Seminar 6: ggplot

Olivia Burge

October, 2014

Introduction

ggplot is a package for R which allows finer control over plots and plotting. It follows the "grammar of graphics", a system initially proposed by Leland Wilkinson. This system essentially means you tell R which data you are (primarily) dealing with, and what the axes represent. From there, you build up or "layer" your plot.

Necessary boring things

First, we need to load some packages. If you don't have them installed, you'll need to delete the "#" sign infront of the `install.packages()` commands and run those too.

```
# install.packages('ggplot2')
# install.packages('vegan')
require(ggplot2)
require(vegan)
require(ggthemes)
require(dplyr)
require(tidyr)
require(grid)
require(gridExtra)
require(reshape2)
```

"require" or "library" mean the same thing (for our purposes). You cannot call a package using either of those terms if you haven't already installed it.

Data set up and inspection

First, we have a look at the mite dataset in vegan, which we'll be using:

```
data(mite)
data(mite.env)
mite_species <- mite
mite_env <- mite.env
mite_all <- cbind(mite_species, mite_env)
```

Run the following yourself to visualise some of the data:

```
head(mite_species[1:5], 3)  #just the first 5 columns
head(mite_env, 3)
head(mite_all)
names(mite_all)
```

In the deep end - intro to plotting

Time to do some plotting. We either set up the plot first as an object, or whack it all into one command. If you create it all at once, and assign it to an object (e.g. `plot1 <- ggplot(...)`), it won't print automatically. To make it print automatically you can:

- surround the object in brackets; or
- add a line of code after creating the object with the name of the object.

```
miteplot1 <- ggplot(data = mite_all, aes(y = Brachy,
    x = WatrCont))
miteplot1 + geom_point()
```

For this figure we wrap the lines in brackets to make it print automatically:

```
(miteplot2 <- ggplot(data = mite_all, aes(y = SubsDens,
    x = WatrCont)) + geom_point(colour = "bisque3"))
```

That all looks easy enough. But often you want to colour by a factor, and not use the horrendous grey background:

```
miteplot3 <- ggplot(data = mite_all,
                    aes(y = SubsDens, x = WatrCont,
                        colour = Substrate)) +
  geom_point(size = 5)+
  theme_bw() +
  scale_colour_brewer(type = "qual",
                      palette = "Set2",
                      guide = guide_legend(nrow = 2)) +
  theme(legend.position = "bottom",
        legend.key.size = unit(0.3, "cm"),
        legend.key = element_blank(),
        legend.text = element_text(size = 7))
miteplot3 # using the name of the object to print it
```

Other `theme_` options include `theme_classic()` (similar to base graphics) and `theme_minimal()`.

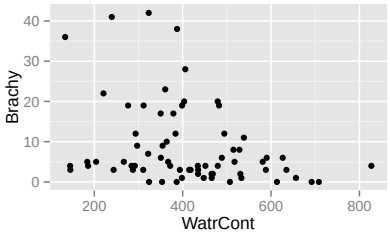

Figure 1: Brachy abundance by water content

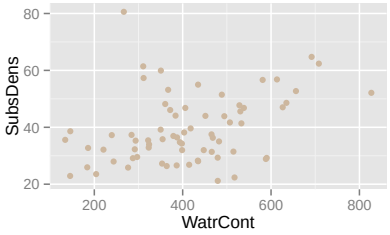

Figure 2: Substrate density against water content

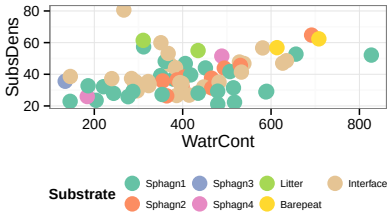

Figure 3: Using a plainer background

The structure of a `ggplot` object

Now that you have a feel for how `ggplot` works, we can consider the structure a little further:

- `geom` sets up the layers of the plot
  - line
  - point
  - bar (etc)
- `scale` **configures** the layers
  - e.g. scale_colour_manual
- `theme` sets up the appearance of the plot

In addition, aes() is used sometimes. The easy way to remember is that you use it where you are referring to things within your data. We used it above for the x, y, and colour by topo instructions, **because they were items ggplot had to find in our data**. When we referring to colours that were not in our data, we left it *outside* the aes() call, but *put it in quotations* - see `geom_point(colour = "bisque3")`, above. Numeric constants (e.g., `size = 2`) do not need quotations.

Different types of plots

The various forms of plots (more than you may even need) may be found at here.[1] Some of the common ones are:

[1] `http://docs.ggplot2.org/current/`

- bar graph - `geom_bar(stat = "identity")` - you will need the `stat` part
- histogram - `geom_histogram()` - don't specify a `y =` in this case
- boxplot - `geom_boxplot()`
- line graph - `geom_line()` - if the lines don't join up you probably need to specify the group
- points - `geom_point()`
- error bars/CIs - `geom_errorbar()`- you need to precalculate these, normally

Have a go at running some of these:

```
ggplot(mite_all, aes(y = Brachy, x = Shrub)) +
    theme_bw() + geom_bar(stat = "identity")
```

```
ggplot(mite_all, aes(x = Brachy)) + theme_bw() +
    geom_histogram()
```

```
#sum of a number of species:
ggplot(mite_all, aes(
  y = Eupelops + HPAV +
    Brachy + PLAG2 + Oppiminu,
  x = Shrub)) +
  theme_bw() +
  ylab("sum counts of many species") +
  geom_boxplot(notch = TRUE)
```

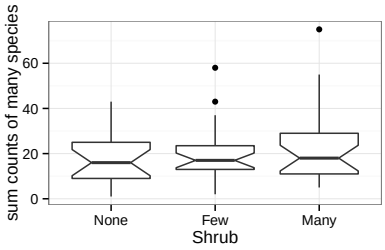

Figure 4: Notched boxplot

```
# can have vertical and horizontal lines too
ggplot(mite_env, aes(y = SubsDens, x = WatrCont)) +
    theme_bw() + geom_vline(aes(xintercept = 500),
    colour = "orange") + geom_point() + geom_hline(aes(yintercept = 50),
    linetype = "dotdash", colour = "royalblue",
    size = 2)

#or specify manually the slope of a line
ggplot(mite_env, aes(y = log(SubsDens), x = log(WatrCont))) +
  theme_bw() +
  geom_point() +
  geom_abline(slope = 0.3, intercept = 2)

# error bars to be covered later
```

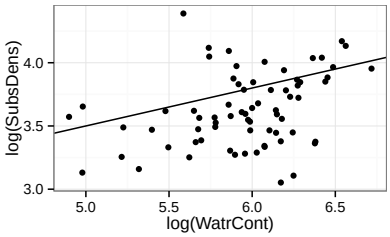

Figure 5: geom_abline()

Customising colour, shape, linetype, alpha and size

We can either set these in relation to a variable in our dataframe (in which case it will go inside the aes() call for the layer), or as a constant (in which case it goes outside.)

```
# for example, setting all the points to one
# colour:
miteplot1 + geom_point(colour = "cornflowerblue",
    aes(shape = Topo))

# for example, setting the points to a shape
# depending on a variable
miteplot1 + geom_point(aes(shape = Topo))
# factor

miteplot1 + geom_point(aes(colour = SubsDens))
# continuous
```

`scale_colour_manual` allows us to specify the colours by name (or number).

`ggplot` uses the same colours as base graphics. You can see them all at this website, [2] or give this a go: `demo("colors")` or for all the names, `colors()`.

[2] https://www.stat.auckland.ac.nz/~ihaka/downloads/R-colours-letter.pdf

```
miteplot1 +
  geom_point(aes(colour = Shrub, shape = Shrub), size = 3) +
  scale_colour_manual(values = c("forestgreen",
                                 "orange",
                                 "royalblue")) +
  scale_shape_manual(values = c(8, 18, 3))+
  theme_classic()
```

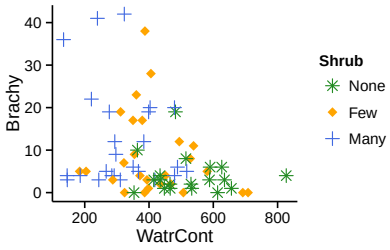

Figure 6: specifying shape & colour

Other examples of customisation

Here, we build on the `miteplot1` by calling it first, then adding some layers and theme adjustments.

```
miteplot1 +
  geom_point(aes(colour = Topo))+
  scale_colour_manual(values = c("royalblue", "orange")) +
  theme(legend.position = "bottom",
        legend.key = element_blank(),
        legend.background = element_rect(colour = "gray"))
```

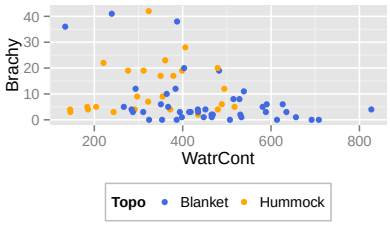

Figure 7: Specifying colours; legend position

We can add basic regression lines (the equivalent of `abline` in base graphics) using the `geom_smooth` option. The default is a loess smooth which is useful to see local changes in the data, although an lm smooth is probably more common.

```
miteplot1 +
  geom_point(aes(colour = Topo))+
  geom_smooth(method = "lm", aes(colour = Topo)) +
    guides(colour=guide_legend(override.aes=list(fill = NA))) +
  scale_colour_manual(values = c("royalblue", "orange")) +
  theme_bw()+
  theme(legend.position = "bottom", legend.key = element_blank(),
        legend.background = element_rect(colour = "gray"))
```

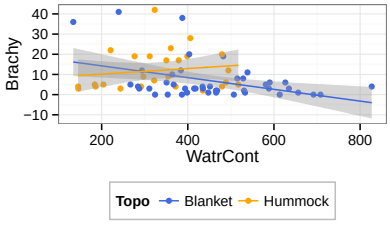

Figure 8: Adding a regression line

Facetting

Sometimes we want to make multiple graphs based on one factor. For example, substrate density, shrub, and topo. For this we use the `facet_wrap` command.

```
ggplot(mite_all, aes(y = Brachy, x = SubsDens)) +
  geom_point(aes(shape = Topo)) +
  geom_smooth(aes(linetype = Topo),
              colour = "black",
              method = lm,
              formula = y ~ poly(x, 2)) +
  scale_shape_manual(values = c(1, 16)) +
  facet_wrap(~Shrub) +
  theme_bw() +
  theme(legend.position = "bottom",
        legend.background = element_rect(colour = "grey"),
        legend.key = element_blank())
```

We can also facet using two variables, using `facet_grid`.

```
ggplot(mite_all, aes(y = Brachy, x = SubsDens)) +
  geom_point(aes(shape = Topo)) +
  geom_smooth(aes(linetype = Topo),
              colour = "black",
              method = lm,
              formula = y ~ poly(x, 2)) +
  scale_shape_manual(values = c(1, 16)) +
  guides(shape = FALSE, linetype = FALSE) +
  facet_grid(Topo~Shrub) +
  theme_bw()
```

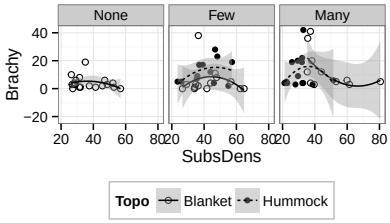

Figure 9: Facet wrap, scale_shape

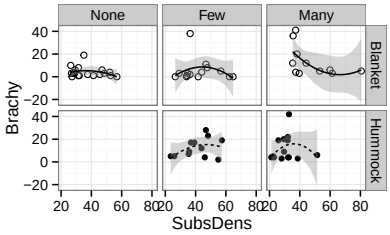

Figure 10: Facet grid, scale_shape

Reshaping data

As another example, we might want to see how three species (Brachy, and two others) react to substrate density and shrub types. To do this, we need to turn the three columns of species into one column. This is "reshaping" our data from wide to long. This example uses the `tidyr` package, but the `reshape2` package is another good option. See the r cookbook [3] for an example.

[3] `http://www.cookbook-r.com/Manipulating_data/Converting_data_between_wide_and_long_format/`

```
# names(mite_all) #check which ones we want
mite_long <- gather(mite_all[c(1:3, 36:40)], key = "species",
    value = "count", Brachy:HPAV)
dim(mite_all)

## [1] 70 40

dim(mite_long)

## [1] 210   7
```

Note that we excluded all the other species. We could have selected *all* the species, it just would have ended in a very large dataframe.

We might like a bar plot by species we use the new dataframe we've created:

```
ggplot(mite_long, aes(y = count, x = Shrub)) +
  geom_bar(aes(fill = species), stat = "summary", fun.y = "sum")+
  facet_wrap(~species) +
  coord_flip()+
  theme_bw() +
  scale_fill_colorblind(guide = FALSE)


#uses colours from ggthemes package
```

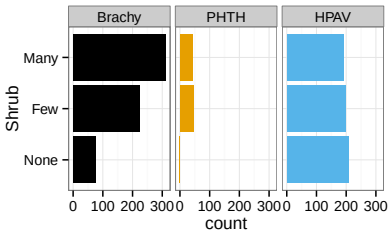

Figure 11: Flipped bar chart

Which is nice. It's summed the multiple observations for each group. But we often want to display the mean by group, and use error bars. We use the `dplyr` package for this.

```
mite_summarised <- mite_long %>%
  group_by(species, Shrub, Topo) %>%
  summarise(
    mean_count = mean(count),
    sd = sd(count),
    se = sd(count)/sqrt(length(count)) # n() is another option you may see instead of length
    )
#head(mite_summarised, 3) # first 3 rows
```

We have returned a dataframe which has the mean (& sd & se) for each combination of `species`, `Shrub`, and `Topo` in our data. We can now use this dataframe to plot:

```
ggplot(mite_summarised, aes(x = Topo, y = mean_count))+
  geom_bar(stat = "identity", aes(fill = species)) +
  geom_errorbar(aes(ymin = mean_count - se,
                    ymax = mean_count + se)) +
  theme_bw() +
  scale_fill_colorblind() +
  facet_wrap(~Shrub)


# well, that didn't work so well. You can see all three species
# are stacked on to each other.
```

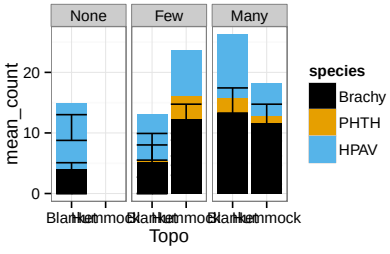

Figure 12: Bar chart with SE, 1

This time, we specify `position_dodge()` for *both* the error bars and the bars to make them separate by a reasonable amount. Here we specify it to be 0.9, you can try other numbers.

```
ggplot(mite_summarised, aes(x = Topo, y = mean_count))+
  geom_bar(aes(fill = reorder(species, mean_count)),
           stat = "identity",
           position = position_dodge(0.9)) +
  geom_errorbar(aes(ymin = mean_count - se, ymax = mean_count + se,
                    group = reorder(species, mean_count)),
                width = 0.3,
                position = position_dodge(0.9)) +
  theme_bw() +
  scale_fill_colorblind() +
  labs(fill = "Species", y = "Mean abundance\n (+/- SE)") +
  facet_wrap(~Shrub) +
  theme(legend.position = "bottom")
```

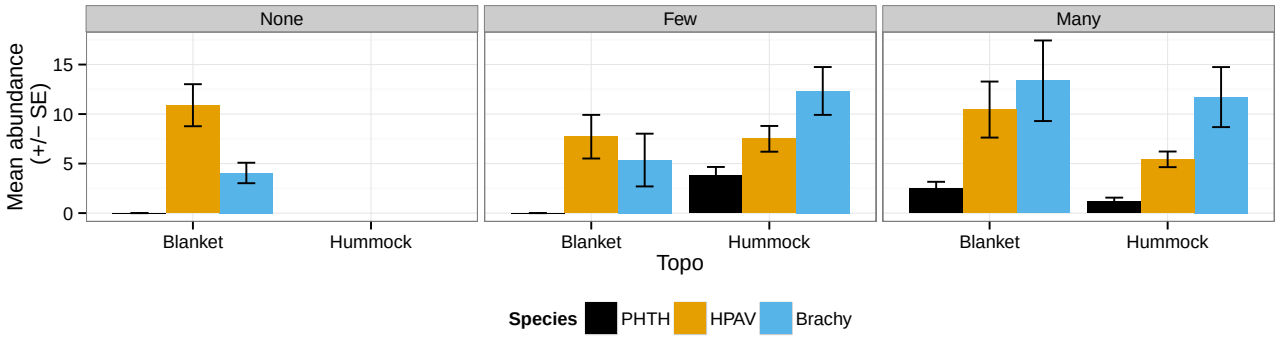

Figure 13: Bar chart with SE, 2

Saving plots, polishing for publication

Colour is nice for presentations. But sometimes you might want something that works for greyscale and resembles baseplot. Here we run a mock glm (`mite_glm`), create a prediction frame using `expand.grid()` and then predict. Have a look at the "Predicting from models" handout that accompanies the "Implementing models in R" seminar earlier in the series[4].

[4] https://github.com/orb16/seminaRs.git

```
names(mite_all) # easy to forget or misspell variable names unless we check

mite_sp <- gather(data = mite_all,  key = "species", value = "count", Brachy:Trimalc2) %>%
  filter()

mite_glm <- glm(count ~ WatrCont * species * Topo, family = poisson, data = mite_sp)

mite_data <- expand.grid(
  WatrCont = seq(from = round(min(mite_sp$WatrCont), digits = -1),
```

```
                to = round(max(mite_sp$WatrCont), digits = -1),
                by = 10),
  species = levels(mite_sp$species),
  Topo = levels(mite_sp$Topo))

mite_data2 <- cbind(mite_data, predict(mite_glm, newdata = mite_data, type = "link",
    se = TRUE))
mite_data2 <- within(mite_data2, {
    Count <- exp(fit)
    LL <- exp(fit - (1.96 * se.fit))
    UL <- exp(fit + (1.96 * se.fit))
})
```

That was all the prediction required. Now we can make the plot:

```
(tidy_plot <- ggplot(subset(mite_data2, mite_data2$species == "Brachy"),
                     aes(x = WatrCont, y = Count)) +
   geom_smooth(stat = "identity",
               colour = "black",
               aes(ymin = LL, ymax = UL, linetype = Topo))+
   scale_linetype_manual(values = c(1, 4))+
   guides(linetype = guide_legend(override.aes=list(fill=NA)))+
   theme_bw() +
   theme(legend.position = c(1,1),
         legend.justification = c(1,1),
         legend.direction= "horizontal",
         legend.key = element_blank(),
         legend.background = element_rect(colour = "grey")) +
   labs(y = "Water content",
        x = expression(paste(italic("B. armetelius "),"abundance")),
        linetype = "Topography") +
   coord_cartesian(x = c(130, 600)))
```

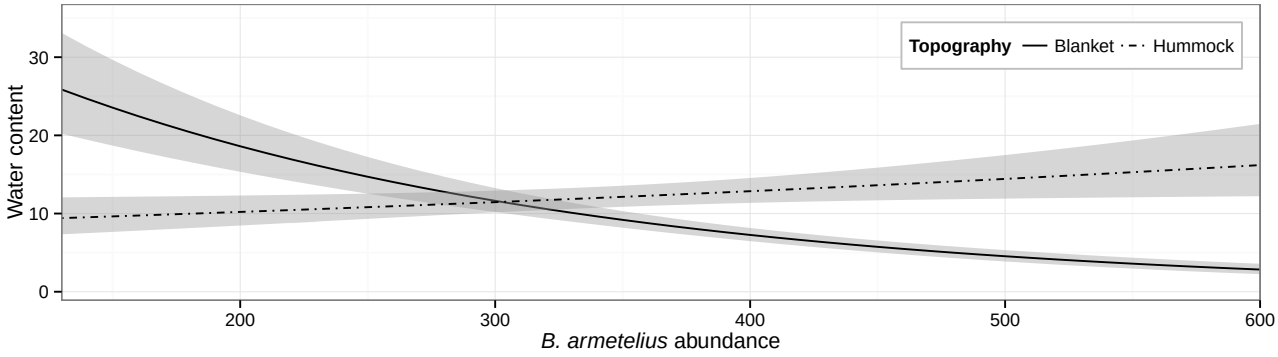

Figure 14: Tidy chart

Saving

You can either:

- use rstudio and export using the menu
- use R and screengrab from the graphics window
- use `pdf()` combined with `dev.off()` - this works for base graphics too
- use `ggsave()` for ggplot objects only.

Rstudio saving

I often do this for quick export. Under the plots pane, click export, then save plot as pdf (or whichever you want). An easy way to scale plots for presentations (if they are not too big) is to save them as landscape A5.

R and screengrabs

Not recommended. The problem with this and the above option is working out which plot goes with which set of code can be problematic (believe me). If you are determined, at least put a `#filename.png` note next to the plot code.

pdf() or png()

Useful because it's universal. Sometimes the R gremlins don't play nicely in operating systems, in which case you might use ggsave() - below.

```
pdf("exciting_mites.pdf", width = 8.3, height = 5.3)  # this sets up the filename, and size (in inches)
tidy_plot  #this tells R the plot we want to save
dev.off()  # this tells R to stop capturing the graphical output (plots)
# we want to save. dev.off() is also useful
# when your graphics window is playing funny
# business
```

The same format applies for .png, except you use png. See ?png for details. Or ?jpeg if you are that way inclined.

ggsave()

This only works for ggplots, but it sometimes it seems to work better. You specify which plot you want to save, the size, and the file name all in one. You can use a bunch of different formats, some of which allow you to specify a different resolution.

```
ggsave(plot = tidy_plot, file = "exciting_mites.pdf",
    width = 210, height = 148, units = "mm", dpi = 600)
```

You can use `arrangeGrob` from the gridExtra package in place of `grid.arrange` when you want to arrange ggplots and still use `ggsave`. For an example of this see the "extended A" ggplot handout.

Conclusions

ggplot produces highly customisable plots. The help pages are your friend, as is google. There is a ggplot mailing list that is worth browsing through or even subscribing too. And `http://www.cookbook-r.com` is gold both for manipulating data, and plotting it.

AND IF ALL ELSE FAILS...

```
ggplot(mite_sp[mite_sp$species %in% c(mite_sp_short$species), ],
       aes(x = Shrub, fill = species, y = count)) +
  geom_boxplot() +
  theme_excel() +
  scale_fill_excel()
```

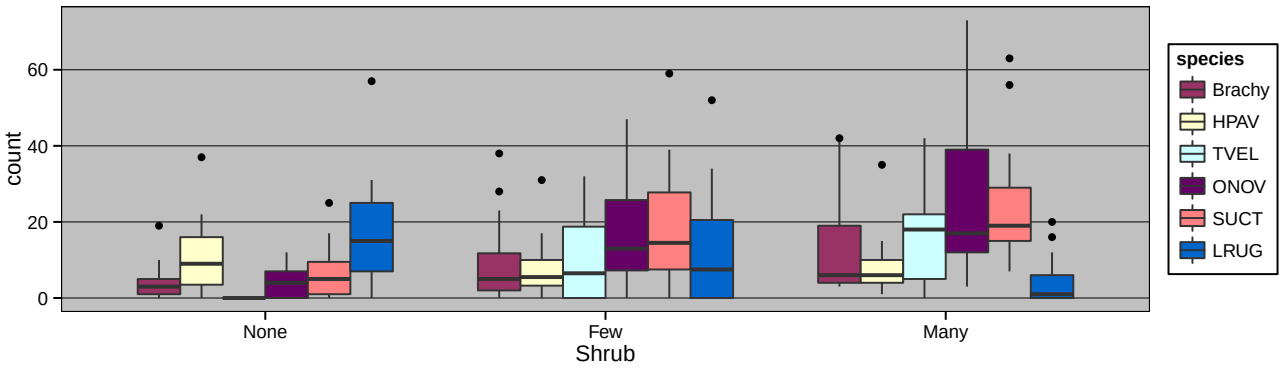

Figure 15: please don't use