R Tutorial #2. April 11, 2011

[1] Using par in plots

When plotting, a whole lot of stuff can be customized by setting "graphical parameters". These include such aspects as the number of plots in the layout, the size of the margins, the style of points and lines, the size of text, the style of axes, etc. Some of these parameters must be set before you plot anything (e.g. the number of plots in the window) and some can be set as you plot (e.g. the plotting character).

All of them can be examined by reading the documentation for the function par().

Here is a quick example of both. Let dat1 and dat2 be two data sets I wish to plot.

```
dat1 <- sort(rnorm(50,0,1))

dat2 <- sort(runif(50,-1,1))

# A call to 'par' is used to change some high-level parameters. In this case, we alter the
# layout so that there will be one row and two columns of plots. The call to 'par' also
# returns a data structure containing all the parameter settings prior to your call. This
# is useful so that we can reset things to how they were once we are done.

oldpar <- par(mfrow=c(1,2))

# now do some plots

plot(seq(1:length(dat1)), dat1, pch=16, cex=2,
    xlab="index",
    ylab="score",
    cex.axis=1.1)


points(seq(1:length(dat2)), dat2, pch=17, cex=2, col="red")

hist(c(dat1, dat2), col=c("blue","green"),
    lty=c(1,2),
    xlab="value",
    main="A histogram")


# and reset the graphical parameters

par(oldpar)
```

Investigate the documentation for 'par' for more information. Refer to it often when customizing your graphics.

[2] Writing your own functions

If you have a rich data set, you will sooner or later want to apply the same series of commands to different subsets repeatedly. Here is an example of a function that takes a data set and returns the z-scores of the data elements. The return value of the function is simply whatever the return value of the last line of code is.

```
zscores <- function(dat) {
        m <- mean(dat)
        sd <- sd(dat)
        (dat - m)/sd
}
```

Once you have read in this function, you can then use it on data:

```
foo <- zscores(dat1)

summary(foo)
```

You can organize functions you use often in separate files. You might have a function that does a test, or that sets up a plot, or that summarizes some data. . . . Some facility at programming will be useful here, as R is a full programming language, with support for objects, loops, etc.

[3] Doing a simple t-test, and similar

Generate two data sets thus:

```
dat1 <- rnorm(50, 5, 1)
dat2 <- rnorm(50, 4.5, 1)
```

These come from different distributions. Usually, of course, you have data and wish to establish whether they plausibly come from the same distribution. Lets look at these.

Create the following function, and source the file with the definition:

```
eda <- function(m1, m2, sd1=1, sd2=2) {

  dat1 <- rnorm(50, m1, sd1)
  dat2 <- rnorm(50, m2, sd2)
  print(summary(dat1))
  print(summary(dat2))
  boxplot(dat1, dat2, notch=T)
  t.test(dat1, dat2)
}
```

Try it out for different means and standard deviations. How close can you make the means and still tell the two sets come from different distributions? Examine the output of the t-test.

In the function definition, default values are given for the two standard deviations. This means that if you do not provide values, they will have these values, but you can over ride them. Examples

```
eda(3, 4)    # uses the defaults
eda(3, 4, 0.5, 0.7)  # does not use the defaults
```

(Notice the use of a single '=' here. This is not 'equality', which is represented as '==')

Revision: remind yourself of the difference between matched sample t-test and independent sample t-test. You can do a matched sample test thus:

```
t.test(dat1, dat2, paired=T)
```

[4] Correlation and regression

Last time, you downloaded these data: http://cspeech.ucd.ie/Statistics/data/smallcor.dat.

Plot X against Y. Is there a correlation? Does knowing a specific value of X allow you to predict something about Y?

```
foo <- read.table("smallcor.dat", header=T)
summary(foo)
plot(foo$X, foo$Y)
cor.test(foo$X, foo$Y)
```

Now we want to fit a straight line to the data. This is of course almost the simplest linear model one can imagine. The function 'lm' is a powerful way to create linear models in R, and it is capable of far more than we can cover here.

We build a linear model by telling R that foo$Y is to be predicted based on foo$X. R knows that this means to build a model of the form y = mx + c, which is a straight line. It always includes the term 'c', which is the Y-axis intercept. We do this thus:

```
mylm <- lm(foo$Y ~ foo$X)
```

You can now look at the model:

```
summary(mylm)
```

What is the slope and intercept of the line?

Add the line to your existing plot thus:

```
abline(mylm)
```

Use the graphical parameters xlim and ylim to extend your plot so that you can verify that the Y-intercept reported is that which this line exhibits. We can do this by adding two more lines to the plot:

```
> plot(foo$X, foo$Y, xlim=c(-2,22), ylim=c(-10,40))
> abline(mylm)
> lines(c(0,0),c(-10,10))
> lines(c(-5,5),c(-1.629,-1.629))
```