ADVANCED GRAPHICS

JUMPINGRIVERS.COM

1
R Graphics

There are a number of packages that improve and tweak R graphics. In this chapter, we will just cover the main packages.

Base Graphics

Base graphics were written by Ross Ihaka based on his experience of implementing the S graphics driver; technically they come with the **graphics** package. If you have created a histogram, scatter plot or boxplot, you've probably used base graphics. Base graphics are generally fast, but have limited scope. For example, you can only draw on top of the plot and cannot edit or alter existing graphics. For example, if you combine the `plot()` and `points()` commands, you have to work out the x- and y- limits before adding the points.

Grid Graphics

Grid graphics were developed by Paul Murrell[1]. Grid grobs (graphical objects) can be represented independently of the plot and modified later. The viewports system makes it easier to construct complex plots. Grid doesn't provide tools for graphics, it provides primitives for creating plots. Lattice and **ggplot2** graphics both use the **grid** package.

Lattice Graphics

The lattice package uses grid graphics to implement the trellis graphics system[2]. It produces nicer plots than base graphics and legends are automatically generated. I initially started using lattice before **ggplot2**. However, I found it a bit confusing and so switched to **ggplot2**. See figure 1.1 for an example **lattice** plot.

ggplot2

ggplot2 started in 2005[3] and follows the "Grammar of Graphics" [4] Like **lattice**, **ggplot2** uses grid to draw graphics, which means you can exercise low-level control over the plot appearance.

To install the course package, please see https://jr-packages.github.io/

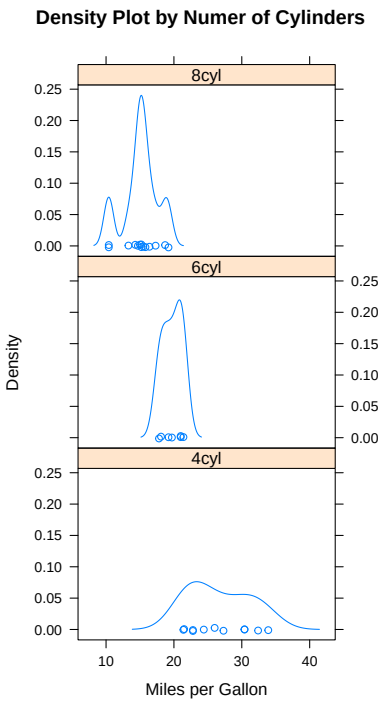

Figure 1.1: An example lattice plot. The give away is the salmon colour box.

[1] P Murrell. *R Graphics*. CRC Press, 2 edition, 2011

[2] D Sarkar. *Lattice: Multivariate Data Visualization with R (Use R!)*. Springer, 1st edition, 2008

[3] H Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, New York, 2009. ISBN 978-0-387-98140-6

[4] We'll come on to that later.

Installing packages in R is straightforward. To install **ggplot2** from the command line just use the `install.packages()` command, i.e.

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

The package website

```
http://ggplot2.tidyverse.org/
```

contains a number of useful links to additional resources.

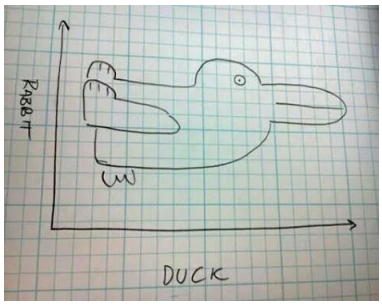

Figure 1.2: Remember: always label your axes.

2
Plot Building

ggplot2 is a bit different from other graphics packages. It roughly follows the *philosophy* of Wilkinson, 1999.[1] Essentially, we think about plots as layers. By thinking of graphics in terms of layers it is easier for the user to iteratively add new components and for a developer to add new functionality.

[1] L Wilkinson. _The Grammar of Graphics. Springer, 1st edition, 1999

The basic plot object

To create an initial ggplot object, we use the `ggplot2()` function. This function has two arguments:

- **data** and
- an aesthetic **mapping**

These arguments set up the defaults for the various layers that are added to the plot and can be empty. For each plot layer, these arguments can be overwritten. The `data` argument is straightforward - it is a data frame[2]. The `mapping` argument creates default aesthetic attributes. For example, the `bond` dataset

[2] **ggplot2** is very strict regarding the `data` argument. It doesn't accept matrices or vectors. The underlying philosophy is that **ggplot2** takes care of plotting, rather than messaging it into other forms. If you want to do some data manipulation, then use other tools.

```
data(bond, package = "jrGgplot2")
```

is a data frame where each row is a particular James Bond movie. So we could *map* the `kills` and `Alcohol_Units` columns to the `x` and `y` coordinates.

```
library("ggplot2")
g = ggplot(data = bond,
           mapping = aes(x = Kills, y = Alcohol_Units))
```

or equivalently[3],

[3] Unlike base graphics, we can store graphs as objects. In this case, the object `g`

```
g = ggplot(bond, aes(Kills, Alcohol_Units))
```

Running the above command generates a blank canvas. Now we'll look at adding layers.

The geom_* functions

Standard aesthetics are x, y, colour and size.

The `geom_` functions are used to perform the actual rendering in a plot. For example, we have already seen that a line geom will create a line plot and a point geom creates a scatter plot. Each geom has a list of aesthetics that it expects. However, some geoms have unique elements. The error-bar geom requires arguments `ymax` and `ymin`. For a full list, see

```
http://ggplot2.tidyverse.org/reference/
```

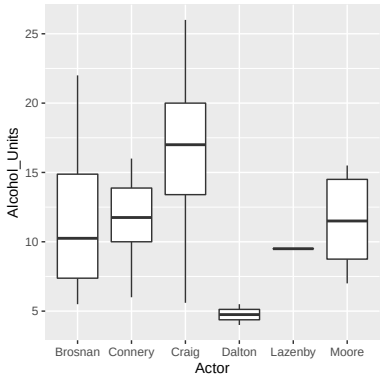

Figure 2.1: Alcohol consumed by various bonds.

Example: combining geoms

Let's look at the `bond` data set in more detail - see `?bond` for a description. We begin by creating a base ggplot object

```
g = ggplot(bond, aes(x = Actor, y = Alcohol_Units))
```

Remember, the above piece of code doesn't do anything[4]. Now we'll create a boxplot using the boxplot geom:

[4] Since `Actor` is discrete we get separate boxplots. If `Actor` was numeric then use the `group` aesthetic.

```
(g1 = g + geom_boxplot())
```

This produces figure 2.1. Notice that the default axis labels are the column headings of the associated data frame. We can have a more colourful boxplot using the `fill` aesthetic

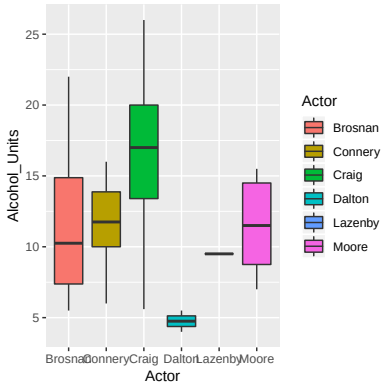

Figure 2.2: More colourful boxplots.

```
## Try colour and group instead of fill
g2 = g + geom_boxplot(aes(fill = Actor))
```

While not that useful in this situation, colour can be useful in boxplots to differentiate different experiment types (say).

Standard Plots

There are a few other standard `geom`'s that are particular useful:

- `geom_point()`: a scatter plot - see figure 2.3.
- `geom_bar()`: produces a standard barplot that counts the `x` values. For example, to generate a bar plot (figure 2.4) of the number of movies per actor, we use the following code:

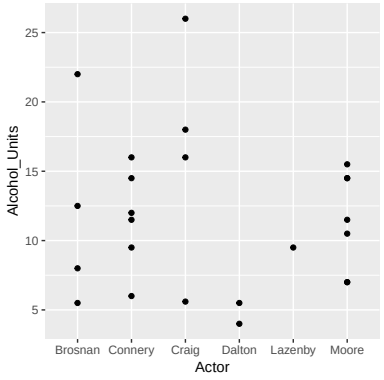

Figure 2.3: Scatter plot of Kills vs Alochol.

```
h = ggplot(bond) + geom_bar(aes(x = Actor))
```

A handy trick with bar charts is to flip the axis

```
h + coord_flip()
```

to get figure 2.5.

- `geom_line()`: a line plot.
- `geom_text()`: adds labels to specified points. This has an additional (required) aesthetic: `label`. Other useful aesthetics, such as `hjust` and `vjust` control the horizontal and vertical position. The `angle` aesthetic controls the text angle.
- `geom_raster()`: Similar to `levelplot()` or `image()`. For example,

```
data(raster_example, package = "jrGgplot2")
g_rast = ggplot(raster_example, aes(x, y)) +
geom_raster(aes(fill = z))
```

generates figure 2.6. If the squares are unequal, then use the (slower) `geom_tile()` function.

What is a geom_*?

A *geom_ _*_* is a single layer that comprises of (at least) four elements:

- an aesthetic and data mapping;
- a statistical transformation (**stat**);
- a geometric object (**geom**);
- and a position adjustment, i.e. how should objects that overlap be handled.

When we use the command

```
geom_point(aes(colour = Actor))
```

this is actually a shortcut for the command:

```
layer(
    data = bond, #inherited
    mapping = aes(colour = Actor), #x,y are inherited
    stat = "identity",
    geom = "point",
    position = "identity",
    params = list(na.rm = FALSE)
)
```

In practice, we **never** use the layer function. Instead, we use

- `geom_*` which creates a layer with an emphasis on the `geom`;
- `stat_*` which create a layer with an emphasis on the `stat`.

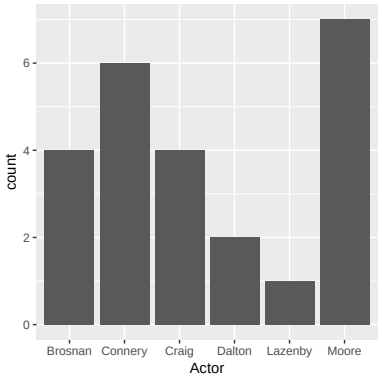

Figure 2.4: Barchart of No. of movies per actor.

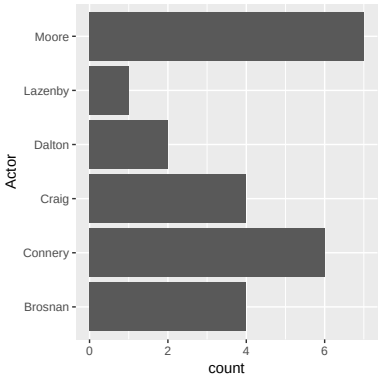

Figure 2.5: Flipped barchart of No. of movies per actor.

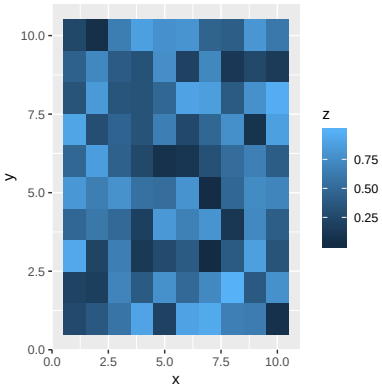

Figure 2.6: Flipped barchart of No. of movies per actor.

Aesthetics

The key to successfully using aesthetics is remembering that the `aes()` function maps data to an aesthetic. If the parameter is not data or is constant, then don't put it in an aesthetic. Only parameters that are inside of an `aes()` will appear in the legend. To illustrate these ideas, we'll generate a simple scatter-plot

```
d = data.frame(x = 1:50, y = 1:50, z = 0:9)
g_aes = ggplot(d, aes(x = x, y = y))
g_aes + geom_point(aes(colour = z))
```

which gives figure 2.6. Here the `z` variable has been mapped to the `colour` aesthetic. Since this parameter is continuous, `ggplot2` uses a continuous colour palette. Alternatively, if we make `z` a factor or a character, `ggplot2` uses a different colour palette

```
g_aes + geom_point(aes(colour = factor(z)))
```

to get figure 2.7. If we set the aesthetic to a constant value (figure 2.8)

```
g_aes + geom_point(aes(colour = "Blue"))
```

the resulting plot is unlikely to be what we intended. The value `'blue'` is just treated as a standard factor. Instead, you probably wanted

```
g_aes + geom_point(colour = "Blue")
```

Another important point, is that when you specify mappings inside `ggplot(aes())`, these mappings are inherited by every subsequent layer. This is fine for `x` and `y`, but can cause trouble for other aesthetics. For example, using the `colour` aesthetic is fine for `geom_line()`, but may not be suitable for `geom_text()`.

There are few standard aesthetics that appear in most, but not all, `geom`'s and `stat`'s (see table 2.1). Individual `geom`'s can have additional optional and required aesthetics. See their help file for further information.

Aesthetic	Description
`linetype`	Similar to `lty` in base graphics
`colour`	Similar to `col` in base graphics
`size`	Similar to `size` in base graphics
`fill`	See figure 2.6.
`shape`	Glyph choice
`alpha`	Control the transparency

Table 2.1: Standard aesthetics. Individual `geom`'s may have other aesthetics. For example, `geom_text` uses `label` and `geom_boxplot` has, amongst other things, `upper`.

The `stat_` functions

The `stat_` functions focus on transforming data. For example, the `stat_smooth()` function uses loess smoother [5] function:

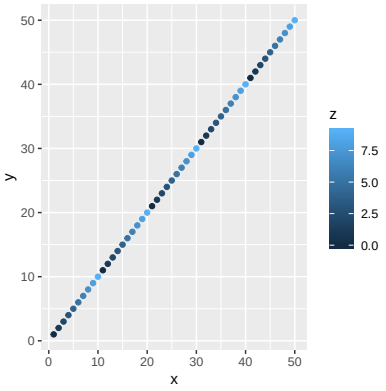

Figure 2.7: Illustration of the continuous colour aesthetic.

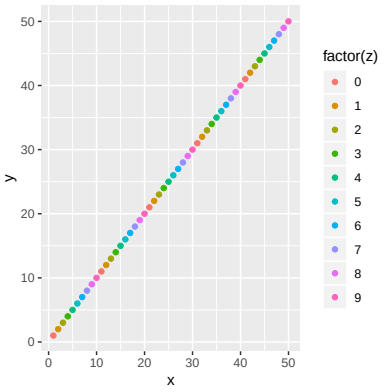

Figure 2.8: Illustration of the continuous discrete aesthetic.

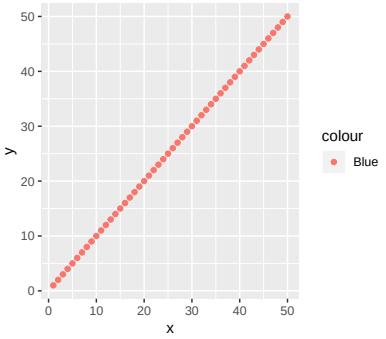

Figure 2.9: Illustration of the continuous constant aesthetic.

[5] A loess smoother is a non-parametric method for smoothing data. It is called local regression because value at point x is weighted toward the data nearest to x.

```
ggplot(bond, aes(x = Alcohol_Units, y = Kills)) +
  stat_smooth()
```

which gives 2.10. Surprisingly(?) the number of fatalities caused by Bond isn't influenced by alcohol.

Figure 2.10: Alcoholic doesn't seem to influence the number of kills in James Bond movies.

[6] Used by both `geom_boxplot` and `stat_boxplot`.

ALL GEOMS HAVE stats and, vice visa, all stats have geoms. A stat takes a dataset as input and returns a dataset as an output. For example, the boxplot stat[6] takes in a data set and produces the following variables:

- lower
- upper
- middle
- ymin: bottom (vertical minimum)
- ymax: top (vertical maximum).

Typically these statistics are used by the boxplot geom. Equally, (some of them) could be used by the error bar geom.

Example: combining stats

Perhaps the easiest `stat_*` to consider is the `stat_summary()` function. This function summarises `y` values at every unique `x` value. This is quite handy, for example, when adding single points that summarise the data or adding error bars.
A simple plot to create, is the mean alcohol consumption per actor (figure 2.11)

```
ggplot(bond, aes(Actor, Alcohol_Units)) +
             stat_summary(geom = "point", fun.y = mean)
```

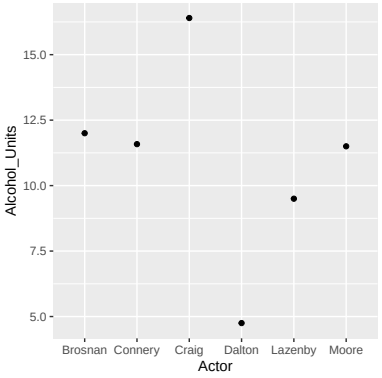

Figure 2.11: Average number of units consumed per actor.

In the above piece of code we calculate the mean number of alcohol units consumed by each Actor. These x-y values are passed to the `point` geom. We can use any function for `fun.y` provided it takes in a vector and returns a single point. For example, we could calculate the range of values, as in figure 2.12:

```
ggplot(bond, aes(Actor, Alcohol_Units)) +
    stat_summary(geom = "point",
                 fun.y= function(i) max(i) - min(i))
```

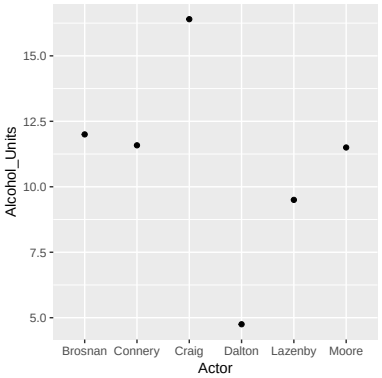

Figure 2.12: Plot of the range for each actor.

Or we could work out confidence intervals for the mean number of Units consumed (figure 2.13):

```
## Standard error function
std_err = function(i)
    dt(0.975, length(i) - 1) * sd(i) / sqrt(length(i))

ggplot(bond, aes(x = Actor, y = Alcohol_Units)) +
    stat_summary(fun.ymin = function(i) mean(i) - std_err(i),
                 fun.ymax = function(i) mean(i) + std_err(i),
```

```
                colour = "steelblue", geom = "errorbar",
                width = 0.2, lwd = 2) +
    ylim(c(0, 20))
```

To calculate the bounds, we work out the standard deviation (`sd(i)`), then number of movies per actor (`length(i)`) and the correct value from the t distributions, with $n - 1$ degrees of freedom.

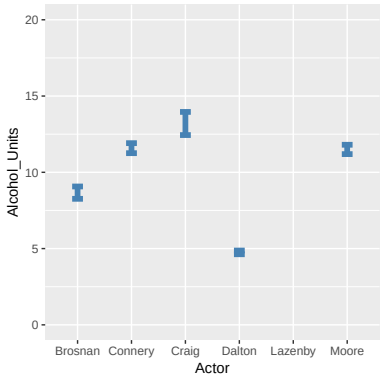

Figure 2.13: Confidence intervals for the mean number of units consumed by each actor.

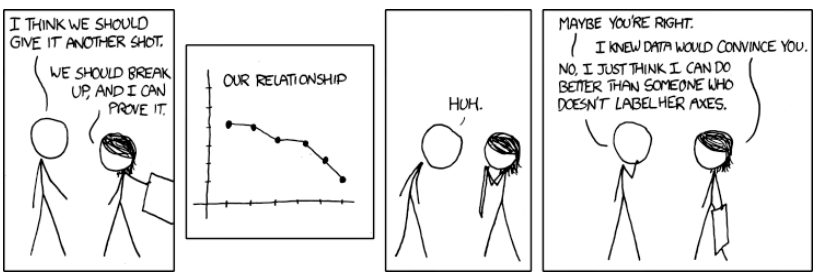

Figure 2.14: http://xkcd.com/833/

3
Facets

Introduction

Faceting is a mechanism for automatically laying out multiple plots on a page. The data is split into subsets, with each subset plotted onto a different panel. **ggplot2** has two types of faceting:

- `facet_grid()`: produces a 2d panel of plots where variables define rows and columns.
- `facet_wrap()`: produces a 1d ribbon of panels which can be wrapped into 2d.

Facet grid

The function `facet_grid()` lays out the plots in a 2d grid. The faceting formula specifies the variables that appear in the columns and rows. Suppose we are interested in movie length. A first plot we could generate is a basic histogram:

```
data(movies, package = "ggplot2movies")
g = ggplot(movies, aes(x=length)) +  xlim(0, 200) +
  geom_histogram(aes(y=..density..), binwidth=3)
```

This produces figure 3.1. Notice that we have altered the x-axis since there are a couple of outlying films and adjusted the binwidth in the histogram. We have also used density as the y-axis scale. This just means that the area under the histogram sums to one. The data is clearly bimodal. Some movies are fairly short, whilst others have an average length of around one hundred minutes.
We will now use faceting to explore the data further.

- `y ~ .`: a single column with multiple rows. This can be handy for double column journals. For example, to create histograms conditional on whether they are comedy films, we use:

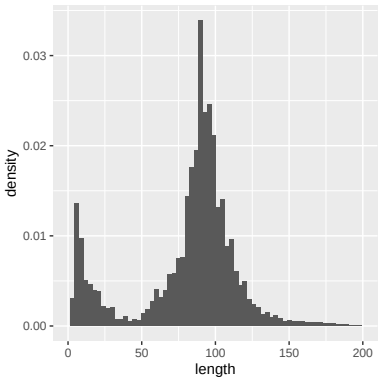

Figure 3.1: A histogram of movie length

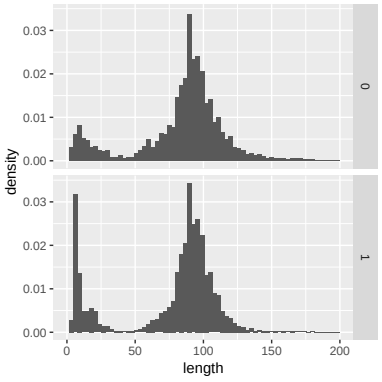

Figure 3.2: Movie length conditional on whether it is a comedy.

```
g + facet_grid(Comedy ~ .)
```

This gives figure 3.2.

- `. ~ x`: a single row with multiple columns. Very useful in wide screen monitors. In this piece of code, we create histograms conditional on whether the movie was animated:

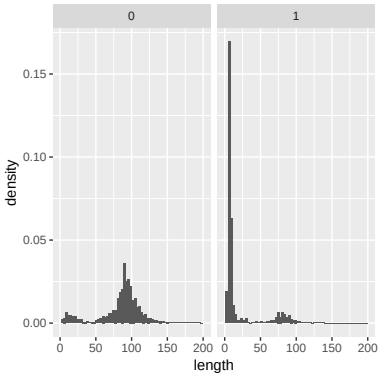

Figure 3.3: Movie length conditional on animation.

```
g + facet_grid(. ~ Animation)
```

From figure 3.3, it's clear that the majority of short films are animations. For illustration purposes, we have used the `geom_density()` function in figure 3.3.

- `y ~ x`: multiple rows and columns. Typically the variable with the greatest number of factors is used for the columns. We can also add marginal plots when using `facet_grid()`. By default, `margin=FALSE`.

```
g + facet_grid(Comedy ~ Animation)
```

Figure 3.4 splits movie length by comedy and animation. The panel labels aren't that helpful - they are either `0` or `1`. By default **ggplot2** uses the values set in the data frame. Typically I use more descriptive names in my data frame so the default is more appropriate.

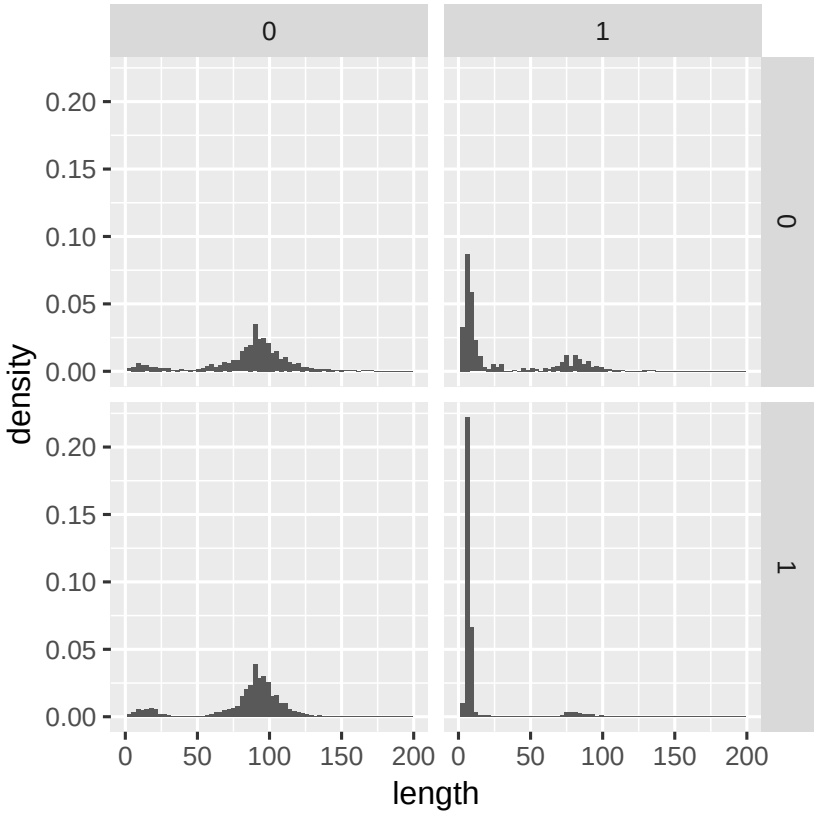

Figure 3.4: Movie length conditional on animation and action status

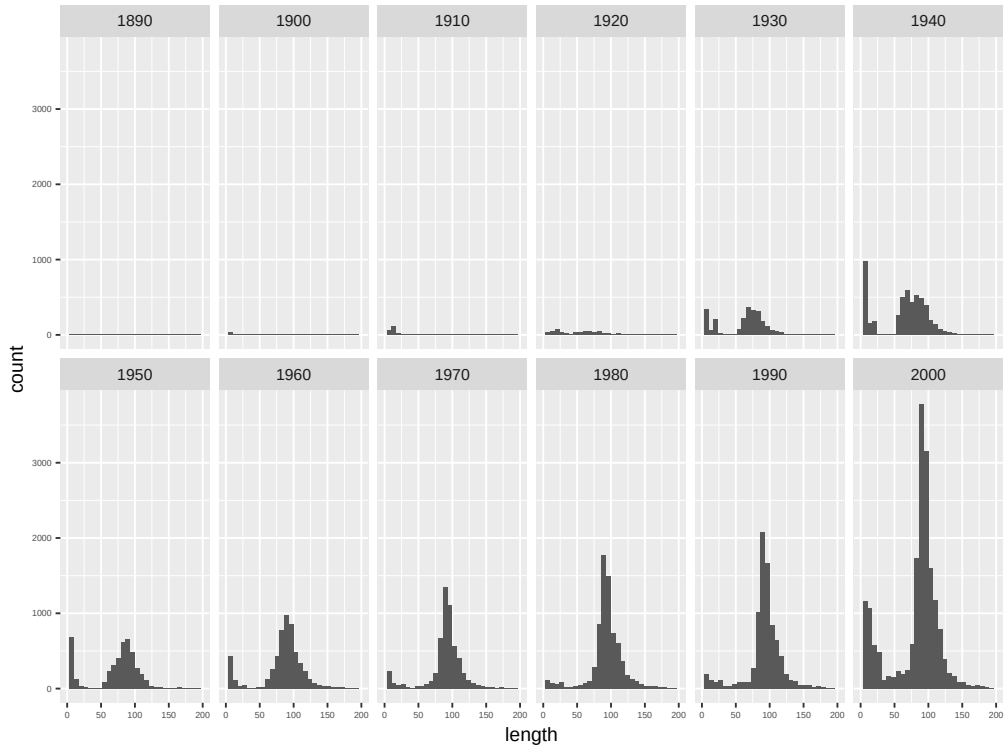

Figure 3.5: Movie length conditional on the decade the movie was created.

Controlling facet scales

For both `facet_grid()` and `facet_wrap()` we can allow the scale to be the same in all panels (fixed) or vary between panels. This is controlled by the `scales` parameter in the `facet_*()` function:

- `scales = 'fixed'`: x and y scales are fixed across all panels (default).
- `scales = 'free'`: x and y scales vary across all panels.
- `scales = 'free_x'`: the x scale is free.
- `scales = 'free_y'`: the y scale is free.

We will experiment with these in the practical session.

Facet wrap

The `facet_wrap()` function creates a 1d ribbon of plots. This can be quite handy when trying to save space. To illustrate, let's examine movie length by decade. First, we a create new variable for the movie decade:

```
movies$decade = round(movies$year/10) * 10
```

Then to generate the ribbon of histograms histograms, we use the `facet_wrap()` function:

```
ggplot(movies, aes(x = length)) + geom_histogram() +
  facet_wrap( ~ decade, ncol=6) + xlim(0, 200)
```

to figure 3.5. As before, we truncate the x-axis. Since we have counts on the

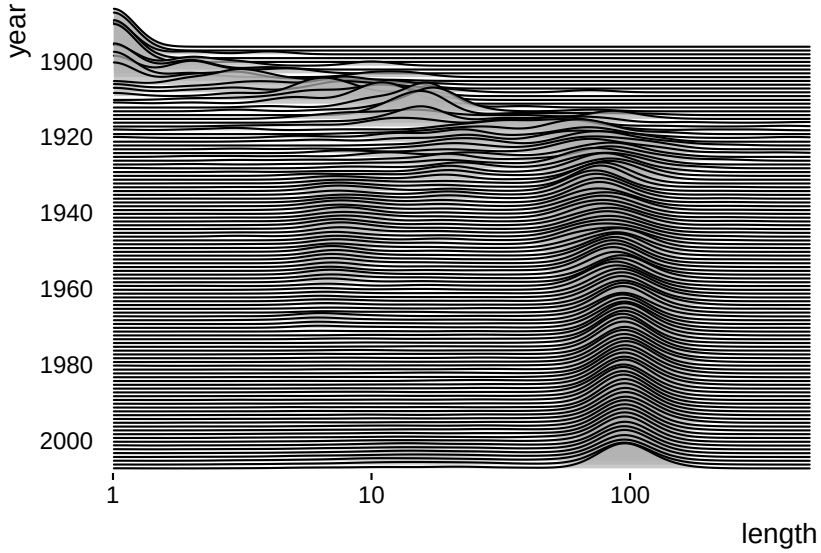

Figure 3.6: A ridge plot of the movie data.

y-axis, we notice that the number of movies made has increased through time. Also, shorter movies were popular in the 1950's and 1960's.

Ridge plots

Ridge plots are partially overlapping line plots that create the impression of a mountain range[1]. They can be quite useful for visualizing changes in distributions over time or space.

[1] This doesn't actually fit that well into this chapter, but Ridge plots are *sort of* facets.

In the following code, we calculate a denisty for movie length for every year. If we tried to display this using facets, we would have over 100 panels. A "Ridge" plot circumvents this issue by having a overlapping plots, and dropping the axis labels

```
library("ggridges")
ggplot(movies,
       aes(x = length, y = year,
           group = year, height = ..density..)) +
    geom_density_ridges(scale = 10, alpha = 0.7) +
    theme_ridges(grid=FALSE) +
    scale_x_log10(limits = c(1, 500),
                  breaks = c(1, 10, 100, 1000),
                  expand = c(0.01, 0)) +
    scale_y_reverse(breaks = seq(2000, 1900, by = -20),
                    expand = c(0.01, 0))
```

In the next chapter we'll cover `theme`'s and `scale`'s.

Figure 3.7: http://xkcd.com/2029/

4
Scales

Axis Scales

When we create complex plots involving multiple layers, **ggplot2** uses an iterative process to calculate the correct scales. For example, if we only plotted the regression lines, **ggplot2** would reduce the y-axis scale. We can specify set scales using the `xlim()` and `ylim()` functions. However, if we use these functions, any data that falls outside of the plotting region isn't plotted **and** isn't used in statistical transformations. For example, when calculating the `binwidth` in histograms. If you want to zoom into a plot region, then use `coord_cartesian(xlim = c(.., ..))` instead.

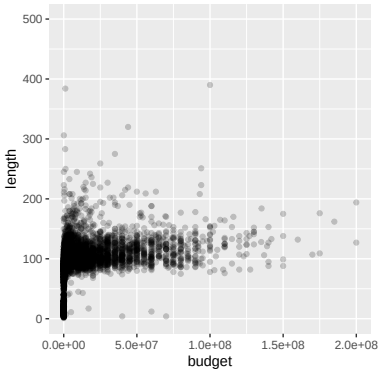

Figure 4.1: Scatter plot of movie budget against length

At times we may want to transform the data. A standard example is the $\log_{10}$ transformation. Suppose we wanted to create a scatter plot of length against budget. We remove any movies that have a zero and known budget. Then we use the following commands

```
data(movies, package = "ggplot2movies")
known_budget = movies[!is.na(movies), ]
h = ggplot(known_budget, aes(y = length)) + ylim(0, 500)
h1 = h + geom_point(aes(budget), alpha = 0.2)
```

to get figure 4.1. Notice that we have changed the alpha transparency value to help with over plotting.

To plot the log budgets, there are two possibilities. First, we could transform the scale

```
h2 = h + geom_point(aes(log10(budget)), alpha = 0.2)
```

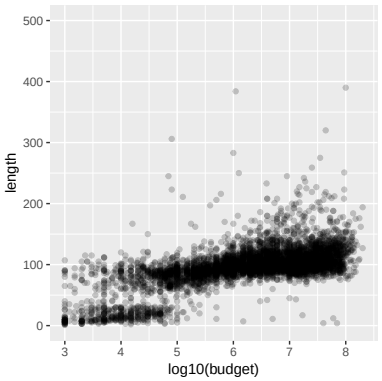

Figure 4.2: Scatter plot of movie log10(budget) against length.

to get figure 4.2. Note that `ylim(0, 500)` is shorthand for

```
scale_y_continuous(limits = c(0, 500))
```

Alternatively, we can transform the data

```
h3 = h1 + scale_x_log10()
## Or equivalently
h1 + scale_x_continuous(trans = "log10")
```

to get figure 4.3. Figures 4.2 and 4.3 are identical, but in figure 4.3 we are still using the original scale. To generate figure 4.3 we used `scale_x_log10()` which is simply the shorthand version of

```
scale_x_continuous(trans = "log10")
```

function. Some standard scale transformations are given in table 4.1. As an aside, the `scale` functions are fundamentally different from `geom`'s, since they don't add a layer to the plot.
The `scale_*()` functions can also adjust the tick marks and labels. For example,

```
h4 = h3 +
  scale_y_continuous(breaks = seq(0, 500, 100),
                     minor_breaks = seq(0, 500, 25),
                     limits = c(0, 500),
                     labels = c(0, "", "", "", "", 500),
                     name = "Movie Length")
```

gives figure 4.4[1]. If you just want to change the x-axis limits or name, then you can use the convenience functions `xlim()` and `xlab()`. There are similar functions for the y-axis.

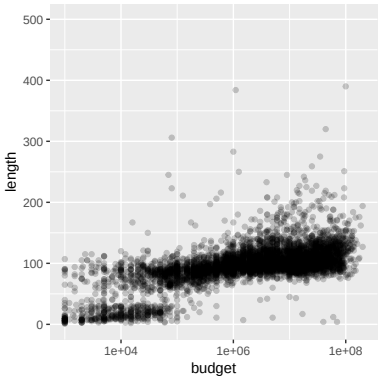

Figure 4.3: Scatter plot of movie budget against length, with the budget data transformed.

[1] Using `scale_y_continous()` gives us more control of tick marks and grid lines.

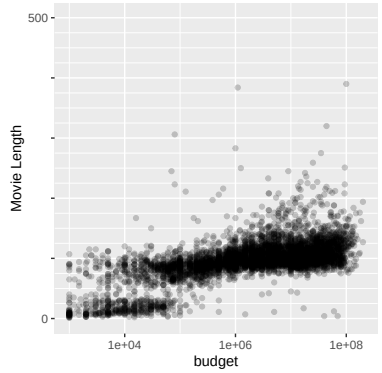

Figure 4.4: Scatter plot of movie budget against length.

Function	Description
`*_continuous(...)`	Main scale function.
`*_log10(...)`	$\log_{10}$ transformation.
`*_reverse(...)`	Reverse the axis.
`*_sqrt(...)`	The square root transformation.
`*_datetime(...)`	Precise control over dates and times.
`*_discrete(...)`	Not usually needed - see §6.3 of Wickham, 2009.

Table 4.1: Standard scales in **ggplot2**. In the above, replace `*` with either `scale_x` or `scale_y`. Common arguments are `breaks`, `labels`, `na.value`, `trans` and `limits`. See the help files for further details.

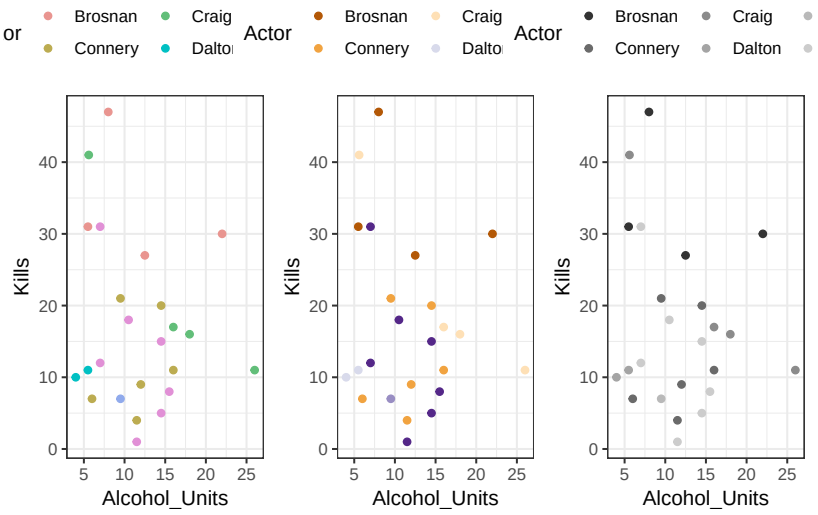

Figure 4.5: Scatter plots of the bond data set showing different colour schemes. The theme has been changed to $theme_bw()$

Colour and fill scales

For discrete data, there are two methods for choosing colour schemes. One that chooses colours in an automated way and another from hand-picked sets. The default is `scale_colour_hue()`, which picks evenly spaced hues around the hcl colour scheme.

Discrete colours

As a test, we will use the scatter plot from chapter 2:

```
data(bond, package = "jrGgplot2")
g = ggplot(bond, aes(x = Alcohol_Units, y = Kills)) +
  geom_point(aes(colour = Actor))
```

We can alter the hue and intensity of the colours (figure 4.5a):[2]

```
g + scale_colour_hue(l = 70, c = 60)
```

or use predefined colour palettes from colour brewer (figure 4.5b):[3]

```
g + scale_colour_brewer(palette = "PuOr", type = "div")
```

or specify our own colour schemes[4]

```
# Colours for amusement rather than clarity
g + scale_colour_manual(values = c(
    "Brosnan" = rgb(192,192,192, maxColorValue = 255), #silver
    "Connery" = "Gold",
    "Craig" = rgb(205, 127, 50, maxColorValue = 255), #Bronze
    "Dalton" = "tomato1",
    "Lazenby" = "tomato2",
    "Moore" = "tomato3"))
```

For black and white, you can always use:

[2] If you want to change the `fill` aesthetic, use `scale_fill_*()`.

[3] There are three possible types: `seq` (sequential), `div` (diverging) and `qual` (qualitative). See `http://colorbrewer2.org/` for other palettes.

[4] I try to avoid specifying my own colours as it's a bit clunky.

```
g + scale_colour_grey()
```

to get figure 4.5c.

Continuous Colour

When we have continuous parameters, we use a gradient of colour, instead of discrete values. There are three types of continuous colour gradients[5]:

[5] The * can be either `fill` or `colour`.

- `scale_*_gradient()`: a two colour gradient, with arguments `low` and `high` to control the end points.
- `scale_*_gradient2()`: a three colour gradient. As above, with additional arguments: `mid` (for the colour) and `midpoint`. The midpoint defaults to 0, but can be set to any value.
- `scale_*_gradientn()`: an n-colour gradient. This requires a vector of colours, which default to being evenly spaced.

See the associated help pages for examples.

Multiple plots

When we want to create a figure in base graphics that contains multiple plots, we use the `par()` function. For example, to create a 2×2 plot, we would use

```
par(mfrow=c(2, 2))
```

In ggplot2, we can do something similar. Using the **gridExtra** package, we have

```
library("gridExtra")
grid.arrange(g1, g2, g3, g4, nrow=2)
```

where `g1`, `g2`, `g3` and `g4` are standard ggplot2 graph objects.

An alternative way of creating figure grids, is to use viewports. First, we load the **grid** package and create a convenience function.

Using viewports gives you more flexibility, but is more complicated.

```
vplayout = function(x, y)
  viewport(layout.pos.row = x, layout.pos.col = y)
```

Next we create a new page, with a 2×2 layout

```
library("grid")
grid.newpage()
pushViewport(viewport(layout = grid.layout(2, 2)))
```

Finally we add the individual graphics. The plot created using the `h` object, is placed on the first row and spans both columns:

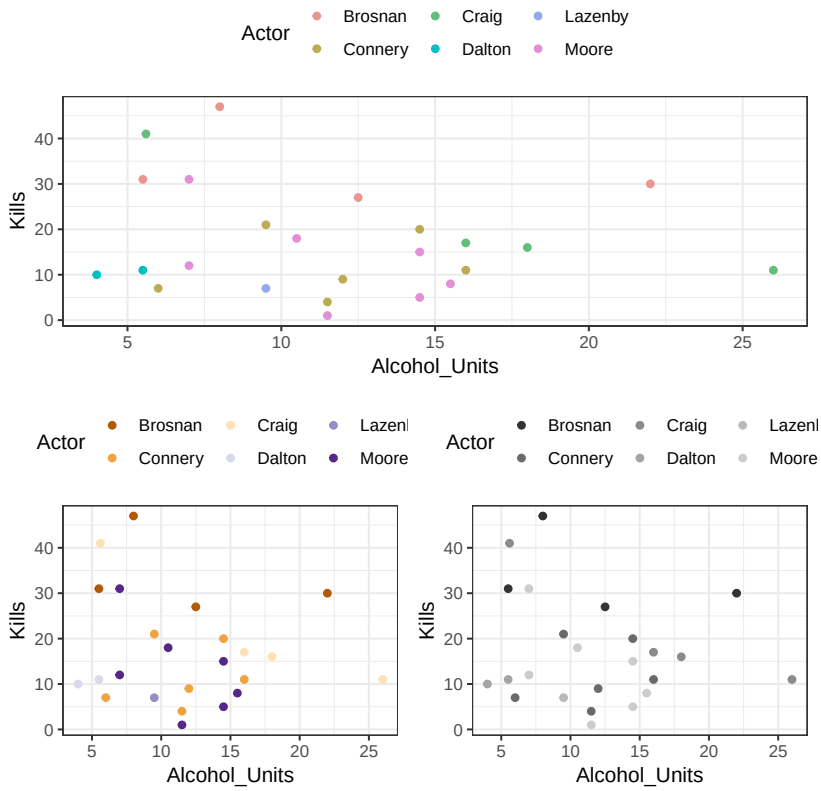

Figure 4.6: An example plot using the viewports. The top plot is spans two columns.

```
print(g1, vp = vplayout(1, 1:2))
```

The other figures are placed on the second row (figure @ref(fig:F5-6}):

```
print(g2, vp = vplayout(2, 1))
print(g3, vp = vplayout(2, 2))
```

5
Themes

A theme in ggplot2 controls the font, legends, background, text and panel. The rationale behind the default, `theme_grey()` was to put the data forward, while still making the grid lines visible. However, many people dislike this style.

What themes are available?

There are eight other themes that come with **ggplot2**. The default is `theme_grey()`[1]. My standard theme is `theme_minimal()`. Figure 5.1 shows the eight different themes for a standard scatter plot.

[1] Or `theme_gray()`

Themes in other packages

The R package **ggthemes**[2] Contains a large number of different themes, in-

[2] Installed in the usual way, `install.package('ggthemes')`.

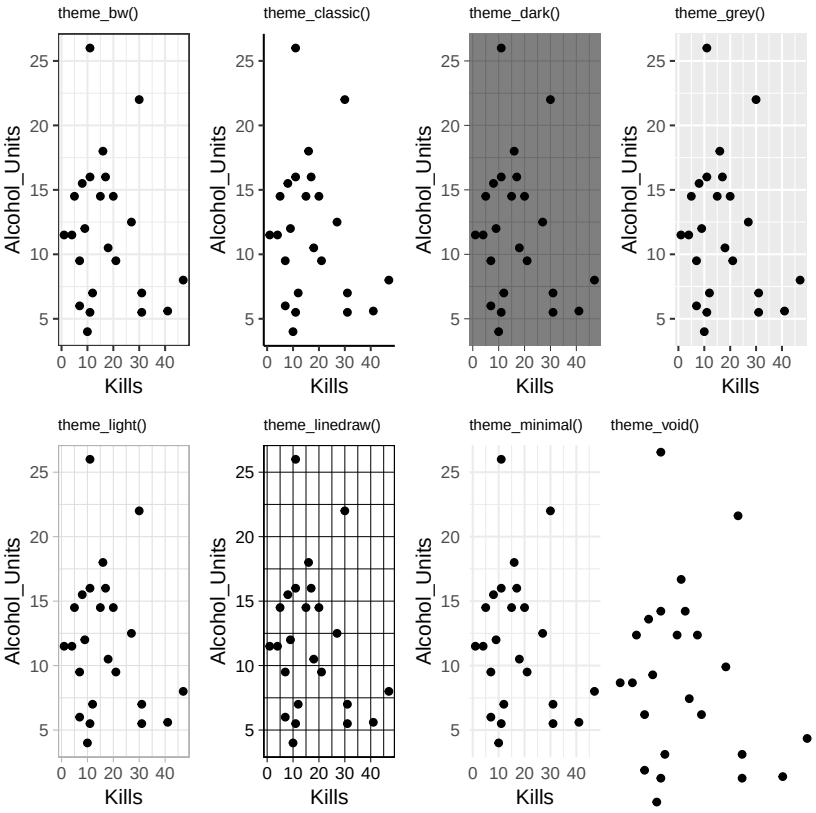

Figure 5.1: The eight ggplot2 themes that come with **ggplot2**. The theme name is above each figure.

cluding `theme_excel()`! The easiest way to view the themes is via the website

```
https://github.com/jrnold/ggthemes
```

Personally, I don't use any of the themes in **ggthemes**. However, the underlying R code

```
https://github.com/jrnold/ggthemes/tree/master/R
```

is very useful for creating your own theme[3]

Similar to **ggthemes**[4] (but far less popular) is **ggthemr**

```
https://github.com/cttobin/ggthemr
```

Again, useful for *borrowing* particular elements of a plot.

The final theme package is **hrbrthemes**. The tag line, "Opinionated, typographic-centric ggplot2 themes and theme components". The package can be installed from CRAN, and example themes are visible at

```
https://github.com/hrbrmstr/hrbrthemes
```

I like and use this theme (although I do have font issues under Linux) As promised, the theme is minimal, but quite nice.

[3] More on this in the next section.

[4] Not on CRAN.

Creating your own theme

A ggplot2 theme is simply an R list that specifies the plot elements. For example, just type

```
theme_minimal()
```

in your console and you can see the individual elements. The theme is made up of main components, .e.g

```
names(theme_minimal())
```

If possible, try to modify an existing theme. For example

```
my_theme = theme_minimal()
my_theme$panel.grid.minor$colour = "red"
g + my_theme
```

For larger changes, consult the **ggplot2** book.