Lesson 6: Stronger visualizations with ggplot2 and lattice

Randall Julian, Adam Zabell

12/28/2017

Plotting Data With Default Graphics

Default R comes with several basic plotting commands – `plot` to draw an X,Y graph, `points` to add X,Y points to the current graph, `barplot` to draw vertical or horizontal bars, `boxplot` to draw box-and-whisker plots, `hist` to build and draw a histogram, and many other plot types or plot-specific additions to plots.

The first major drawback to using these plots is that each requires learning a slightly different syntax to decorate the graph. For example, here are three plots basd on the January sample data, showing the ion ratios for all compounds and samples which exhibit and quant and qual peak. The first is a simple series plot, changing the default plot color to blue.

```
jan.s <- read_csv(here::here("class_data", "2017-01-06.csv"))
jan.s$idx <- c(1:nrow(jan.s))
hasIonRatio <- jan.s$ionRatio > 0
plot(jan.s$ionRatio[which(hasIonRatio)],col='blue')
```

If you want a histogram instead of a sequential series, the function changes but based on how `plot` looked, the coloring results may not be what you expected.

```
hist(jan.s$ionRatio[which(hasIonRatio)],col='blue')
```

In order to plot the histogram with blue outline, to match the blue open circles of the first plot, you need to specify a different variable.

```
hist(jan.s$ionRatio[which(hasIonRatio)],border='blue',main='Histogram')
```

The second drawback is that these plots, while drawn quickly, require detailed sort and select mechanisms in order to display complex data on a single graph. Plotting a matrix of graphs (as shown below) is even more difficult and you may spend more time troubleshooting the graph than actually analyzing the data. Here is a simple example which colors the series data by compound.

```
compounds <- unique(jan.s$compoundName)
for (i in 1:length(compounds)) {
  if (i == 1) {
    plot(
      jan.s$ionRatio[hasIonRatio & jan.s$compoundName == compounds[i]],
      col = i,
      main = "color by compound"
    )
  } else {
    points(
      jan.s$ionRatio[hasIonRatio & jan.s$compoundName == compounds[i]],
      col = i
    )
  }
}
```

Plotting Data with *lattice*

Working with large datasets, especially data you want to slice by one or more variables, may require moving to another graphing package. Using **lattice** provides a simplified faceting functionality, with syntax more typical of the default graphics package. Instead of plotting every compound on top of each other, here we can show each compound in it's own graph and color by sample type.

```
xyplot(ionRatio ~ idx | compoundName,
       data=jan.s[hasIonRatio,],
       groups=sampleType,
       auto.key=TRUE)
```

Instead of showing the data as a set of individual data points, we can instead show it as a smoothed trace.

```
xyplot(ionRatio ~ idx | compoundName,
       data=jan.s[hasIonRatio,],
       groups=sampleType,
       auto.key=TRUE,
       type=c("l","spline"))
```

The default parameters don't look very smooth when using the default terms, so additional refinement of the spline variables would be required to see something beyond this jagged green line.

Finally, the command for drawing a histogram of the results, separated by both compound name and sample type.

```
histogram( ~ ionRatio | compoundName + sampleType,
           data=jan.s[hasIonRatio,])
```

Benchmarks for running time within R

Sometimes there is a need to optimize code for timing as well as readability, and that usually occurs when plotting is involved. Assuming you can't reduce the complexity with a filter (e.g. plotting only one compound, or across a smaller time scale) the choice to display 100k-10m datapoints will simply take the time it takes. Knowing how long to expect a process to complete is good for future users of your code, and requires benchmarking. Here we run into an awkward difficulty within R and Rstudio, where 'execute the command' and 'render the figure' are two different tasks. We'll discuss functions in Lesson 9, but for now just consider it a 'modular unit' of commands we can call which **R** will treat as a single command. When we call *system.time* on this command, it will report the userTime (the **R** session) and systemTime (the OS kernel). Notice how we don't get the returned result of our function unless we wrap it in a *print* statement.

```
system.time(print("Anything seen as a single function can be tested in this way"))

anyUserFunction <- function(someText) {
  hasNumber <- grepl("[:digit:]", someText)
  if (hasNumber) {
    editedText <- "including their pros and cons"
    unusedVariable <- rnorm(1e6)
  } else {
    editedText <- "have you looked ahead yet"
    unusedVariable <- rnorm(5e6)
  }
  return(editedText)
}
system.time(anyUserFunction("we will talk about user functions in lesson 9"))
system.time(print(anyUserFunction("we will talk about user functions in lesson 9")))
```

There are purpose-built packages that can make this measurement easier (the libraries **tictoc** and **microbenchmark** are popular) but the above methodology is usually sufficient.

Plotting Data With *ggplot2*

To maintain the tidy focus of the tidyverse, the **ggplot2** package keeps the same syntax for all graphing schemes, has arguably prettier default graphs, and a frankly intuitive means for layering/faceting of the underlying data. The main drawback is that plotting from a large data.frame is still measured in minutes. The mock data in this course definitely qualifies as a large dataset, so we recommend that plotting be used judiciously if you're not applying a filter (see below).

Syntax follows the format of {'define the data' {+ 'describe the visualization'}} where each description is called a *geom* and multiple geoms can be stacked together. Definitions for the aesthetic mappings (e.g. plotTerms, color, iconShape, lineType) can be supplied when defining the data and are applied to the subsequent stack of geoms. Any mappings can be overridden within an individual geom.

Our first two examples show the ggplot version of the per component plots previously done with lattice. Notice that defining the data can be done as a variable (here it is `g`) and that definition can be used later for any number of geoms.

```
g <- jan.s %>%
  filter(ionRatio > 0) %>%
  ggplot(aes(x = idx, y = ionRatio, colour = sampleType))
g + geom_point() + facet_wrap(~compoundName) + scale_x_continuous(labels = NULL)
g + geom_smooth() + facet_wrap(~compoundName)
```

For the histogram, we override the aesthetic because this plot only uses 'one dimension' of the source data.

```
g +
  geom_histogram(mapping=aes(x=ionRatio,colour=sampleType),inherit.aes=FALSE) +
  facet_wrap(~compoundName)
```

We could easily spend the whole class session on this package, but the above plots showcase the basic syntax. The cheatsheet downloadable from the link at the end of this lesson provides additional examples of what can be done.

> Exercise 1: Draw a better histogram
> The default histogram paramaters for ggplot will stack the sample types in the same bin, making it difficult to determine if the trend for qc and standard samples is the same as the unknowns. The first plot in this exercise makes adjacent bars, but what does the second plot do?

```
g <- jan.s %>%
  filter(ionRatio > 0) %>%
  ggplot(aes(x = ionRatio, colour = sampleType, fill = sampleType))
# g + geom_histogram(position='dodge', bins= ) + facet_wrap(~compoundName)
# g + geom_histogram(aes(y=..density..), bins= ) + facet_grid(sampleType~compoundName)
```

> Exercise 2: Plot timing
> There is a longstanding community opinion that ggplot "takes longer" than the other two plotting mechanisms, but how much longer is it really? Is the time savings from **lattice** worth learning that syntax?

```
oneYearSamples <- list.files(here::here("class_data"), pattern = "csv$") %>%
  file.path(here::here("class_data"), .) %>%
  map_dfr(read_csv)
oneYearSamples$idx <- 1:nrow(oneYearSamples)
coreR <- function(oneYearSamples) {
  sampleTypes <- unique(oneYearSamples$sampleType)
  for (i in 4:1) {
    oneType <- which(oneYearSamples$sampleType == sampleTypes[i])
    if (i == 4) {
      plot(oneYearSamples$idx[oneType], oneYearSamples$concentration[oneType], col = i)
    } else {
      points(oneYearSamples$idx[oneType], oneYearSamples$concentration[oneType], col = i)
    }
  }
}
g <- ggplot(oneYearSamples, aes(x = idx, y = concentration, color = sampleType)) + geom_point()
l <- xyplot(
  concentration ~ idx,
  data = oneYearSamples,
  groups = sampleType,
  auto.key = TRUE
)
# system.time(coreR(oneYearSamples))
# dev.off()
# system.time(print(g))
# dev.off()
# system.time(print(l))
```

Summary

- ggplot2 cheatsheat
- lattice overview
- download a PDF comparison of both packages
- download an older PDF showing the system.time comparison