R Computing, Ed 160, Fall 2008
Lab 1: A Quick Introduction to R
Wednesday, September 24, 2008

Purposes

The purposes of this lab are:
1. to familiarize you with the R computing environment, and
2. to introduce you to some handy R functions for descriptive and graphical statistics.

What is R?

R is a statistics package freely available at http://www.r-project.org. It is both a command line environment and a programming language, similar in some respects to MATLAB. Since R is a programming language, it is flexible. But flexibility comes with the price of a learning curve.

The basics, however, are straightforward to grasp and much can be accomplished with just a little practice. Practice is the key—one learns R by doing.

Downloading R

The newest version of R (R-2.7.2) is installed on the computers in both Little Tree and Big Tree in CERAS. You can download a copy of R onto your own machine by visiting http://cran.cnr.berkeley.edu/ and following the directions for the operating system that you use (Linux, MacOS X[1], Windows). Note: for Windows, you will want to download the "base" package, and for Mac you will want to download the installer, which can be found at the link entitled R-2.7.2.dmg (latest version). For Mac users, downloading should begin automatically. The installer should open automatically when it has been completely downloaded (if not, find the installer on your desktop). Follow the instructions for installation.

Starting R

In Windows or Mac, start R by clicking the R icon in the programs list. In Linux, start R by opening a terminal and typing `R -g tk` (this opens R using the Tcl/Tk GUI). At any command window, you can also type `R -help` to get a listing of command line options.
There are two basic ways to execute commands in R. The most straightforward is to open a terminal, start R, and just type in commands following the prompt (">"). For example:

```
> 3 + 5 # you type
[1] 8 # R responds
```

The "`[1]`" in the R response simply designates the first element in the response (in this case the response is a vector with only one element, 8).

[1] R supports both PowerPC and Intel based Macs. The corresponding binaries of R packages are now available for both architectures as well. Starting with R 2.3.1, CRAN binaries support Mac OS X 10.4 (Tiger) and higher only. It is, however, possible to compile binaries for earlier OS X versions from sources.

Getting Help

R has great documentation. If you are unsure about how to use a function in R, try

```
> help(function name)
```
or
```
> ? function name
```

Also look at the R manual and any other manuals listed under the "help" tab in the R GUI. Our course website also has links to many good R help references. What is a function? For our purposes it is a command that when called performs some action. Functions can take zero or more arguments as input and then optionally return a data item.

Elementary Operations

Give a try at using R as a fancy calculator:

```
> 8 - 6
[1] 2

> 3*5
[1] 15

> 10/2
[1] 5

> sqrt(100)
 [1] 10
```

How do you raise 2 to the 3rd power? Try it.

Assigning variables allows us to better manage our work.
Note that "#" is used to denote comments and R will ignore any commands after a "#" in a given line of code.

```
> a = 5  # now the value 5 is assigned to the variable a . An alternative form is: a <- 5
> b = 6
> a  # lets see what a holds
[1] 5
> b
[1] 6

> a+b
[1] 11

> d = c(a,b)  # combine values into a vector
> d
[1] 5 6
```

A vector is simply a structure for holding a one-dimensional ordered set of items, usually numeric or character items. We will often hold information about a variable, measured across many individuals (or observations) in vectors. For example, lets say that the ages of your nine best friends are 23, 23, 27, 22, 22, 22, 24, 26, and 22. To store this data in a vector, type this (the c stands for "combine"):

```
> age = c(23,23,27,22,22,22,24,26,22)
```

You've now created a vector, entitled "age". To view this vector, simply type the name of the vector:

```
> age
[1] 23 23 27 22 22 22 24 26 22
```

You would like to know how your age compares, so you decided to calculate their mean (average) age. To do this type:

```
> mean(age)
[1] 23.44444
```

To better understand the distribution of ages, you can also calculate the median and standard deviation of *age*, as well as calculate the number of items in *age* and other summary statistics.

```
> age
[1] 23 23 27 22 22 22 24 26 22

> median(age)
[1] 23

> sd(age)
[1] 1.878238

> length(age)
[1] 9

> summary(age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  22.00   22.00   23.00   23.44   24.00   27.00
```

You know that time does not stand still, and you wonder what their ages and mean age will be in four years.

```
> age + 4
[1] 27 27 31 26 26 26 28 30 26
> mean(age+4)
[1] 27.44444
```

You wonder whether you could have simply added 4 to the mean, instead of first adding 4 to each age and then calculating the mean (for more on this topic, see the definition of "linear transformations" on page 91 in AF):

```
> mean(age+4)
[1] 27.44444

> mean(age) + 4
[1] 27.44444
```

In this simple case, you can visually inspect (above) to see that these two ways of calculating the mean age in four years are equivalent. You also could have asked R to tell you if this statement was true, using a logical test statement:

```
> mean(age)+4 == mean(age+4)          ## Use <, <=, = =, >= , or > to test logical expressions
[1] TRUE
```

Now, after all that work, you realize that you mistyped the third entry in the vector. What to do? First, we want to identify the third element of the vector:

```
> age
[1] 23 23 27 22 22 22 24 26 22

> age[3]
[1] 27
```

Now, we want to make that element equal a new value, 30:

```
> age[3]=30

> age
[1] 23 23 30 22 22 22 24 26 22
```

We can see, above, that the third value in the vector entitled age has changed from 27 to 30.

The indexing function, served by the [bracket] syntax can be useful if you want to examine only certain observations. Below, we ask R to show us only the 1st, 2nd, 3rd, and 6th elements of the vector age:

```
> age[c(1,2,3,6)]                     ## A vector of indices can be passed in the brackets
[1] 23 23 30 22
```

Reading in Data from a File or the Web

Of course, in most real cases we're interested in a larger data set, as opposed to a short vector that we can type in manually. Let's consider a dataset often used in early childhood educational research:

The Early Childhood Longitudinal Study, Kindergarten Class of 1998-99 (ECLS-K) is an ongoing study that focuses on children's early school experiences beginning with kindergarten and following children through middle school. The ECLS-K provides descriptive information on children's status at entry to school, their transition into school, and their progression through 8th grade. The longitudinal nature of the ECLS-K data enables researchers to study how a wide range of family, school, community, and individual factors are associated with school performance. Students' cognitive skill is assessed in the fall and spring of every school year, beginning in kindergarten. The dataset used in this lab is a simple random sample of 100 students who were assessed in their second grade year.

Urban Variable. The "urban" variable is a dummy variable (meaning it only takes on values of 0 and 1). When "urban" = 1, the student attended a school located in an urban setting. When "urban"=0, the student attended a school located in a non-urban setting.

Math Score Variable. The mathematics assessment is designed to measure conceptual knowledge, procedural knowledge, and problem solving within specific content strands. Across the *full* set of grade level assessments, the content strands include number sense, properties, and operations; measurement; geometry and spatial sense; data analysis, statistics, and probability; and patterns, algebra, and functions. Of these,

number sense, properties, and operations strand is the largest of all the content strands at all grade level assessments. Herein, the math concepts included in the math direct cognitive assessment are relevant to kindergarten content.

In your web browser, open up the eclsk data file located at: http://www-stat.stanford.edu/~rag/ed160/eclskLab1 to see what an R data file looks like.

There are two options for reading your data into R. The easier option for accessing the data is to simply read the dataset directly from the web site using the `read.table(url)` function:

```
eclsk = read.table("http://www-stat.stanford.edu/~rag/ed160/eclskLab1")
```

Alternatively, you can save the file to your hard drive and load it from there into R, using the `read.table(filename)` function. This will allow you save any changes you make to the dataset. Here are the steps:

1. Save the file to your hard drive (on BigTree computers, save to "My Documents" folder)
2. Read the file into R, naming the dataset "eclsk":

The file path name will be different for each person, but here's what it looks like on my computer:

```
> eclsk = read.table(file="C:\\Documents and Settings\\Allison\\My
Documents\\ED160\\eclsk.dat", header= TRUE)
```

(Important: Note that you must add an additional "\" in order for R to recognize the location!)

Use of the `read.table()` function will create a data frame, which is a useful structure that can store numerous columns (i.e., vectors) of data. Each column of data in a data frame can be of a different mode. That is, one can be "numeric" while another one is "character".

You might want to begin to examine your new dataset be running a summary command on the dataset. This will return basic descriptives on all your variables:

```
> summary(eclsk)
    childid          mathsc          urban
 Min.   :101.0   Min.   :14.41   Min.   :0.00
 1st Qu.:125.8   1st Qu.:22.87   1st Qu.:0.00
 Median :150.5   Median :28.08   Median :0.00
 Mean   :150.5   Mean   :29.49   Mean   :0.19
 3rd Qu.:175.2   3rd Qu.:34.60   3rd Qu.:0.00
 Max.   :200.0   Max.   :54.23   Max.   :1.00
```

You'll have noticed, when we read in the data, that we use the option `header=TRUE` to tell R that the first line in the file gives the column names. If column names are not contained in the file, then specify `header=FALSE`. The first line in the summary gives the names of each column. To view column names, use:

```
> names(eclsk)
[1] "childid" "mathsc"  "urban"
```

To access individual columns of the data frame, use the $ symbol. First, type the data frame name (here, eclsk), "$", and then the column you wish to view:

```
> eclsk$mathsc
  [1] 30.364 24.722 22.072 20.726 45.395 54.082 30.235 23.367 30.955
27.355 47.718 23.273 24.584 28.247 35.572 39.445 23.458 22.273 31.349
34.001 22.233
 [22] 36.398 22.777 38.740 26.184 27.848 24.214 34.745 32.856 36.265
32.470 27.728 54.231 25.338 33.927 27.158 30.651 22.391 22.227 32.524
36.055 32.068
 [43] 52.524 29.581 23.950 19.646 38.739 15.237 30.132 22.902 19.216
34.170 25.618 33.360 26.528 32.336 27.913 22.329 32.675 27.082 19.156
25.430 30.795
 [64] 20.099 22.721 37.990 23.495 37.490 29.841 43.721 35.684 41.606
35.030 22.597 23.374 27.535 32.401 22.019 15.368 17.168 23.794 14.413
22.264 48.767
 [85] 22.497 20.977 30.479 29.057 30.655 23.562 34.580 34.650 35.968
17.426 23.710 43.934 18.803 38.705 22.087 37.213
```

I can again use element indexing brackets to view only the first four elements of the "mathsc" column. The ":" in between 1 and 4 tells R to return all elements from 1 *through* 4:

```
> eclsk$mathsc[1:4]
[1] 30.364 24.722 22.072 20.726
```

Typing too many $'s can get tiresome, however. An easy way to lighten the typing load is to use the `attach()` function. Thereafter, you can refer to columns by their column name. For example:

```
> attach (eclsk) #Note the space after the command
> mathsc[1:4]
[1] 30.364 24.722 22.072 20.726
```

Elementary Graphics & Visual Displays

In the class lecture, we talked about constructing stem-and-leaf plots, histograms, and boxplots. These are easily constructed as follows:

```
> stem(mathsc)

  The decimal point is 1 digit(s) to the right of the |

  1 | 4
  1 | 5577999
  2 | 00112222222222333333333444444
  2 | 5555667777888889
  3 | 000000111112222333344 4
  3 | 55556666667789999
  4 | 244
  4 | 589
  5 | 344
```

Above, we can see that the lowest value has been rounded to 14, and the highest value is 54. Perhaps we would rather have the decimal point be located at the | . To do so, we can alter the scale of the stem (one of the suboptions of the “stem” command”

```
> stem(mathsc, scale=2)

  The decimal point is at the |

  14 | 424
  16 | 24
  18 | 8226
  20 | 170
  22 | 01122333456789344556789
  24 | 267346
  26 | 251245789
  28 | 2168
  30 | 124577803
  32 | 134557949
  34 | 02667067
  36 | 013425
  38 | 07774
  40 | 6
  42 | 79
  44 | 4
  46 | 7
  48 | 8
  50 |
  52 | 5
  54 | 12
```

Now we have a little more detail (more decimals) in our stem and leaf plot above: the lowest value is 14.4 and the highest value is 54.2.

Now, to make a boxplot, we use the “boxplot” command. We’re actually going to make side-by-side boxplots for students from urban schools, versus non-urban schools. In order to make our graphic more readable, we are also going to add labels to the x and y axis, as well as a main title to the graphic:

```
> boxplot(mathsc ~ urban, xlab="School Type (1= Urban)", ylab="Math
Score", main="Boxplots: Math Scores in Urban vs. Non-Urban Schools")
```

Can you see what did the options xlab, ylab, and main do?

The boxplot above is created using the quantiles we saw previously when we examined the 5-number summary using the "summary" command. We can also use the quantile command to retrieve these numbers:

```
> quantile(mathsc)
      0%      25%      50%      75%     100%
14.41300 22.87075 28.08000 34.59750 54.23100
```

The tapply Function

Often in education research (and many fields that use statistics), we will want to compare different groups' values of variables to each other. The "tapply" function allows you to execute a given function repeatedly on different groups. For example, suppose we want to average the math score data separately for students from urban and suburban schools. We can use our "tapply" function. We will give R the relevant information in the following order tapply(X, group, function):

```
> tapply(mathsc, urban, mean)
       0        1
30.54869 24.98821
```

We could also use our "tapply" function to retrieve basic descriptives (mean, median, min, max, etc.) for both groups:

```
> tapply(mathsc, urban, summary)
$`0`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  17.17   23.56   30.13   30.55   35.57   54.23

$`1`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  14.41   20.19   23.46   24.99   30.04   48.77
```

What if we want to just look at the summary for urban students, not also separately for non-urban students? We can use the following syntax to tell R we only want to execute the command over a certain subset:

```
> summary(mathsc[urban==0])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  17.17   23.56   30.13   30.55   35.57   54.23

> summary(mathsc[urban==1])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  14.41   20.19   23.46   24.99   30.04   48.77
```

Note that these numbers agree with the numbers we found using our "tapply" command.

Exercises

(1) Create a histogram of the mathsc column by using the `hist()` function. Can you give the

histogram a title and nice labels? If you have syntax questions, you can always investigate by typing help(hist).

(2) Draw a boxplot of the mathsc column, not subdivided by group.

(3) Try downloading another dataset from the website on your own. Try renaming the variable columns. And replace the third value of every column with the mean of the column.

(4) View information about a data frame using `str(dataframe)`. View attributes of a data frame by using `attributes(dataframe)`.

Extended Material: Sequences and Quantiles

R has a shorthand for denoting *sequence* vectors like `c(1,2,3,4,5,6,7,8,9,10)`:

```
> z = 1:10
> z
[1] 1 2 3 4 5 6 7 8 9 10
```

Now, lets create a vector entitled "ww" that is a sequence from 10 to 20 that skips every other value:

```
> ww = seq(from=10, to=20, by=2) # "seq" stands for "sequence"
> ww
[1] 10 12 14 16 18 20
```

R knows that you will tell it the parameters of your desired sequence in a particular order. It will expect you to first provide the first number in the sequence, then the last, and thirdly the increment:

```
> myseq = seq(0, 1, .1)
> myseq
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

> seq2 = seq(from=0, to=1, by = .05)     ## The same as seq2 = seq(0,1,.05)
> seq2
[1] 0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50 ...
```

These sequence vectors are useful for manipulating the quantile function. By default and definition, the "quantile" command will provide you with the 0% 25% 50% 75% 100% values:

```
> quantile(mathsc,na.rm=TRUE)
      0%      25%      50%      75%     100%
14.41300 22.87075 28.08000 34.59750 54.23100
```

(Note that, we used `na.rm=TRUE` to remove NA values, if there are any.)
Suppose that, instead, you want to get *deciles*—the 10%, 20%, 30%, 40%, 50%...100% percentiles—of the ECLS math score data instead. In order to do so, you must pass in a sequence vector:

```
> quantile(mathsc, myseq, na.rm=TRUE)
0%     10%     20%     30%     40%     50%     60%     70%     80%     90%
100%
14.4130 20.6633 22.3786 23.4839 25.5428 28.0800 30.7110 33.0072 35.7408 38.8105
54.2310
```

Created by Rudy Angeles, 2005.
Edited by George Chang (2006), John Boik (2007), Allison Atteberry/Heather Hough (2008).