based on the value of `group`:

```
pg <- PlantGrowth                 # Make a copy of the PlantGrowth data
pg$hl <- "no"                     # Set all to "no"
pg$hl[pg$group=="trt2"] <- "yes"  # If group is "trt2", set to "yes"
```

Then we'll plot it with manually specified colors and with no legend (Figure 7-13):

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

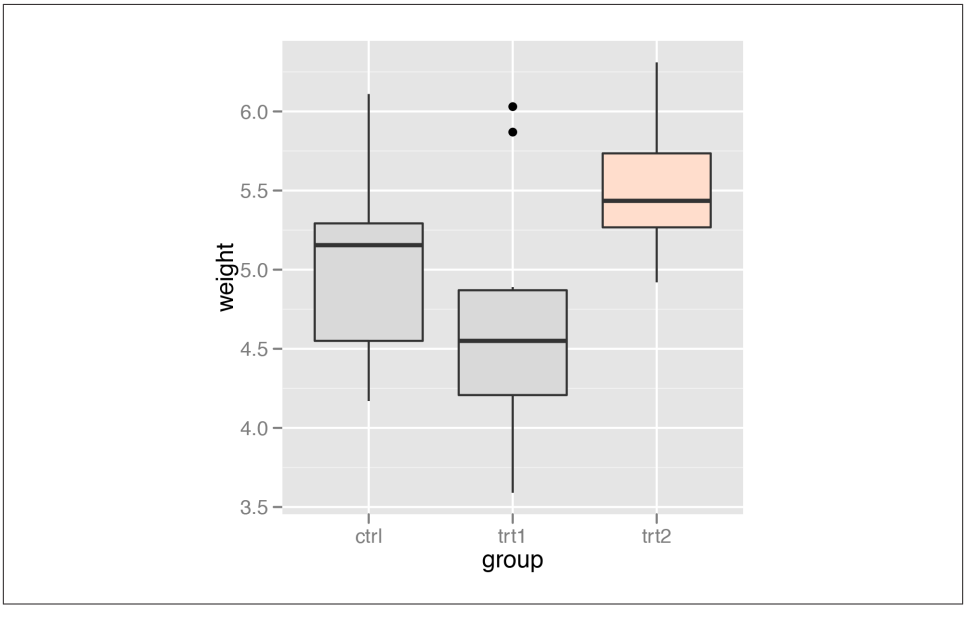

Figure 7-13. Highlighting one item

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 default *y* range is different for bars and lines, though):

```
library(gcookbook) # For the data set
# Take a subset of the cabbage_exp data for this example
ce <- subset(cabbage_exp, Cultivar == "c39")

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

# With a line graph
ggplot(ce, 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)
```

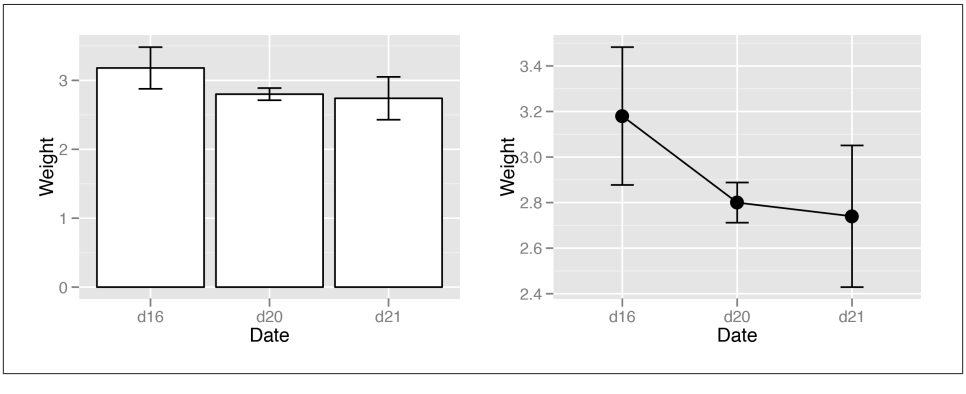

Figure 7-14. Left: error bars on a bar graph; right: on a line graph

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
```

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

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

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_bar(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_bar(position="dodge") +
    geom_errorbar(aes(ymin=Weight-se, ymax=Weight+se),
                  position=position_dodge(0.9), width=.2)
```

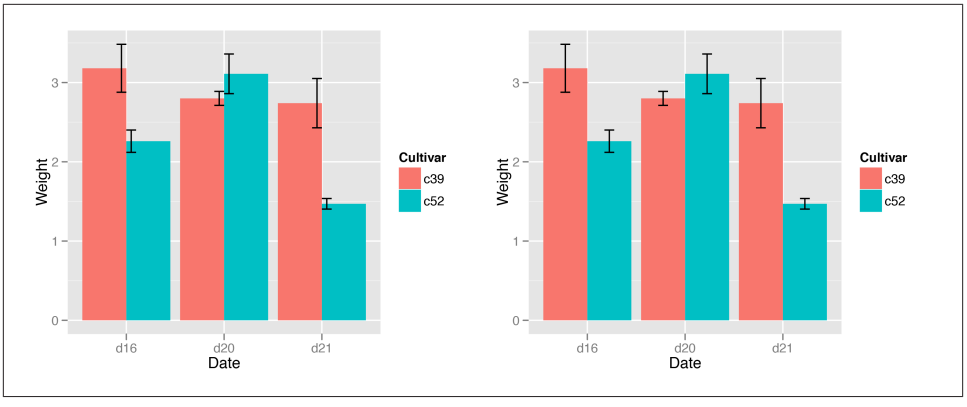

Figure 7-15. Left: error bars on a grouped bar graph without dodging width specified; right: with dodging width specified

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
```

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`.

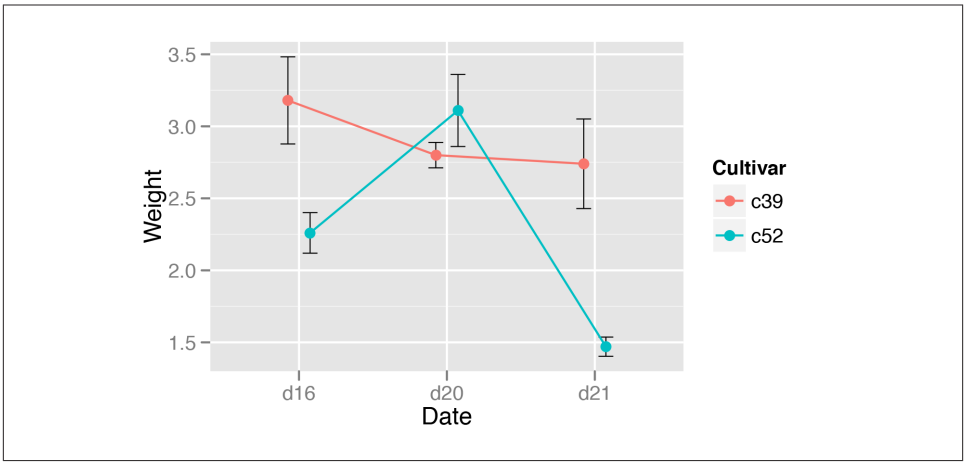

Figure 7-16. Error bars on a line graph, dodged so they don't overlap

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):

```
# The base plot
p <- 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"))

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

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

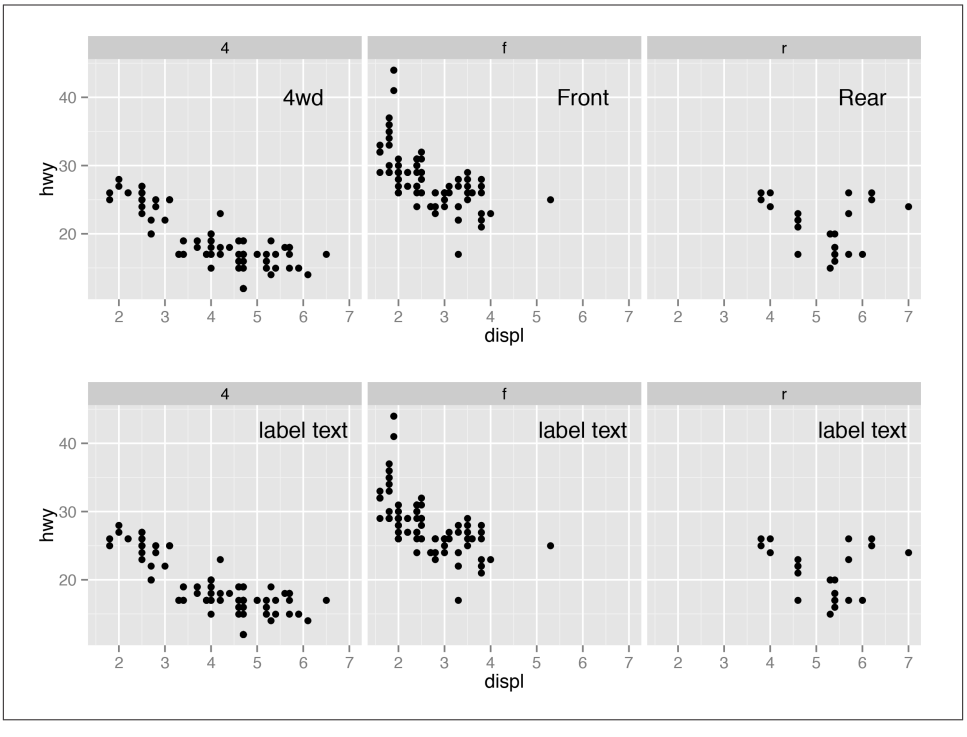

Figure 7-17. Top: different annotations in each facet; bottom: the same annotation in each facet

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 `ddply()` 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(plyr) # For the ddply() function
labels <- ddply(mpg, "drv", lm_labels)
labels
```

```
 drv                                formula                 r2
   4 italic(y) == 30.68 -2.88 * italic(x) italic(R^2) == 0.65
   f italic(y) == 37.38 -3.60 * italic(x) italic(R^2) == 0.36
   r italic(y) == 25.78 -0.92 * italic(x) italic(R^2) == 0.04
```

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

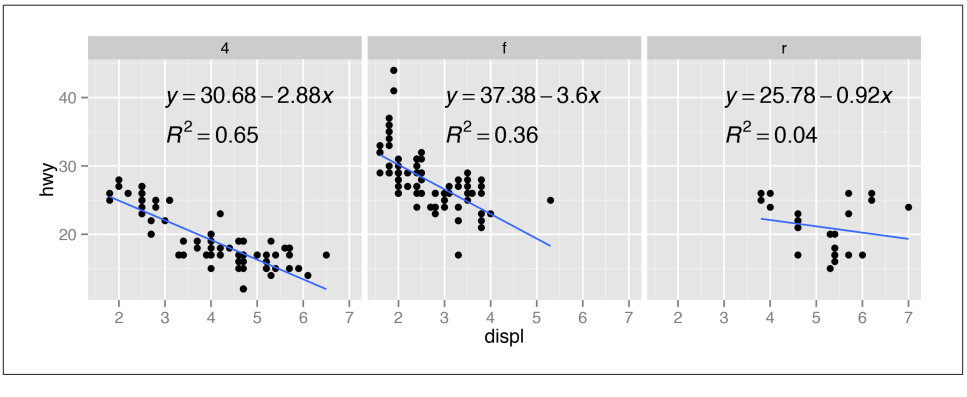

Figure 7-18. Annotations in each facet with information about the data

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 `ddply()` with the `summarise()` function and then passing additional arguments for `summarise()`:

```
# Find r^2 values for each group
labels <- ddply(mpg, "drv", summarise, r2 = cor(displ, hwy)^2)
labels$r2 <- sprintf("italic(R^2) == %.2f", labels$r2)
```

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.

CHAPTER 8
Axes

The x- and y-axes provide context for interpreting the displayed data. Ggplot2 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()
```

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 ggplot2 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.

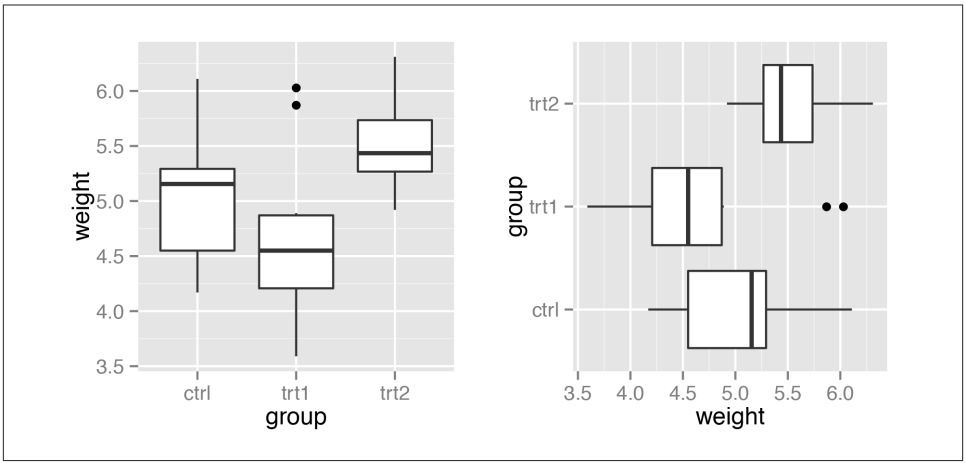

Figure 8-1. Left: a box plot with regular axes; right: with swapped axes

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)))
```

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:

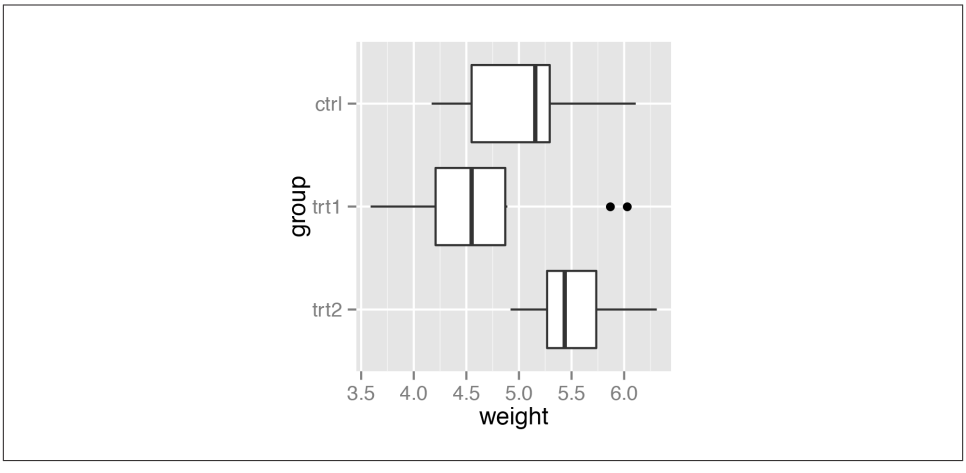

Figure 8-2. A box plot with swapped axes and x-axis order reversed

```
p <- ggplot(PlantGrowth, aes(x=group, y=weight)) + geom_boxplot()
# Display the basic graph
p

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

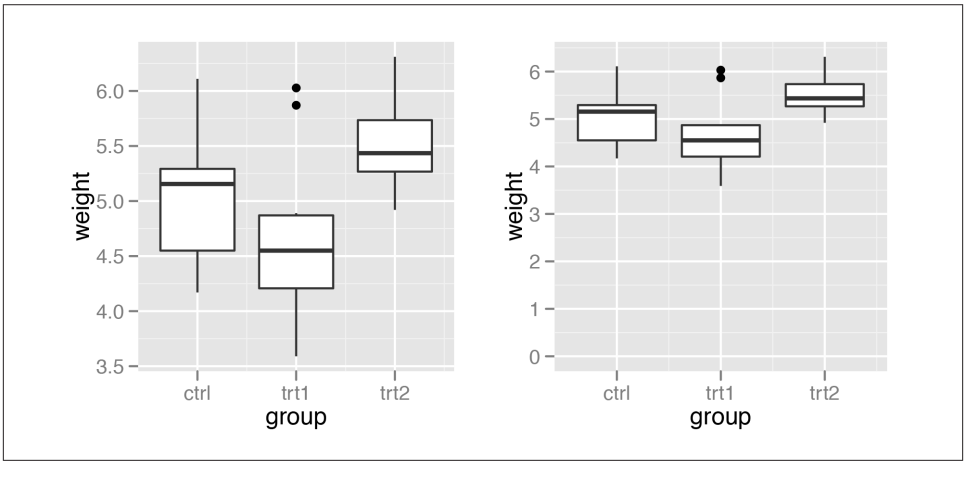

Figure 8-3. Left: box plot with default range; right: with manually set range

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:

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

p + 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()`:

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

In ggplot2, 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.

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:

```
p + scale_y_continuous(limits = c(5, 6.5))  # Same as using ylim()

p + coord_cartesian(ylim = c(5, 6.5))
```

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:

```
p + expand_limits(y=0)
```

8.3. Reversing a Continuous Axis

Problem

You want to reverse the direction of a continuous axis.

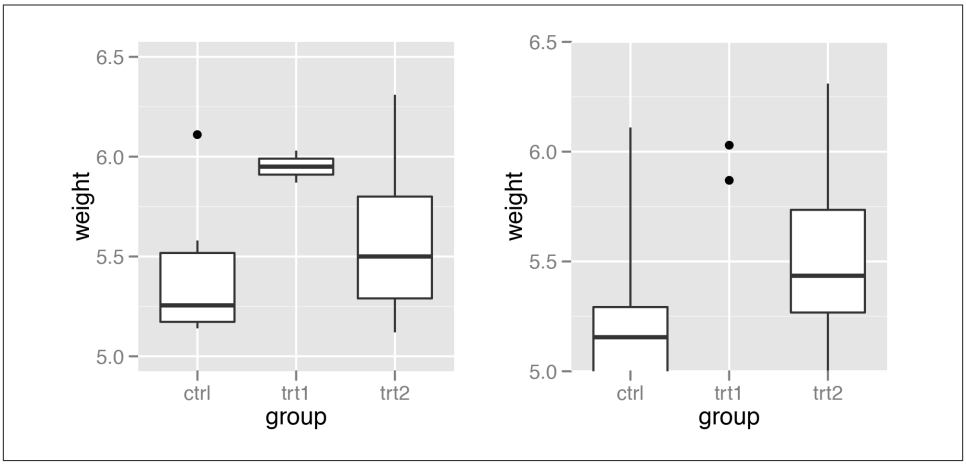

Figure 8-4. Left: smaller y range using a scale (data has been dropped, so the box plots have changed shape); right: "zooming in" using a coordinate transform

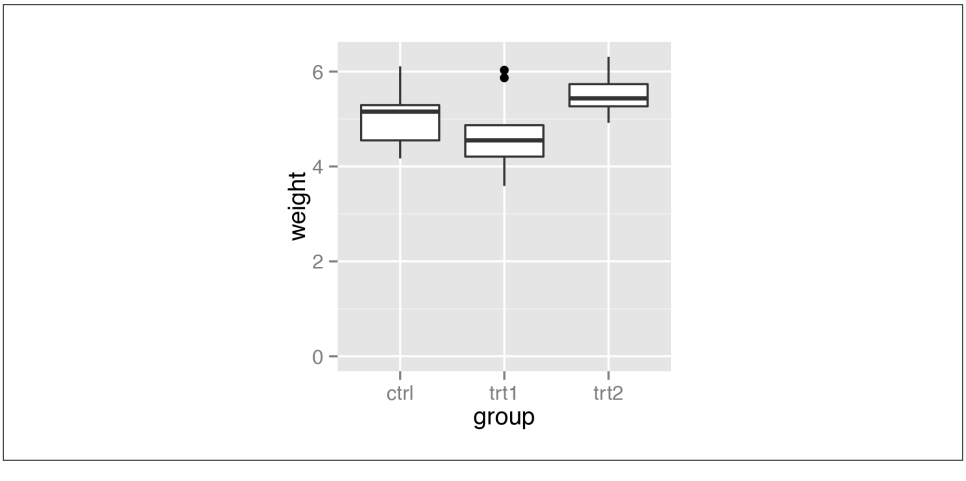

Figure 8-5. Box plot on which y range has been expanded to include 0

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:

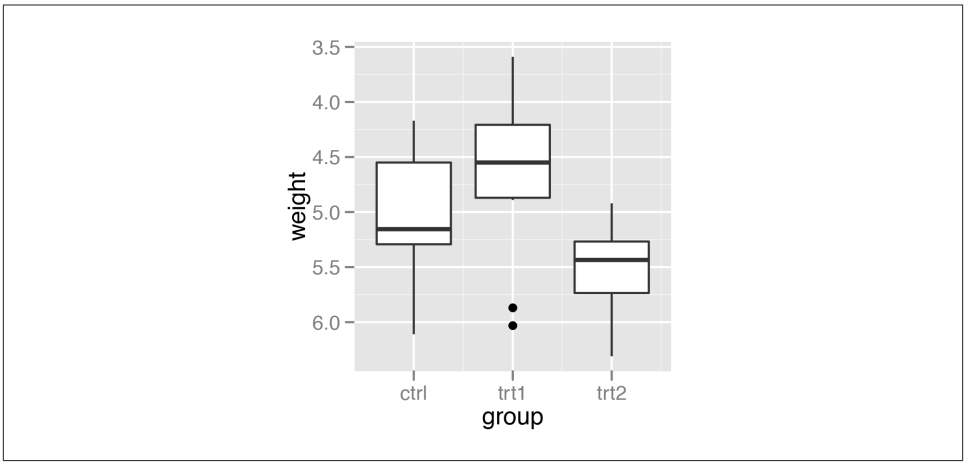

Figure 8-6. Box plot with reversed y-axis

```
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)
```

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))
```

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.

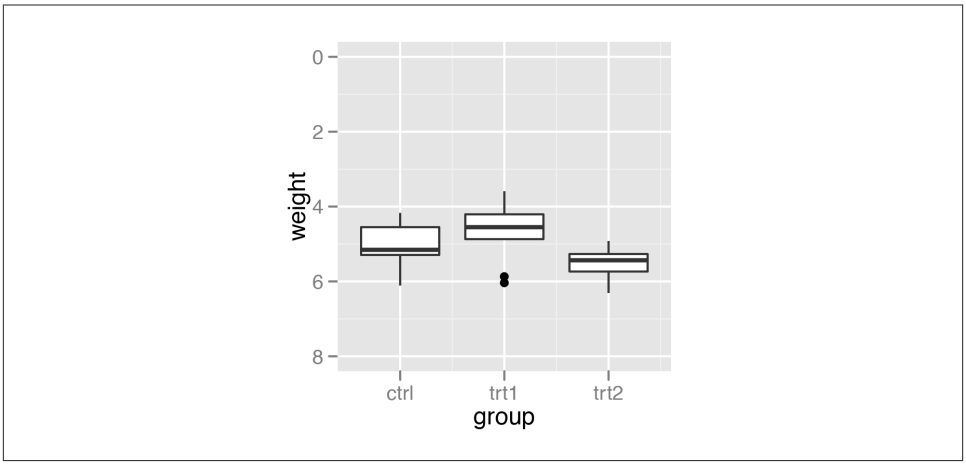

Figure 8-7. Box plot with reversed y-axis with manually set limits

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:

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

p + 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):

```
p + scale_x_discrete(limits=c("ctrl","trt1"))
```

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:

```
p + scale_x_discrete(limits=rev(levels(PlantGrowth$group)))
```

See Also

To reorder factor levels based on data values from another column, see Recipe 15.9.

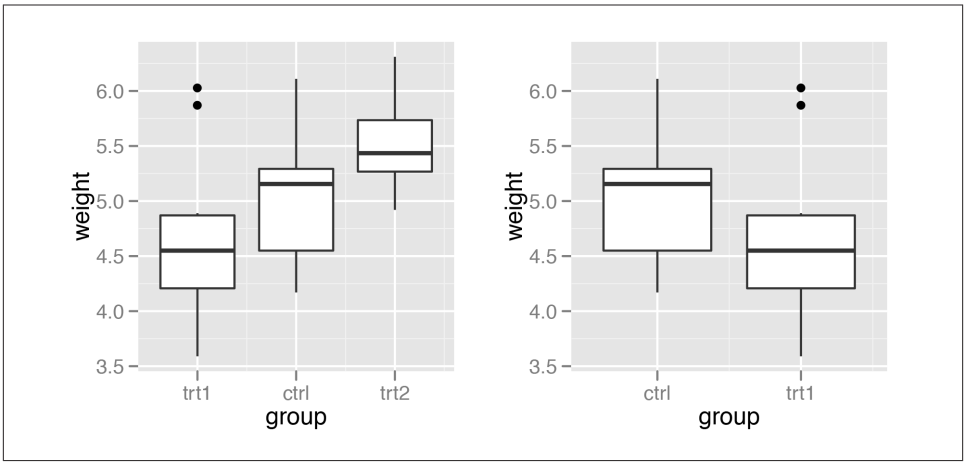

Figure 8-8. Left: box plot with manually specified items on the x-axis; right: with only two items

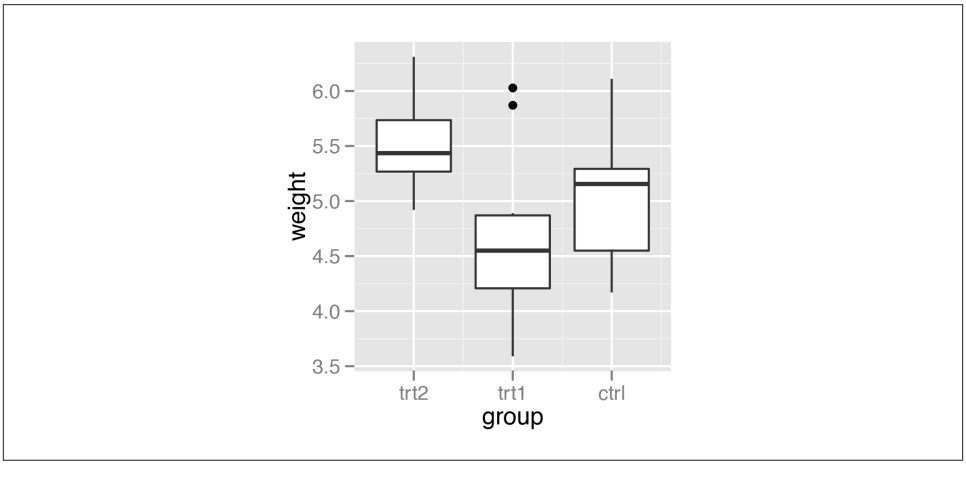

Figure 8-9. Box plot with order reversed on the x-axis

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) # For the data set

sp <- ggplot(marathon, aes(x=Half,y=Full)) + geom_point()

sp + coord_fixed()
```

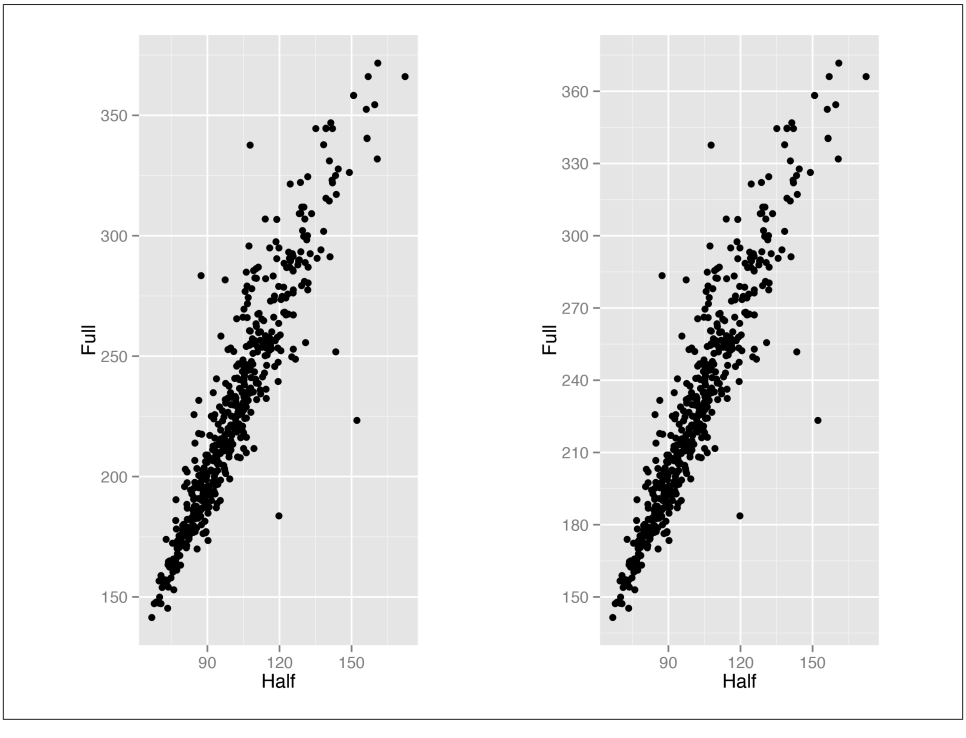

Figure 8-10. Left: scatter plot with equal scaling of axes; right: with tick marks at specified positions

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):

```
sp + coord_fixed() +
    scale_y_continuous(breaks=seq(0, 420, 30)) +
    scale_x_continuous(breaks=seq(0, 420, 30))
```

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:

```
sp + coord_fixed(ratio=1/2) +
    scale_y_continuous(breaks=seq(0, 420, 30)) +
    scale_x_continuous(breaks=seq(0, 420, 15))
```

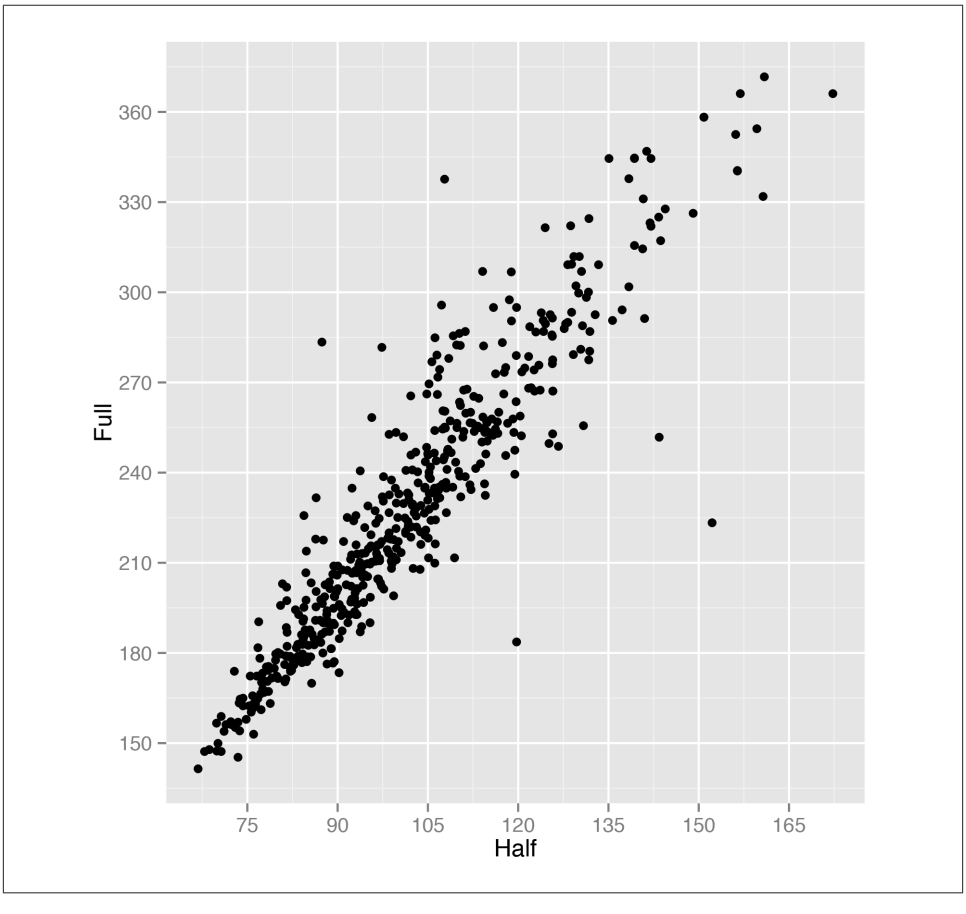

Figure 8-11. Scatter plot with a 1/2 scaling ratio for the axes

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))
```

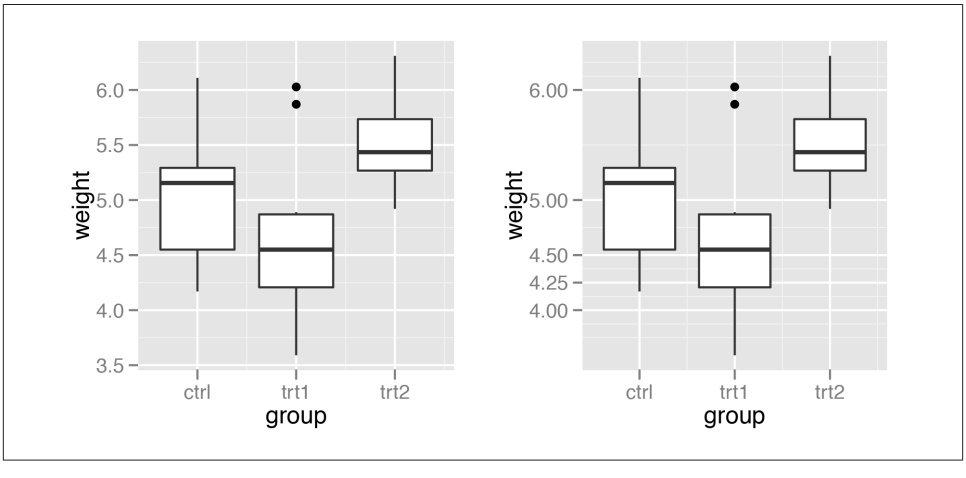

Figure 8-12. Left: box plot with automatic tick marks; right: with manually set tick marks

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)
```

```
4.0 4.5 5.0 5.5 6.0 6.5 7.0
```

```
5:10
```

```
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`:

```
# 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")
```

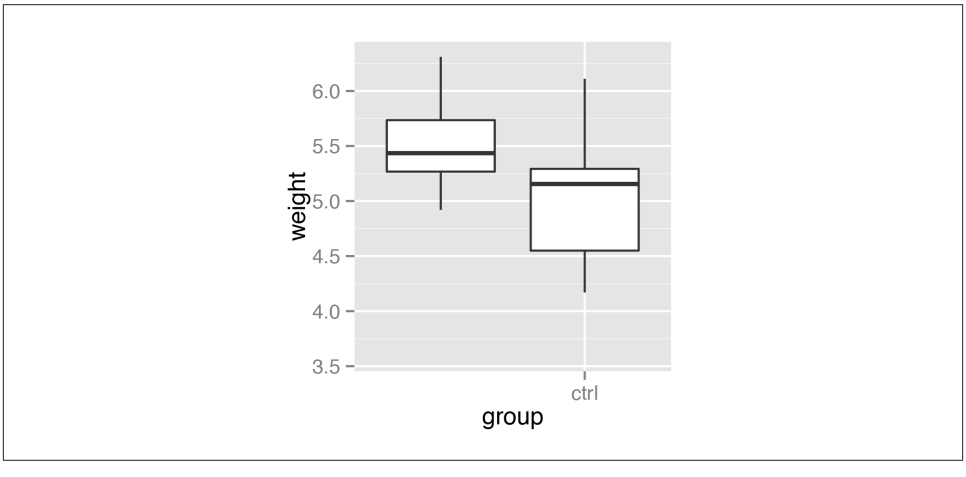

Figure 8-13. For a discrete axis, setting limits reorders and removes items, and setting breaks controls which items have labels

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:

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

p + theme(axis.text.y = element_blank())
```

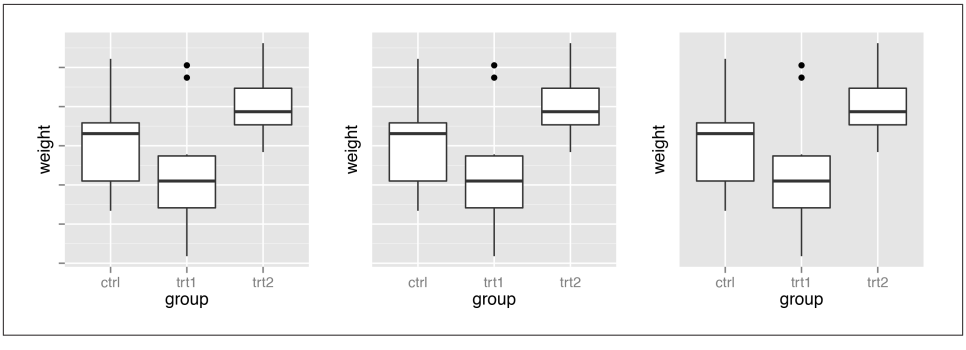

Figure 8-14. Left: no tick labels on y-axis; middle: no tick marks and no tick labels on y-axis; right: with breaks=NULL

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):

```
p + 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):

```
p + scale_y_continuous(breaks=NULL)
```

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) # For the data set

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

hwp
```

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:

```
hwp + scale_y_continuous(breaks=c(50, 56, 60, 66, 72),
                         labels=c("Tiny", "Really\nshort", "Short",
                                  "Medium", "Tallish"))
```

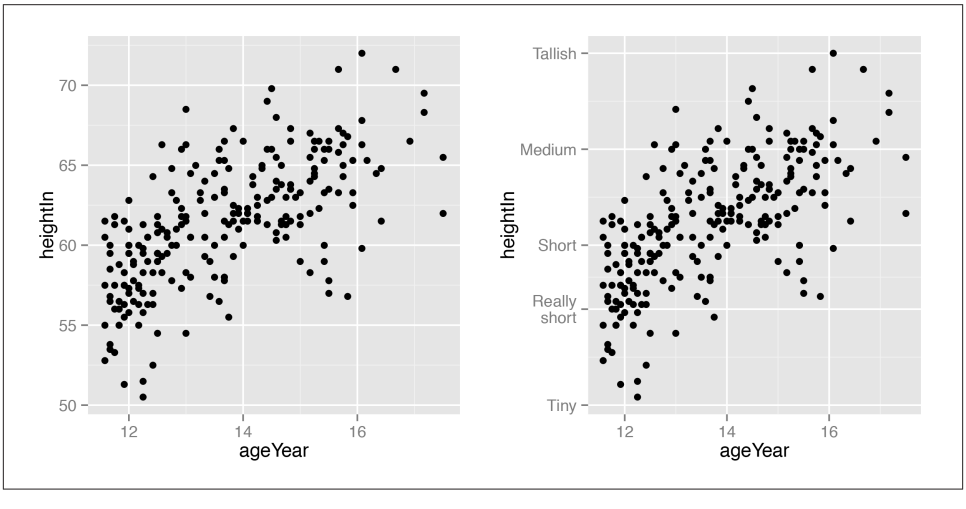

Figure 8-15. Left: scatter plot with automatic tick labels; right: with manually specified labels on the y-axis

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)
```

```
 "4'8\""  "4'9\""  "4'10\"" "4'11\"" "5'0\""  "5'1\""  "5'2\""  "5'3\""  "5'4\""
```

Now we can pass our function to the scale, using the `labels` parameter (Figure 8-16):

```
hwp + 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):

```
hwp + scale_y_continuous(breaks=seq(48, 72, 4), labels=footinch_formatter)
```

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))                  # Round to nearest second
    lab <- sprintf("%02d:%02d:%02d", h, m, s) # Format the strings as HH:MM:SS
    lab <- gsub("^00:", "", lab)             # Remove leading 00: if present
    lab <- gsub("^0", "", lab)               # Remove leading 0 if present
    return(lab)
}
```

Running it on some sample numbers yields:

```
timeHMS_formatter(c(.33, 50, 51.25, 59.32, 60, 60.1, 130.23))
```

```
 "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:

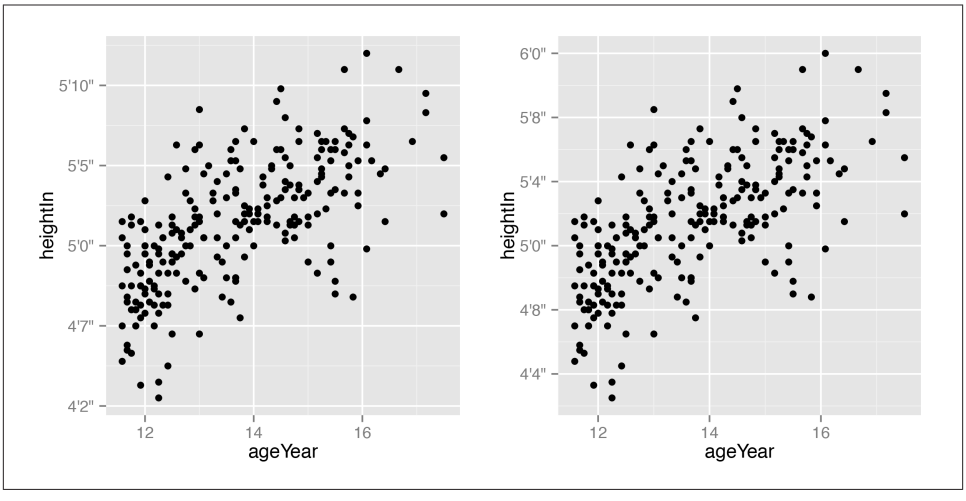

Figure 8-16. Left: scatter plot with a formatter function; right: with manually specified breaks on the y-axis

- `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:

```
bp <- ggplot(PlantGrowth, aes(x=group, y=weight)) + geom_boxplot() +
      scale_x_discrete(breaks=c("ctrl", "trt1", "trt2"),
                       labels=c("Control", "Treatment 1", "Treatment 2"))
bp
```

To rotate the text 90 degrees counterclockwise (Figure 8-17, middle), use:

```
bp + 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:

```
bp + theme(axis.text.x = element_text(angle=30, hjust=1, vjust=1))
```

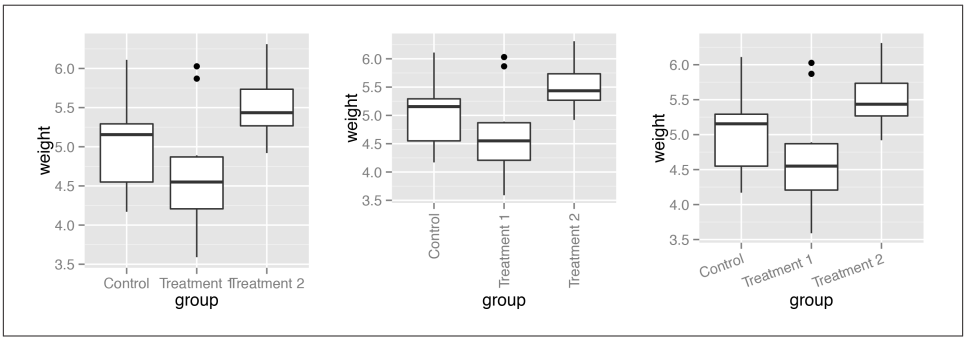

Figure 8-17. X-axis tick labels rotated 0 (left), 90 (middle), and 30 degrees (right)

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:

```
bp + theme(axis.text.x = element_text(family="Times", face="italic",
                                      colour="darkred", size=rel(0.9)))
```

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.

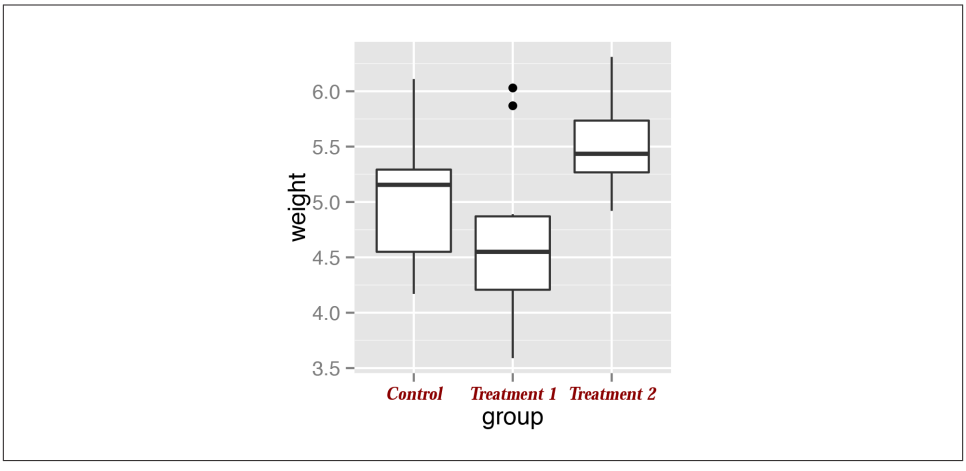

Figure 8-18. X-axis tick labels with manually specified appearance

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) # For the data set

hwp <- ggplot(heightweight, aes(x=ageYear, y=heightIn, colour=sex)) +
       geom_point()
# With default axis labels
hwp

# Set the axis labels
hwp + xlab("Age in years") + ylab("Height in inches")
```

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()`:

```
hwp + labs(x = "Age in years", y = "Height in inches")
```

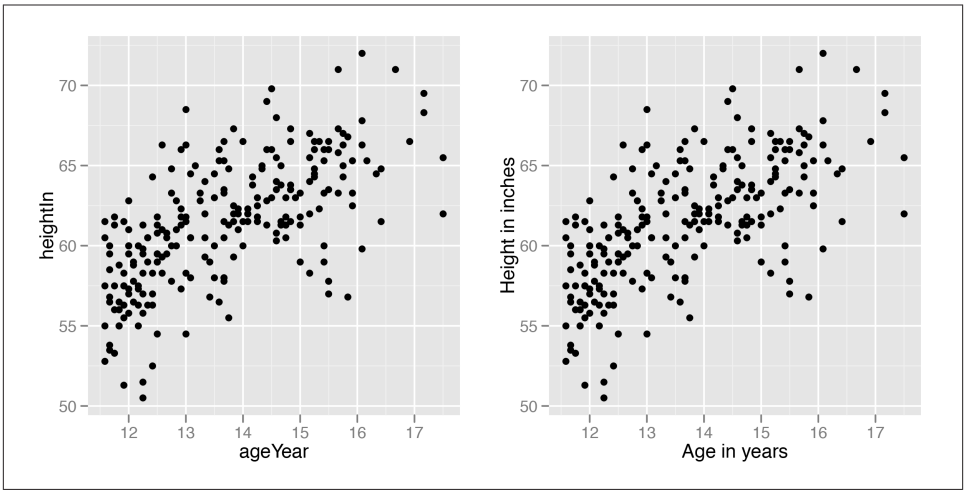

Figure 8-19. Left: scatter plot with the default axis labels; right: manually specified labels for the x- and y-axes

Another way of setting the axis labels is in the scale specification, like this:

```
hwp + 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:

```
hwp + scale_x_continuous(name="Age\n(years)")
```

8.11. Removing Axis Labels

Problem

You want to remove the label on an axis.

Solution

For the x-axis label, use `theme(axis.title.x=element_blank())`. For the y-axis label, do the same with `axis.title.y`.

We'll hide the x-axis in this example (Figure 8-21):

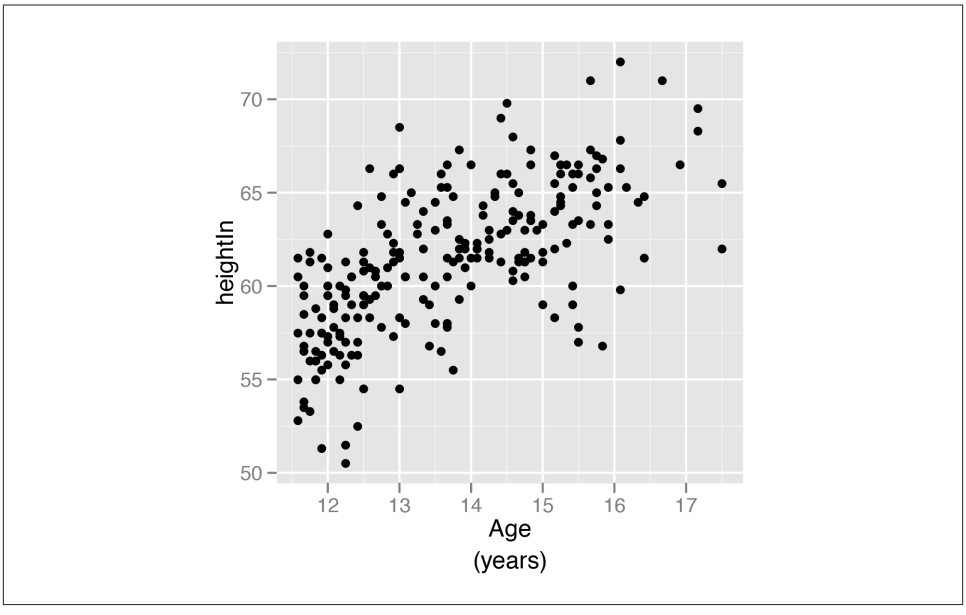

Figure 8-20. X-axis label with a line break

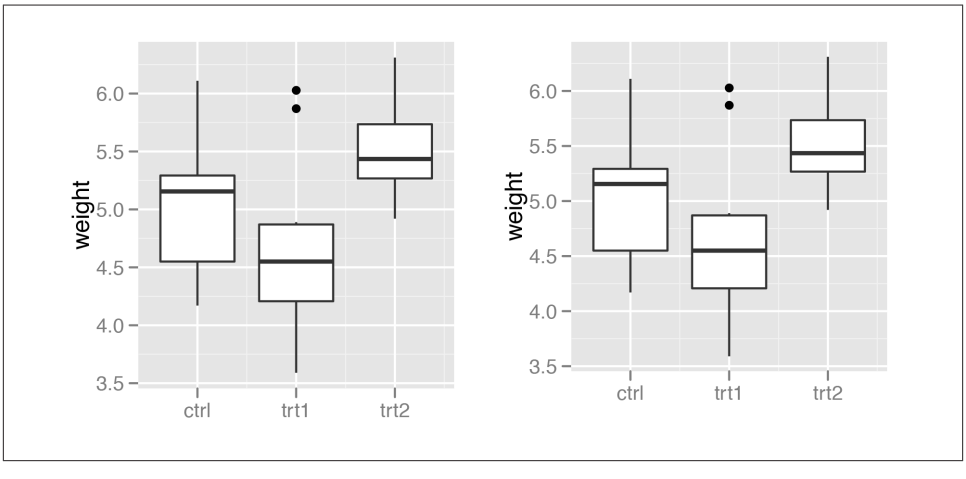

Figure 8-21. Left: x-axis label with element_blank(); right: with the label set to “”

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

p + theme(axis.title.x=element_blank())
```

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:

```
p + xlab("")
```

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) # For the data set

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

hwp + theme(axis.title.x=element_text(face="italic", colour="darkred", size=14))
```

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:

```
hwp + 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):

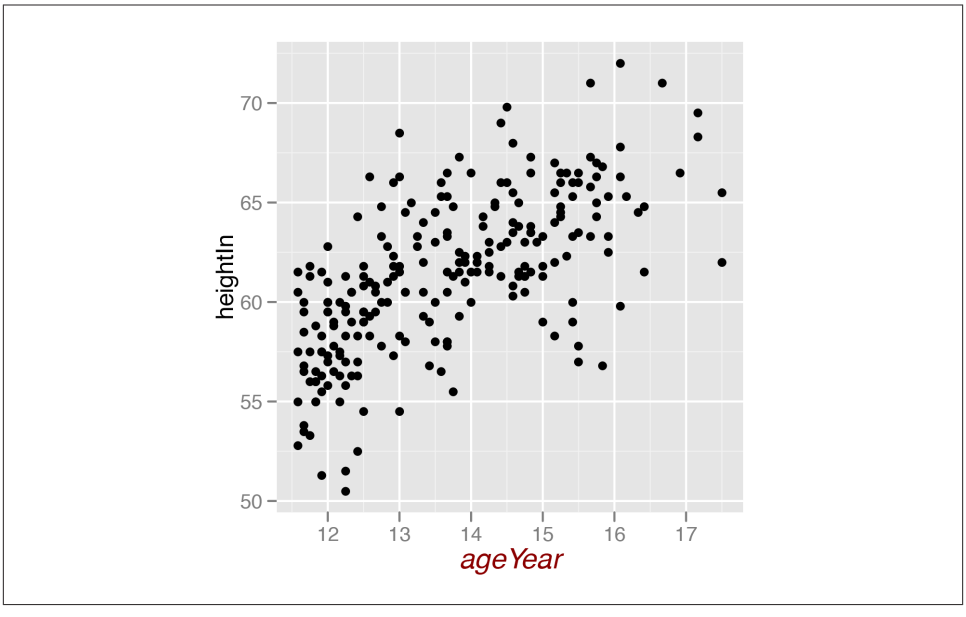

Figure 8-22. X-axis label with customized appearance

```
hwp + ylab("Height\n(inches)") +
    theme(axis.title.y=element_text(angle=90, face="italic", colour="darkred",
                                    size=14))
```

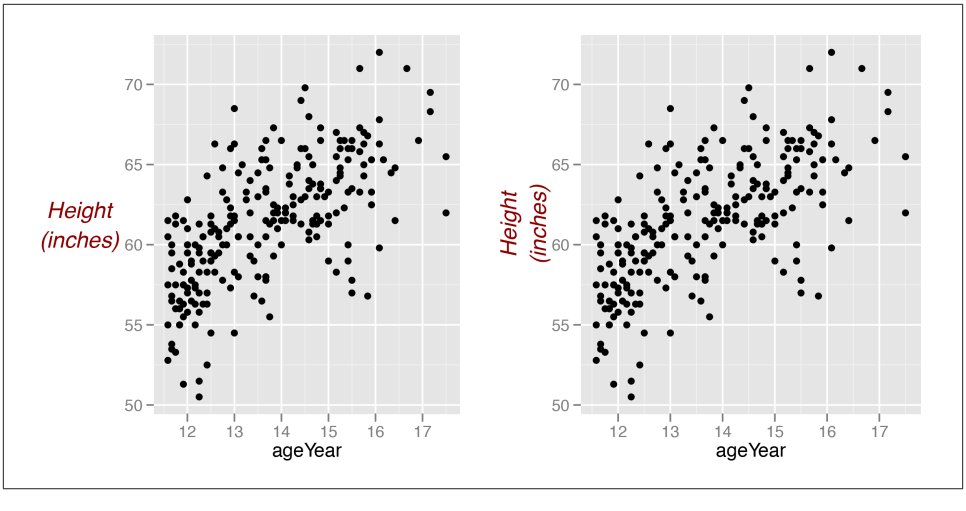

Figure 8-23. Left: y-axis label with angle=0; right: with angle=90

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) # For the data set

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

p + theme(axis.line = element_line(colour="black"))
```

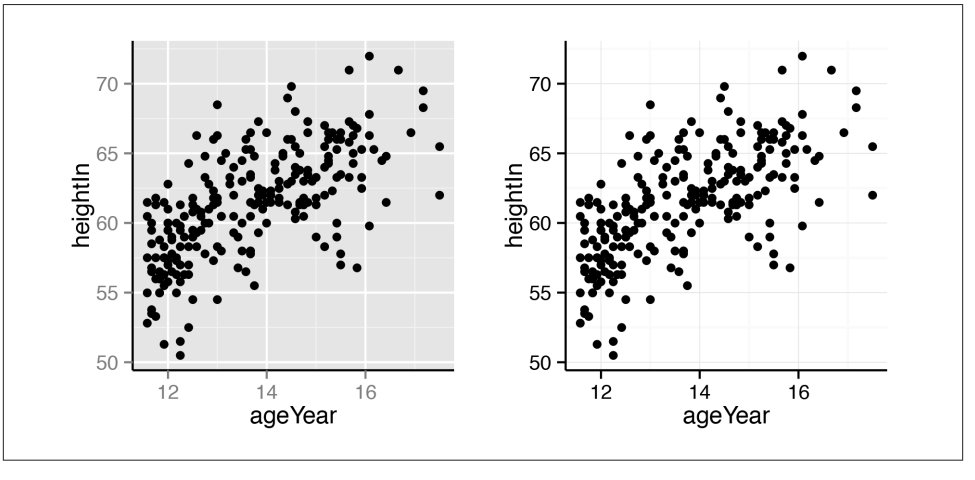

Figure 8-24. Left: scatter plot with axis lines; right: with theme_bw(), panel.border must also be made blank

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):

```
p + theme_bw() +
    theme(panel.border = element_blank(),
          axis.line = element_line(colour="black"))
```

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
p + theme_bw() +
    theme(panel.border = element_blank(),
          axis.line = element_line(colour="black", size=4))

# Full overlap
p + theme_bw() +
    theme(panel.border = element_blank(),
          axis.line = element_line(colour="black", size=4, lineend="square"))
```

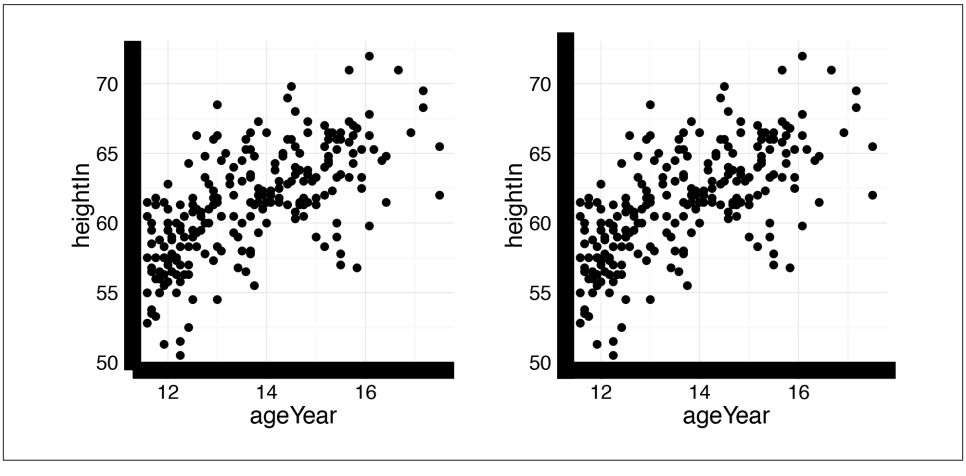

Figure 8-25. Left: with thick lines, the ends don't fully overlap; right: full overlap with lineend="square"

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) # For the data set

# The base plot
p <- ggplot(Animals, aes(x=body, y=brain, label=rownames(Animals))) +
     geom_text(size=3)
p

# With logarithmic x and y scales
p + scale_x_log10() + scale_y_log10()
```

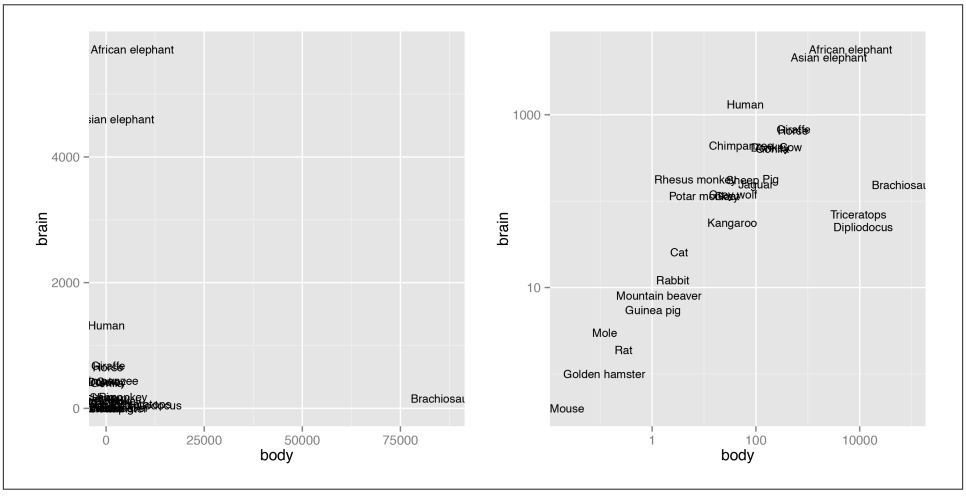

Figure 8-26. Left: exponentially distributed data with linear-scaled axes; right: with logarithmic axes

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` library 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**

```
...
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.

Ggplot2 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   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)
```

```
 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):

```
p + 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)
p + 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)))
```

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)
```

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:

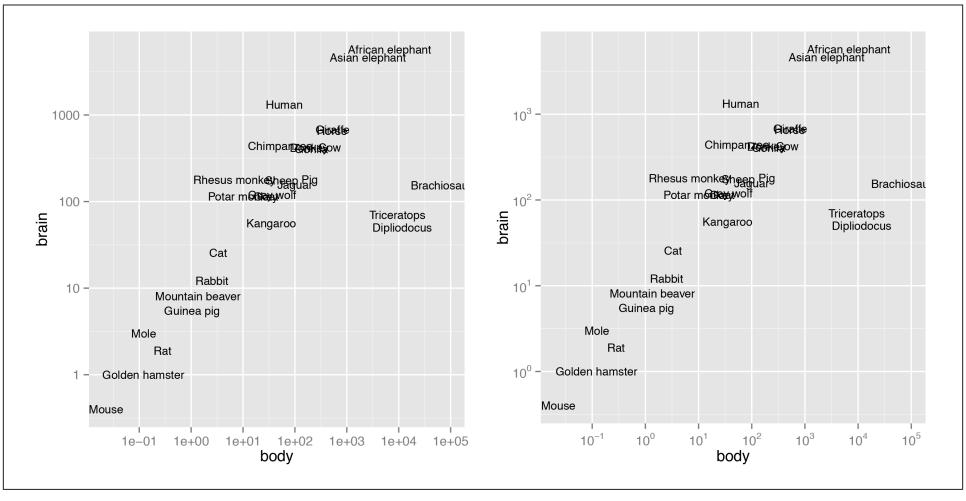

Figure 8-27. Left: scatter plot with $\log_{10}$ x- and y-axes, and with manually specified breaks; right: with exponents for the tick labels

```
library(scales)

# Use natural log on x, and log2 on y
p + 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)))
```

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) # For the 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))
```

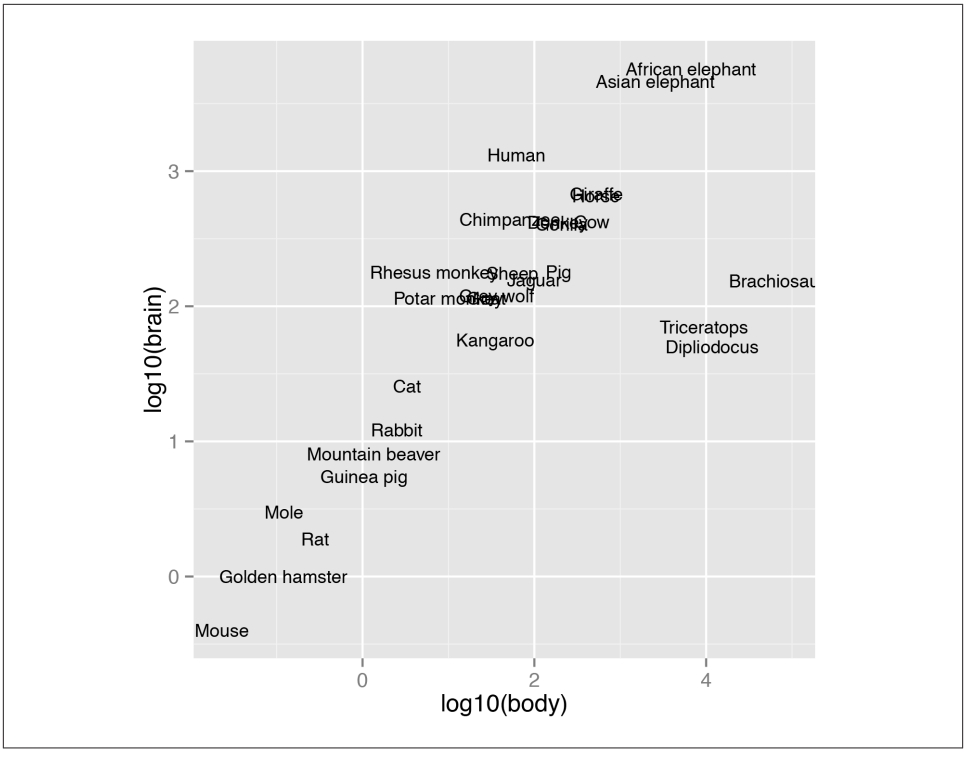

Figure 8-28. Plot with log transform before mapping to x- and y-axes

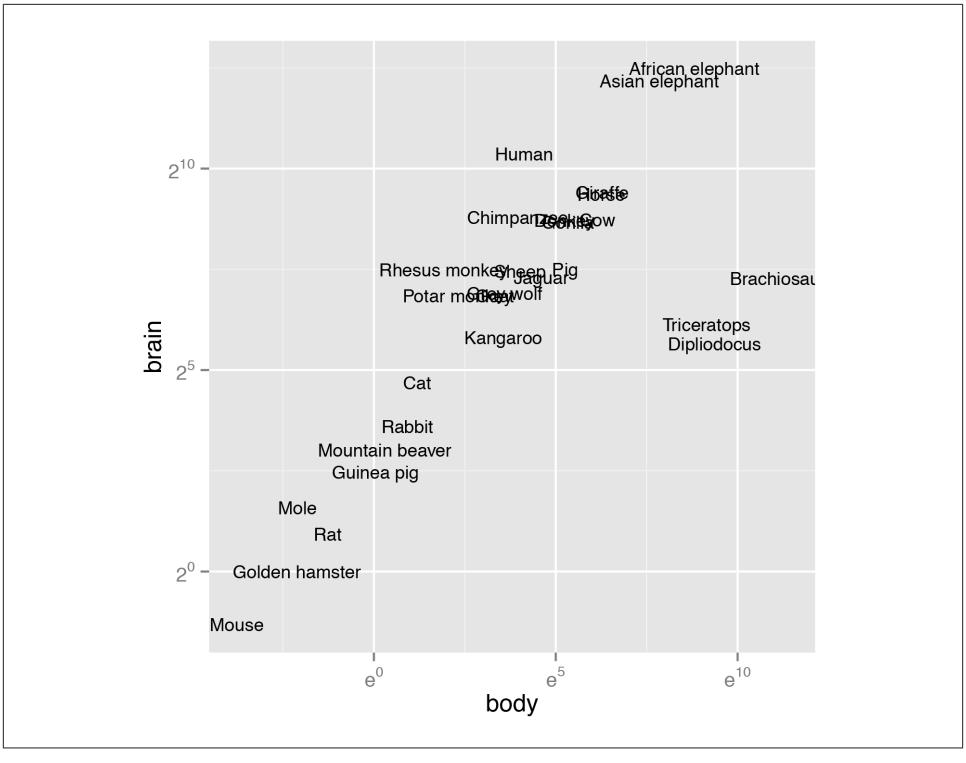

Figure 8-29. Plot with exponents in tick labels. Notice that different bases are used for the x and y axes.

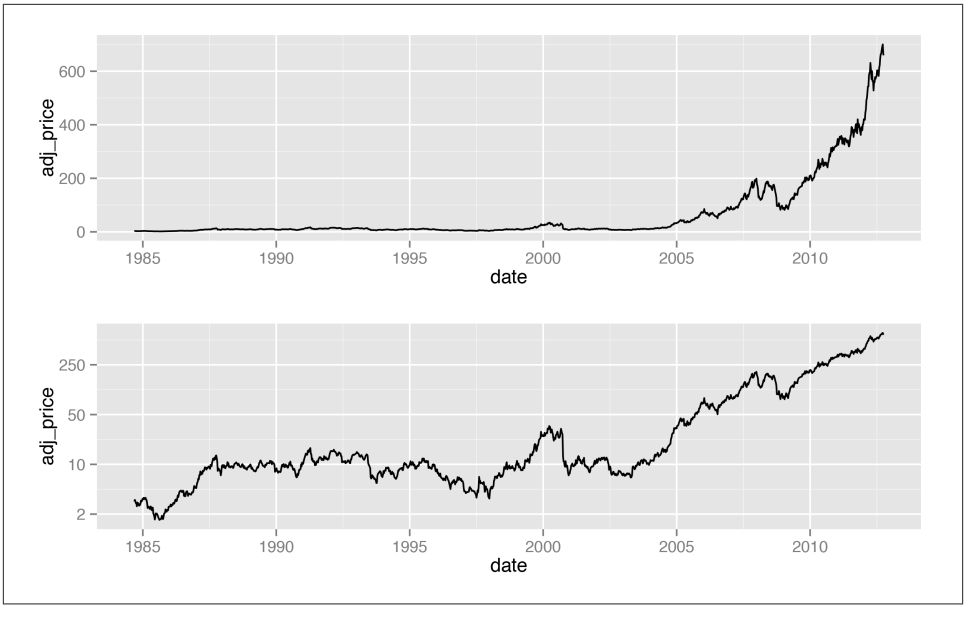

Figure 8-30. Top: a stock chart with a linear x-axis and log y-axis; bottom: with manual breaks

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)   # For the data set
library(scales) # For the trans and format functions
ggplot(Animals, aes(x=body, y=brain, label=rownames(Animals))) +
    geom_text(size=3) +
    annotation_logticks() +
    scale_x_log10(breaks = trans_breaks("log10", function(x) 10^x),
                  labels = trans_format("log10", math_format(10^.x))) +
    scale_y_log10(breaks = trans_breaks("log10", function(x) 10^x),
                  labels = trans_format("log10", math_format(10^.x)))
```

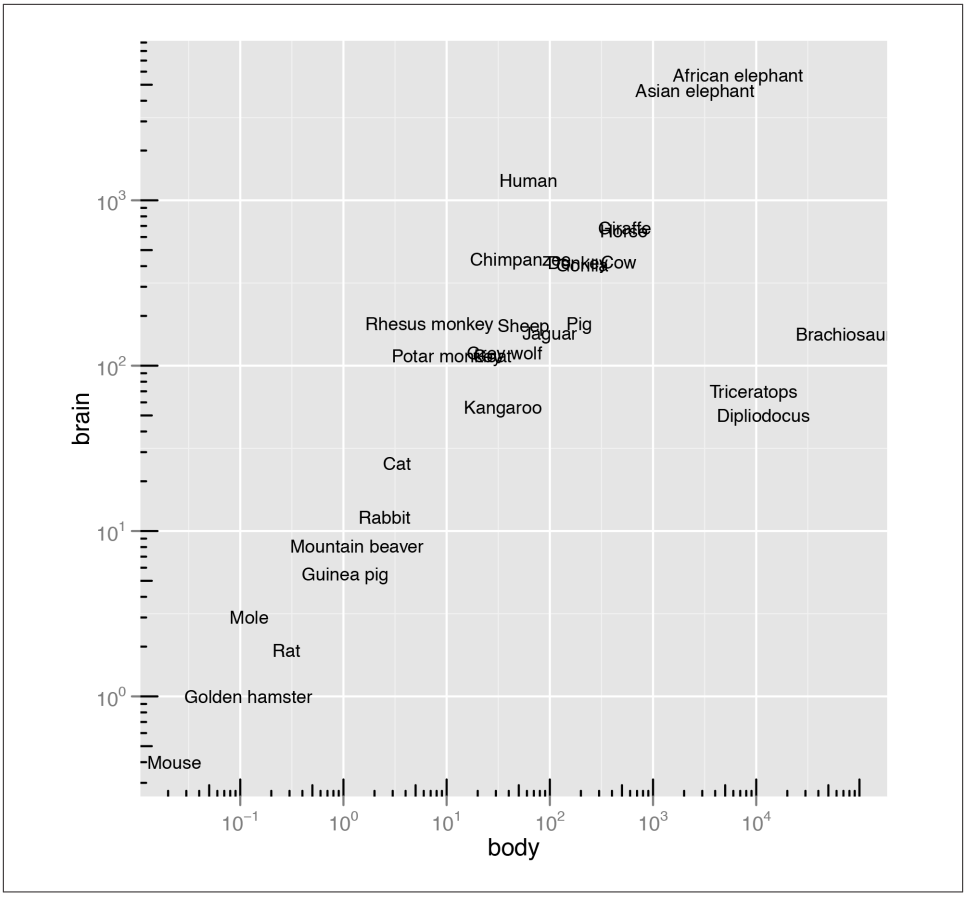

Figure 8-31. Log axes with diminishing tick marks

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, you can manually set the scale's `minor_breaks`. To do this, we need to set them to `log10(5*10^(minpow:maxpow))`, which reduces to `log10(5) + minpow:maxpow` (Figure 8-32):

```
ggplot(Animals, aes(x=body, y=brain, label=rownames(Animals))) +
    geom_text(size=3) +
    annotation_logticks() +
    scale_x_log10(breaks = trans_breaks("log10", function(x) 10^x),
                  labels = trans_format("log10", math_format(10^.x)),
                  minor_breaks = log10(5) + -2:5) +
    scale_y_log10(breaks = trans_breaks("log10", function(x) 10^x),
                  labels = trans_format("log10", math_format(10^.x)),
                  minor_breaks = log10(5) + -1:3) +
    coord_fixed() +
    theme_bw()
```

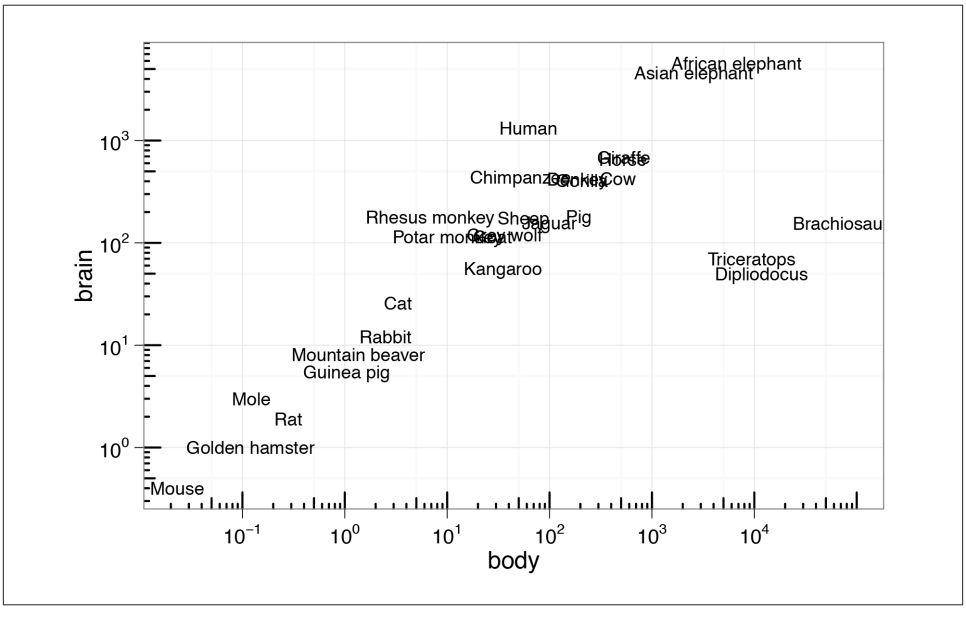

Figure 8-32. Log axes with ticks at each 5, and fixed coordinate ratio

See Also

For more on controlling the scaling ratio of the x- and y-axes, see Recipe 8.5.

8.16. Making a Circular Graph

Problem

You want to make a circular graph.

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) # For the data set
wind
```

```
 TimeUTC  Temp WindAvg WindMax WindDir SpeedCat DirCat
       0  3.54    9.52   10.39      89    10-15     90
       5  3.52    9.10    9.90      92     5-10     90
      10  3.53    8.73    9.51      92     5-10     90
  ...
    2335  6.74   18.98   23.81     250      >20    255
    2340  6.62   17.68   22.05     252      >20    255
    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, origin=-7.5) +
    coord_polar() +
    scale_x_continuous(limits=c(0,360))
```

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, origin=-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()
```

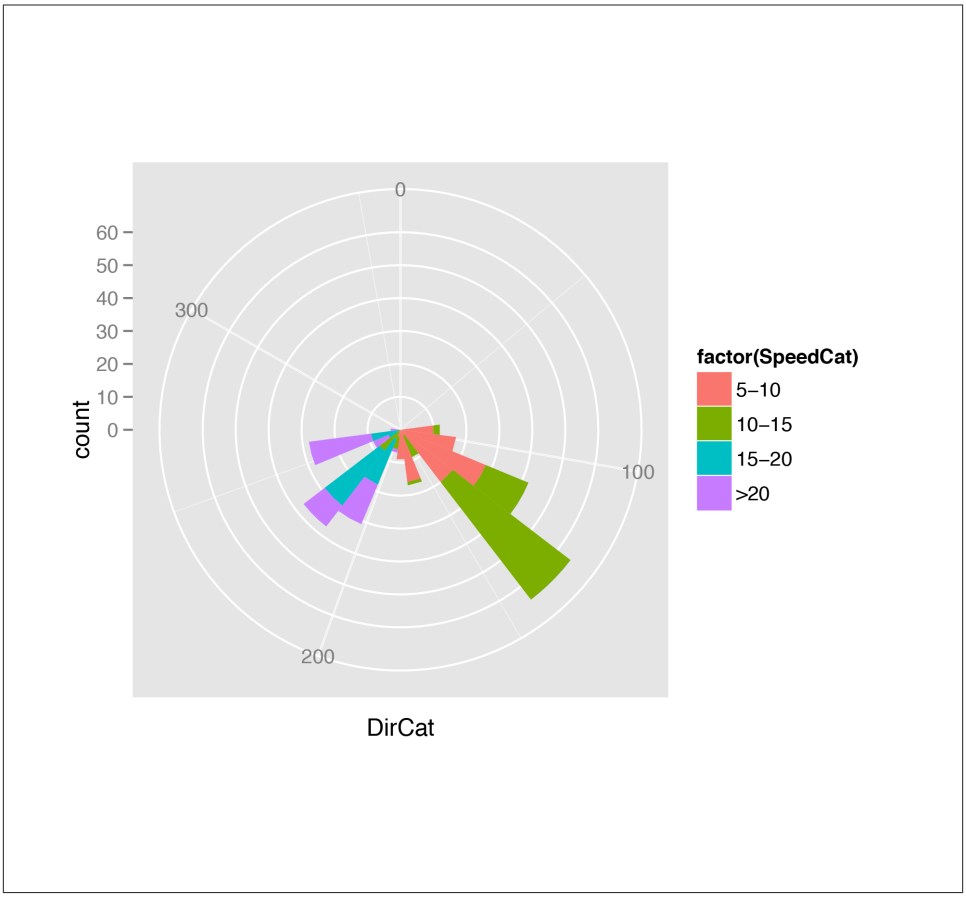

Figure 8-33. Polar plot

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.

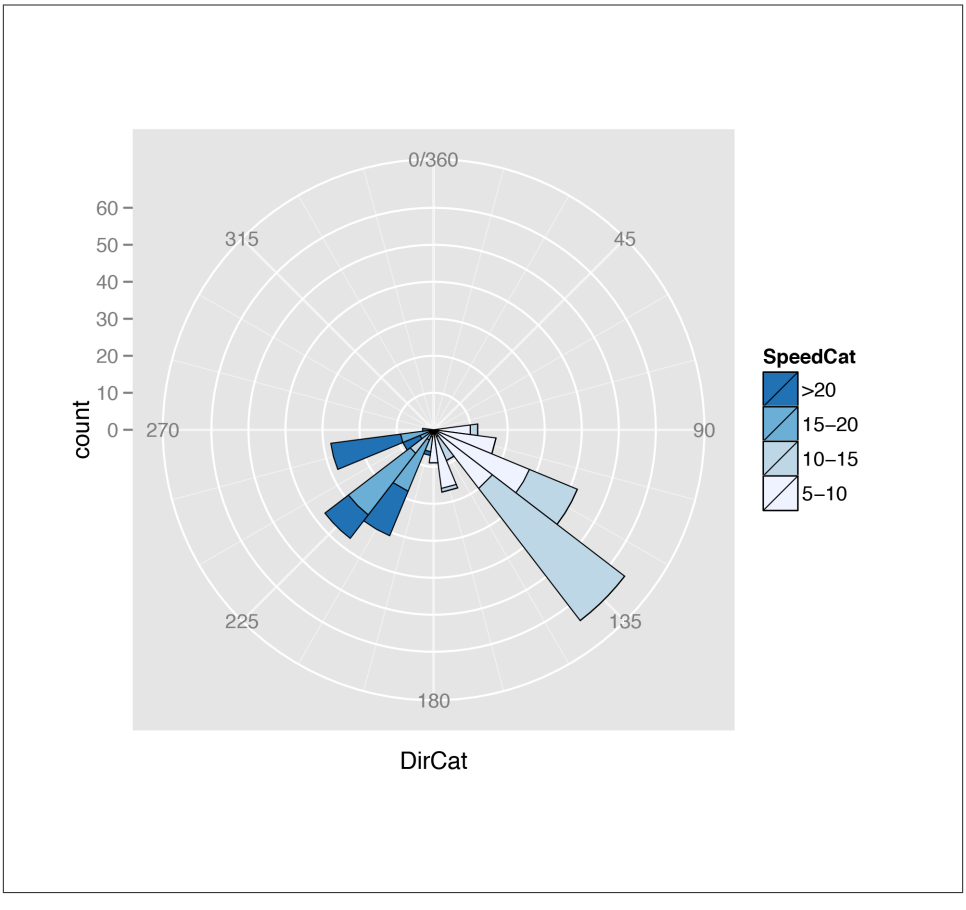

Figure 8-34. Polar plot with different colors and breaks

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).

We'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
md <- data.frame(deaths = as.numeric(mdeaths),
                 month  = as.numeric(cycle(mdeaths)))
```

```
# Calculate average number of deaths in each month
library(plyr) # For the ddply() function
md <- ddply(md, "month", summarise, deaths = mean(deaths))
md
```

```
 month   deaths
     1 2129.833
     2 2081.333
 ...
    11 1377.667
    12 1796.500
```

```
# Make the base plot
p <- ggplot(md, aes(x=month, y=deaths)) + geom_line() +
     scale_x_continuous(breaks=1:12)

# With coord_polar
p + coord_polar()
```

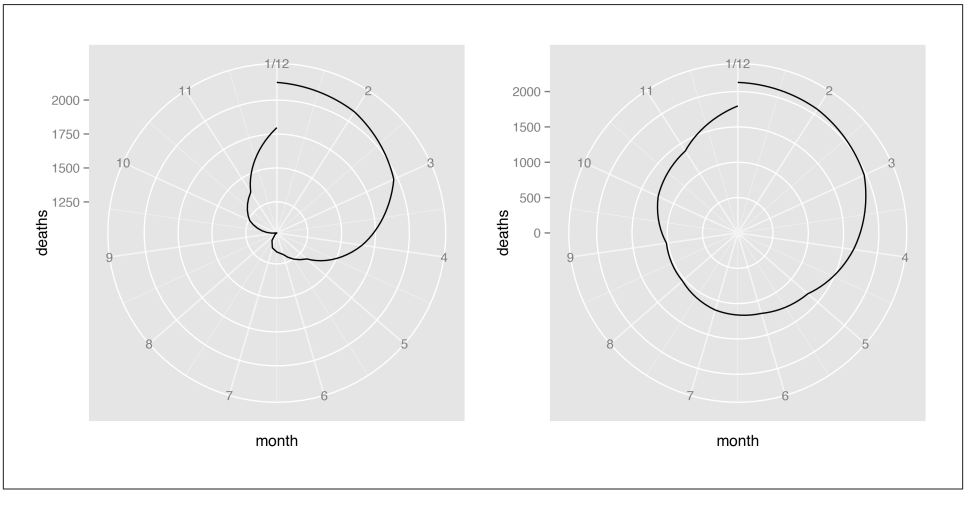

Figure 8-35. Left: polar plot with line (notice the data range of the radius); right: with the radius representing a data range starting from zero

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 setting 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
p + coord_polar() + ylim(0, max(md$deaths))
```

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):

```
p + coord_polar() + ylim(0, max(md$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
mdx <- md[md$month==12, ]
mdx$month <- 0
mdnew <- rbind(mdx, md)

# Make the same plot as before, but with the new data, by using %+%
p %+% mdnew + coord_polar() + ylim(0, max(md$deaths))
```

Figure 8-36. Left: polar plot with theta representing x values from 0 to 12; right: the gap is filled in by adding a dummy data point for month 0

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 `md` to `mdnew`.

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:

```
# Look at the structure
str(economics)
```

```
'data.frame':   478 obs. of  6 variables:
 $ date    : Date, format: "1967-06-30" "1967-07-31" ...
 $ pce     : num  508 511 517 513 518 ...
 $ pop     : int  198712 198911 199113 199311 199498 199657 199808 199920 ...
 $ psavert : num  9.8 9.8 9 9.8 9.7 9.4 9 9.5 8.9 9.6 ...
 $ uempmed : num  4.5 4.7 4.6 4.9 4.7 4.8 5.1 4.5 4.1 4.6 ...
 $ unemploy: int  2944 2945 2958 3143 3066 3018 2878 3001 2877 2709 ...
```

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()
```

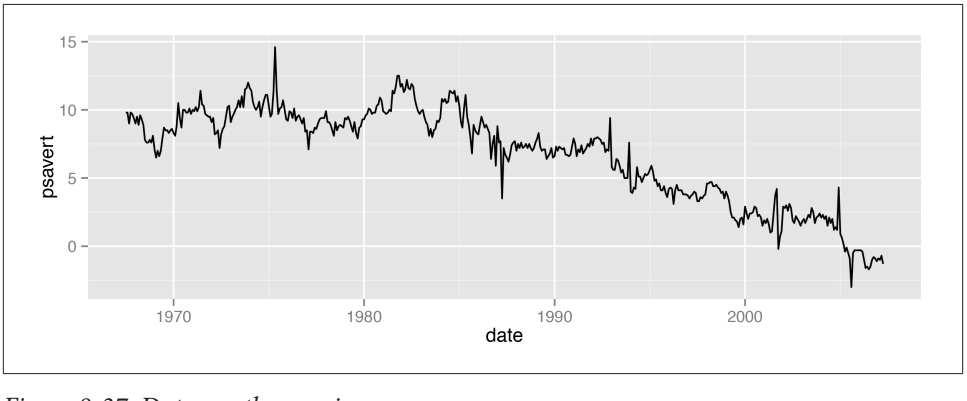

Figure 8-37. Dates on the x-axis

Discussion

ggplot2 handles two kinds of time-related objects: dates (objects of class `Date`) and datetimes (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):

```
# Take a subset of economics
econ <- subset(economics, date >= as.Date("1992-05-01") &
                          date <  as.Date("1993-06-01"))

# Base plot - without specifying breaks
p <- ggplot(econ, aes(x=date, y=psavert)) + geom_line()
p
```

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
p + scale_x_date(breaks=datebreaks) +
    theme(axis.text.x = element_text(angle=30, hjust=1))
```

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)
p + scale_x_date(breaks=datebreaks, labels=date_format("%Y %b")) +
    theme(axis.text.x = element_text(angle=30, hjust=1))
```

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)

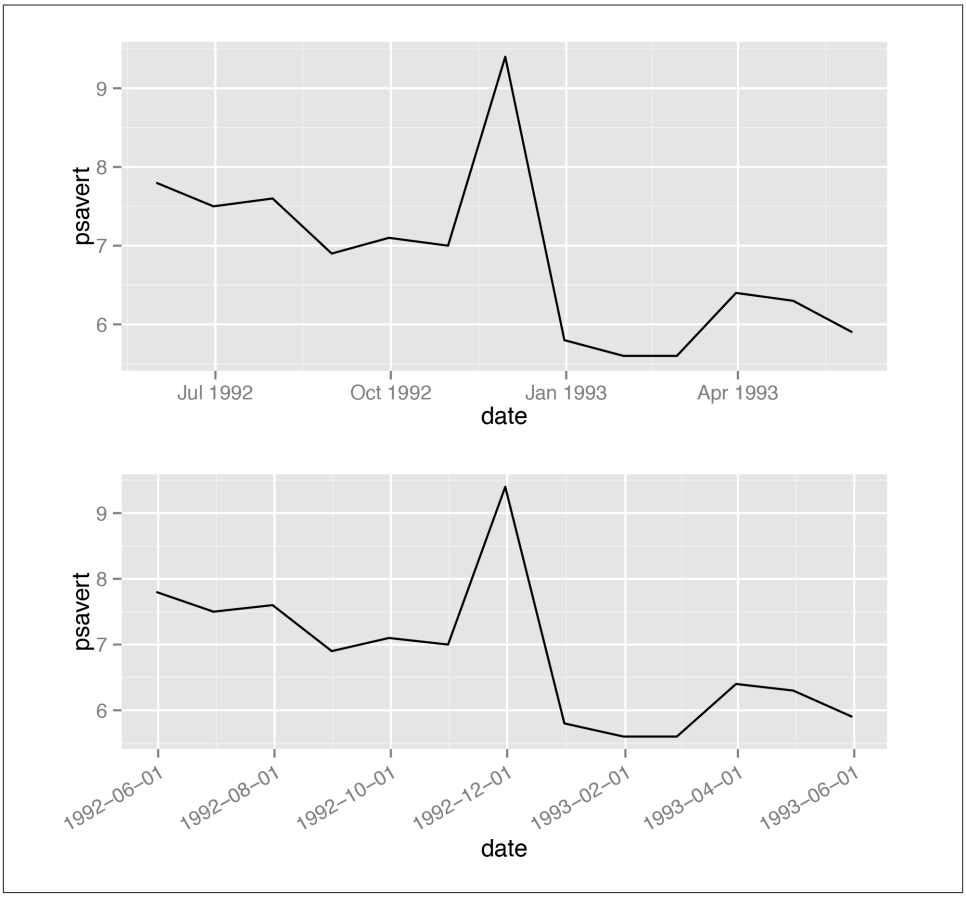

Figure 8-38. Top: with default breaks on the x-axis; bottom: with breaks specified

Option	Description
%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)

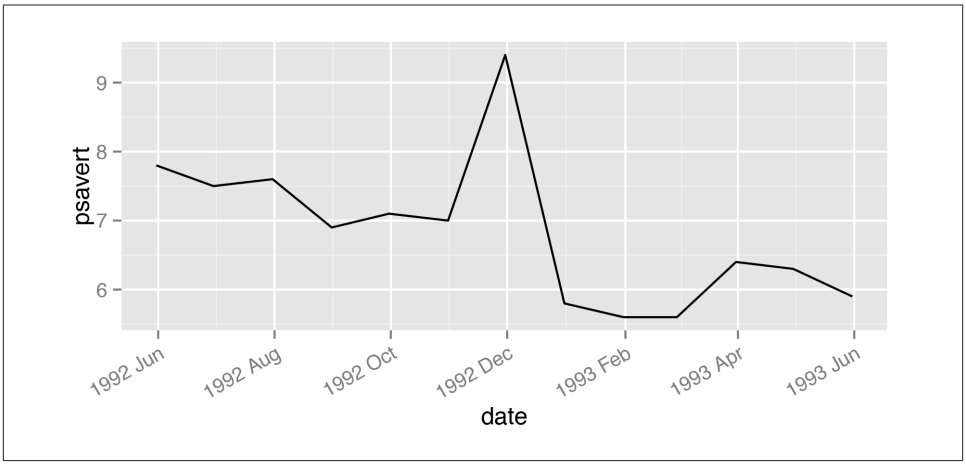

Figure 8-39. Line graph with date format specified

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)
```

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))
```

```
 "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)                       # Round to nearest second
    lab <- sprintf("%02d:%02d:%02d", h, m, s) # Format the strings as HH:MM:SS
```

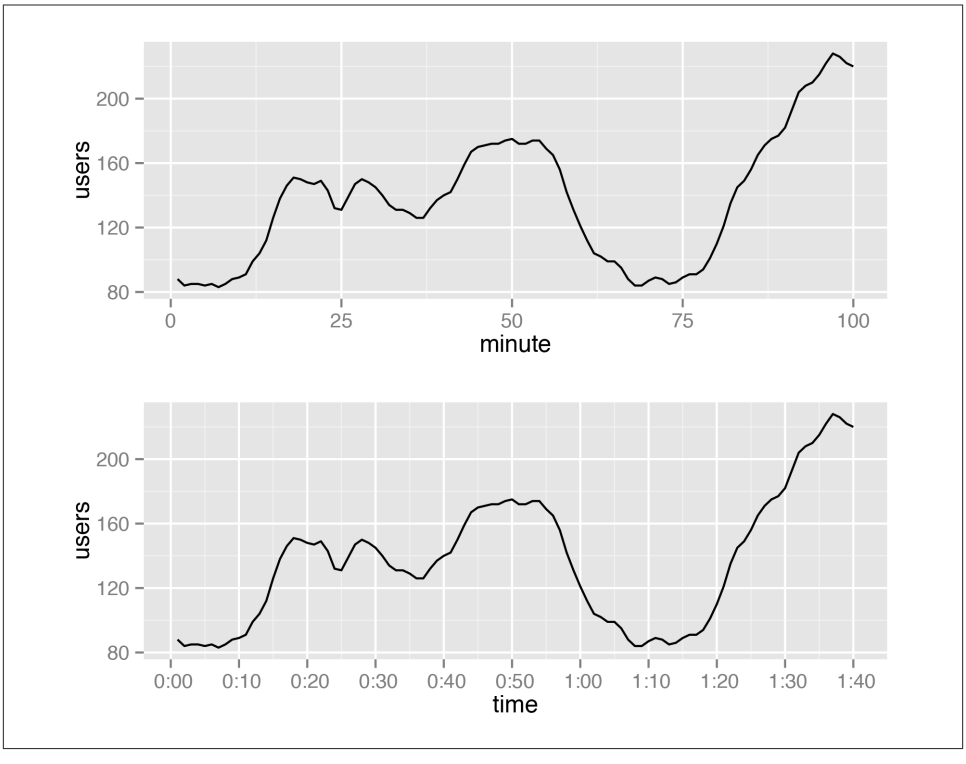

Figure 8-40. Top: relative times on x-axis; bottom: with formatted times

```
    lab <- sub("^00:", "", lab)              # Remove leading 00: if present
    lab <- sub("^0", "", lab)                # Remove leading 0 if present
    return(lab)
}
```

Running it on some sample numbers yields:

```
timeHMS_formatter(c(20, 3000, 3075, 3559.2, 3600, 3606, 7813.8))
```

```
 "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.

CHAPTER 9

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 non-data 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 set the title of a graph.

Solution

Set `title` with `ggtitle()`, as shown in Figure 9-1:

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

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

p + ggtitle("Age and Height of Schoolchildren")

# Use \n for a newline
p + ggtitle("Age and Height\nof Schoolchildren")
```

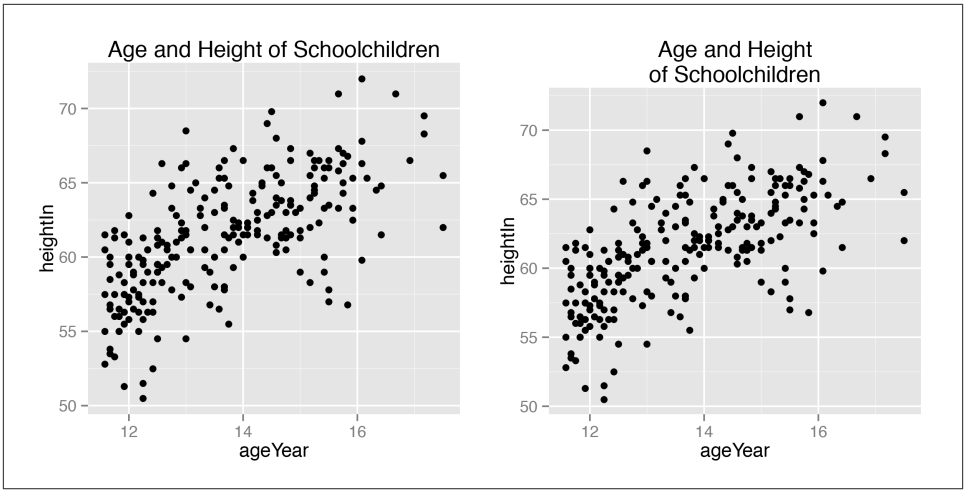

Figure 9-1. Left: scatter plot with a title added; right: with a /n for a newline

Discussion

`ggtitle()` is equivalent to using `labs(title = "Title text")`.

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 9-2). 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.

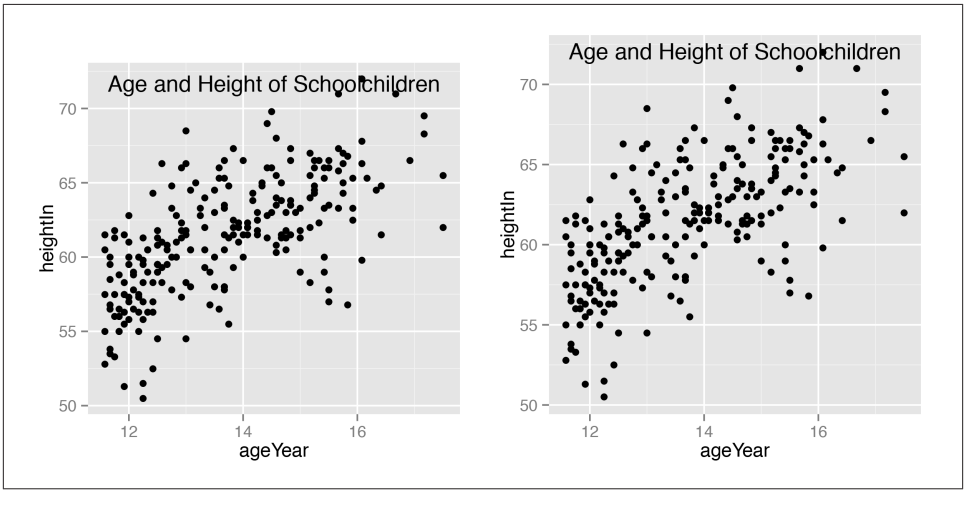

Figure 9-2. Left: title with ggtitle and a negative vjust() value (note the extra space above the plotting area); right: with a text annotation at the top of the figure

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
p + ggtitle("Age and Height of Schoolchildren") +
    theme(plot.title=element_text(vjust = -2.5))

# Use a text annotation instead
p + annotate("text", x=mean(range(heightweight$ageYear)), y=Inf,
             label="Age and Height of Schoolchildren", vjust=1.5, size=6)
```

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 9-3, left):

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

# Base plot
p <- ggplot(heightweight, aes(x=ageYear, y=heightIn)) + geom_point()

# Controlling appearance of theme items
p + theme(axis.title.x=element_text(size=16, lineheight=.9, family="Times",
                                    face="bold.italic", colour="red"))

p + 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.
```

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 9-3, right):

```
p + annotate("text", x=15, y=53, label="Some text", size = 7, family="Times",
        fontface="bold.italic", colour="red")

p + geom_text(aes(label=weightLb), size=4, family="Times", colour="red")

# For text geoms, font size is in mm
```

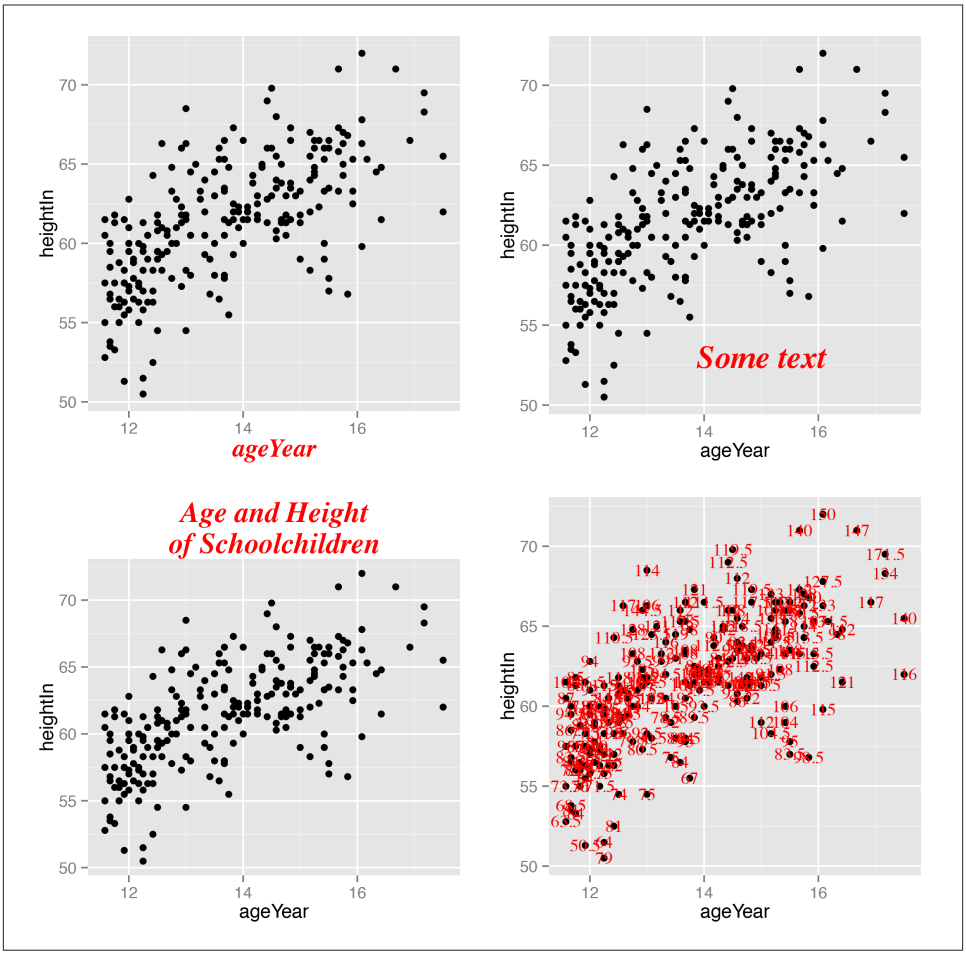

Figure 9-3. Counterclockwise from top left: axis.title.x, plot.title, geom_text(), and annotate("text")

Discussion

There are two kinds of text items in ggplot2: theme elements and text geoms. Theme elements are all the non-data elements in the plot: the title, legends, and axes. Text geoms are things that are part of the plot itself.

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 9-4.

Table 9-2. Theme items 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
`axis.ticks`	Appearance of tick labels on both axes
`axis.ticks.x`	Appearance of x tick labels
`axis.ticks.y`	Appearance of y tick labels
`legend.title`	Appearance of legend title
`legend.text`	Appearance of legend items
`plot.title`	Appearance of overall plot title
`strip.text`	Appearance of facet labels in both directions
`strip.text.x`	Appearance of horizontal facet labels
`strip.text.y`	Appearance of vertical facet labels

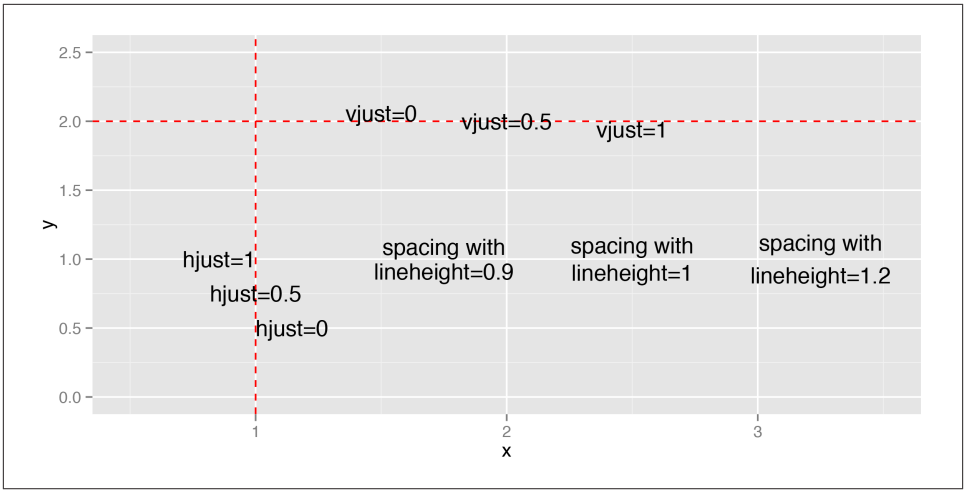

Figure 9-4. Aligning with hjust and vjust, and spacing with lineheight

9.3. Using Themes

Problem

You want to use premade themes to control the overall plot appearance.

Solution

To use a premade theme, add `theme_bw()` or `theme_grey()` (Figure 9-5):

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

# Base plot
p <- ggplot(heightweight, aes(x=ageYear, y=heightIn)) + geom_point()

# Grey theme (the default)
p + theme_grey()

# Black-and-white theme
p + theme_bw()
```

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.

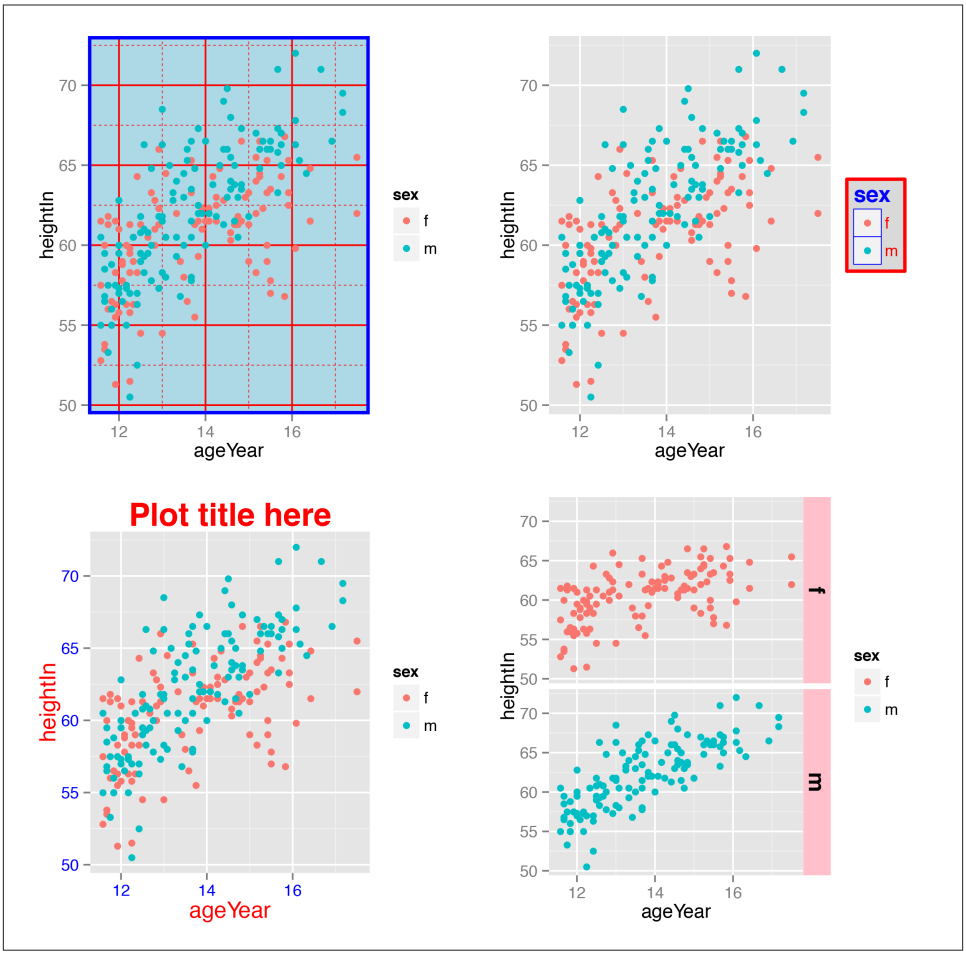

Figure 9-5. Left: scatter plot with theme_grey() (the default); right: with theme_bw()

The two included themes are `theme_grey()` and `theme_bw()`, but 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):

```
p + theme_grey(base_size=16, base_family="Times")
```

You can set the default theme for the current R session with `theme_set()`:

```
# Set default theme for current session
theme_set(theme_bw())

# This will use theme_bw()
```

```
p

# Reset the default theme back to theme_grey()
theme_set(theme_grey())
```

See Also

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 9-6):

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

# Base plot
p <- ggplot(heightweight, aes(x=ageYear, y=heightIn, colour=sex)) + geom_point()

# Options for the plotting area
p + 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 text items
p + 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 the legend
p + 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 facets
p + 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
```

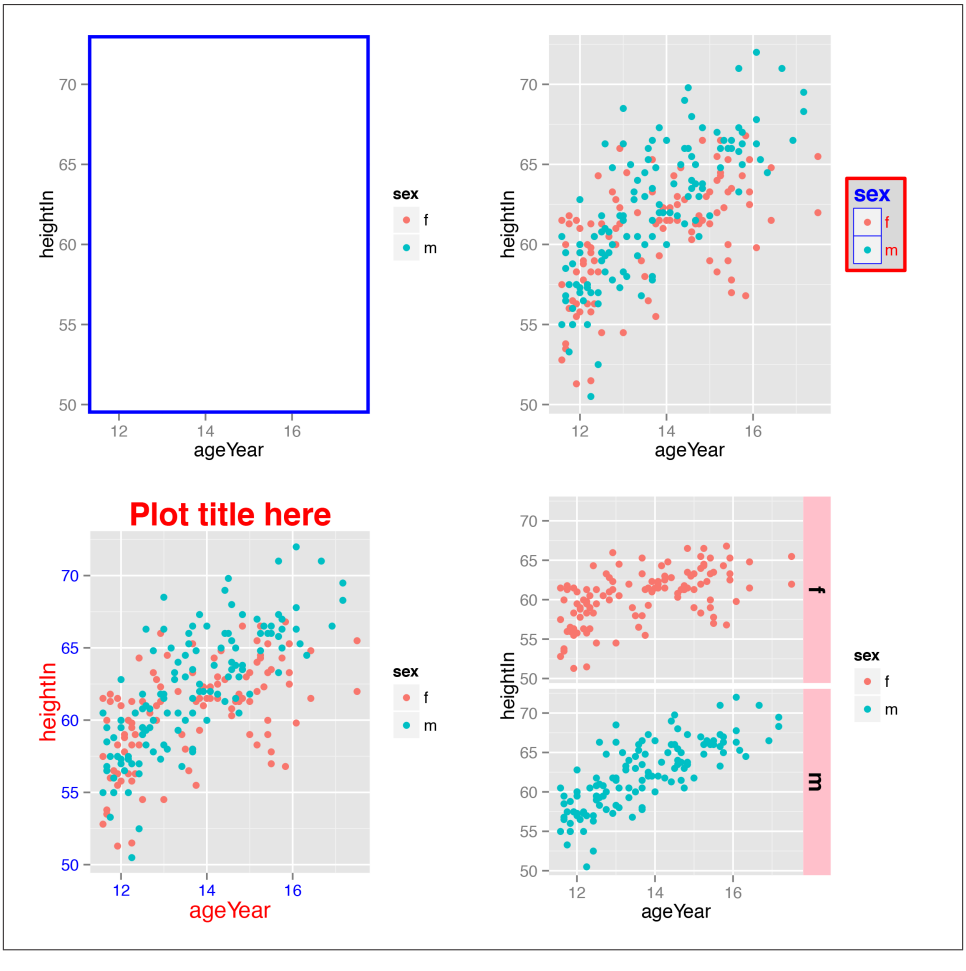

Figure 9-6. Clockwise from top left: modifying theme properties for the plotting area, the legend, the facets, and the text items

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
p + theme(axis.title.x = element_text(colour="red")) + theme_bw()

# theme() works if after a compete theme
p + 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
`text`	All text elements	`element_text()`
`rect`	All rectangular elements	`element_rect()`
`line`	All line elements	`element_line()`
`axis.line`	Lines along axes	`element_line()`
`axis.title`	Appearance of both axis labels	`element_text()`
`axis.title.x`	X-axis label appearance	`element_text()`
`axis.title.y`	Y-axis label appearance	`element_text()`
`axis.text`	Appearance of tick labels on both axes	`element_text()`
`axis.text.x`	X-axis tick label appearance	`element_text()`
`axis.text.y`	Y-axis tick label appearance	`element_text()`
`legend.background`	Background of legend	`element_rect()`
`legend.text`	Legend item appearance	`element_text()`
`legend.title`	Legend title appearance	`element_text()`
`legend.position`	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)
`panel.background`	Background of plotting area	`element_rect()`
`panel.border`	Border around plotting area	`element_rect(linetype="dashed")`
`panel.grid.major`	Major grid lines	`element_line()`
`panel.grid.major.x`	Major grid lines, vertical	`element_line()`
`panel.grid.major.y`	Major grid lines, horizontal	`element_line()`
`panel.grid.minor`	Minor grid lines	`element_line()`
`panel.grid.minor.x`	Minor grid lines, vertical	`element_line()`
`panel.grid.minor.y`	Minor grid lines, horizontal	`element_line()`

Name	Description	Element type
`plot.background`	Background of the entire plot	`element_rect(fill = "white", colour = NA)`
`plot.title`	Title text appearance	`element_text()`
`strip.background`	Background of facet labels	`element_rect()`
`strip.text`	Text appearance for vertical and horizontal facet labels	`element_text()`
`strip.text.x`	Text appearance for horizontal facet labels	`element_text()`
`strip.text.y`	Text appearance for vertical facet labels	`element_text()`

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 9-7):

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

# Base plot
p <- ggplot(heightweight, aes(x=ageYear, y=heightIn)) + geom_point()

# Plot with modified theme
p + mytheme
```

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.

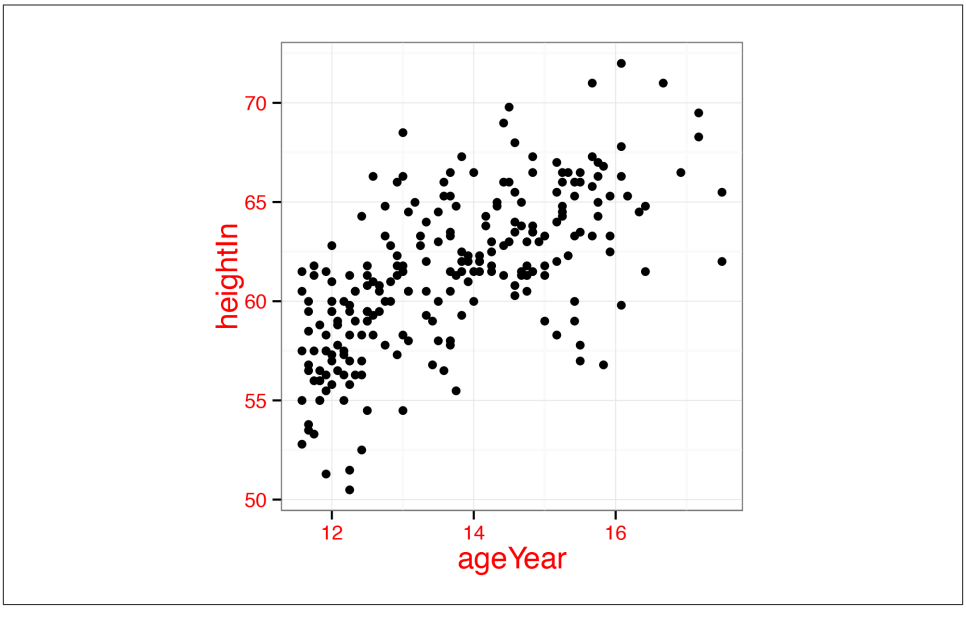

Figure 9-7. A modified default theme

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 9-8 (left):

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

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

p + theme(panel.grid.major = element_blank(),
          panel.grid.minor = element_blank())
```

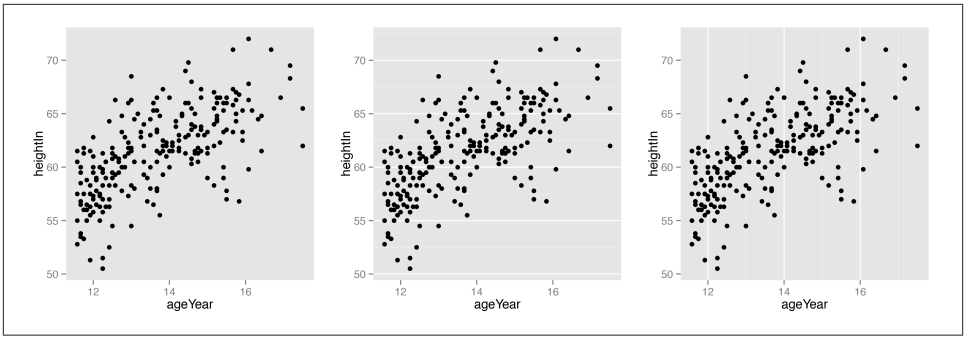

Figure 9-8. Left: no grid lines; middle: no vertical lines; right: no horizontal lines

Discussion

It's possible to hide just the vertical or horizontal grid lines, as shown in the middle and righthand graphs in Figure 9-8, 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)
p + theme(panel.grid.major.x = element_blank(),
          panel.grid.minor.x = element_blank())

# Hide the horizontal grid lines (which intersect with the y-axis)
p + theme(panel.grid.major.y = element_blank(),
          panel.grid.minor.y = element_blank())
```

CHAPTER 10
Legends

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):

```
# The base plot (with legend)
p <- ggplot(PlantGrowth, aes(x=group, y=weight, fill=group)) + geom_boxplot()
p

# Remove the legend for fill
p + guides(fill=FALSE)
```

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
p + 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:

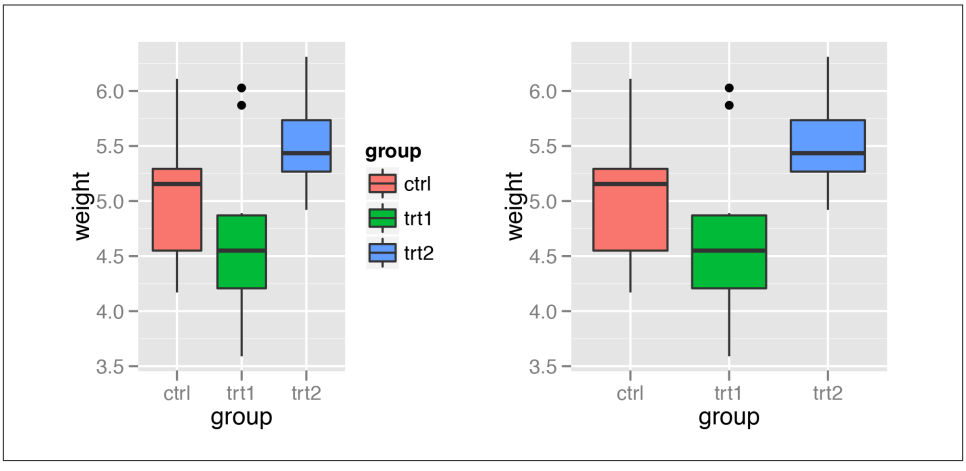

Figure 10-1. Left: default appearance; right: with legend removed

```
p + 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):

```
p <- ggplot(PlantGrowth, aes(x=group, y=weight, fill=group)) + geom_boxplot() +
    scale_fill_brewer(palette="Pastel2")

p + theme(legend.position="top")
```

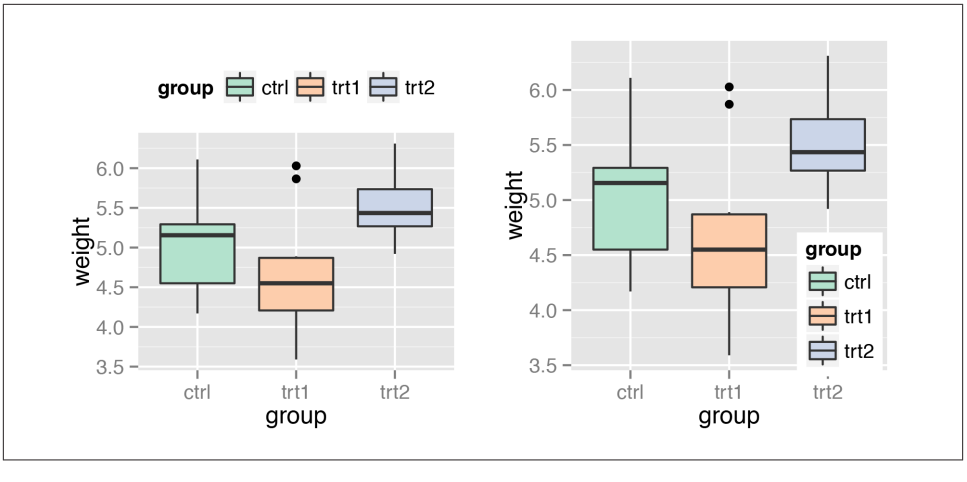

Figure 10-2. Left: legend on top; right: legend inside of graphing area

The legend can also be placed inside the graphing area by specifying a coordinate position, as in `legend.position=c(1,0)` (Figure 10-2, right). The coordinate space starts at (0, 0) in the bottom left and goes to (1, 1) in the top 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 graphing area (1,0):

```
p + theme(legend.position=c(1,0), legend.justification=c(1,0))
```

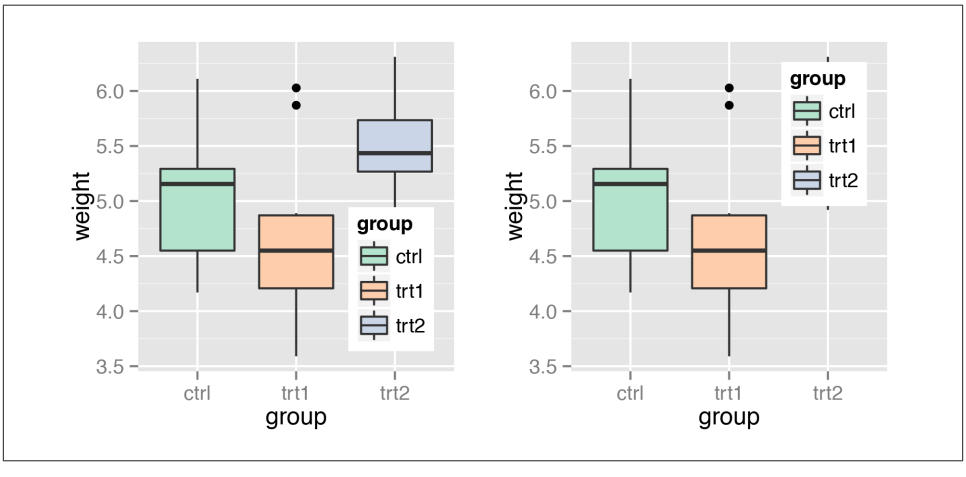

Figure 10-3. Left: legend in bottom-right corner; right: legend in top-right corner.

And this will place the top-right corner of the legend in the top-right corner of the graphing area, as in the graph on the right in Figure 10-3:

```
p + theme(legend.position=c(1,1), legend.justification=c(1,1))
```

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):

```
p + 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):

```
p + theme(legend.position=c(.85,.2)) +
    theme(legend.background=element_blank()) +  # Remove overall border
    theme(legend.key=element_blank())           # Remove border around each item
```

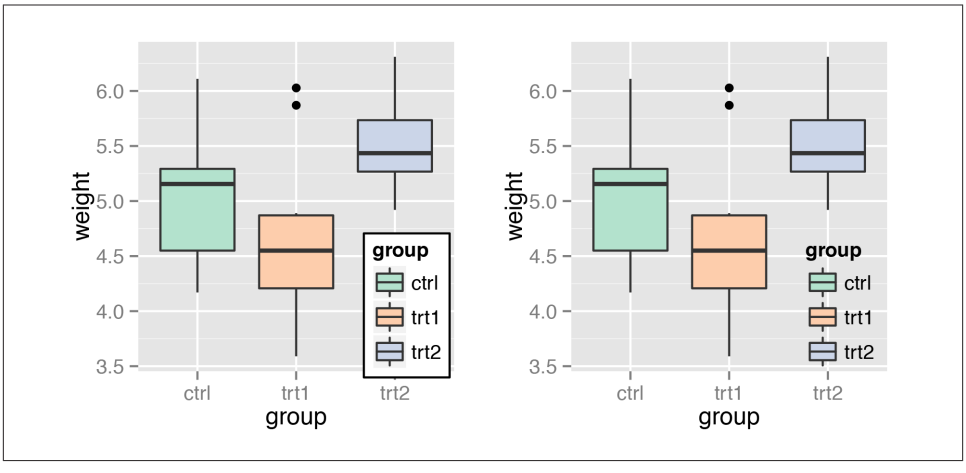

Figure 10-4. Left: legend with opaque background and outline; right: with no background or outlines

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):

```
# The base plot
p <- ggplot(PlantGrowth, aes(x=group, y=weight, fill=group)) + geom_boxplot()
p

# Change the order of items
p + scale_fill_discrete(limits=c("trt1", "trt2", "ctrl"))
```

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).

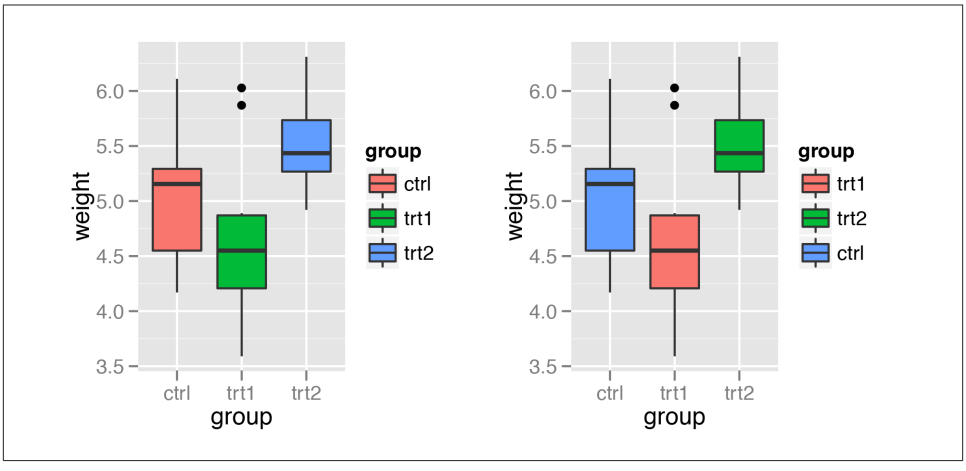

Figure 10-5. Left: default order for legend; right: modified order

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):

```
p + scale_fill_grey(start=.5, end=1, limits=c("trt1", "trt2", "ctrl"))
```

Or we could use a palette from RColorBrewer (Figure 10-6, right):

```
p + scale_fill_brewer(palette="Pastel2", limits=c("trt1", "trt2", "ctrl"))
```

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()`

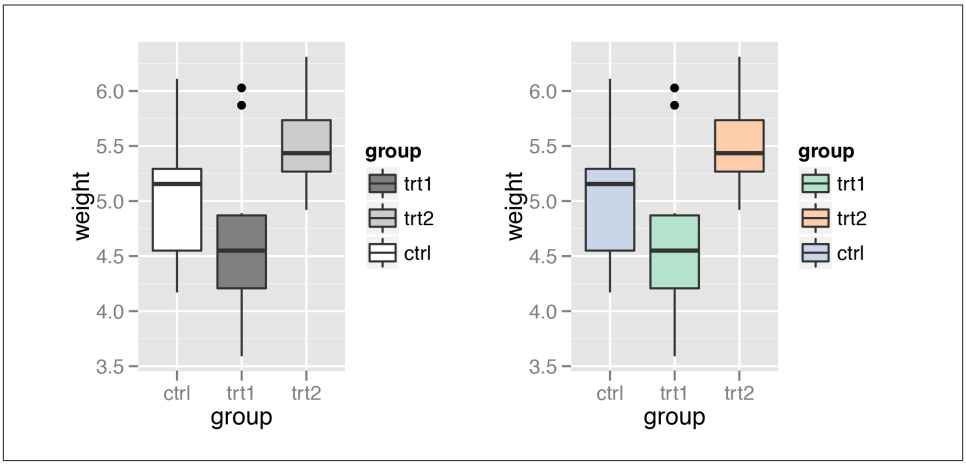

Figure 10-6. Left: modified order with a grey palette; right: with a palette from RColorBrewer

- `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`):

```
# The base plot
p <- ggplot(PlantGrowth, aes(x=group, y=weight, fill=group)) + geom_boxplot()
p

# Reverse the legend order
p + guides(fill=guide_legend(reverse=TRUE))
```

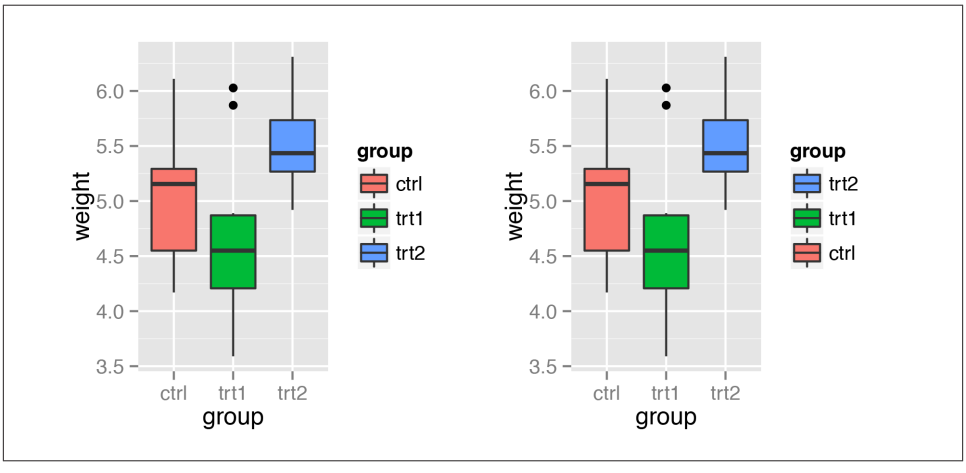

Figure 10-7. Left: default order for legend; right: reversed order

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):

```
# The base plot
p <- ggplot(PlantGrowth, aes(x=group, y=weight, fill=group)) + geom_boxplot()
p

# Set the legend title to "Condition"
p + labs(fill="Condition")
```

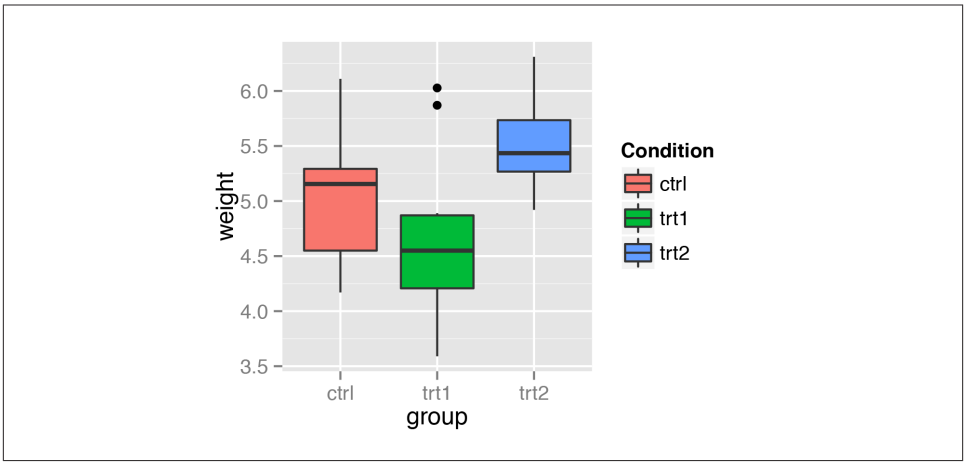

Figure 10-8. With the legend title set to "Condition"

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:

```
p + 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) # For the data set

# Make the base plot
hw <- ggplot(heightweight, aes(x=ageYear, y=heightIn, colour=sex)) +
      geom_point(aes(size=weightLb)) + scale_size_continuous(range=c(1,4))

hw

# With new legend titles
hw + labs(colour="Male/Female", size="Weight\n(pounds)")
```

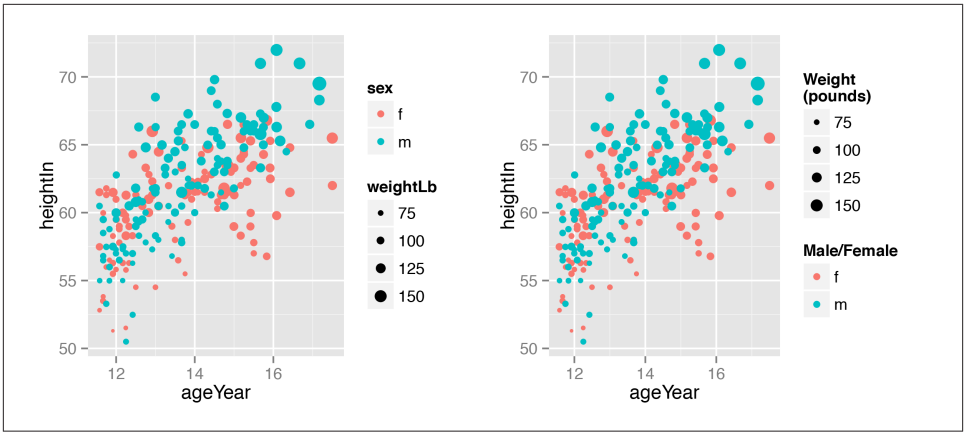

Figure 10-9. Left: two legends with original titles; right: with new titles

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):

```
hw1 <- ggplot(heightweight, aes(x=ageYear, y=heightIn, shape=sex, colour=sex)) +
       geom_point()

hw1
```

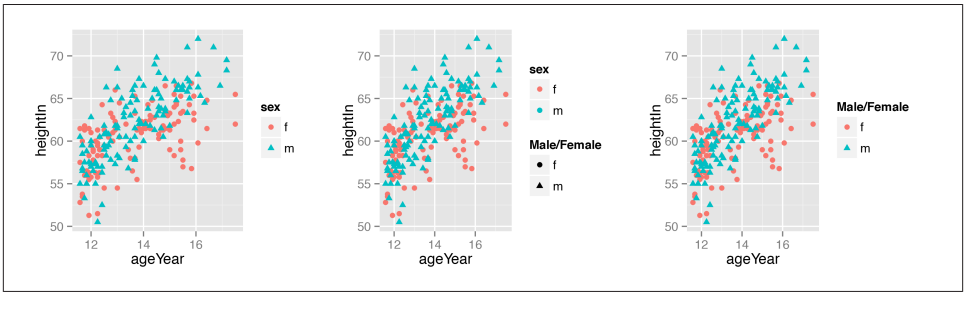

Figure 10-10. Left: default legend with a variable mapped to shape and colour; middle: with shape renamed; right: with both shape and colour renamed

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
hw1 + labs(shape="Male/Female")

# Change both shape and colour
hw1 + 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:

```
p + guides(fill=guide_legend(title="Condition"))
```

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):

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

p + theme(legend.title=element_text(face="italic", family="Times", colour="red",
                                    size=14))
```

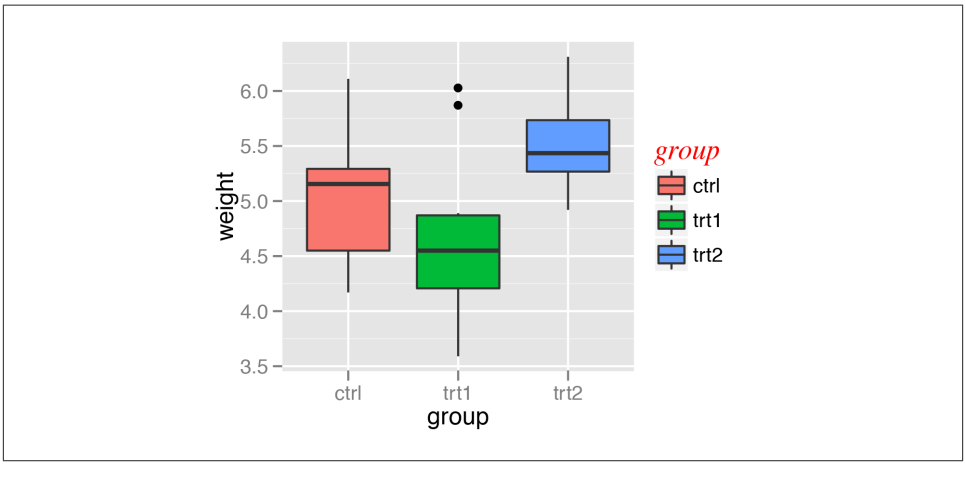

Figure 10-11. Customized legend title appearance

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:

```
p + 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))
```

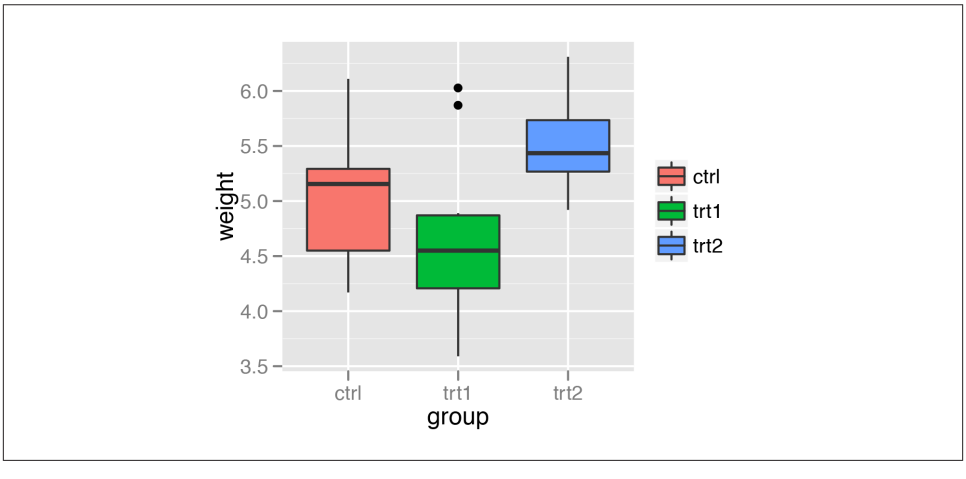

Figure 10-12. Box plot with no legend title

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) # For the data set

# The base plot
p <- ggplot(PlantGrowth, aes(x=group, y=weight, fill=group)) + geom_boxplot()

# Change the legend labels
p + scale_fill_discrete(labels=c("Control", "Treatment 1", "Treatment 2"))
```

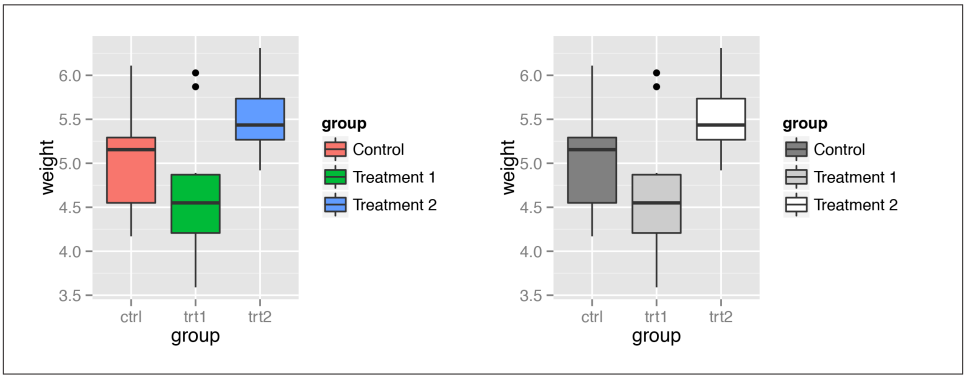

Figure 10-13. Left: manually specified legend labels with the default discrete scale; right: manually specified labels with a different scale

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:

```
p + scale_fill_grey(start=.5, end=1,
                    labels=c("Control", "Treatment 1", "Treatment 2"))
```

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):

```
p + scale_fill_discrete(limits=c("trt1", "trt2", "ctrl"),
                        labels=c("Treatment 1", "Treatment 2", "Control"))
```

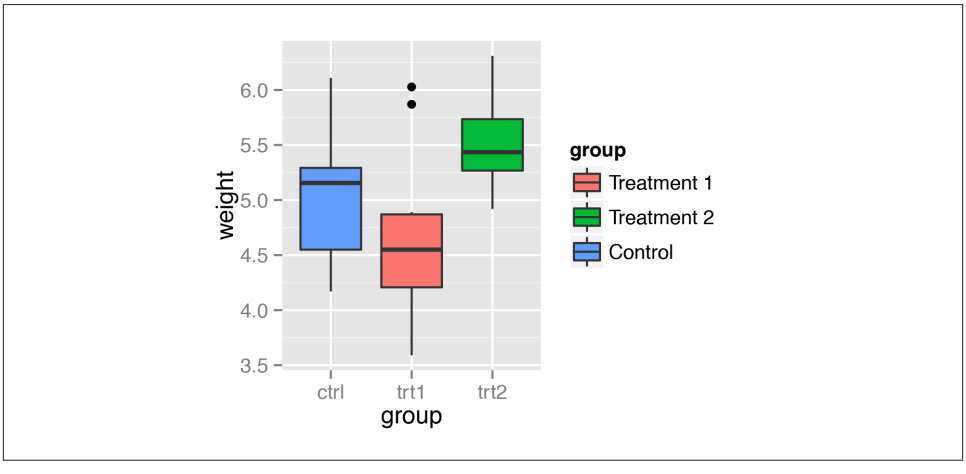

Figure 10-14. Modified legend label order and manually specified labels (note that the x-axis labels and their order are unchanged)

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:

```
# The base plot
p <- ggplot(heightweight, aes(x=ageYear, y=heightIn, shape=sex, colour=sex)) +
     geom_point()
p

# Change the labels for one scale
p + scale_shape_discrete(labels=c("Female", "Male"))

# Change the labels for both scales
p + scale_shape_discrete(labels=c("Female", "Male")) +
    scale_colour_discrete(labels=c("Female", "Male"))
```

Figure 10-15. Left: a variable mapped to shape and colour; middle: with new labels for shape; right: with new labels for both shape and colour

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_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.

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):

```
# The base plot
p <- ggplot(PlantGrowth, aes(x=group, y=weight, fill=group)) + geom_boxplot()

# Change the legend label appearance
p + theme(legend.text=element_text(face="italic", family="Times", colour="red",
                                   size=14))
```

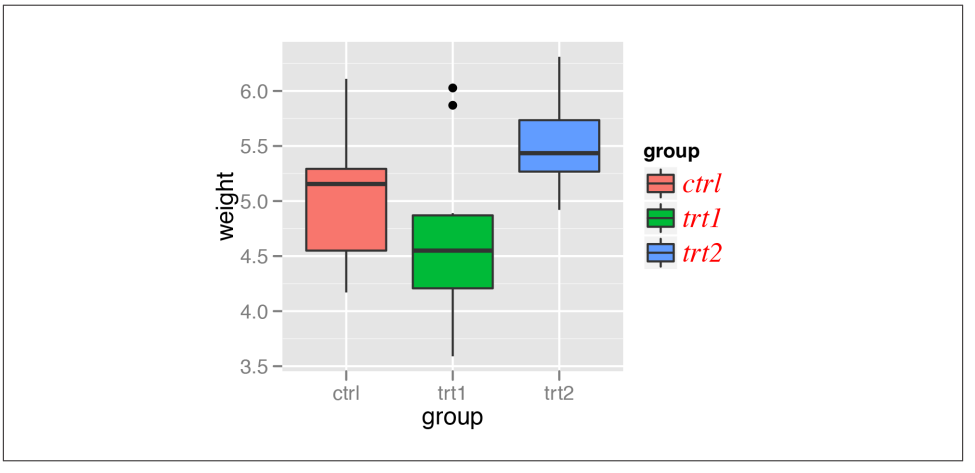

Figure 10-16. Customized legend label appearance

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
p + guides(fill=guide_legend(label.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):

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

# Labels that have more than one line
p + scale_fill_discrete(labels=c("Control", "Type 1\ntreatment",
                                 "Type 2\ntreatment"))
```

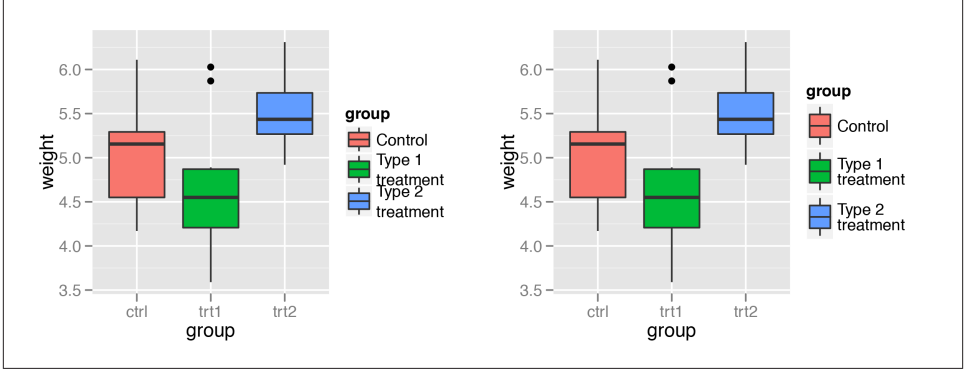

Figure 10-17. Left: multiline legend labels; right: with increased key height and reduced line spacing

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)
p + scale_fill_discrete(labels=c("Control", "Type 1\ntreatment",
                                 "Type 2\ntreatment")) +
    theme(legend.text=element_text(lineheight=.8),
          legend.key.height=unit(1, "cm"))
```

CHAPTER 11
Facets

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, color, 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):

```
# The base plot
p <- ggplot(mpg, aes(x=displ, y=hwy)) + geom_point()

# Faceted by drv, in vertically arranged subpanels
p + facet_grid(drv ~ .)

# Faceted by cyl, in horizontally arranged subpanels
```

```
p + facet_grid(. ~ cyl)

# Split by drv (vertical) and cyl (horizontal)
p + facet_grid(drv ~ cyl)
```

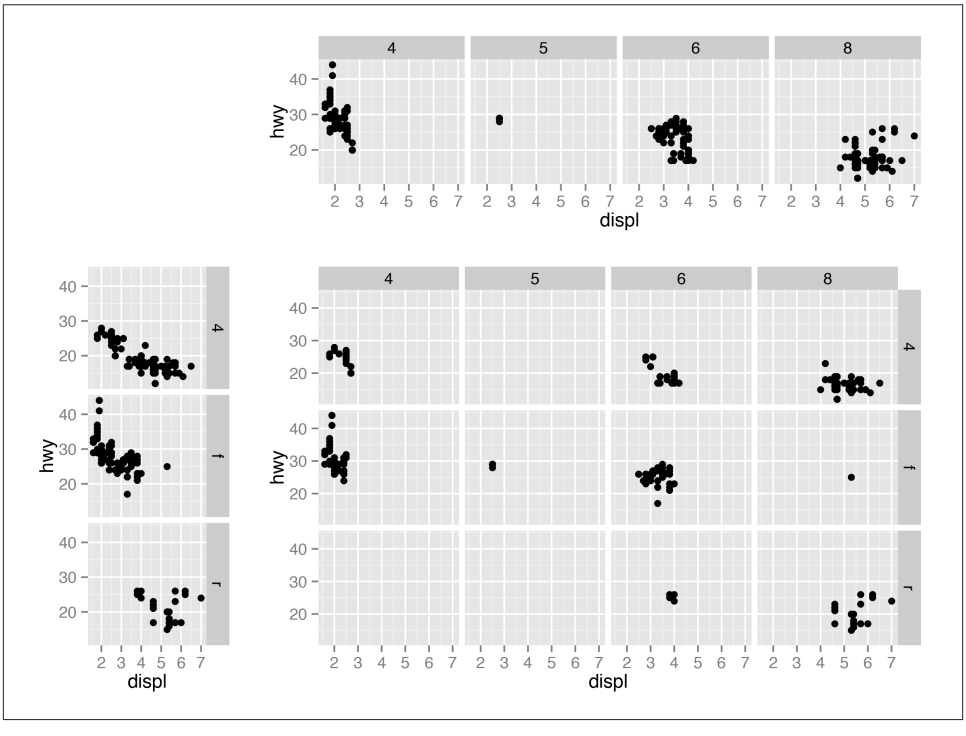

Figure 11-1. Top: faceting horizontally by drv; bottom left: faceting vertically by cyl; bottom right: faceting in both directions, with both variables

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 there is nothing before the tilde
p + facet_wrap( ~ 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`:

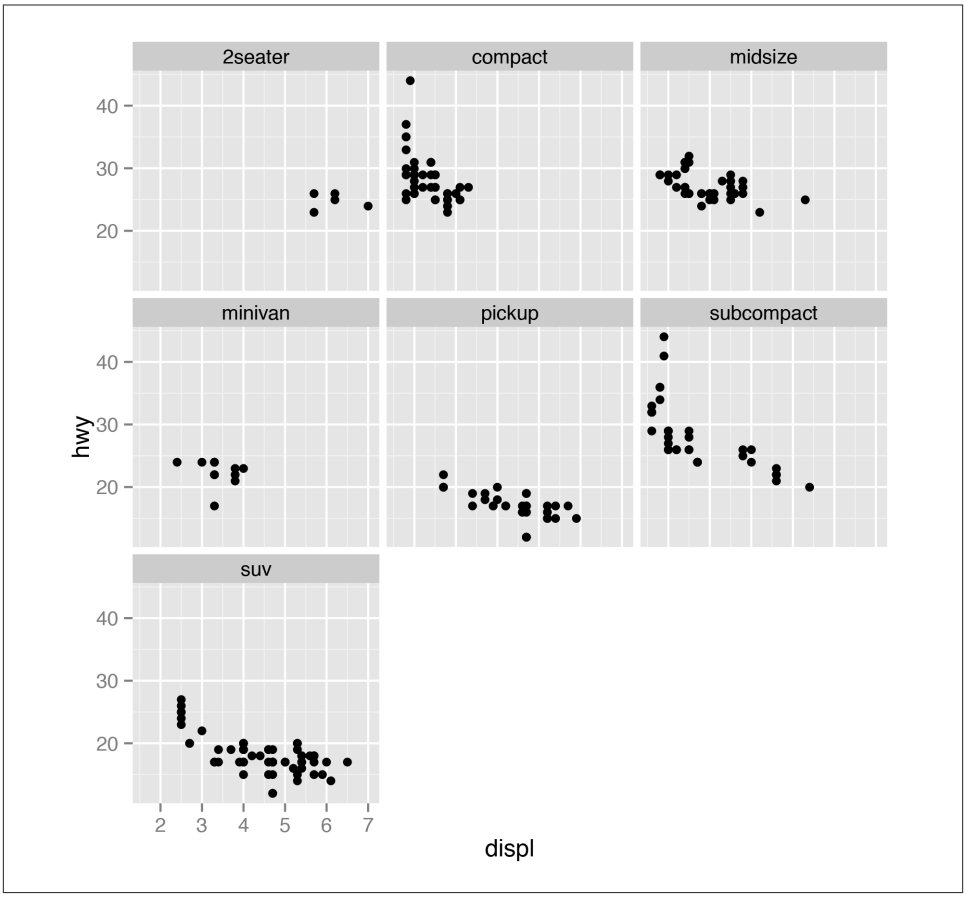

Figure 11-2. A scatter plot with facet_wrap() on class

```
# These will have the same result: 2 rows and 4 cols
p + facet_wrap( ~ class, nrow=2)
p + 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):

```
# The base plot
p <- ggplot(mpg, aes(x=displ, y=hwy)) + geom_point()

# With free y scales
p + facet_grid(drv ~ cyl, scales="free_y")

# With free x and y scales
p + facet_grid(drv ~ cyl, scales="free")
```

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):

```
mpg2 <- mpg  # Make a copy of the original data

# Rename 4 to 4wd, f to Front, r to Rear
levels(mpg2$drv)[levels(mpg2$drv)=="4"]  <- "4wd"
levels(mpg2$drv)[levels(mpg2$drv)=="f"]  <- "Front"
```

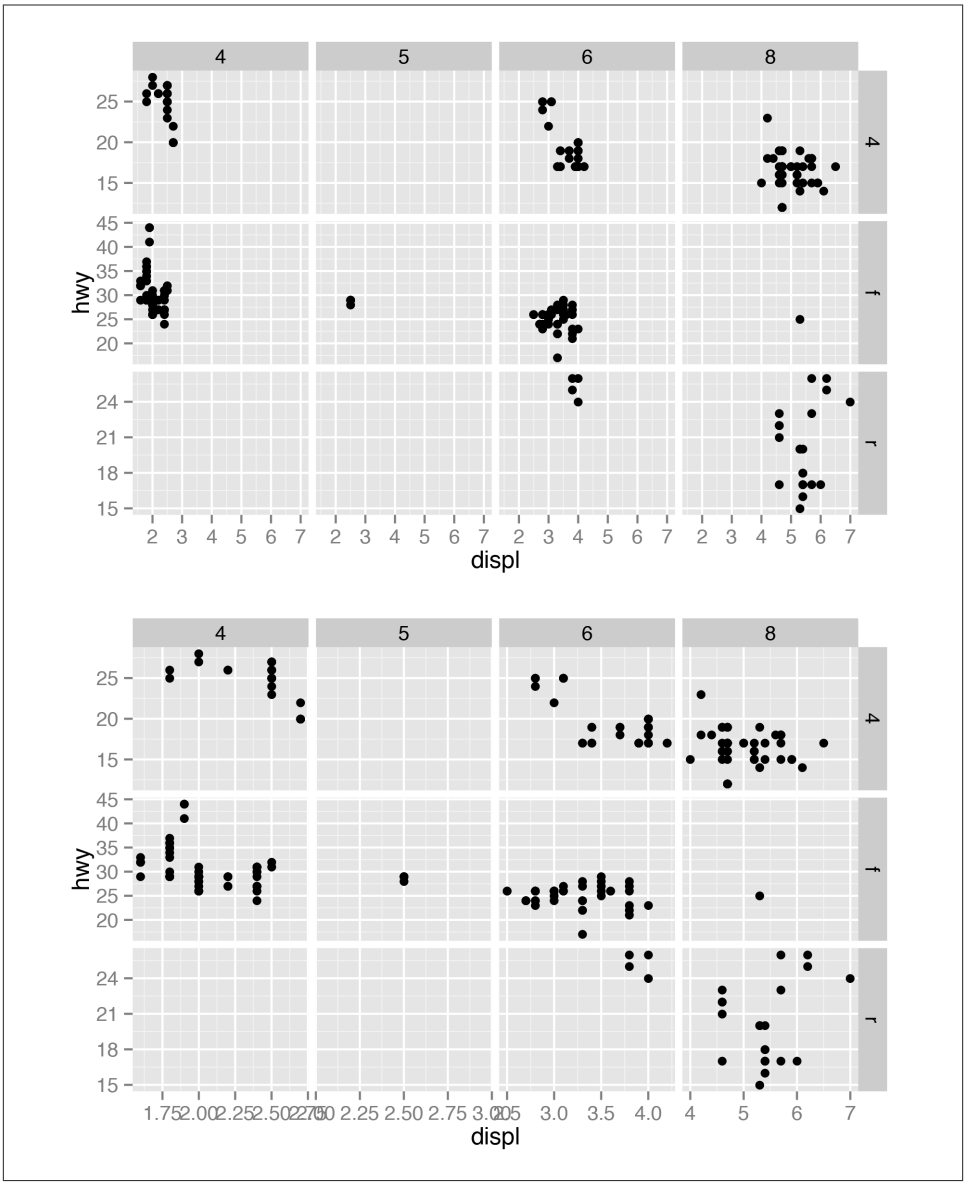

Figure 11-3. Top: with free y scales; bottom: with free x and y scales

```
levels(mpg2$drv)[levels(mpg2$drv)=="r"]  <- "Rear"

# Plot the new data
ggplot(mpg2, aes(x=displ, y=hwy)) + geom_point() + facet_grid(drv ~ .)
```

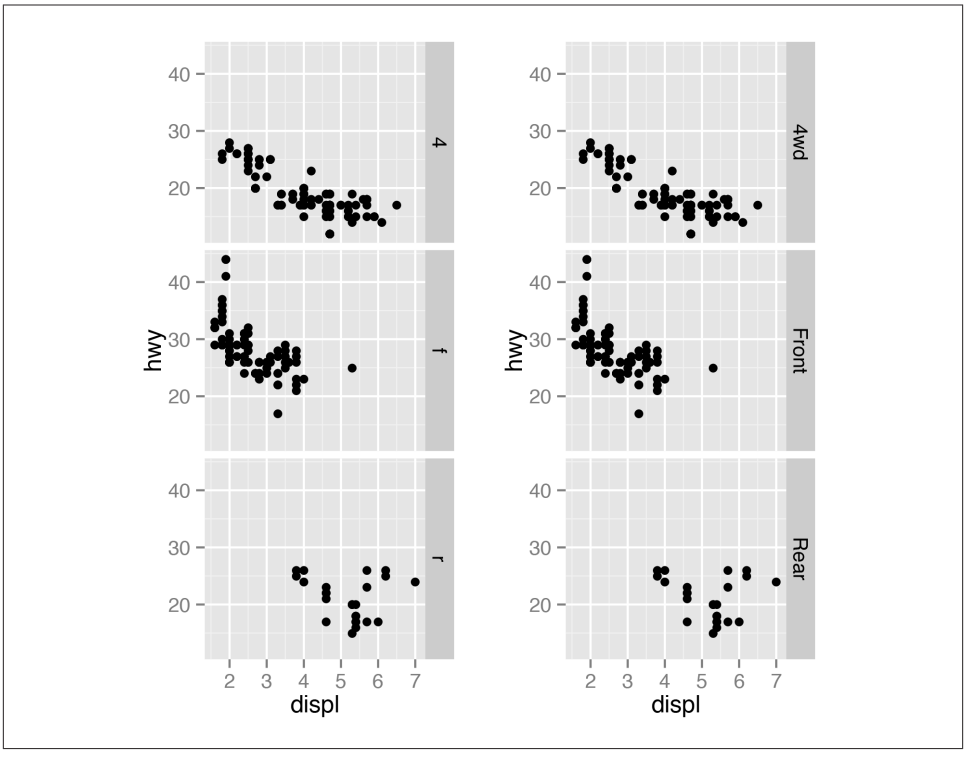

Figure 11-4. Left: default facet labels; right: modified facet labels

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()` (but not `facet_wrap()`, at this time), 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(mpg2, 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):

```
mpg3 <- mpg

levels(mpg3$drv)[levels(mpg3$drv)=="4"]  <- "4^{wd}"
```

```
levels(mpg3$drv)[levels(mpg3$drv)=="f"]  <- "- Front %.% e^{pi * i}"
levels(mpg3$drv)[levels(mpg3$drv)=="r"]  <- "4^{wd} - Front"

ggplot(mpg3, aes(x=displ, y=hwy)) + geom_point() +
    facet_grid(drv ~ ., labeller = label_parsed)
```

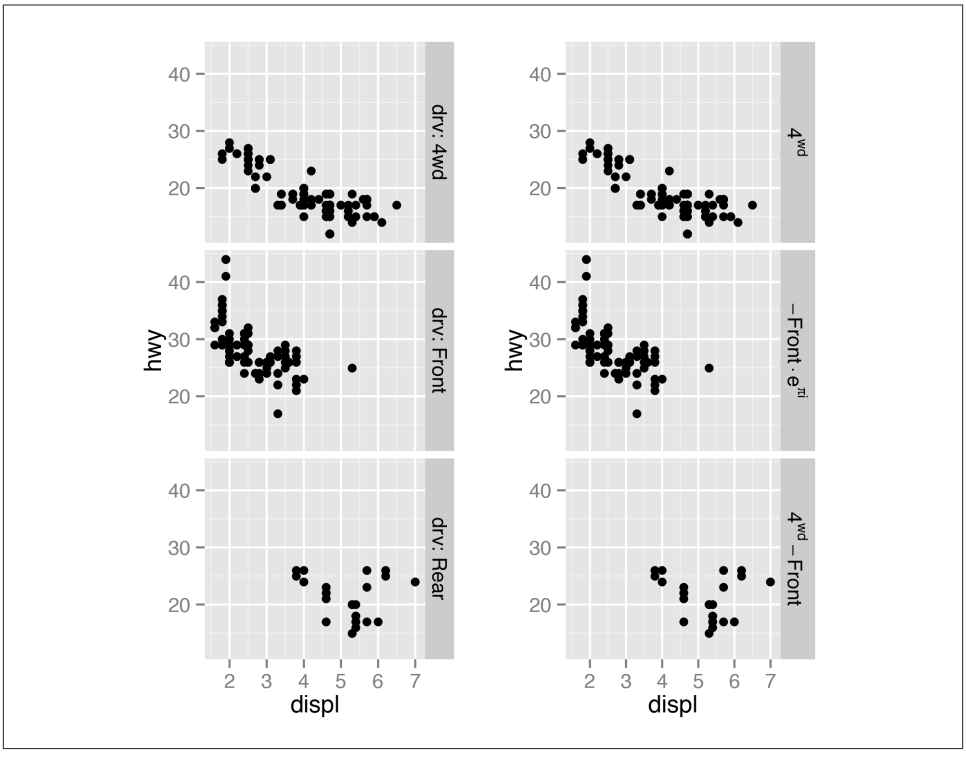

Figure 11-5. Left: with label_both(); right: with label_parsed() for mathematical expressions

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)  # For the data set

ggplot(cabbage_exp, aes(x=Cultivar, y=Weight)) + geom_bar(stat="identity") +
    facet_grid(. ~ Date) +
    theme(strip.text = element_text(face="bold", size=rel(1.5)),
          strip.background = element_rect(fill="lightblue", colour="black",
                                          size=1))
```

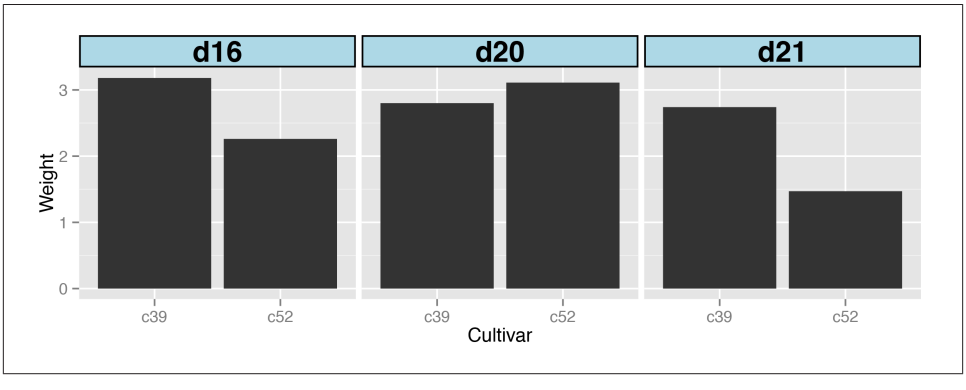

Figure 11-6. Customized appearance for facet labels

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.

CHAPTER 12

Using Colors in Plots

In ggplot2's implementation of the grammar of graphics, color 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):

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

library(MASS) # For the data set
ggplot(birthwt, aes(x=bwt)) + geom_histogram(fill="red"), colour="black"
```

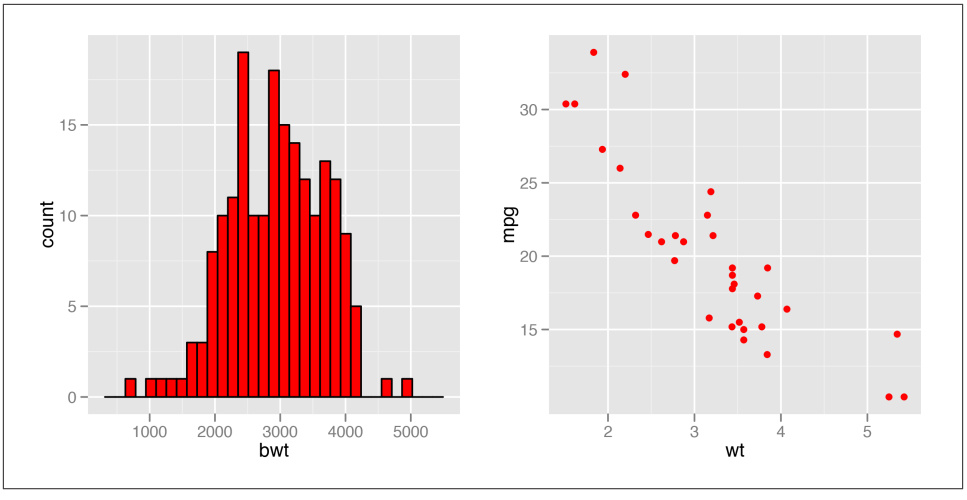

Figure 12-1. Left: setting fill and colour; right: setting colour for points

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.

See Also

For more information about point shapes, see Recipe 4.5.

See Recipe 12.4 for more on specifying colors.

12.2. Mapping Variables to 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, set the value of `colour` or `fill` to the name of one of the columns in the data (Figure 12-2):

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

# These both have the same effect
ggplot(cabbage_exp, aes(x=Date, y=Weight, fill=Cultivar)) +
    geom_bar(colour="black", position="dodge")

ggplot(cabbage_exp, aes(x=Date, y=Weight)) +
    geom_bar(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))
```

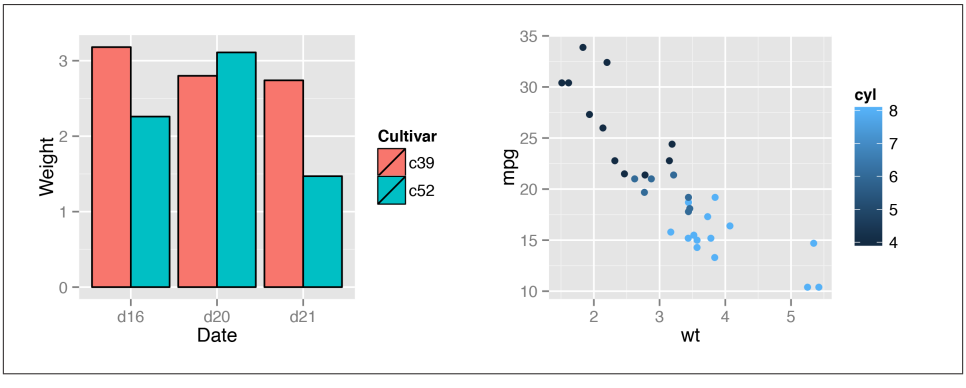

Figure 12-2. Left: mapping a variable to fill; right: mapping a variable to colour for points

When the mapping is specified in `ggplot()` it sets the default mapping, which is inherited by all the geoms. The default mappings can be overridden by specifying mappings within a geom.

Discussion

In the `cabbage_exp` example, the variable `Cultivar` is mapped to `fill`. The `Cultivar` column in `cabbage_exp` is a factor, so ggplot2 treats it as a discrete 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()`, or you can modify the data so that the column is a character vector or factor (Figure 12-3):

```
# 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
m <- mtcars              # Make a copy of mtcars
m$cyl <- factor(m$cyl)   # Convert cyl to a factor
ggplot(m, aes(x=wt, y=mpg, colour=cyl)) + geom_point()
```

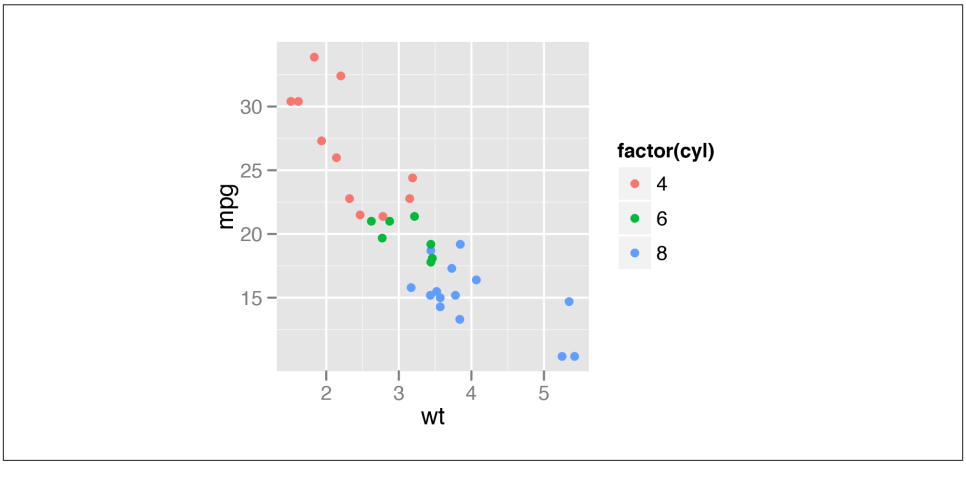

Figure 12-3. Mapping to colour with a continuous variable converted to a factor

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 12.3 and 12.4.

12.3. 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
`scale_fill_discrete()`	`scale_colour_discrete()`	Colors evenly spaced around the color wheel (same as hue)
`scale_fill_hue()`	`scale_colour_hue()`	Colors evenly spaced around the color wheel (same as discrete)
`scale_fill_grey()`	`scale_colour_grey()`	Greyscale palette
`scale_fill_brewer()`	`scale_colour_brewer()`	ColorBrewer palettes
`scale_fill_manual()`	`scale_colour_manual()`	Manually specified colors

In the example here we'll use the default palette (hue), and a ColorBrewer palette (Figure 12-4):

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

# Base plot
p <- ggplot(uspopage, aes(x=Year, y=Thousands, fill=AgeGroup)) + geom_area()

# These three have the same effect
p
p + scale_fill_discrete()
p + scale_fill_hue()

# ColorBrewer palette
p + scale_fill_brewer()
```

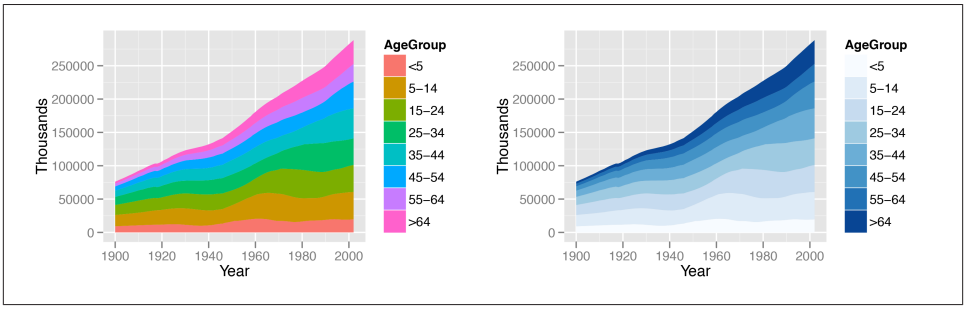

Figure 12-4. Left: default palette (using hue); right: a ColorBrewer palette

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 12-5 (right), set the value of `l` (luminance/lightness):

```
# Basic scatter plot
h <- ggplot(heightweight, aes(x=ageYear, y=heightIn, colour=sex)) +
    geom_point()

# Default lightness = 65
h

# Slightly darker
h + scale_colour_hue(l=45)
```

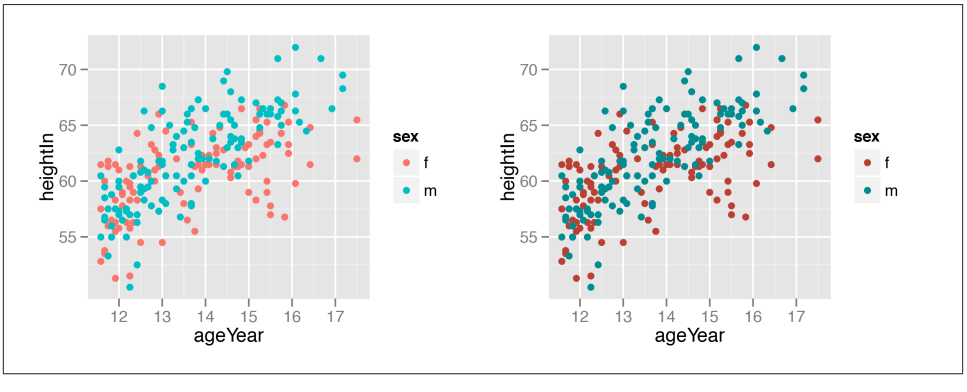

Figure 12-5. Left: points with default lightness; right: with lightness set to 45

The ColorBrewer package provides a number of palettes. You can generate a graphic showing all of them, as shown in Figure 12-6:

```
library(RColorBrewer)
display.brewer.all()
```

The ColorBrewer palettes can be selected by name. For example, this will use the `Oranges` palette (Figure 12-7):

```
p + scale_fill_brewer(palette="Oranges")
```

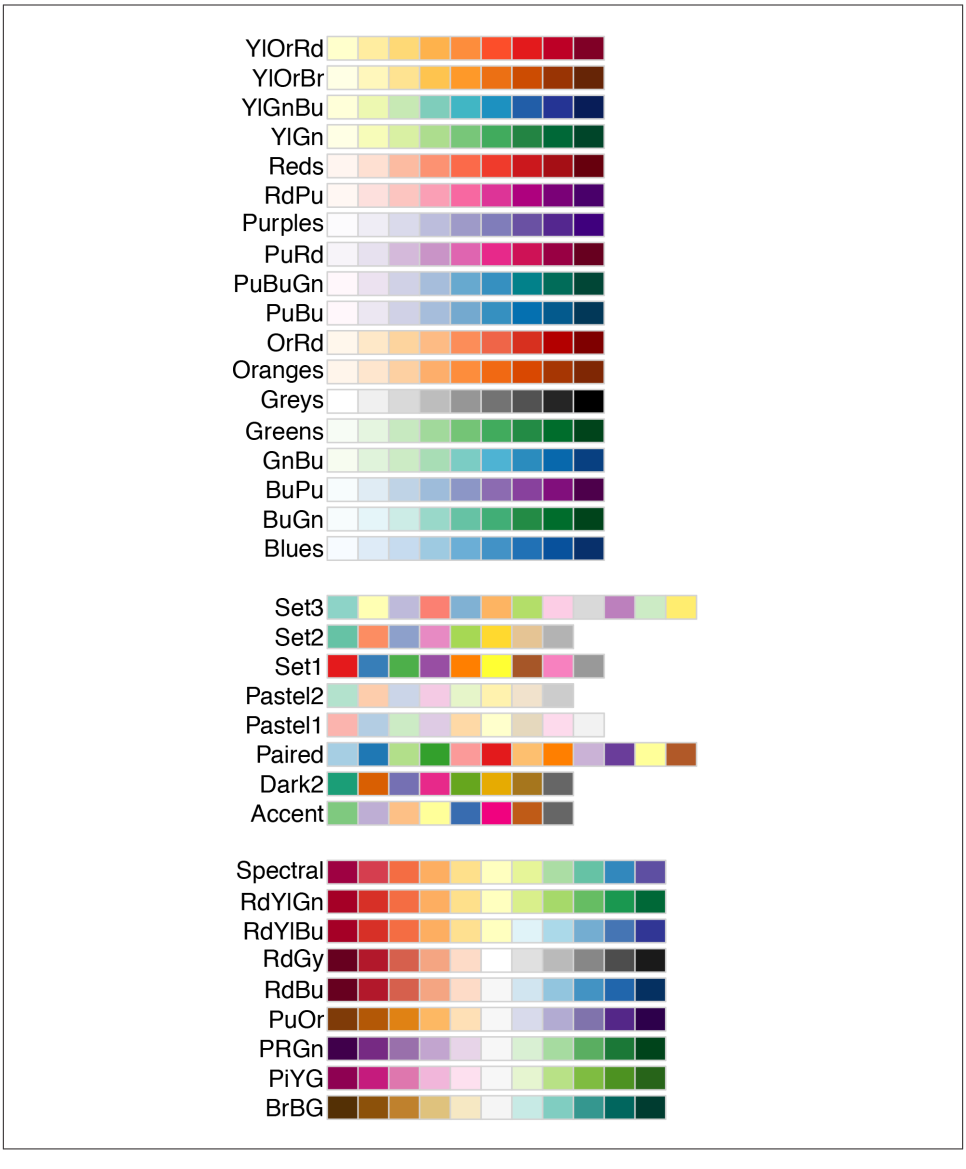

Figure 12-6. All the ColorBrewer palettes

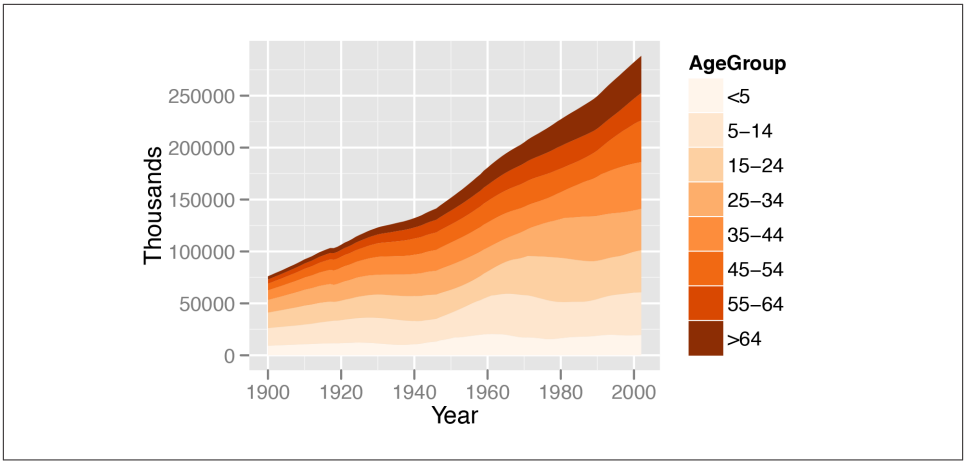

Figure 12-7. Using a named ColorBrewer palette

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 12-8.

```
p + scale_fill_grey()

# Reverse the direction and use a different range of greys
p + scale_fill_grey(start=0.7, end=0)
```

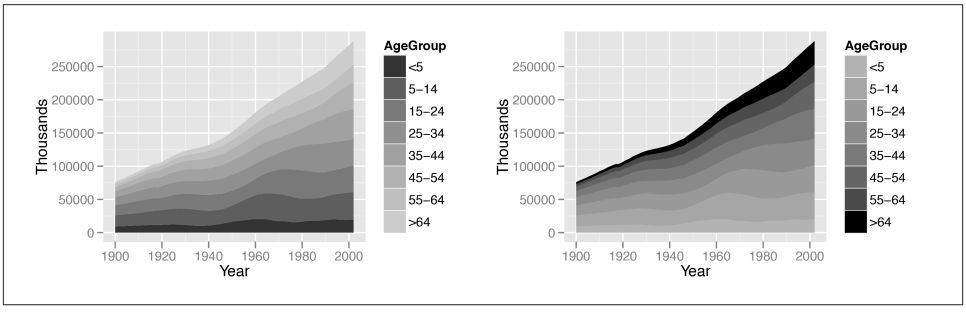

Figure 12-8. Left: using the default grey palette; right: a different grey palette

See Also

See Recipe 10.4 for more information about reversing the legend.

To select colors manually, see Recipe 12.4.

For more about ColorBrewer, see *http://colorbrewer.org*.

12.4. 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 12-9). The colors can be named, or they can be specified with RGB values:

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

# Base plot
h <- ggplot(heightweight, aes(x=ageYear, y=heightIn, colour=sex)) + geom_point()

# Using color names
h + scale_colour_manual(values=c("red", "blue"))

# Using RGB values
h + scale_colour_manual(values=c("#CC6666", "#7777DD"))
```

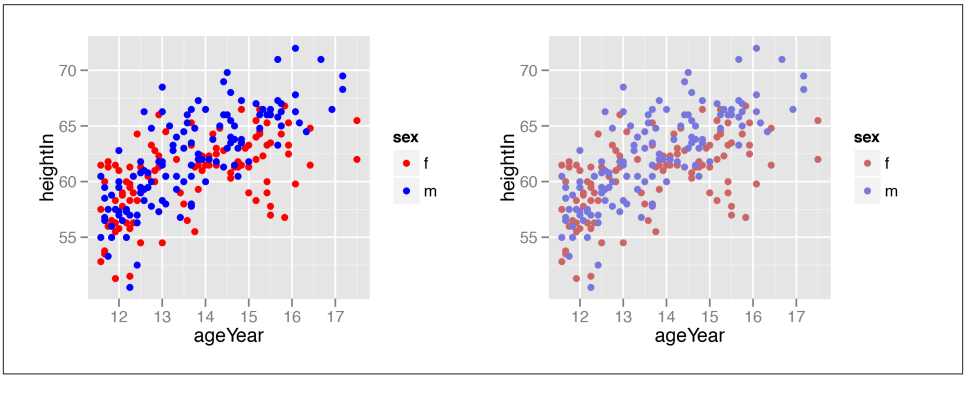

Figure 12-9. Left: scatter plot with named colors; right: with slightly different RGB colors

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)
```

```
"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:

```
h + 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.

See Also

A chart of RGB color codes.

12.5. Using a Colorblind-Friendly Palette

Problem

You want to use colors that can be distinguished by colorblind viewers.

Solution

Use the palette defined here (`cb_palette`) with `scale_fill_manual()` (Figure 12-10):

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

# Base plot
p <- ggplot(uspopage, aes(x=Year, y=Thousands, fill=AgeGroup)) + geom_area()

# The palette with grey:
cb_palette <- c("#999999", "#E69F00", "#56B4E9", "#009E73", "#F0E442",
                "#0072B2", "#D55E00", "#CC79A7")

# Add it to the plot
p + scale_fill_manual(values=cb_palette)
```

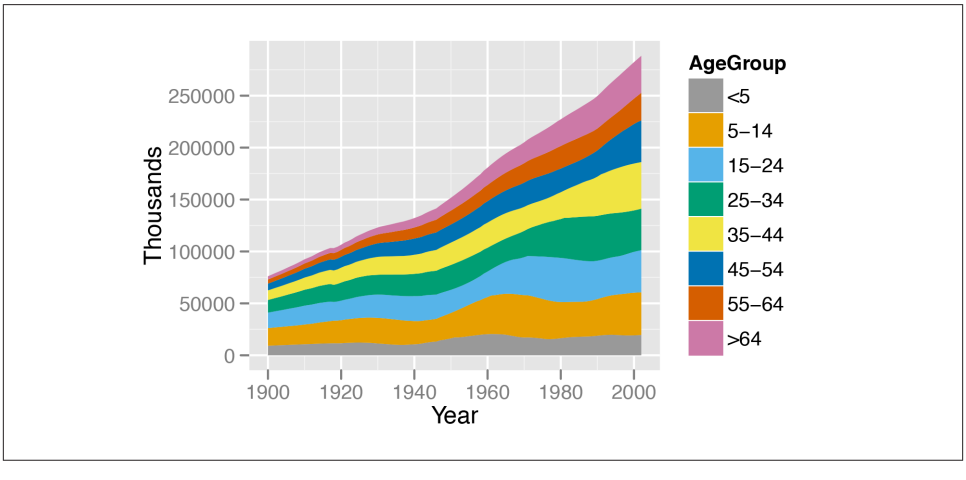

Figure 12-10. A graph with the colorblind-friendly palette

A chart of the colors is shown in Figure 12-11.

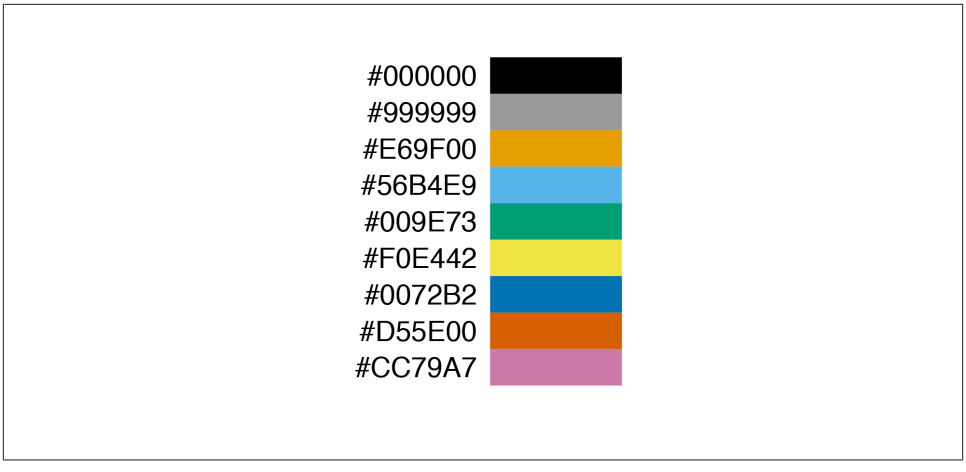

Figure 12-11. Colorblind palette with RGB values

In some cases it may be better to use black instead of grey. To do this, replace the `"#999999"` with `"#000000"` or `"black"`:

```
c("#000000", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00",
  "#CC79A7")
```

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 palette here is 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.)

See Also

The source of this palette.

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!

12.6. 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) # For the data set

# Base plot
p <- ggplot(heightweight, aes(x=ageYear, y=heightIn, colour=weightLb)) +
     geom_point(size=3)

p

# With a gradient between two colors
p + scale_colour_gradient(low="black", high="white")

# A gradient with a white midpoint
library(scales)
p + scale_colour_gradient2(low=muted("red"), mid="white", high=muted("blue"),
    midpoint=110)

# A gradient of n colors
p + 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`.

Discussion

Mapping continuous values to a color scale requires a continuously changing 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
`scale_fill_gradient()`	`scale_colour_gradient()`	Two-color gradient
`scale_fill_gradient2()`	`scale_colour_gradient2()`	Gradient with a middle color and two colors that diverge from it
`scale_fill_gradientn()`	`scale_colour_gradientn()`	Gradient with *n* colors, equally spaced

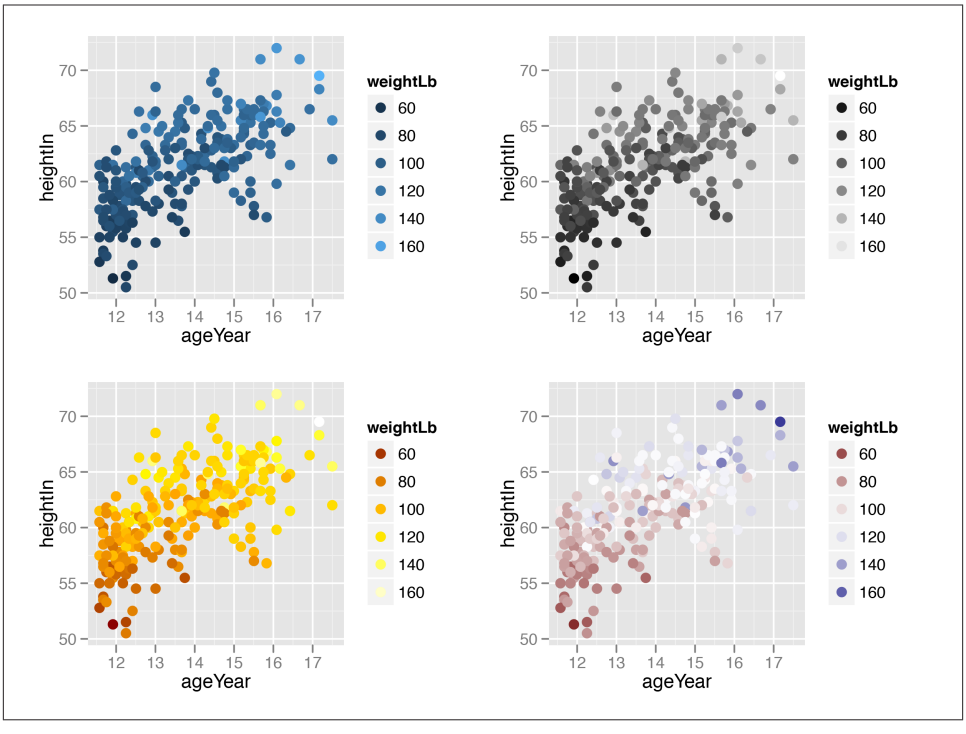

Figure 12-12. Clockwise from top left: default colors, two-color gradient with scale_colour_gradient(), three-color gradient with midpoint with scale_colour_gradient2(), four-color gradient with scale_colour_gradientn()

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) # For the data set

cb <- subset(climate, Source=="Berkeley")

cb$valence[cb$Anomaly10y >= 0] <- "pos"
cb$valence[cb$Anomaly10y < 0]  <- "neg"

cb
```

```
   Source Year Anomaly1y Anomaly5y Anomaly10y Unc10y valence
 Berkeley 1800        NA        NA     -0.435  0.505     neg
 Berkeley 1801        NA        NA     -0.453  0.493     neg
 Berkeley 1802        NA        NA     -0.460  0.486     neg
 ...
 Berkeley 2002        NA        NA      0.856  0.028     pos
 Berkeley 2003        NA        NA      0.869  0.028     pos
 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(cb, aes(x=Year, y=Anomaly10y)) +
    geom_area(aes(fill=valence)) +
    geom_line() +
    geom_hline(yintercept=0)
```

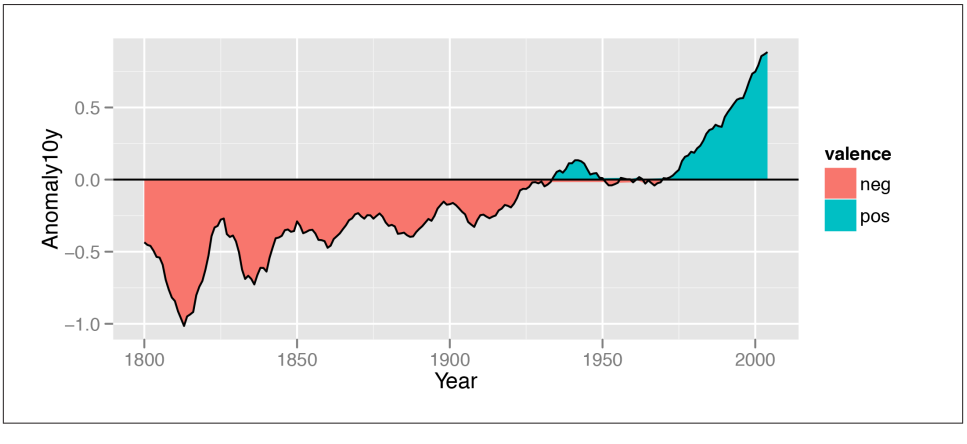

Figure 12-13. Mapping valence to fill color—notice the red area under the zero line around 1950

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(cb$Year, cb$Anomaly10y, n=1000)

# Put in a data frame and recalculate valence
cbi <- data.frame(Year=interp$x, Anomaly10y=interp$y)
cbi$valence[cbi$Anomaly10y >= 0] <- "pos"
cbi$valence[cbi$Anomaly10y < 0]  <- "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))
```

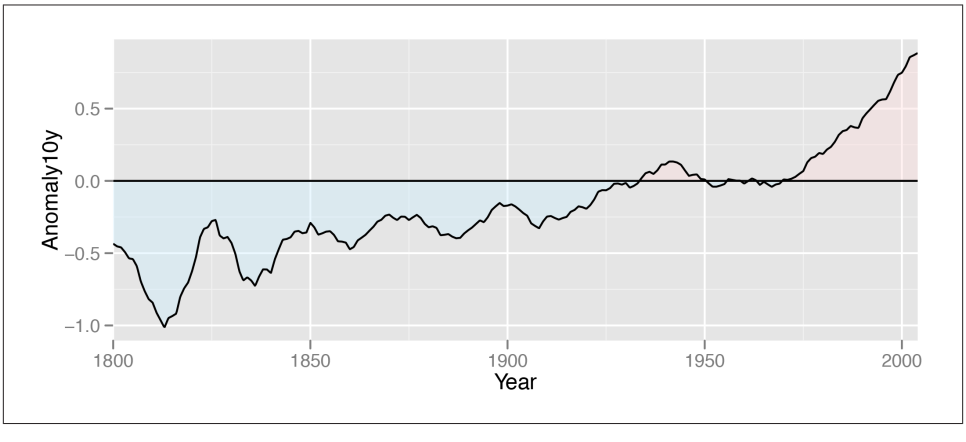

Figure 12-14. Shaded regions with interpolated data

CHAPTER 13

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   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
...
Ferrari Dino      19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E        21.4   4 121.0 109 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
hp   -0.78  0.83  0.79  1.00 -0.45  0.66 -0.71 -0.72 -0.24 -0.13  0.75
drat  0.68 -0.70 -0.71 -0.45  1.00 -0.71  0.09  0.44  0.71  0.70 -0.09
wt   -0.87  0.78  0.89  0.66 -0.71  1.00 -0.17 -0.55 -0.69 -0.58  0.43
qsec  0.42 -0.59 -0.43 -0.71  0.09 -0.17  1.00  0.74 -0.23 -0.21 -0.66
vs    0.66 -0.81 -0.71 -0.72  0.44 -0.55  0.74  1.00  0.17  0.21 -0.57
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 option `use="complete.obs"` or `use="pairwise.complete.obs"`.

To graph the correlation matrix (Figure 13-1), we'll use the `corrplot` package, which first must be installed with `install.packages("corrplot")`:

```
library(corrplot)

corrplot(mcor)
```

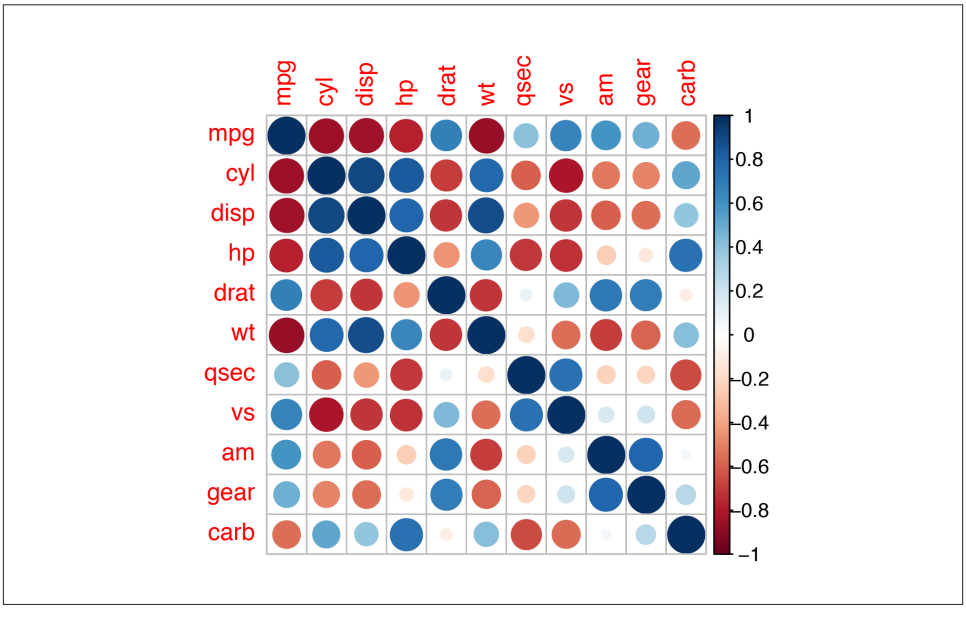

Figure 13-1. A correlation matrix

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)
```

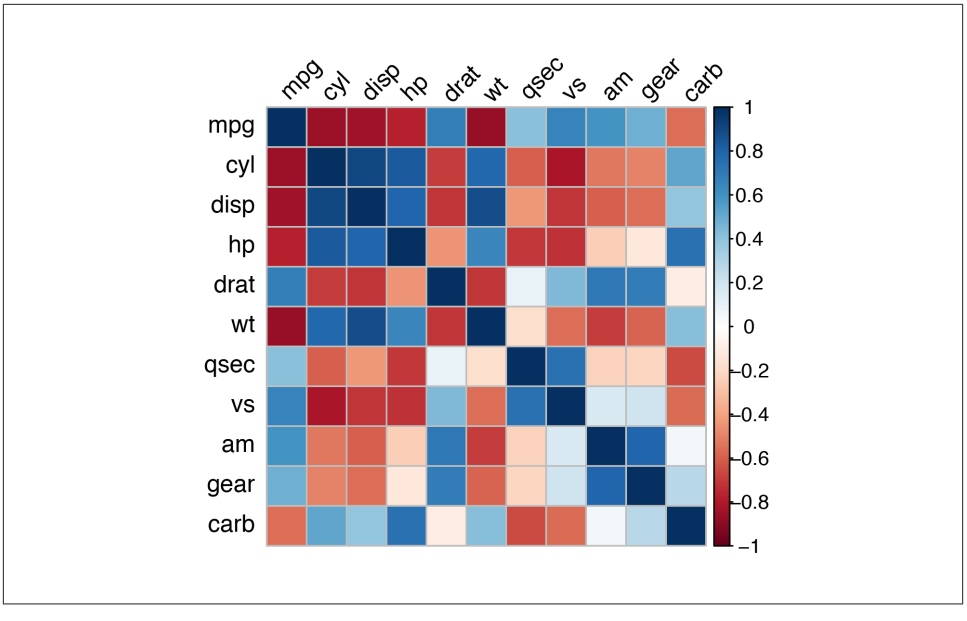

Figure 13-2. Correlation matrix with colored squares and black, rotated labels

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", addcolorlabel="no", order="AOE")
```

Like many other standalone graphing functions, `corrplot()` has its own menagerie of options, which can't all be illustrated here. Table 13-1 lists some useful options.

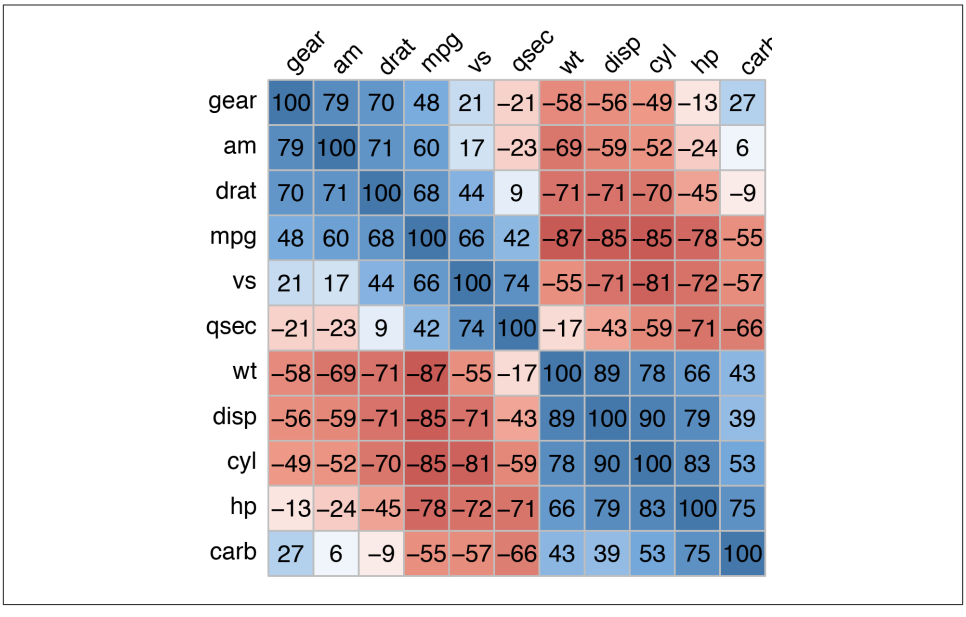

Figure 13-3. Correlation matrix with correlation coefficients and no legend

Table 13-1. Options for corrplot()

Option	Description
`type={"lower"|"upper"}`	Only use the lower or upper triangle
`diag=FALSE`	Don't show values on the diagonal
`addshade="all"`	Add lines indicating the direction of the correlation
`shade.col=NA`	Hide correlation direction lines
`method="shade"`	Use colored squares
`method="ellipse"`	Use ellipses
`addCoef.col="color"`	Add correlation coefficients, in *`color`*
`tl.srt="number"`	Specify the rotation angle for top labels
`tl.col="color"`	Specify the label color
`order={"AOE"|"FPC"|"hclust"}`	Sort labels using angular order of eigenvectors, first principle 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)
```

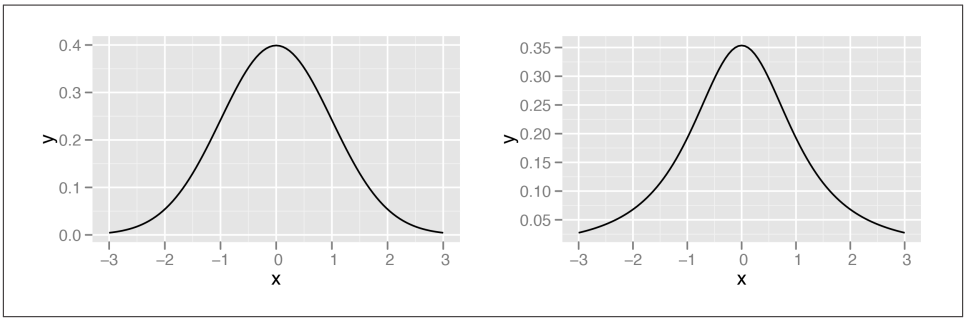

Figure 13-4. Left: the normal distribution; right: the t-distribution with df=2

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))
```

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)
```

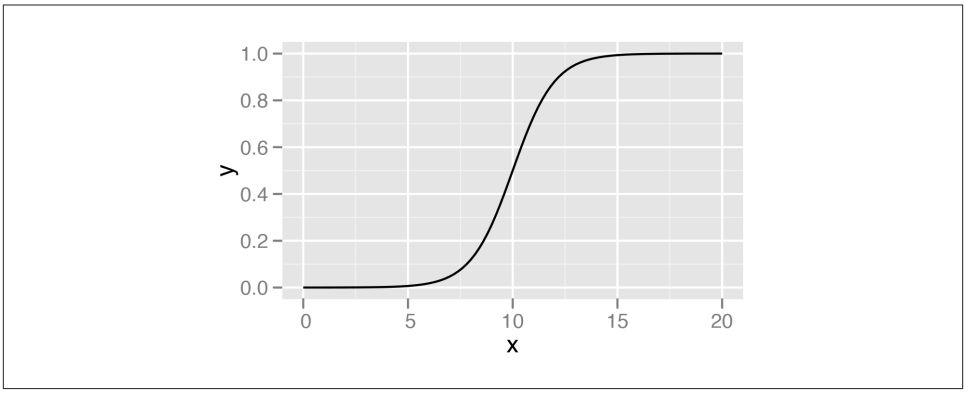

Figure 13-5. A user-defined function

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)
```

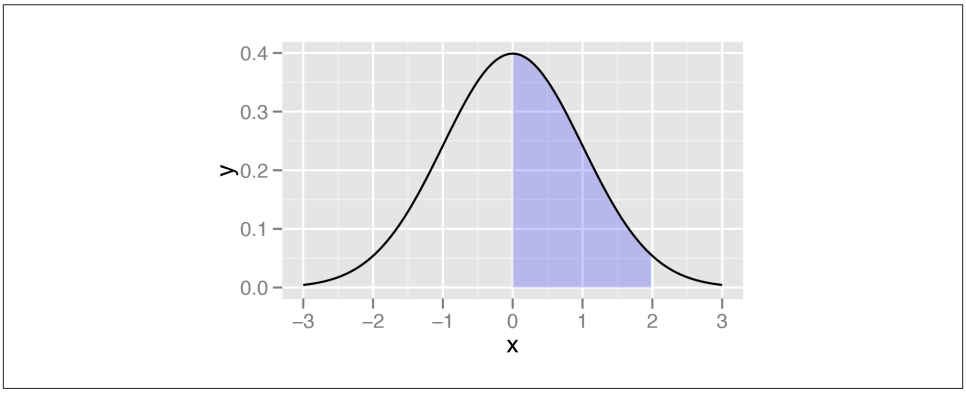

Figure 13-6. Function curve with a shaded region

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 hardcoded 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)
```

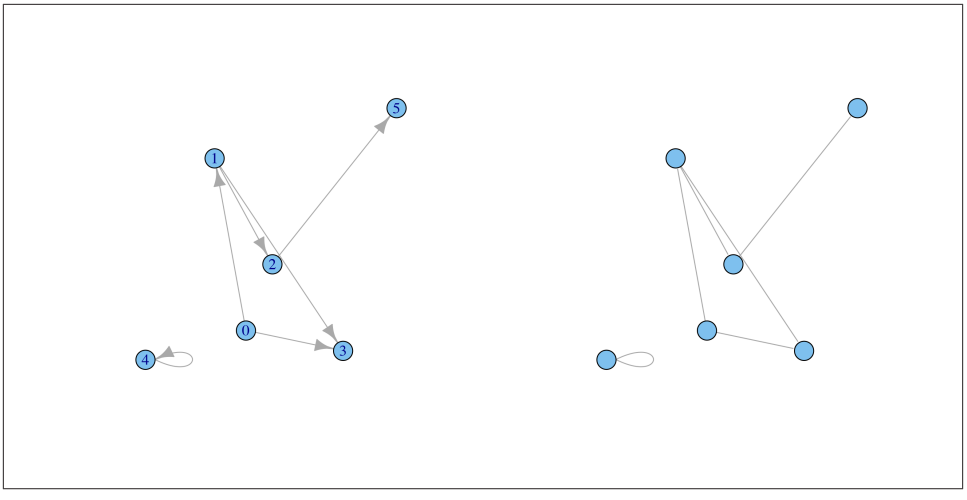

Figure 13-7. Left: a directed graph; right: an undirected graph, with no vertex labels

This is the structure of each of the graph objects:

```
str(gd)

IGRAPH D--- 6 6 --
+ edges:
[1] 1->2 2->3 2->4 1->4 5->5 3->6

str(gu)

IGRAPH U--- 6 6 --
+ edges:
[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 rows 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:

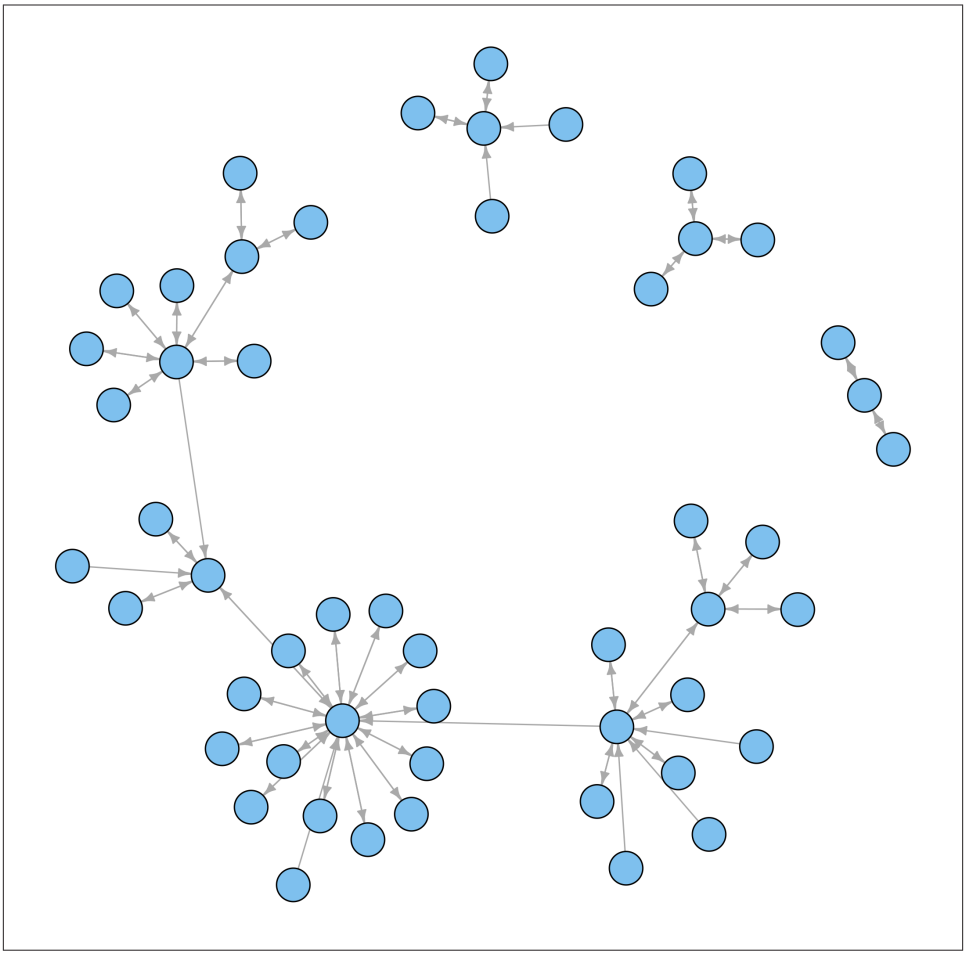

Figure 13-8. A directed graph from a data frame, with the Fruchterman-Reingold algorithm

```
library(gcookbook) # For the data set
madmen2
```

```
       Name1                Name2
 Abe Drexler          Peggy Olson
     Allison           Don Draper
 Arthur Case        Betty 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)
```

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)
```

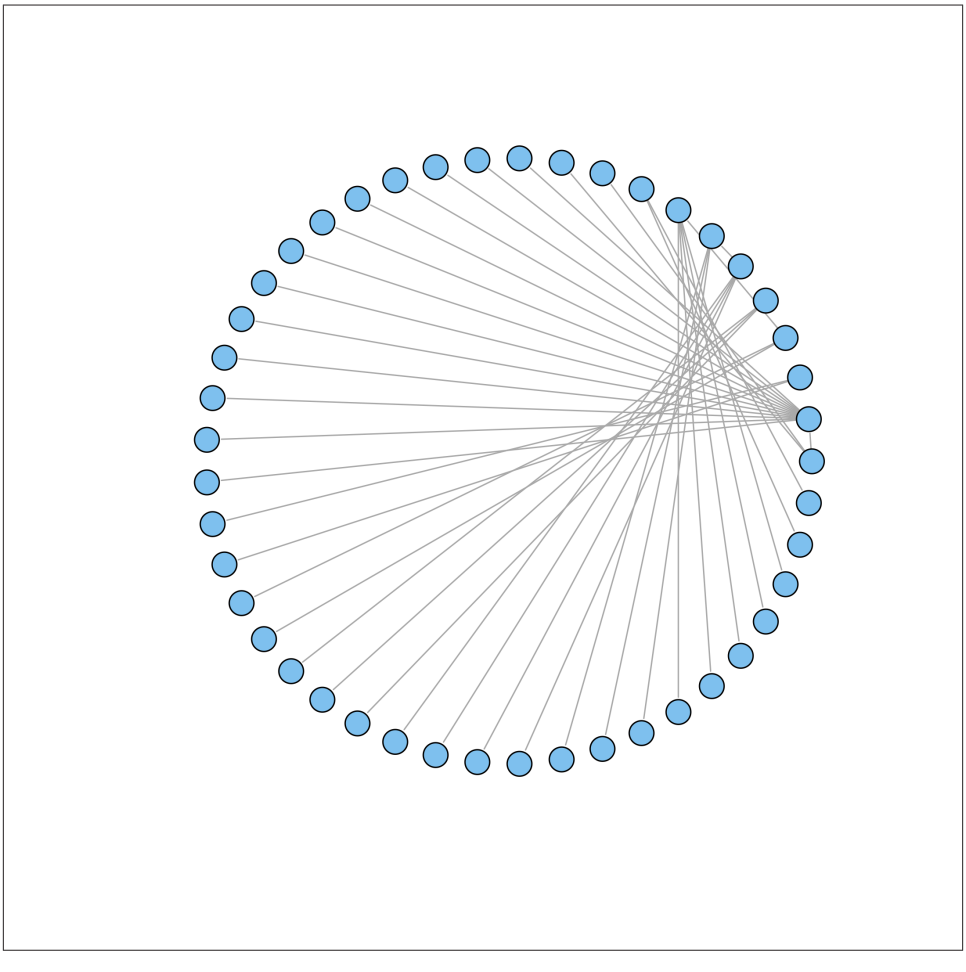

Figure 13-9. A circular undirected graph from a data frame

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 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"       "Joan Holloway"
 [5] "Lane Pryce"      "Peggy Olson"       "Pete Campbell"     "Roger Sterling"
 [9] "Sal Romano"      "Henry Francis"     "Allison"           "Candace"
[13] "Faye Miller"     "Megan Calvet"      "Rachel Menken"     "Suzanne Farrell"
[17] "Hildy"           "Franklin"          "Rebecca Pryce"     "Abe Drexler"
[21] "Duck Phillips"   "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")
```

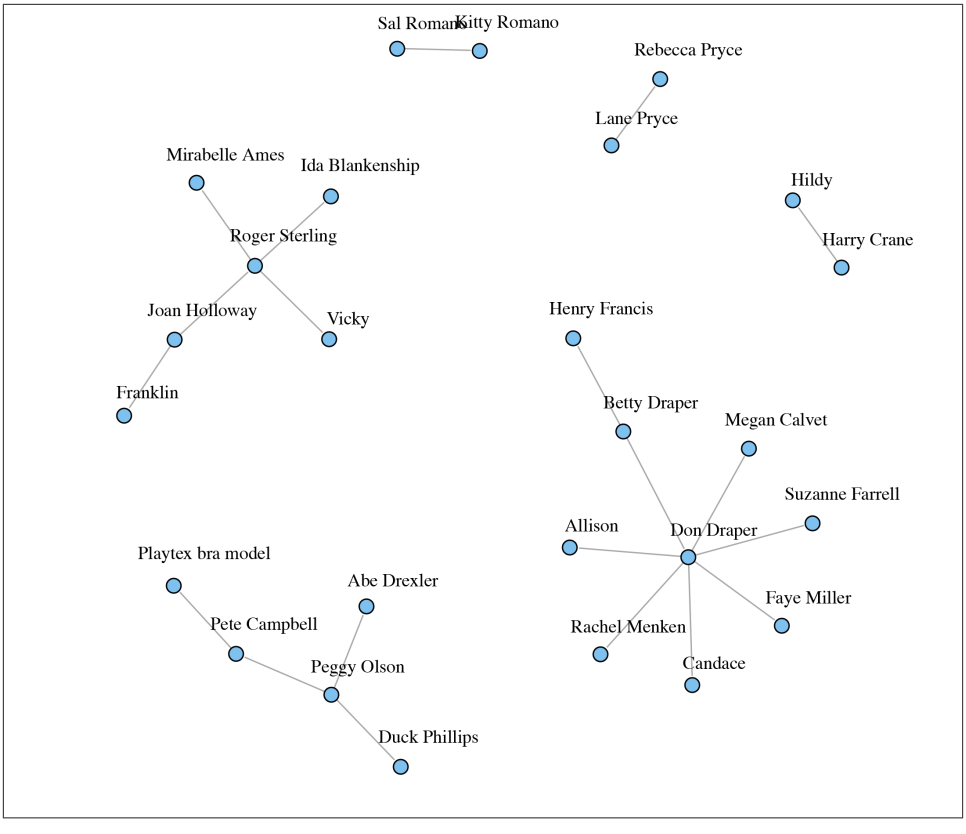

Figure 13-10. A network graph with labels

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()$`*`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)

# 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)
```

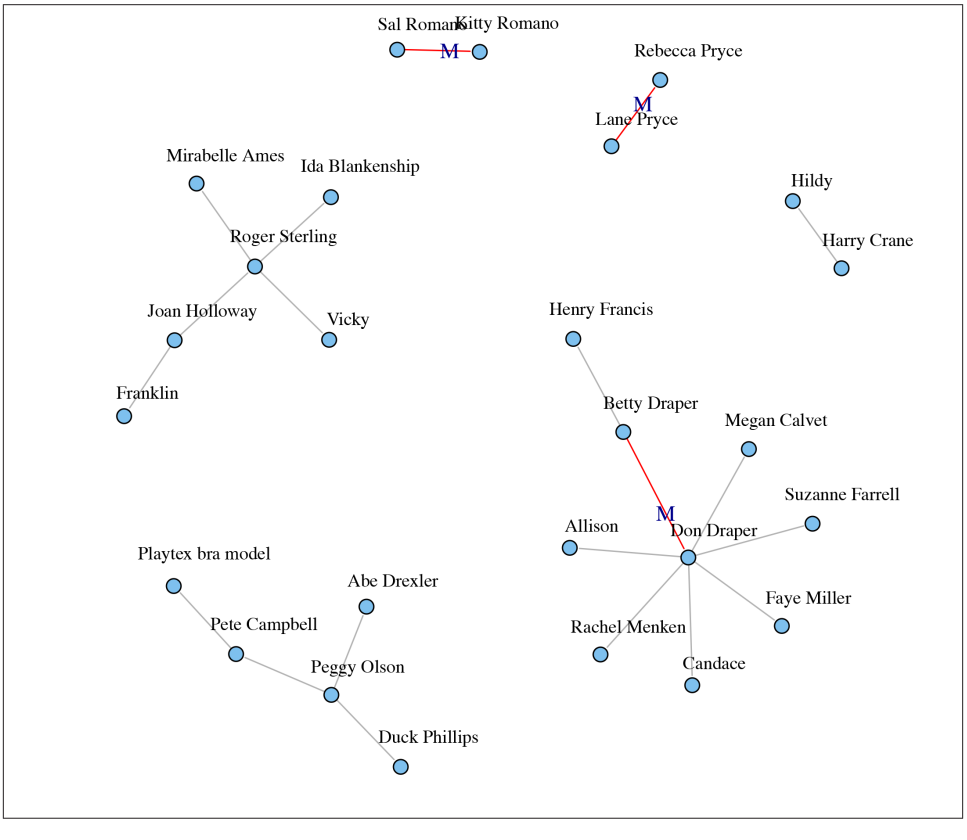

Figure 13-11. A network graph with labeled and colored edges

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
 ...
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
     NA 1945       1
     87 1945       2
     82 1945       3
 ...
     25 1974       2
     24 1974       3
     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()
```

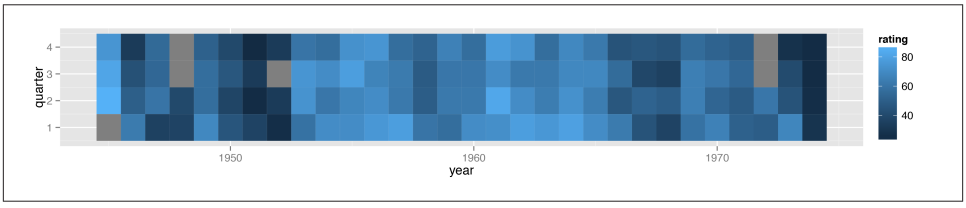

Figure 13-12. A heat map—the grey squares represent NAs in the data

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. 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)) +
    scale_y_reverse() +
    scale_fill_gradient2(midpoint=50, mid="grey70", limits=c(0,100))
```

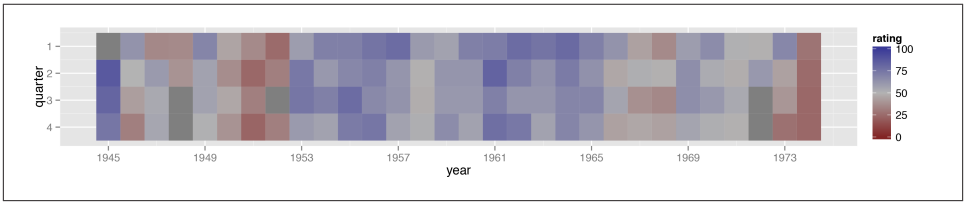

Figure 13-13. A heat map with customized appearance

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 (3D) 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$wt, mtcars$disp, mtcars$mpg, type="s", size=0.75, lit=FALSE)
```

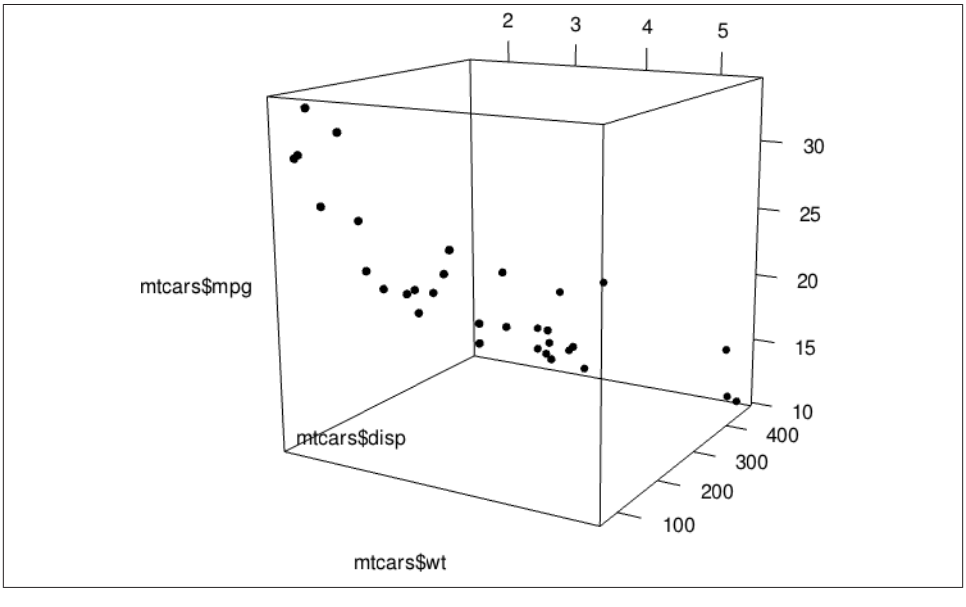

Figure 13-14. A 3D scatter plot

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, we used `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")
```

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")
```

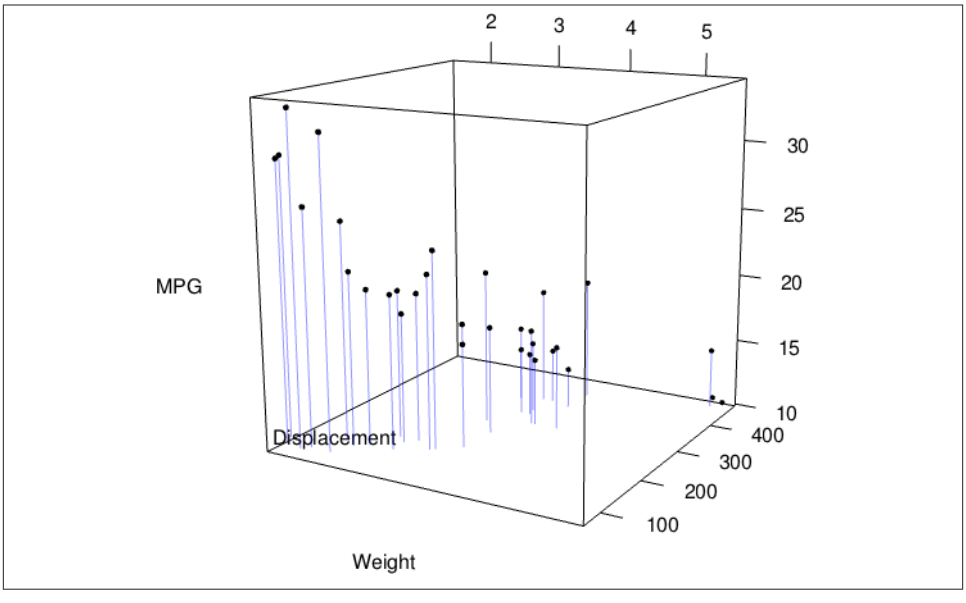

Figure 13-15. A 3D scatter plot with vertical lines for each point

```
# 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)
```

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.

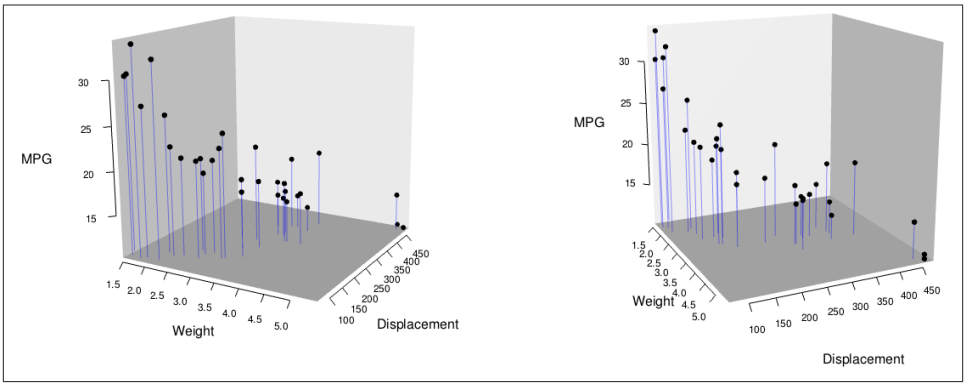

Figure 13-16. Left: 3D scatter plot with axis ticks and labels repositioned; right: from a different point of view

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")
```

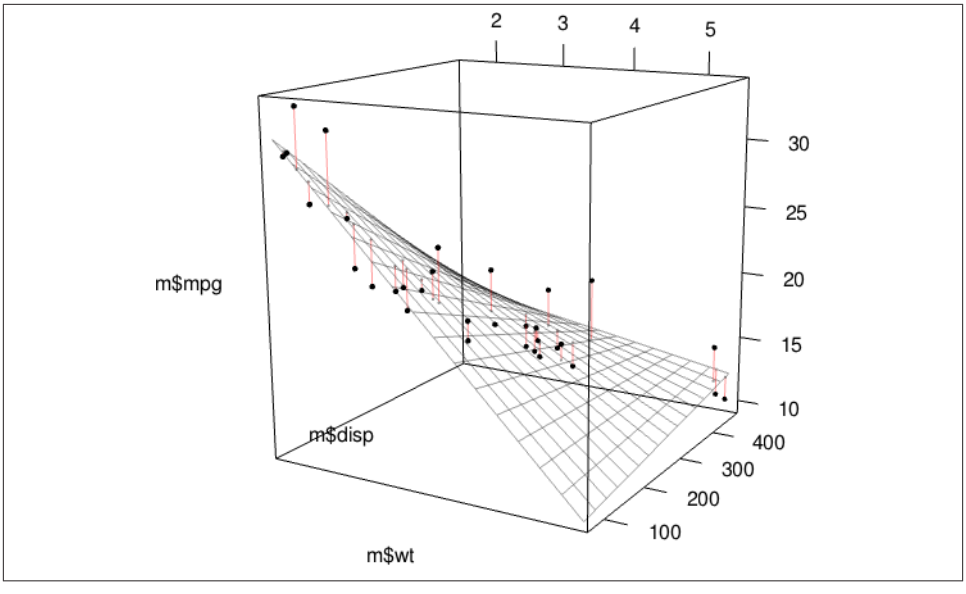

Figure 13-17. A 3D scatter plot with a prediction surface

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
```

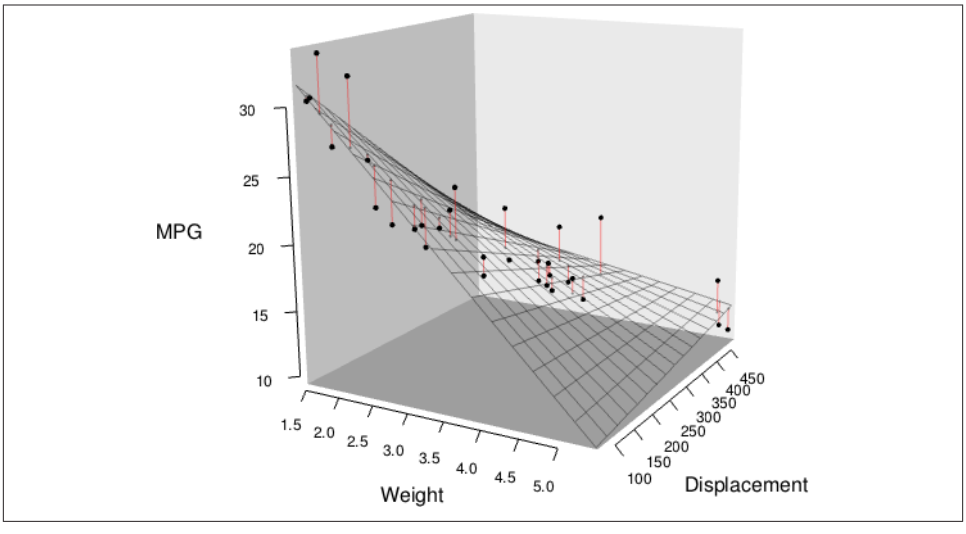

Figure 13-18. Three-dimensional scatter plot with customized appearance

```
# 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)
```

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('figs/miscgraph/3dplot.pdf', fmt='pdf')

rgl.postscript('figs/miscgraph/3dplot.ps', fmt='ps')
```

Postscript and PDF output does not support many features of the OpenGL library on which rgl is based. For example, it does 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)

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))
```

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(gcookbook) # For the data set

# Get data from year 2009
c2 <- subset(countries, Year==2009)

# Drop rows that have any NA values
c2 <- c2[complete.cases(c2), ]

# Pick out a random 25 countries
# (Set random seed to make this repeatable)
set.seed(201)
c2 <- c2[sample(1:nrow(c2), 25), ]

c2
```

```
                Name Code Year        GDP laborrate  healthexp infmortality
6731         Mongolia  MNG 2009  1690.4170      72.9   74.19826         27.8
1733           Canada  CAN 2009 39599.0418      67.8 4379.76084          5.2
 ...
5966    Macedonia, FYR  MKD 2009  4510.2380      54.0  313.68971         10.6
10148     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
Mongolia      1690.4170      72.9   74.19826         27.8
Canada       39599.0418      67.8 4379.76084          5.2
 ...
Macedonia, FYR  4510.2380      54.0  313.68971         10.6
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
Mongolia         -0.6783472  1.15028714 -0.6341393599  -0.08334689
Canada            1.7504703  0.59747293  1.9736219974  -0.88014885
 ...
Macedonia, FYR   -0.4976803 -0.89837729 -0.4890859471  -0.68976254
Turkmenistan     -0.5489228  0.61915192 -0.6324002997   0.62883892
attr(,"scaled:center")
         GDP    laborrate    healthexp infmortality
   12277.960       62.288     1121.198       30.164
attr(,"scaled:scale")
         GDP    laborrate    healthexp infmortality
15607.852864     9.225523  1651.056974    28.363384
```

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)
```

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.)

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".

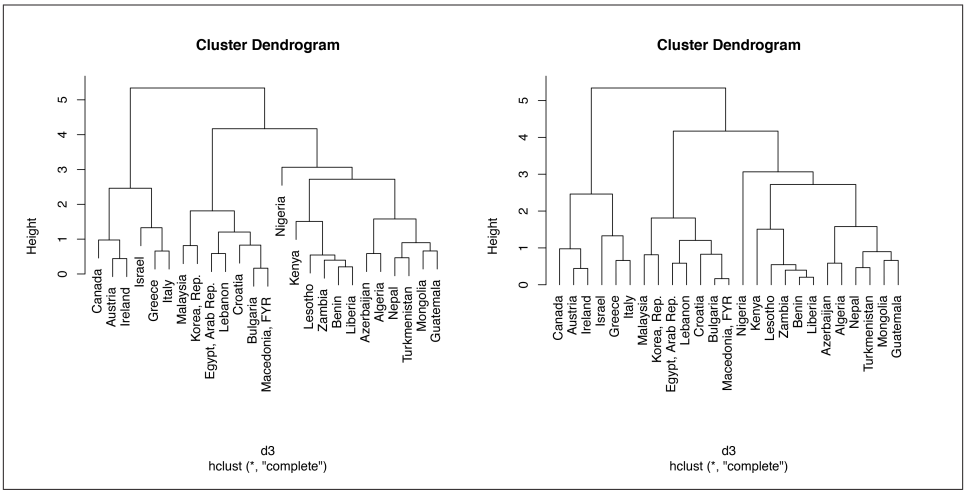

Figure 13-19. Left: a dendrogram; right: with text aligned

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 data set
isabel
```

```
        x        y      z  vx  vy  vz  t  speed
-83.00000 41.70000  0.035  NA  NA  NA NA     NA
-83.00000 41.62786  0.035  NA  NA  NA NA     NA
-83.00000 41.55571  0.035  NA  NA  NA NA     NA
```

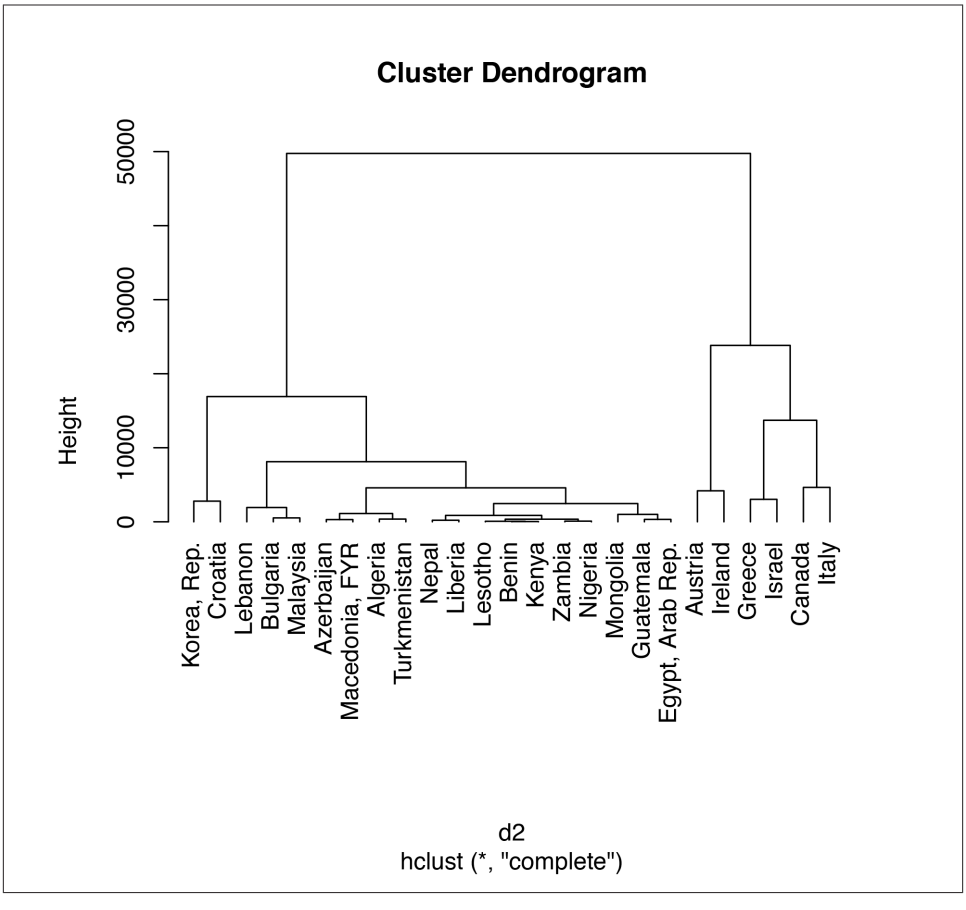

Figure 13-20. Dendrogram with unscaled data—notice the much larger Height values, which are largely due to the unscaled GDP values

```
...
-62.04208 23.88036 18.035 -12.54371 -5.300128 -0.045253485 -66.96269 13.61749
-62.04208 23.80822 18.035 -12.56157 -5.254994 -0.020277001 -66.98840 13.61646
-62.04208 23.73607 18.035 -12.78071 -5.259613  0.005555035 -67.00575 13.82064
```

`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 <- subset(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
```

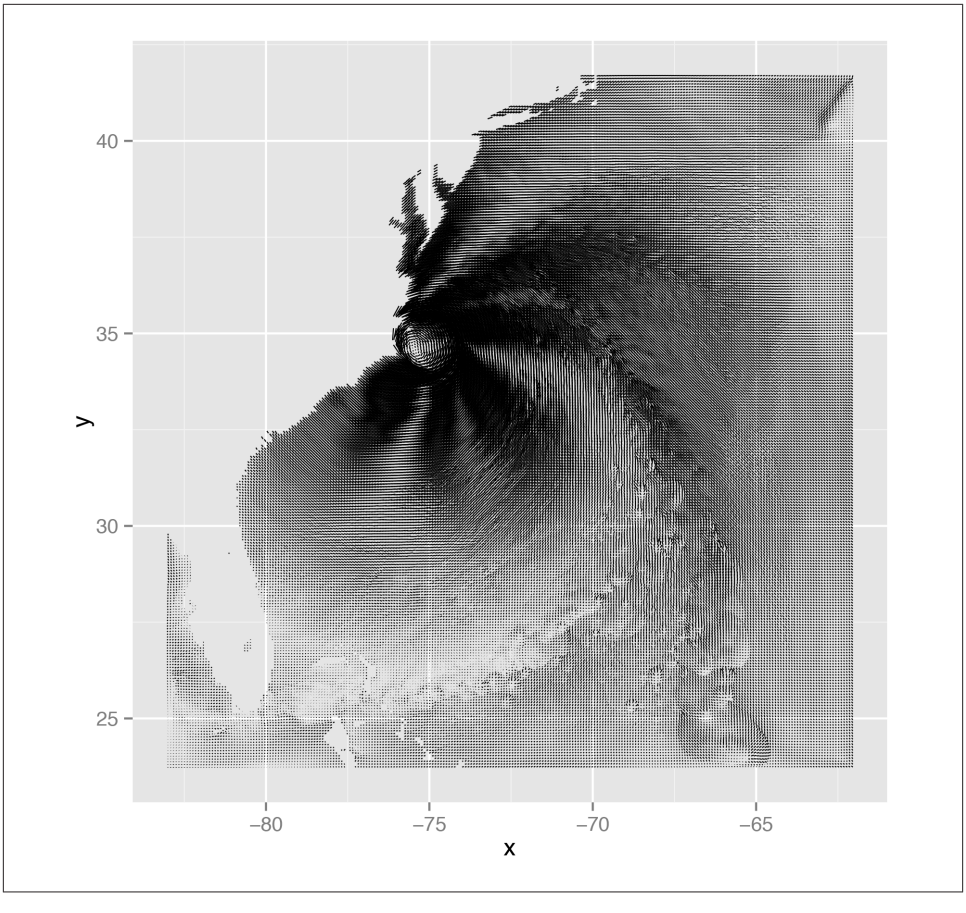

Figure 13-21. First attempt at a vector field—the resolution of the data is too high, but it does hint at some interesting patterns not visible in graphs with a lower data resolution

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 <- subset(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 <- subset(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)
```

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):

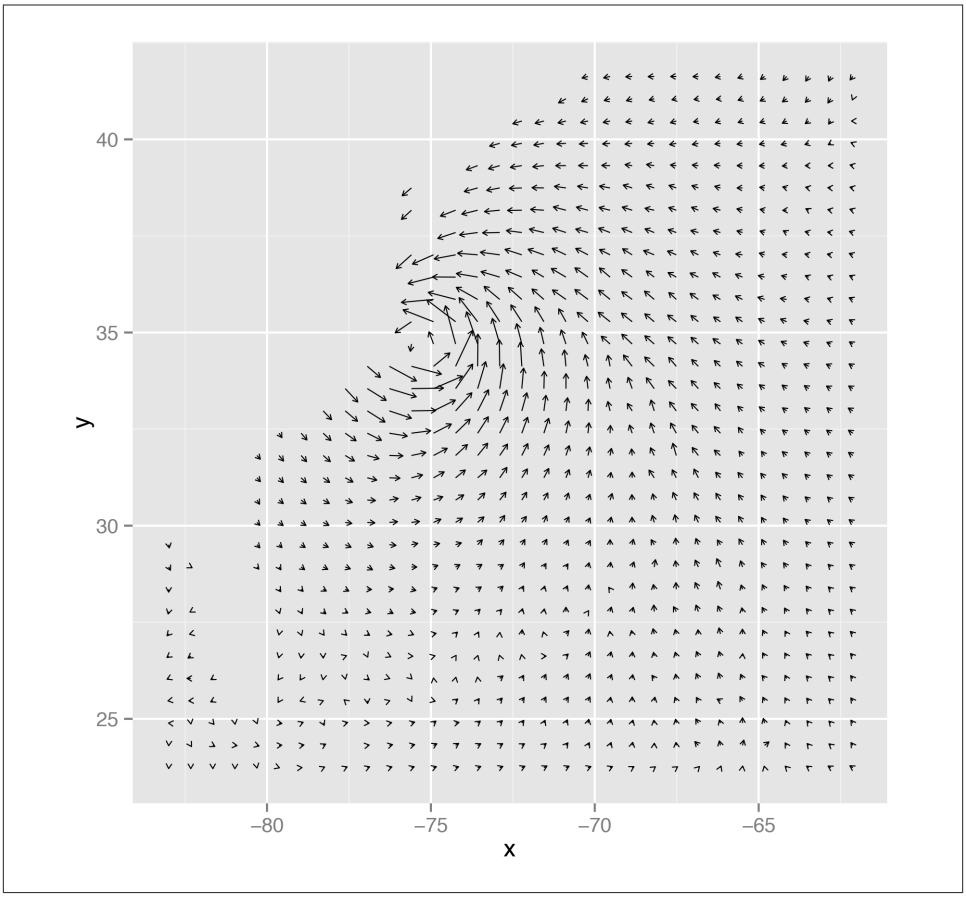

Figure 13-22. Vector field with arrowheads

```
# 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))
```

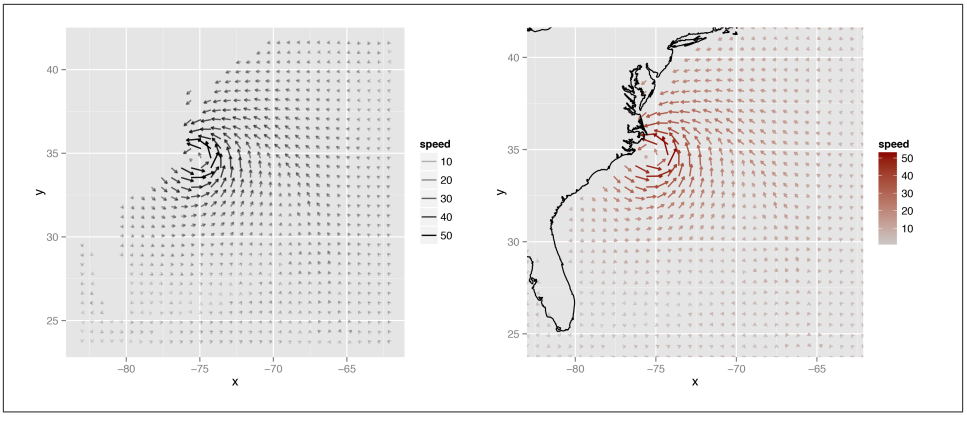

Figure 13-23. Left: vector field with speed mapped to alpha; right: with speed mapped to colour

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 <- subset(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)
```

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.

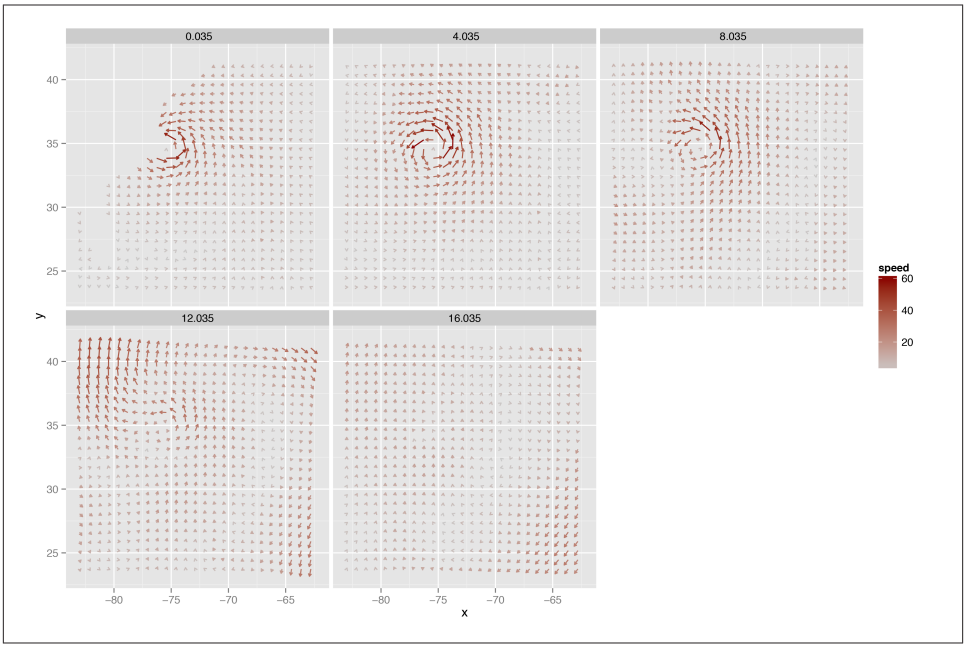

Figure 13-24. Vector field of wind speeds faceted on z

Solution

Use `qqnorm()` to compare to a normal distribution. Give `qqnorm()` a vector of numerical values, and add a theoretical distribution line with `qqline()` (Figure 13-25):

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

# QQ plot of height
qqnorm(heightweight$heightIn)
qqline(heightweight$heightIn)

# QQ plot of age
qqnorm(heightweight$ageYear)
qqline(heightweight$ageYear)
```

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.

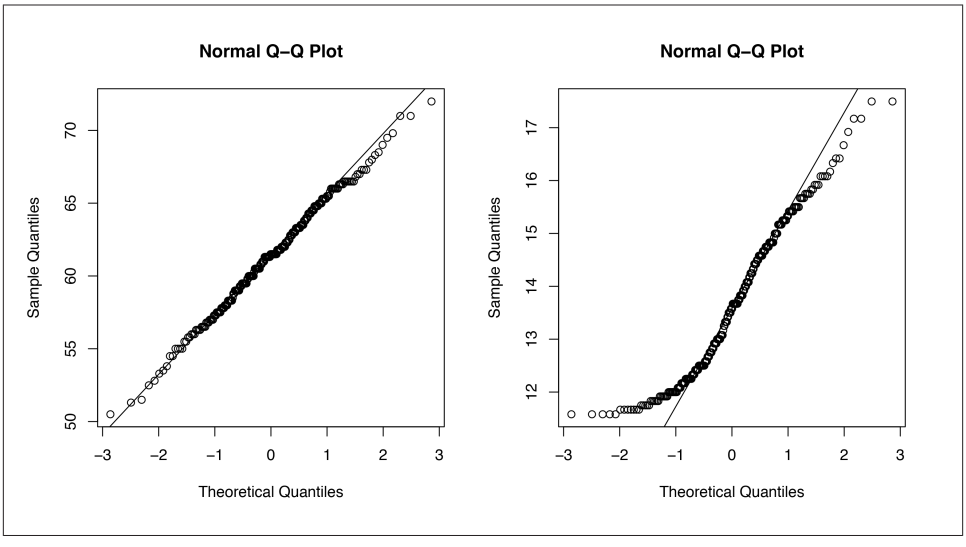

Figure 13-25. Left: QQ plot of height, which is close to normally distributed; right: QQ plot of age, which is not normally distributed

See Also

See `?qqplot` for information on comparing data to theoretical distributions other than the normal distribution.

ggplot2 has a `stat_qq()` function, but it doesn't provide an easy way to draw the QQ line.

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()
```

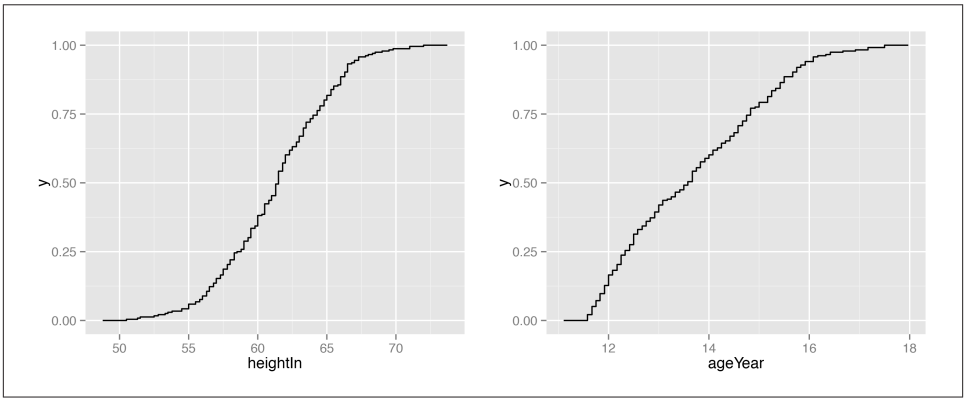

Figure 13-26. Left: ECDF of height; right: ECDF of age

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
```

```
 ... [four other Depts]

, , Dept = F

          Gender
Admit      Male Female
  Admitted   22     24
  Rejected  351    317

# 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)
```

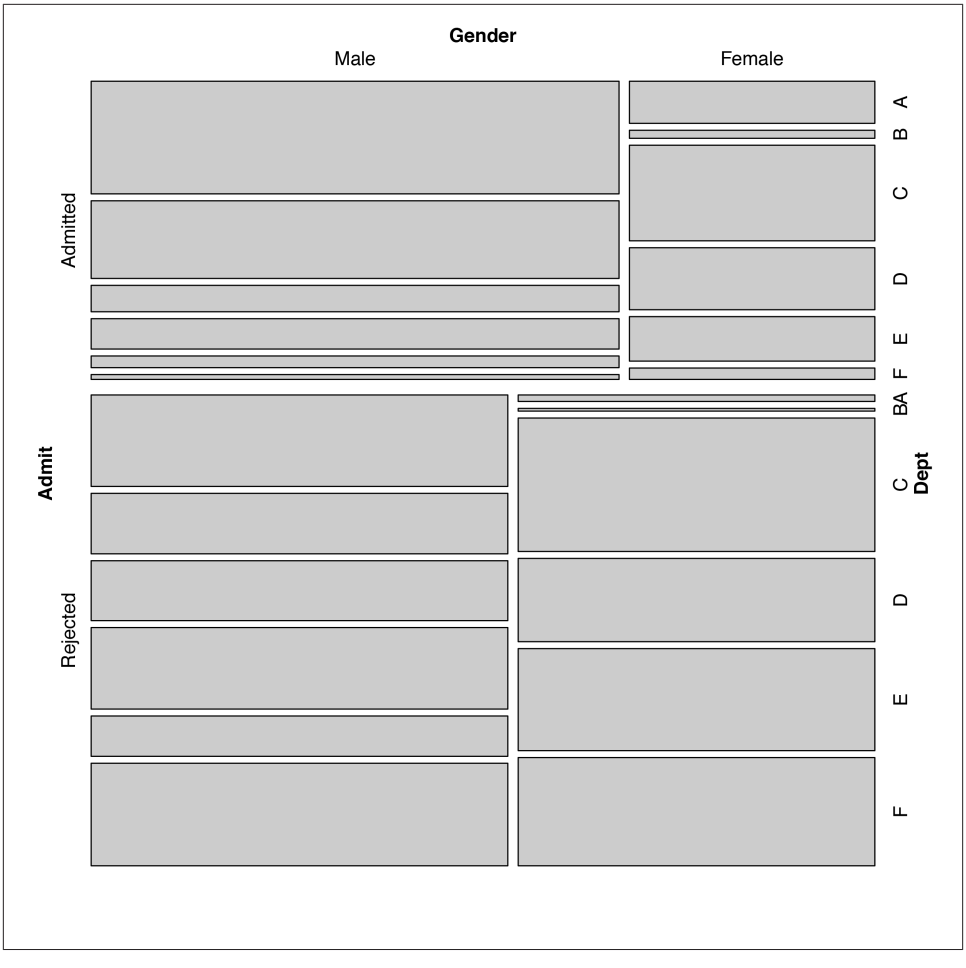

Figure 13-27. Mosaic plot of UC-Berkeley admissions data—the area of each rectangle is proportional to the number of cases in that cell

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=UCBAdmissions,
    highlighting="Admit", highlighting_fill=c("lightblue", "pink"),
    direction=c("v","h","v"))
```

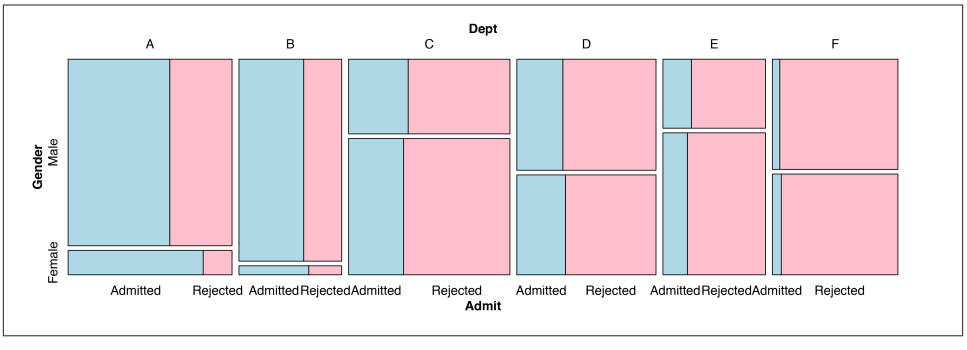

Figure 13-28. Mosaic plot with a different variable splitting order: first department, then gender, then admission status

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=UCBAdmissions,
    highlighting="Admit", highlighting_fill=c("lightblue", "pink"),
    direction=c("v", "v", "h"))

# This order makes it difficult to compare male and female
mosaic( ~ Dept + Gender + Admit, data=UCBAdmissions,
    highlighting="Admit", highlighting_fill=c("lightblue", "pink"),
    direction=c("v", "h", "h"))
```

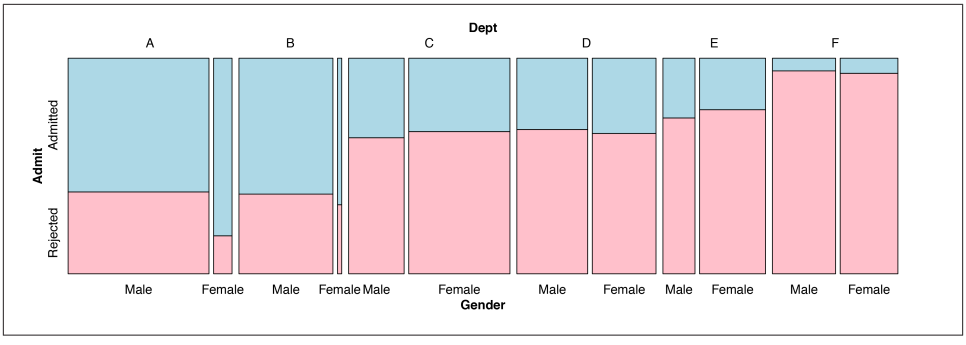

Figure 13-29. Splitting Dept vertically, Gender vertically, and Admit horizontally

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 `UCBerkeley` 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 `?mosiacplot` 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.