Introduction to the R Language

Plotting

Biostatistics 140.776

Plotting

The plotting and graphics capabilities in R are encapsulated in a few base and recommend packages:

- **graphics**: contains plotting functions for the "base" graphing systems, including `plot`, `hist`, and many others.
- **lattice**: contains code for producing Trellis graphics, which are independent of the "base" graphics system; includes functions like `xyplot`, `bwplot`, `levelplot`
- **grid**: implements a different graphing system independent of the "base" system; the **lattice** package builds on top of **grid**; we seldom call functions from the **grid** package directly
- **grDevices**: contains all the code implementing the various graphics devices, including X11, PDF, PostScript, PNG, etc.

The Process of Making a Plot

When making a plot one must first make a few choices (not necessarily in this order):

- To what device will the plot be sent? The default in Unix is `x11`; on Windows it is `windows`; on Mac OS X it is `quartz`
- Is the plot for viewing temporarily on the screen, or will it eventually end up in a paper? Are you using it in a presentation? Plots included in a paper/presentation need to use a file device rather than a screen device.
- Is there a large amount of data going into the plot? Or is it just a few points?
- Do you need to be able to resize the graphic?

The Process of Making a Plot

- What graphics system will you use: base or grid/lattice? These generally cannot be mixed.
- Base graphics are usually constructed piecemeal, with each aspect of the plot handled separately through a series of function calls; this is often conceptually simpler and allows plotting to mirror the thought process
- Lattice/grid graphics are usually created in a single function call, so all of the graphics parameters have to specified at once; specifying everything at once allows R to automatically calculate the necessary spacings and font sizes.

Base Graphics

Base graphics are used most commonly and are a very powerful system for creating 2-D graphics.

- Calling `plot(x, y)` or `hist(x)` will launch a graphics device (if one is not already open) and draw the plot on the device
- If the arguments to `plot` are not of some special class, then the *default method* for `plot` is called; this function has *many* arguments, letting you set the title, x axis lable, y axis label, etc.
- The base graphics system has *many* parameters that can set and tweaked; these parameters are documented in `?par`; it wouldn't hurt to memorize this help page!

Some Important Base Graphics Parameters

The `par` function is used to specify global graphics parameters that affect all plots in an R session. These parameters can often be overridden as arguments to specific plotting functions.

- pch: the plotting symbol (default is open circle)
- lty: the line type (default is solid line), can be dashed, dotted, etc.
- lwd: the line width, specified as an integer multiple
- col: the plotting color, specified as a number, string, or hex code; the `colors` function gives you a vector of colors by name
- las: the orientation of the axis labels on the plot

Some Important Base Graphics Parameters

- bg: the background color
- mar: the margin size
- oma: the outer margin size (default is 0 for all sides)
- mfrow: number of plots per row, column (plots are filled row-wise)
- mfcol: number of plots per row, column (plots are filled column-wise)

Some Important Base Graphics Parameters

Some default values.

```
> par("lty")
[1] "solid"
> par("lwd")
[1] 1
> par("col")
[1] "black"
> par("pch")
[1] 1
```

Some Important Base Graphics Parameters

Some default values.

```
> par("bg")
[1] "transparent"
> par("mar")
[1] 5.1 4.1 4.1 2.1
> par("oma")
[1] 0 0 0 0
> par("mfrow")
[1] 1 1
> par("mfcol")
[1] 1 1
```

Some Important Base Plotting Functions

- `plot`: make a scatterplot, or other type of plot depending on the class of the object being plotted
- `lines`: add lines to a plot, given a vector x values and a corresponding vector of y values (or a 2-column matrix); this function just connects the dots
- `points`: add points to a plot
- `text`: add text labels to a plot using specified x, y coordinates
- `title`: add annotations to x, y axis labels, title, subtitle, outer margin
- `mtext`: add arbitrary text to the margins (inner or outer) of the plot
- `axis`: adding axis ticks/labels

Useful Graphics Devices

The list of devices is found in `?Devices`; there are also devices created by users on CRAN

- `pdf`: useful for line-type graphics, vector format, resizes well, usually portable
- `postscript`: older format, also vector format and resizes well, usually portable, can be used to create encapsulated postscript files, Windows systems often don't have a postscript viewer
- `xfig`: good of you use Unix and want to edit a plot by hand

Useful Graphics Devices

- `png`: bitmapped format, good for line drawings or images with solid colors, uses lossless compression (like the old GIF format), most web browsers can read this format natively, good for plotting many many many points, does not resize well
- `jpeg`: good for photographs or natural scenes, uses lossy compression, good for plotting many many many points, does not resize well, can be read by almost any computer and any web browser, not great for line drawings
- `bitmap`: needed to create bitmap files (png, jpeg) in certain situations (uses Ghostscript), also can be used to create a variety of other bitmapped formats not mentioned
- `bmp`: a native Windows bitmapped format

Copying Plots

There are two basic approaches to plotting.

1. Launch a graphics device
2. Make a plot; annotate if needed
3. Close graphics device

Or

1. Make a plot on a screen device (default); annotate if needed
2. Copy the plot to another device if necessary (not an exact process)

Copying Plots

Copying a plot to another device can be useful because some plots require a lot of code and it can be a pain to type all that in again for a different device.

- `dev.copy`: copy a plot from one device to another
- `dev.copy2eps`: copy a plot to an Encapsulated PostScript File
- `dev.list`: show the list of open graphics devices
- `dev.next`: switch control to the next graphics device on the device list
- `dev.set`: set control to a specific graphics device
- `dev.off`: close the current graphics device

Lattice Functions

- `xyplot`: this is the main function for creating scatterplots
- `bwplot`: box-and-whiskers plots ("boxplots")
- `histogram`: histograms
- `stripplot`: like a boxplot but with actual points
- `dotplot`: plot dots on "violin strings"
- `splom`: scatterplot matrix; like `pairs` in base graphics system
- `levelplot`, `contourplot`: for plotting "image" data

Lattice Functions

Lattice functions generally take a formula for their first argument, usually of the form

```
y ~ x | f * g
```

- On the left of the ~ is the y variable, on the right is the x variable
- After the | are *conditioning variables* — they are optional; the * indicates an interaction
- The second argument is the data frame or list from which the variables in the formula should be obtained.
- If no data frame or list is passed, then the parent frame is used.
- If no other arguments are passed, there are defaults that can be used.

Lattice Behavior

Lattice functions behave differently from base graphics functions in one critical way.

- Base graphics functions plot data directly the graphics device
- Lattice graphics functions return an object of class `trellis`.
- The print methods for lattice functions actually do the work of plotting the data on the graphics device.
- Lattice functions return "plot objects" that can, in principle, be stored (but it's usually better to just save the code + data).
- On the command line, `trellis` objects are *auto-printed* so that it appears the function is plotting the data

Calling Lattice Functions

```
p <- xyplot(y ~ x | f, subscripts = TRUE,
            ylim = lattice:::extend.limits(range(lo, hi)),
            panel = function(x, y, subscripts, ...) {
                panel.xyplot(x, y, ...)
                lsegments(x, lo[subscripts],
                          x, hi[subscripts])
                panel.abline(h = 0, lty = 2)
            },
            xlab = NULL,
            scales = list(x = list(at = 1:nmodels, labels =
                                   rot = 90, alternating =
                          y = list(alternating = 3)),
            ylab = list(label = expression("% increase in a
            )
print(p)
```

Lattice Panel Functions

Lattice functions have a `panel` function which controls what happens inside each panel of the entire plot.

```
x <- rnorm(100)
y <- x + rnorm(100, sd = 0.5)
f <- gl(2, 50, labels = c("Group 1", "Group 2"))
xyplot(y ~ x | f)
```

plots y vs. x conditioned on f.

Lattice Panel Functions

```
xyplot(y ~ x | f,
       panel = function(x, y, ...) {
               panel.xyplot(x, y, ...)
               panel.abline(h = median(y),
                            lty = 2)
       })
```

plots y vs. x conditioned on f with horizontal (dashed) line drawn at the median of y for each panel.

Lattice Panel Functions

Adding a regression line

```
xyplot(y ~ x | f,
       panel = function(x, y, ...) {
               panel.xyplot(x, y, ...)
               panel.lmline(x, y, col = 2)
       })
```

fits and plots a simple linear regression line to each panel of the plot.

Using Subscripts

Sometimes you need to access objects outside the panel envrionment.

```
y <- c(rnorm(10), rnorm(10, 2))
x <- rep(1:10, 2)
std <- rep(1, 20)
rng <- range(y - 1.96 * std, y + 1.96 * std)
f <- gl(2, 10)

xyplot(y ~ x | f, subscripts = TRUE, ylim = rng,
       panel = function(x, y, subscripts, ...) {
               panel.xyplot(x, y, ...)
               lsegments(x, y - 1.96 * std[subscripts],
                         x, y + 1.96 * std[subscripts])
               panel.abline(h = 0, lty = 2)
       })
```

Mathematical Annotation

R can produce LaTeX-like symbols on a plot for mathematical annotation. This is very handy and is useful for making fun of people who use other statistical packages.

- Math symbols are "expressions" in R and need to be wrapped in the `expression` function
- There is a set list of allowed symbols and this is documented in ?plotmath
- Plotting functions that take arguments for text generally allow expressions for math symbols

Mathematical Annotation

Some examples.

```
plot(0, 0, main = expression(theta == 0),
     ylab = expression(hat(gamma) == 0),
     xlab = expression(sum(x[i] * y[i], i==1, n)))
```

Pasting strings together.

```
x <- rnorm(100)
hist(x,
     xlab=expression("The mean (" * bar(x) * ") is " *
                       sum(x[i]/n,i==1,n)))
```

Substituting

What if you want to use a computed value in the annotation?

```
x <- rnorm(100)
y <- x + rnorm(100, sd = 0.5)
plot(x, y,
     xlab=substitute(bar(x) == k, list(k=mean(x))),
     ylab=substitute(bar(y) == k, list(k=mean(y)))
     )
```

Or in a loop of plots

```
par(mfrow = c(2, 2))
for(i in 1:4) {
        x <- rnorm(100)
        hist(x, main=substitute(theta==num,list(num=i)))
}
```

Summar of Important Help Pages

- ?par
- ?plot
- ?xyplot
- ?plotmath
- ?axis