R for Statistics 571

Bret Larget

September 6, 2010

1 Preliminaries

Before the first discussion section, you ought to install R onto your computer. If you have laptop, install R on the laptop and bring it to discussion section.

1.1 Installing R

To install R, connect your computer to the web and go to the Comprehensive R Archive Network (CRAN)

http://cran.r-project.org/

or its US mirror site

http://cran.us.r-project.org/

Click on the link to Linux, Mac OS X, or Windows, depending on your computer. (If you are a power user who wants to compile your own version from source code, there is a link for you too and you do not need these instructions.)

1.1.1 Mac OS X

Click on the link to the latest version of the software (R-2.11.1.pkg as of this writing), download the 40 MB file, double-click on the resulting icon, and follow onscreen instructions.

1.1.2 Windows

If you have a 32-bit system, click on the link named `base` and then click on `Download R 2.11.1 for Windows` (or a more recent version). Double-click on the executable file `R-2.11.1-win32.exe` (or more recent version).

If you have a 64-bit system, click on the link `here` for a 64-bit Windows port, then on the link `base`, and then click on `Download R 2.11.1 for Windows` (or a more recent version). Double-click on the executable file `R-2.11.1-win64.exe` (or more recent version).

1.2 Installing Packages

The easiest way to install a package is from within R when your computer is connected to the web. We will use the package `lattice` in the class. From the command line, you can type

```
> install.packages("lattice")
```

which will open a dialog asking you to pick a server (there are several in the US, or travel somewhere exotic around the world). Pick the server, wait a few moments, and the package and any other packages which `lattice` depends on will also be insalled. If successful, there will be a message to the screen indicating what was installed and the prompt will return.

In Windows and Macs, you can also install packages through a menu. On a Mac, the menu is named *Packages & Data* from which you select *Package Installer*. When you click on the Get List button, a box opens ask you to pick a CRAN server (unless you have already been to one this session). Over 1000 package names will appear. You can use the search box to find the name of the package you are seeking, select it, and then click the Install Selected button. A similar process works in Windows.

This is an excellent example of how the command line is easier than the menu, but only if you already know the right command to type.

To prepare for discussion section next week, install R and the `lattice` package on your computer.

2 Basics

Prompts. When you start R, the first window that pops up is a console window with a prompt `>`. You type commands at the prompt, press return, and something happens. If you ever see a prompt `+`, this means that the previous command was incomplete and R is waiting for you to complete it. Most likely, your previous command included a left parenthesis '(' that was not matched by a right one ')'. Type something to complete the command and then continue.

Output. When R writes out an array of numbers to the screen, it labels each line with the position in the array of the first element of the array between square brackets (for example, `[1]`). This label is not part of the array.

```
> 100
```

```
[1] 100
```

```
> 1:100
```

```
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100
```

Changing your workspace. R keeps all of the variables you keep in memory as it runs. When you end an R session, you may save your workspace. This allows you to have variables you have previously defined available without the need to create them all again from scratch. My recommendation, however, is to *not save* the workspace, but rather keep a text file with the commands needed to read in data and do any necessary manipulations, as well as analyses. In this way, you have a record of what is there and can create the workspace from scratch. (More on this later.)

By default, R will use as its workspace the folder in which the executable program exists. You will most likely want to change the workspace to a new folder where you might keep data from the textbook and your homework. You change the workspace for R under Windows by using the `File` menu and selecting `Change Directory...` with your mouse. For Macs, the menu `Misc` has the selection `Change Working Directory...`. My advice is to have a folder where you keep work for this course and to change the worspace to this folder each time you start R. You may want a separate folder for each project or assignment.

Quitting R. To quit, you can type `q()` on a command line or you can quit through the `File` menu. R will prompt you if you want to save your workspace.

Calculating with numbers. You can use R like a calculator. The `*` symbol stands for multiplication and the `^` symbol stands for exponentiation. The colon operator `:` creates an array of numbers from the first to the second. R has a number of built-in functions such as `mean()`, `sum()`, `median()`, `sd()`, `sqrt()`, `log()`, and `exp()` that have obvious meaning. Note that `log()` computes the natural (base e) logarithm; use a second argument (`log(1000,10)` or `log(1024,2)`, for example) to compute the logarithm with a different base. Try these example.

```
> 2 + 2
> 12 * 3 - 10/2 + sqrt(16)
> 3^2
> 1:10
> sum(1:10)
> mean(1:10)
> sd(1:10)
> log(1000)
> log(1000, 10)
> log(1024, 2)
```

Calculating with arrays. R can do arithmetic operations on arrays. If you multiply an array of numbers by a single number, the multiplication happens separately for each number. You can also add or multiply equal-sized arrays of numbers.

```
> 2 * (1:15)
> (1:10) + (10:1)
> (1:4)^2
```

Assigning variables. You can use the `=` sign to create new variables. Typing the name of a variable displays it.

```
> a = 1:10
> mean(a)
```

An alternative (and the original) to the `=` syntax is to use the key combination `<-` which was created to look like an arrow. Much documentation may use this instead of the equal sign, but both are valid methods. One point of view is that you are only blessed with a certain number of key-strokes in life, and you do not want to waste them. On the other hand, writing comments for code and using long and meaningful variable names is worthwhile.

3 Entering Data

3.1 Entering Data Directly

Single variables can be entered directly into R using `c()` for concatenation.

```
> milk = c(3.46, 3.55, 3.21, 3.78)
> treatment = factor(c("Control", "Low", "Medium", "High"))
```

This is useful for very small data sets, but it generally more useful to use either a word processor or Excel to enter larger data sets in a format that can later be read into R.

3.2 Entering Data from a Text File

The file `cows.txt` contains the cow data we encountered in class. This file is in a plain text file (not a Word or rich-text formatted file) which can be created in Windows using Notepad or on a Mac using Text Edit (or with another program). The first row contains variable names, separated by *white space*, which are spaces or tabs. Subsequent rows contain the data. Each row must contain the same number of *fields*, but it is not necessary to line up all of the data into neat columns.

The function that reads data into R from a text file in this format is `read.table()`. For historical reasons, the default is to not include a header line, so we add the argument `header=T` (T for true) to let R know that the first line of the file contains a header row with variable names.

```
> cows = read.table("cows.txt", header = T)
> str(cows)
```

```
'data.frame':       50 obs. of  11 variables:
 $ treatment     : Factor w/ 4 levels "control","high",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ level         : num  0 0 0 0 0 0 0 0 0 0 ...
 $ lactation     : int  3 3 2 2 2 1 1 1 3 3 ...
 $ age           : int  49 47 36 33 31 22 34 21 65 61 ...
 $ initial.weight: int  1360 1498 1265 1190 1145 1035 1090 960 1495 1439 ...
 $ dry           : num  15.4 18.8 17.9 18.3 17.3 ...
 $ milk          : num  45.6 66.2 63 68.4 59.7 ...
 $ fat           : num  3.88 3.4 3.44 3.42 3.01 2.97 2.99 3.54 2.65 4.04 ...
 $ solids        : num  8.96 8.44 8.7 8.3 9.04 8.6 8.46 8.78 9.04 8.51 ...
 $ final.weight  : int  1442 1565 1315 1285 1182 1043 1030 1057 1520 1300 ...
 $ protein       : num  3.67 3.03 3.4 3.37 3.61 3.03 3.31 3.48 3.42 3.27 ...
```

The function `str()` shows the *structure* of the data we just read in. Notice that numerical variables and categorical variables (factors) are distinguished. If levels of a categorical variable had been stored as numbers, we would have needed to tell R to reclassify the variable as a factor.

R calls a rectangular array of data where rows are observations and columns are variables a *data frame*.

3.3 Entering Data from an Excel Worksheet

Many of you may be more comfortable using Excel then a plain text editor. To enter data into an excel spreadsheet for subsequent entry into R, use the first row as a header row with variable names and put the values of each variable in a column. After the data is entered, save the file as a *comma-separated-variable file* (CSV file, for short). Excel will ask if you really mean to do this and warn you of all of the things you will lose of you do so, but disregard the warning and save the data in this format nevertheless. The resulting file is a plain text file where each field is separated by a comma rather than white space. This file can be read into R using the function `read.csv()`. There is no need with this function to specify `header=T`.

```
> cows = read.csv("cows.csv")
> str(cows)
```

```
'data.frame':        50 obs. of  11 variables:
 $ treatment     : Factor w/ 4 levels "control","high",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ level         : num  0 0 0 0 0 0 0 0 0 0 ...
 $ lactation     : int  3 3 2 2 2 1 1 1 3 3 ...
 $ age           : int  49 47 36 33 31 22 34 21 65 61 ...
 $ initial.weight: int  1360 1498 1265 1190 1145 1035 1090 960 1495 1439 ...
 $ dry           : num  15.4 18.8 17.9 18.3 17.3 ...
 $ milk          : num  45.6 66.2 63 68.4 59.7 ...
 $ fat           : num  3.88 3.4 3.44 3.42 3.01 2.97 2.99 3.54 2.65 4.04 ...
 $ solids        : num  8.96 8.44 8.7 8.3 9.04 8.6 8.46 8.78 9.04 8.51 ...
 $ final.weight  : int  1442 1565 1315 1285 1182 1043 1030 1057 1520 1300 ...
 $ protein       : num  3.67 3.03 3.4 3.37 3.61 3.03 3.31 3.48 3.42 3.27 ...
```

4 Working with Data Frames

4.1 Access to Variables

The operator $ is used to specify variables within a data frame. For example, we can work with the variable *milk* by typing `cows$milk`.

```
> cows$milk
```

```
 [1] 45.552 66.221 63.032 68.421 59.671 44.045 55.153 46.957 63.948 65.994
[11] 57.603 63.254 57.053 69.699 71.337 68.276 74.573 66.672 72.237 58.168
[21] 48.063 60.412 45.128 53.759 52.799 76.604 64.536 71.771 59.323 62.484
[31] 70.178 48.013 60.140 56.506 40.245 45.791 59.373 54.281 71.558 56.226
[41] 49.543 55.351 64.509 74.430 68.030 46.888 53.164 53.096 50.471 66.619
```

```
> mean(cows$milk)
```

```
[1] 59.54314
```

Assuming that this variable is measured in kg/day and that the density of milk is 1.03 kg/liter, we could add a new variable *volume* to the cow data set equal to the number of liters of milk produced on average each day.

```
> cows$volume = cows$milk/1.03
> str(cows)
```

```
'data.frame':        50 obs. of  12 variables:
 $ treatment     : Factor w/ 4 levels "control","high",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ level         : num  0 0 0 0 0 0 0 0 0 0 ...
 $ lactation     : int  3 3 2 2 2 1 1 1 3 3 ...
 $ age           : int  49 47 36 33 31 22 34 21 65 61 ...
 $ initial.weight: int  1360 1498 1265 1190 1145 1035 1090 960 1495 1439 ...
 $ dry           : num  15.4 18.8 17.9 18.3 17.3 ...
 $ milk          : num  45.6 66.2 63 68.4 59.7 ...
 $ fat           : num  3.88 3.4 3.44 3.42 3.01 2.97 2.99 3.54 2.65 4.04 ...
 $ solids        : num  8.96 8.44 8.7 8.3 9.04 8.6 8.46 8.78 9.04 8.51 ...
 $ final.weight  : int  1442 1565 1315 1285 1182 1043 1030 1057 1520 1300 ...
 $ protein       : num  3.67 3.03 3.4 3.37 3.61 3.03 3.31 3.48 3.42 3.27 ...
 $ volume        : num  44.2 64.3 61.2 66.4 57.9 ...
```

4.2 Subsets

It is frequently useful to partition data into smaller groups, often on the basis of the levels of a categorical variable. For example, with the cows data, we may want to calculate the mean protein level for cows by treatment group. In R, we can get subsets of a data frame using the square brackets `[` and `]`. For example, to display the protein numbers for all cows in the control group, we can do the following.

```
> cows$protein[cows$treatment == "control"]
```

```
 [1] 3.67 3.03 3.40 3.37 3.61 3.03 3.31 3.48 3.42 3.27 3.31 3.32
```

Note that two equal signs without a space is a comparison operator. The array `cows$treatment=="control"` has length 50 (the length of the `cows$treatment` variable) and the values are `True` and `False`. Inside the square brackets, only those elements corresponding to `True` are retained.

```
> cows$treatment == "control"
```

```
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[49] FALSE FALSE
```

We could find the mean of each group in turn.

```
> mean(cows$protein[cows$treatment == "control"])
```

```
[1] 3.351667
```

```
> mean(cows$protein[cows$treatment == "low"])
```

```
[1] 3.378462
```

```
> mean(cows$protein[cows$treatment == "medium"])
```

```
[1] 3.242308
```

```
> mean(cows$protein[cows$treatment == "high"])
```

```
[1] 3.341667
```

There is a shortcut using the functions `split()` which partitions a variable into a list for each level of a factor and `sapply()` which applies a function to each element of a list.

```
> sapply(split(cows$protein, cows$treatment), mean)
```

```
 control     high      low   medium 
3.351667 3.341667 3.378462 3.242308 
```

Notice that the ordering of the levels of treatment is alphabetical. Here, it makes sense to order by level. The `reorder()` function in lattice can be used for this purpose.

```
> cows$treatment = reorder(cows$treatment, cows$level)
> sapply(split(cows$protein, cows$treatment), mean)
```

```
 control      low   medium     high 
3.351667 3.378462 3.242308 3.341667 
```

The square brackets can also be used to find subsets of a data frame. Here, the command has the form *data frame[row subset,column subset]*. For example, to show columns 1, 7, and 11 for the first five cows, we could do the following.

```
> cows[1:5, c(1, 7, 11)]
```

```
  treatment   milk protein
1   control 45.552    3.67
2   control 66.221    3.03
3   control 63.032    3.40
4   control 68.421    3.37
5   control 59.671    3.61
```

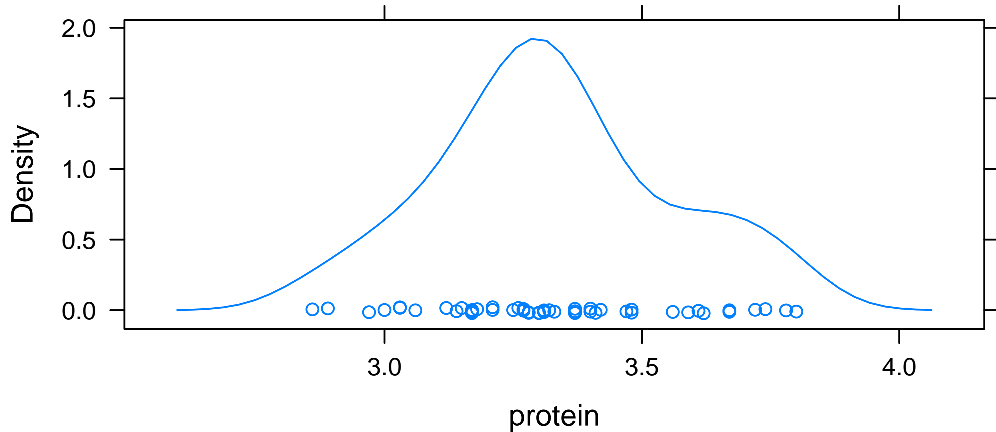
4.3 Simple Lattice Plots

The lattice package is loaded using this command.

```
> library("lattice")
```

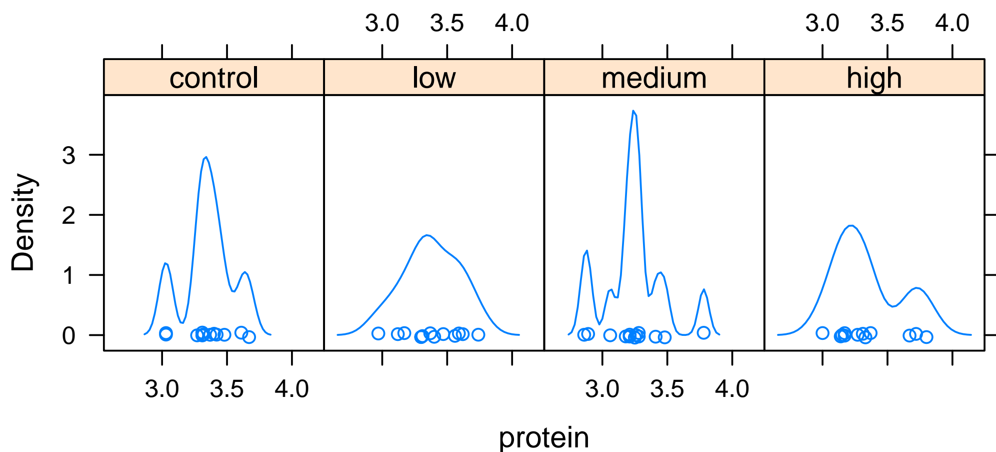
The graphics function in lattice are particularly useful for displaying data separately for each group of a categorical variable. Compare the commands for showing a density plot of all of the protein measurements

```
> plot(densityplot(~protein, cows))
```



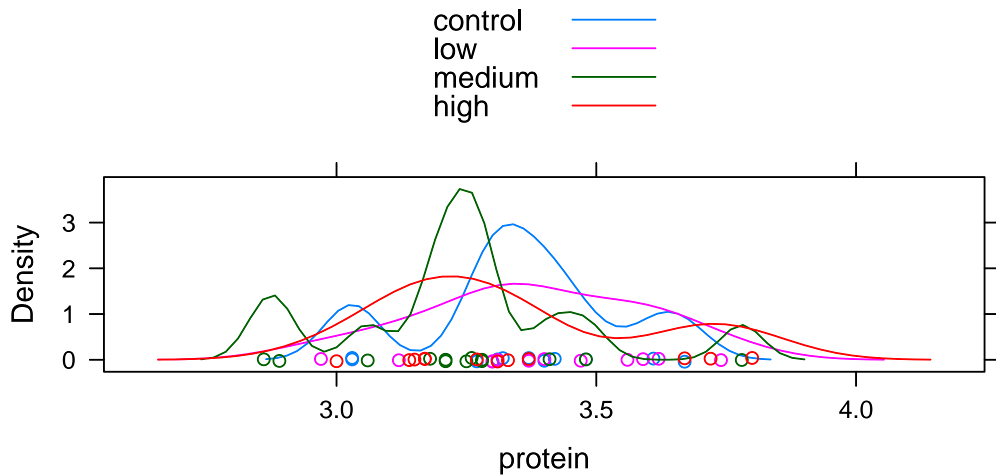
with density plot of protein for each treatment group in a different panel

```
> plot(densityplot(~protein | treatment, cows))
```



with a plot with the density plots overlayed.

```
> plot(densityplot(~protein, cows, groups = treatment, auto.key = T))
```



The raw data can be displayed with `dotplot()` with a layout with one column and four rows as follows.

```
> plot(dotplot(~protein | treatment, cows, layout = c(1, 4)))
```