An Introduction to R

Christopher R. Bilder
University of Nebraska-Lincoln
Department of Statistics

September 16, 2011

www.chrisbilder.com/workshop

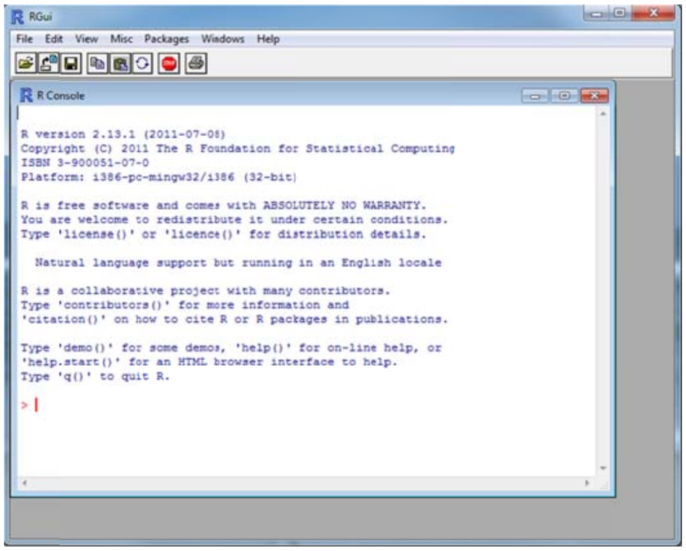
Table of Contents

I. Basics

The R installation file for Windows is available at http://cran.r-project.org/bin/windows/base/. Select the "Download R 2.*.* for Windows" link. You can simply execute the file to install it (all the installation defaults are o.k. to use).

R Console window

After starting R, you will obtain the following window:



```
R version 2.13.1 (2011-07-08)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: i386-pc-mingw32/i386 (32-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

The *R Console* window is where commands are typed, and it can be used much like a calculator:

```
> 2+2
[1] 4
> (2-3)/6
[1] -0.1666667
> 2^2
[1] 4
> sin(pi/2)
[1] 1
> cos(pi/2)
[1] 6.123032e-17
> log(1)
[1] 0
> qchisq(0.95,1)
[1] 3.841459
> pnorm(1.96)
[1] 0.9750021
```

Results from these calculations can be stored in an *object*. A `<-` (less than and minus symbols) is used to make the *assignment*, and it is read as the word "gets". For example,

```
> save<-2+2
> save
[1] 4
```

The = symbol can be used to make the assignment too, but `<-` is much more frequently used.

Objects are stored in R's database, which is kind of like the SAS WORK library. When you close R, you will be

asked to save or delete the objects. I usually delete them because they can be easily reproduced through my code. To see a listing of all objects, use one of the following:

```
> ls()
[1] "save"
> objects()
[1] "save"
```

To delete an object, use `rm(<object name>)`, where the appropriate object name is substituted for `<object name>`.

Functions

R performs calculations using *functions*. For example, the `qchisq()` and the `pnorm()` commands used earlier are functions. Writing your own function is fairly simple. For example, suppose you would like a function to calculate the standard deviation. Below is an example where 5 observations are saved to an *object* using the *concatenate* or *combine* function. A function called `sd2()` is written that finds the standard deviation by using the square root of the variance. The `sd2` object is now stored in the R database.

```
> x<-c(1,2,3,4,5)

> sd2<-function(numbers) {
   sqrt(var(numbers))
 }

> sd2(x)
[1] 1.581139

> save<-sd2(x)
> save
[1] 1.581139
```

Note that there already is a function called `sd()` in R to calculate the standard deviation.

When a function has multiple lines of code in it, the last line corresponds to the returned value. For example,

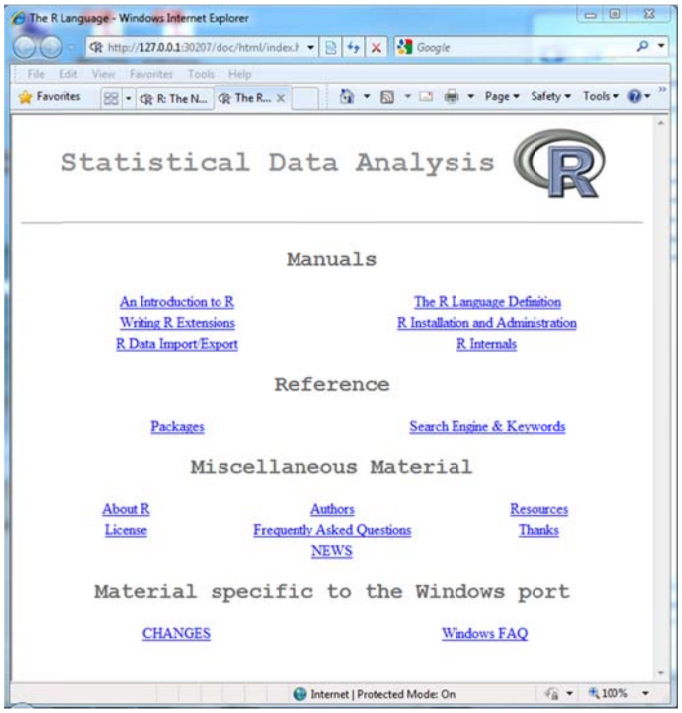
```
> x<-c(1,2,3,4,5)
```

```
> sd2<-function(numbers) {
   cat("Print the data: \n", numbers, "\n")
   sqrt(var(numbers))
 }

> save<-sd2(x)
Print the data
1 2 3 4 5

> save
[1] 1.581139
```

The `cat()` function within `sd2()` prints text and the `\n` is a special escape character that moves printed text to the next line.

Help

To view a list of R's functions, open the Help by selecting HELP > HTML HELP from the main R menu bar. The Help will open in your default web browser:


The R Language - Windows Internet Explorer
http://127.0.0.1:30207/doc/html/index.h
File Edit View Favorites Tools Help
Favorites
R: The N...
The R...
Page Safety Tools
Done
Internet | Protected Mode: On
100%

Statistical Data Analysis

Manuals

An Introduction to R
The R Language Definition
Writing R Extensions
R Installation and Administration
R Data Import/Export
R Internals

Reference

Packages
Search Engine & Keywords

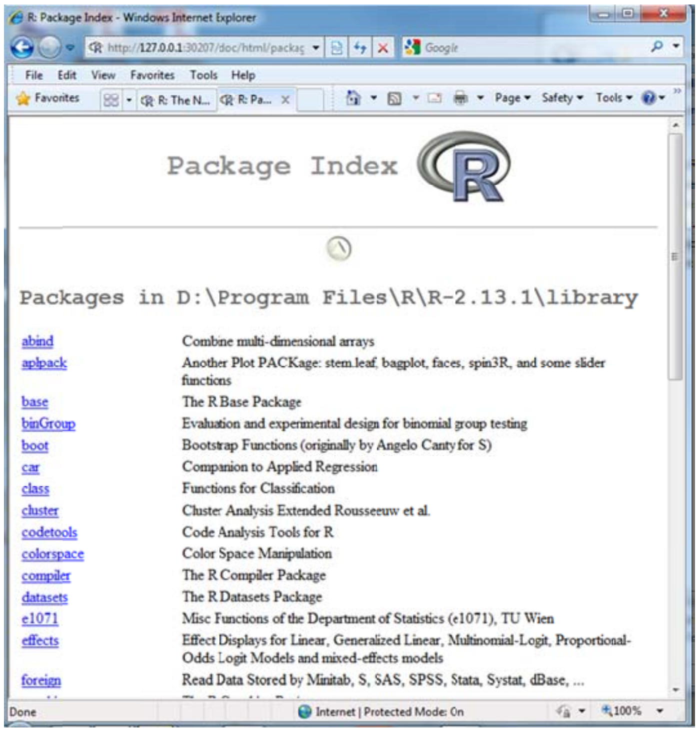
Miscellaneous Material

About R
Authors
Resources
License
Frequently Asked Questions
Thanks
NEWS

Material specific to the Windows port

CHANGES
Windows FAQ

Under REFERENCE, select the link PACKAGES to open the window below:


R: Package Index - Windows Internet Explorer
http://127.0.0.1:30207/doc/html/packag
Google
File Edit View Favorites Tools Help
Favorites
R: The N...
R: Pa...
Page Safety Tools
Done
Internet | Protected Mode: On
100%

Package Index

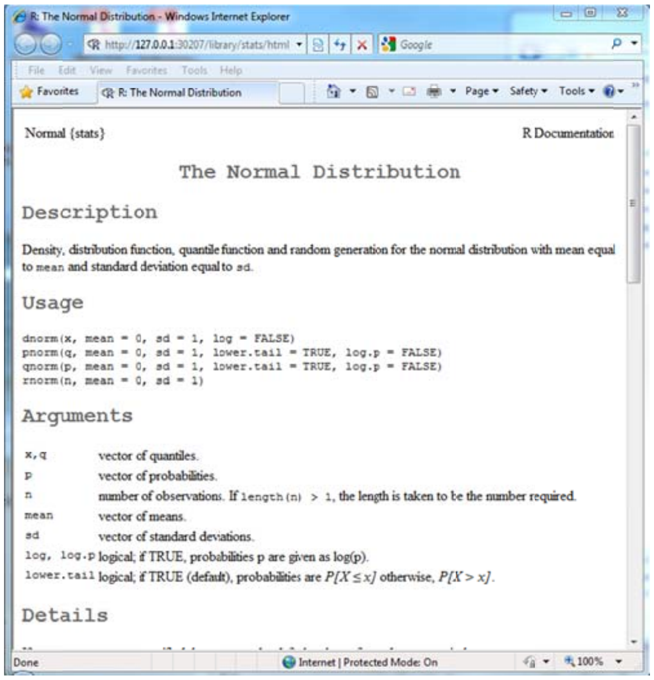
Packages in D:\Program Files\R\R-2.13.1\library

abind	Combine multi-dimensional arrays
aplpack	Another Plot PACKage: stem.leaf, bagplot, faces, spin3R, and some slider functions
base	The R Base Package
binGroup	Evaluation and experimental design for binomial group testing
boot	Bootstrap Functions (originally by Angelo Canty for S)
car	Companion to Applied Regression
class	Functions for Classification
cluster	Cluster Analysis Extended Rousseeuw et al.
codetools	Code Analysis Tools for R
colorspace	Color Space Manipulation
compiler	The R Compiler Package
datasets	The R Datasets Package
e1071	Misc Functions of the Department of Statistics (e1071), TU Wien
effects	Effect Displays for Linear, Generalized Linear, Multinomial-Logit, Proportional-Odds Logit Models and mixed-effects models
foreign	Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, ...

All built in R functions are stored in a *package*. Some packages are automatically included with your R

installation, and some need to be downloaded from R's website (more on this later).

We have been using functions from the *base* and *stats* packages. Through selecting stats and scrolling down to the link for `pnorm()`, we obtain the help web page for this function:


R: The Normal Distribution - Windows Internet Explorer
http://127.0.0.1:30207/library/stats/html
Google
File Edit View Favorites Tools Help
Favorites
R: The Normal Distribution
Page Safety Tools
Done
Internet | Protected Mode: On
100%

Normal {stats} R Documentation

The Normal Distribution

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

`x,q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`mean`	vector of means.
`sd`	vector of standard deviations.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$ otherwise, $P[X > x]$.

Details

The full syntax for `pnorm()` is

```
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p=FALSE)
```

and it evaluates the cumulative distribution function for the normal distribution (i.e., F(x) for a random variable X). The q *argument* corresponds to the 1.96 that was entered earlier. Thus,

```
> pnorm(1.96)
[1] 0.9750021
> pnorm(q = 1.96)
[1] 0.9750021
> pnorm(q = 1.96, mean = 0, sd = 1)
[1] 0.9750021
> pnorm(1.96, 0, 1)
[1] 0.9750021
```

produce the same results. The other arguments within the function have default values. For example, the standard normal distribution is the default, because `mean = 0` and `sd = 1` (standard deviation). If you use argument values without argument names (last example), you MUST have the correct order for the argument values. For this reason, I strongly recommend always using the argument names in all but the most basic functions.

You can see help for other functions involving the normal distribution. They are

- `dnorm()` – Finds the normal probability density function value (i.e., f(x) for a random variable X)
- `qnorm()` – Computes a quantile from a normal distribution (i.e., find q in F(q) = α for a known value of α)
- `rnorm()` – Simulates data from a normal distribution

There are many functions available for other probability distributions. All functions have the same leading letter: d, p, q, and r, that correspond to what they do. Help files for many other distributions are available on your computer at http://127.0.0.1:14149/library/stats/html/Distributions.html.

All help web pages have the same general format. The end of each web page gives code examples that you can copy and paste into your R Console window.

If you know the exact name of the function, simply type `help(<function name>)` at the R Console command prompt to open its help web page. For example,

```
> help(pnorm)
```

opens the same help as before for `pnorm()`.

Vectorized calculations

Many R functions work directly on *vectors*. We saw an example of a vector earlier when we created the object x with

```
> x<-c(1,2,3,4,5)
```

As an example of how R takes advantage of working with vectors, below is how to find more than one probability or quantile at a time from a probability distribution:

```
> pnorm(q = c(-1.96, 1.96))
[1] 0.02499790 0.97500210

> qt(p = c(0.025, 0.975), df = 9)
[1] -2.262157  2.262157
```

The `qt()` function computes the 0.025 and 0.975 quantiles from a t-distribution with 9 degrees of freedom.

For a little more complex example, suppose we want a 95% confidence interval for a population mean:

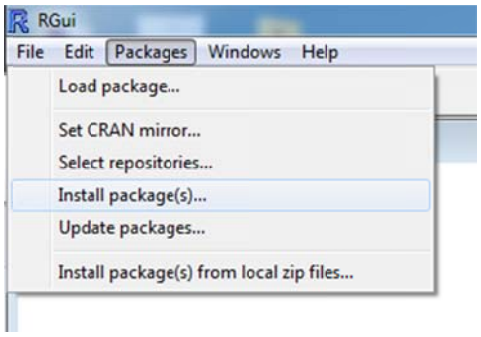
```
> x<-c(3.68, -3.63, 0.80, 3.03, -9.86, -8.66, -2.38, 8.94,
      0.52, 1.25)
> x
 [1]  3.68 -3.63  0.80  3.03 -9.86 -8.66 -2.38
      8.94  0.52  1.25

> var.xbar<-var(x)/length(x)
> mean(x) + qt(p = c(0.025, 0.975), df = length(x) - 1) *
    sqrt(var.xbar)
```

```
[1] -4.707033  3.445033

> t.test(x = x, mu = 2, conf.level = 0.95)

        One Sample t-test

data:  x
t = -1.4602, df = 9, p-value = 0.1782
alternative hypothesis: true mean is not equal to 2
95 percent confidence interval:
 -4.707033  3.445033
sample estimates:
mean of x
   -0.631
```

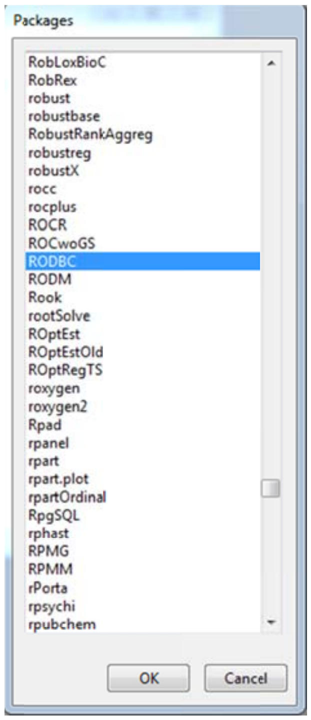
In this example, a random sample of size 10 is taken from a population and put into an object called `x`. The "`mean(x) + …` " line of code shows how the calculations are performed automatically even though the `qt()` function produces a vector with two elements in it. I checked my confidence interval calculation with the results from `t.test()`, which calculates the confidence interval and does a hypothesis test for a specified mean (`mu`). Be careful when intermixing vectors and scalar values when doing calculations like this so that unintended results do not occur.

Packages

A set of functions can be combined into a package. For those packages not already installed, R can download them from the Comprehensive R Archive Network (CRAN) and install them. For example, we will use the RODBC package later to read in Excel files containing our data. While in the R console, select PACKAGES > INSTALL PACKAGE(S) from the main menu.



A number of locations from around the world will be shown in a window. Choose one location close to you (I usually choose USA(IA), which is at Iowa State U.). Next, the list of packages will appear. Highlight the RODBC package and select OK.



The package now will be installed onto your computer. This only needs to be done once for a computer and particular version of R. To load the package into your current R session, type `library(package = RODBC)` at the R Console prompt. This needs to be done only

once per R session. If you close R and reopen, you need to use the `library()` function again.

The availability of these packages is one of the strengths of R. Users submit their own packages to CRAN, so that other users can then download them. There are more than 3,000 packages available! Packages also provide a convenient way to disseminate research. For example, a user will write a paper for a statistics journal and include the corresponding R code in a package. One example of this includes the binGroup package, which I am an author.

A list of all R packages is at http://cran.rproject.org/web/packages. One way to find a package containing functions of interest to you is by searching for a keyword. For example, searching for "group testing" leads to my package.

Characters

Object names can include periods and underscores. For example, "mod.fit" could be a name of an object and it is often read as "mod dot fit".

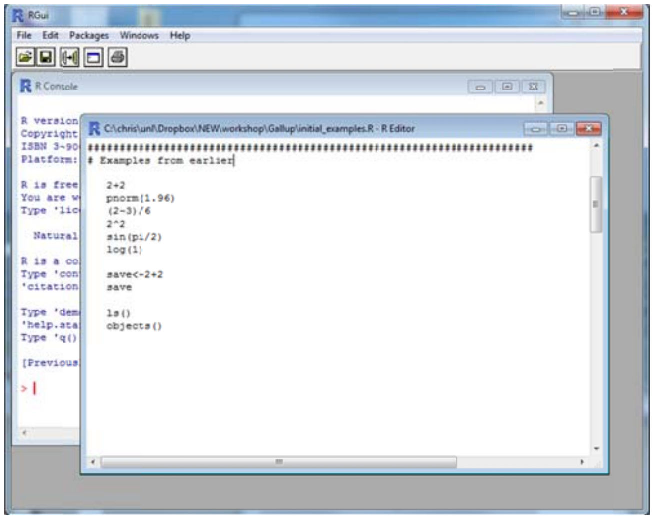
R is case sensitive!

II. Program editors

When there is a set of R code that you would like to execute all at once, you can save the code into a *program* and then run it. A text editor like Notepad or even Word will work as a place to type and then save the R code. Code from the editor can be copied and pasted into R. There are other editors available that make code reading and transferring much easier.

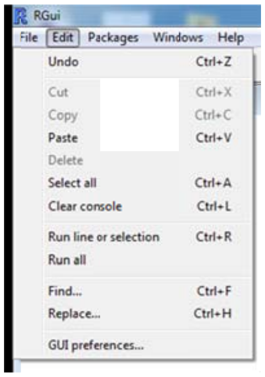
R's program editor

Starting with R 2.0, a VERY limited program editor was incorporated into it. Select FILE > NEW SCRIPT to create a new program. Below is what the editor looks like with some of the past examples:


RGui
File Edit Packages Windows Help
R Console
C:\chris\unl\Dropbox\NEW\workshop\Gallup\initial_examples.R - R Editor

```
#########################################################################
# Examples from earlier

  2+2
  pnorm(1.96)
  (2-3)/6
  2^2
  sin(pi/2)
  log(1)

  save<-2+2
  save

  ls()
  objects()
```

To run the current line of code (where the cursor is positioned) or a set of highlighted code, select EDIT > RUN LINE OR SELECTION.



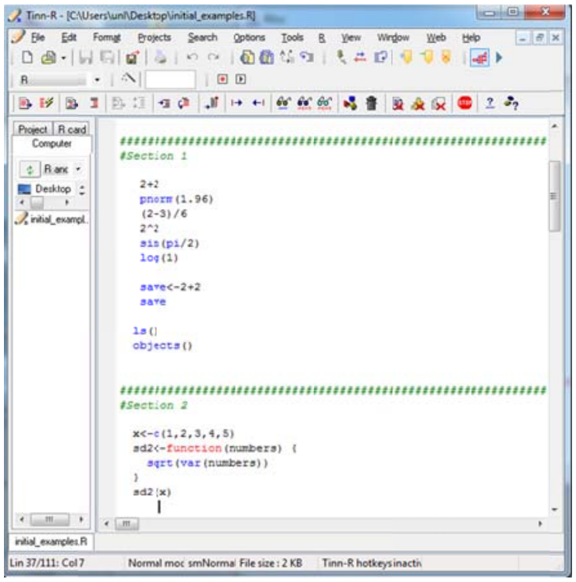
To run all of a program, select EDIT > RUN ALL. To save code as a program outside of R, select FILE > SAVE and make sure to use a .R extension on the file name. To open a program, select

FILE > OPEN SCRIPT. Note that you can have more than one program open at the same time.

There are much better program editors available! Each of the editors described next have color coding for the program code. This makes reading code much easier!

Tinn-R

Tinn-R (http://www.sciviews.org/Tinn-R/index.html) is a free, Windows-based program editor that is a separate software package outside of R. One of its most significant features is syntax highlighting, which means code is colorized to its purpose. For example, comments are green and text within quotes is burgundy. Note that a program needs to be saved with the .R extension for syntax highlighting to appear by default. Below is a screen capture of Tinn-R version 1.17.2.4:


Tinn-R - [C:\Users\unl\Desktop\initial_examples.R]
File Edit Format Projects Search Options Tools R View Window Web Help
Project R card
Computer
R.exe
Desktop
initial_exampl...
initial_examples.R

```
############################################################
#Section 1

  2+2
  pnorm(1.96)
  (2-3)/6
  2^2
  sin(pi/2)
  log(1)

  save<-2+2
  save

 ls()
 objects()


############################################################
#Section 2

 x<-c(1,2,3,4,5)
 sd2<-function(numbers) {
   sqrt(var(numbers))
 }
 sd2(x)
```

```
Lin 37/111: Col 7   Normal moc smNormal File size: 2 KB   Tinn-R hotkeys inactiv
```

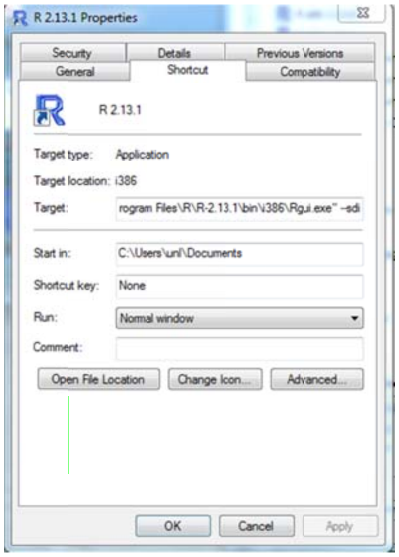
This is not the most up-to-date version, but I like it more than newer versions (reason to be discussed shortly).

Notes:

- Tinn-R has a database containing the syntax of many R functions. When you start typing a function name, the editor shows the syntax – similar to how Excel works when typing a function.
- To send part of a program from Tinn-R to an open R window, highlight the code and select the "Send selection" icon (, 4th from the left on the R toolbar). To send the entire program and see the results displayed in R, select the "Send all" icon (, 2nd from the left on the R toolbar).
- After sending code from Tinn-R to R, the Tinn-R window will come back as the top window. This is not ideal if both windows are open in the same area (your results in the R window would be hidden behind Tinn-R). To prevent this from occurring, select OPTIONS > RETURN FOCUS AFTER SENDING TO R.
- Syntax highlighting can be maintained with code that is copied and pasted into a word processing program. After highlighting the desired code to copy, select EDIT > COPY FORMATTED (TO EXPORT) > RTF.
- A good way to use Tinn-R and R is with two monitors. Open Tinn-R open on your primary monitor, and open R on your secondary monitor, so that both windows are viewable at the same time. This is how I use Tinn-

R and R. Alternatively, if you have one large monitor, open both windows side-by-side. Windows 7 makes this easy by dragging the windows as far as possible to either side of your monitor screen, and the windows will be re-sized appropriately.

There are two ways that R can be run: 1) *MDI* mode and 2) *SDI* mode. The MDI mode is the default, and this is what I am running now. “M”ultiple windows are contained within the R GUI (graphical user interface) including the R Console and plots (to be discussed later). The SDI mode has the R Console in a “s”ingle window and plots are in separate windows outside of the normal R GUI. You can determine the current mode by selecting EDIT > GUI PREFERENCES and examining the top line of the “Rgui Configuration Editor”. If you want to use the SDI mode, the easiest way is to add a “--sdi" (there are two hyphens before sdi) to a R shortcut target path in Windows. For example, I can right click on a R 2.13.1 shortcut to add to its Target path:



R 2.13.1

Target type: Application

Target location: i386

Target: rogram Files\R\R-2.13.1\bin\i386\Rgui.exe" --sdi

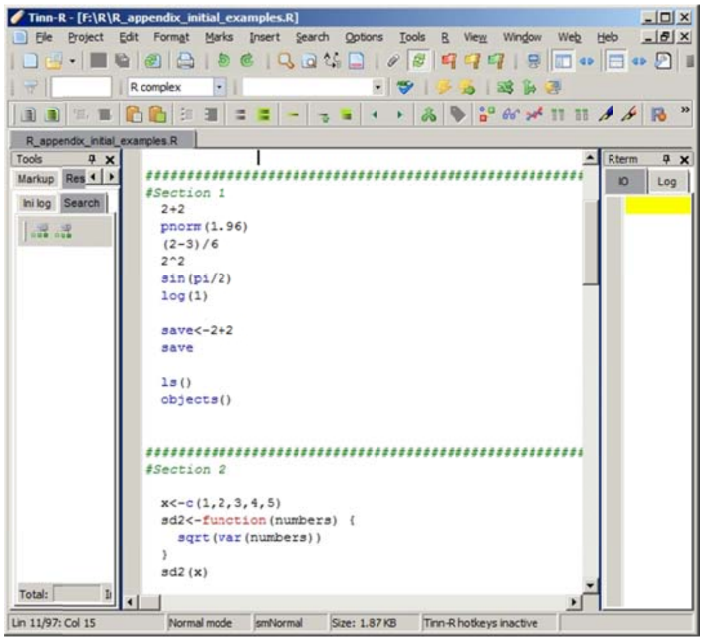
Start in: C:\Users\unl\Documents

Shortcut key: None

Run: Normal window

Comment:

Versions of Tinn-R greater than 1.17.2.4 require the SDI mode. Below is a screen capture of what version 2.3.5.2 looks like:



```
##############################################################
#Section 1
  2+2
  pnorm(1.96)
  (2-3)/6
  2^2
  sin(pi/2)
  log(1)

  save<-2+2
  save

  ls()
  objects()


##############################################################
#Section 2

  x<-c(1,2,3,4,5)
  sd2<-function(numbers) {
    sqrt(var(numbers))
  }
  sd2(x)
```

If Tinn-R is run first without R opened, R can be started by selecting the “R Control: gui (start/close)” icon (, R with an “x” in a red circle) from the R toolbar. Tinn-R will reposition its window and the R Console window to make both viewable simultaneously. Below are some additional comments about Tinn-R 2.3.5.2:

- Program code in Tinn-R can be run in R by selecting specific icons on Tinn-R's R toolbar. For example, a

highlighted portion of code is transferred to R by selecting the “R SEND: cursor to end line” icon (, arrow tip pointing to the right) on the R toolbar.
- After sending code from Tinn-R to R, the Tinn-R window will come back as the top window. If you want to prevent this from occurring, select OPTIONS > RETURN FOCUS TO EDITOR (AFTER SEND/CONTROL RGUI) or the appropriate icon (two circular arrows) on the Misc toolbar.
- By default, the line containing the cursor is highlighted in yellow. To turn this option off, select OPTIONS > COLORS (PREFERENCE) and uncheck the ACTIVE LINE (CHOICE) box.
- Using both Tinn-R 2.3.5.2 and Tinn-R 1.17.2.4 on the same computer does not work well.

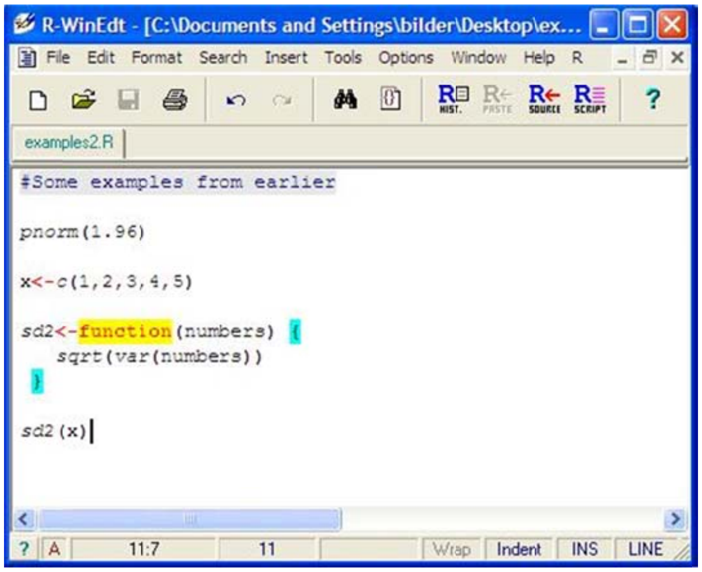
WinEdt

I used the WinEdt editor (version 5.5) with R's RWinEdt package as my main program editor for many years. I recently switched to Tinn-R because RWinEdt was not available for 64-bit processors at the time I purchased a new computer. The package is now available for 64-bit processors, so WinEdt with RWinEdt provide a nice alternative to Tinn-R. One downside is that WinEdt is shareware (30-day free trial).

Below is a brief description of the installation process:

1) Download WinEdt from http://www.winedt.com (see Downloads on left menu) and install on your computer.
2) Assuming R is already installed on your computer, install the RWinEdt package within R.
3) Type `library(package = RWinEdt)` at the command prompt to complete the installation. You can ignore any messages about running R in MDI mode unless you want to run R in a language other than English. Note that you may need to run R as an Administrator before doing this step (right click on an R shortcut and select RUN AS ADMINISTRATOR).
4) An additional menu heading in R named "R-WinEdt" will be available now. Select R-WinEdt and click on the SET AND START R-WinEdt option. This will automatically start WinEdt with the R add-on!

5) In the future, you can type `library(package = RWinEdt)` at the R Console prompt to start WinEdt within R.


R-WinEdt - [C:\Documents and Settings\bilder\Desktop\ex...
File Edit Format Search Insert Tools Options Window Help R
HIST. PASTE SOURCE SCRIPT
examples2.R
11:7
11
Wrap Indent INS LINE

```
#Some examples from earlier

pnorm(1.96)

x<-c(1,2,3,4,5)

sd2<-function(numbers) {
   sqrt(var(numbers))
 }

sd2(x)
```

To transfer code from WinEdt to R, highlight the code and select the PASTE button.

There are other ways to start WinEdt with its additional R components. In the past, I have done the following:

- Use this target path to start WinEdt:

```
"C:\Program Files\WinEdt Team\WinEdt\WinEdt.exe"
   -C="R-WinEdt" -e=r.ini
```

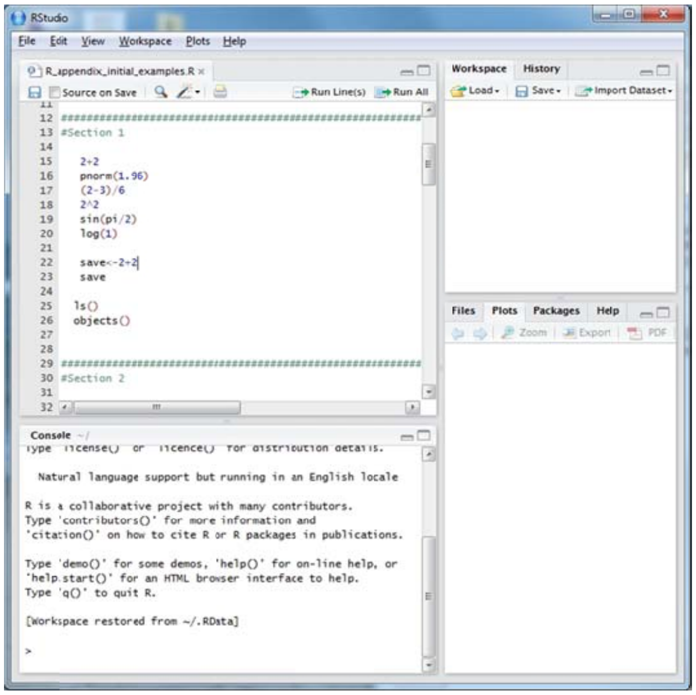
- Add this line of code:

```
options(defaultPackages = c(getOption("defaultPackages"),
  "RWinEdt"))
```

to the "Rprofile.site" file at "C:\Program Files\R\R-2.13.1\etc" (for R 2.13.1). Whenever R starts, it will automatically run `library(package = RWinEdt)`.

RStudio

RStudio (www.rstudio.org) is still in beta testing, but it offers a nice interface for R. The software is free and runs on multiple operating systems. Below is a screen capture of it for version 0.92.23:



```
##############################################################
#Section 1

   2+2
   pnorm(1.96)
   (2-3)/6
   2^2
   sin(pi/2)
   log(1)

   save<-2+2
   save

  ls()
  objects()


##############################################################
#Section 2
```

```
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace restored from ~/.RData]

>
```

The software package combines a program editor, R Console window, graphics window, working database listing, and other items within one overall window. To create a new program, select FILE > NEW > R SCRIPT or open an existing program by selecting FILE > OPEN FILE. To run a segment of code, highlight it and then select the "Run" icon in the program editor window.

Other editors

Please see http://www.sciviews.org/_rgui/projects/Editors.html for a listing of other editors. In particular, the Emacs editor (http://www.gnu.org/software/emacs) with the Emacs Speaks Statistics (http://ess.r-project.org/) add-on are popular especially for Unix users.

III. Regression Example

Suppose you would like to estimate an individual's college GPA by their high school GPA through a simple linear regression model. The corresponding R program for this example is gpa.R and the data files are gpa.txt (plain text file using space delimiters), gpa.csv (plain text file using comma delimiters) and gpa.xls.

Data management

Below is the code used to read the data into R:

```
> ##########################################################
> # NAME:  Chris Bilder                                    #
> # DATE:  8-14-11                                         #
> # PURPOSE: Simple data analysis example in R using the #
> #          gpa data set                                  #
> #                                                        #
> # NOTES:                                                 #
> ##########################################################

> #Read in the data with spaces separating variable values
> gpa<-read.table(file = "C:\\chris\\unl\\Dropbox\\NEW\\
    workshop\\Gallup\\gpa.txt", header=TRUE,  sep = "")

> #Print data set
> gpa
   HS.GPA  College.GPA
1     3.04          3.1
2     2.35          2.3
3     2.70          3.0
4     2.05          1.9
5     2.83          2.5
6     4.32          3.7
```

```
7     3.39          3.4
8     2.32          2.6
9     2.69          2.8
10    0.83          1.6
11    2.39          2.0
12    3.65          2.9
13    1.85          2.3
14    3.83          3.2
15    1.22          1.8
16    1.48          1.4
17    2.28          2.0
18    4.00          3.8
19    2.28          2.2
20    1.88          1.6

> head(gpa)
  HS.GPA  College.GPA
1   3.04          3.1
2   2.35          2.3
3   2.70          3.0
4   2.05          1.9
5   2.83          2.5
6   4.32          3.7
```

Notes:

- The `#()` symbol is used to start *comment* lines in R. Comments are helpful to include in programs to describe the code.
- The `read.table()` function reads the data into R. Pay special attention to the syntax used with the file argument. The "\\" are needed between folder names rather than only "\" ("/" can be used too). Also, because the variable names are at the beginning of the data file, the `header = TRUE` option is given. Finally, the `sep = ""` option specifies that white space (spaces, tabs, …) separates variable values in the data file.

- The `gpa` object is referred to as a *data.frame* in R's terminology. It can be printed by typing its name and then Enter.
- The `head()` function is a simple way to print the first few lines of an object as a quick check. A `tail()` function also exists to print the last few lines.

Alternative data formats include:
- Comma delimited – Use `sep = ","` with `read.table()` or use `read.csv()`.

```
> gpa.comma1<-read.table(file = "C:\\chris\\unl\\Dropbox
    \\NEW\\workshop\\Gallup\\gpa.csv", header=TRUE, sep =
    ",")
> head(gpa.comma1)
  HSGPA CollegeGPA
1  3.04        3.1
2  2.35        2.3
3  2.70        3.0
4  2.05        1.9
5  2.83        2.5
6  4.32        3.7

> #Another way
> gpa.comma2<-read.csv(file = “C:\\chris\\unl\\Dropbox
    \\NEW\\workshop\\Gallup\\gpa.csv", header=TRUE)
> head(gpa.comma2)
  HSGPA CollegeGPA
1  3.04        3.1
2  2.35        2.3
3  2.70        3.0
4  2.05        1.9
5  2.83        2.5
6  4.32        3.7
```

- Excel files – Use the RODBC package

```
> library(package = RODBC)
> z<-odbcConnectExcel(xls.file = "C:\\chris\\unl\\Dropbox
    \\NEW\\workshop\\Gallup\\gpa.xls")
> gpa.excel<-sqlFetch(channel = z, sqtable = "sheet1")
> close(z)

> head(gpa.excel)
  HSGPA CollegeGPA
1  3.04        3.1
2  2.35        2.3
3  2.70        3.0
4  2.05        1.9
5  2.83        2.5
6  4.32        3.7
```

The `write.table()` and `write.csv()` functions export data out of R:

```
> write.csv(x = gpa, file = "C:\\chris\\unl\\Dropbox\\NEW\\
   workshop\\Gallup\\temp.csv")
```

The `summary()` function provides a simple data summary:

```
> summary(gpa)
     HS.GPA        College.GPA   
 Min.   :0.830   Min.   :1.400  
 1st Qu.:2.007   1st Qu.:1.975  
 Median :2.370   Median :2.400  
 Mean   :2.569   Mean   :2.505  
 3rd Qu.:3.127   3rd Qu.:3.025  
 Max.   :4.320   Max.   :3.800  
```

Once data is in a data.frame, one variable at a time can be accessed by using `<data.frame>$<variable>`. For example,

```
> names(gpa)
[1] "HS.GPA"      "College.GPA"
> gpa$HS.GPA
 [1] 3.04 2.35 2.70 2.05 2.83 4.32 3.39 2.32 2.69 0.83 2.39
     3.65 1.85 3.83 1.22 1.48
[17] 2.28 4.00 2.28 1.88
```

Notice that the `names()` function provides a list of variables included in the data.frame. We will use this function again later for more complex data objects!

Parts of the data.frame can also be accessed through using a matrix-like reference. For example,

```
> gpa[1,1]
[1] 3.04
> gpa[,1]
 [1] 3.04 2.35 2.70 2.05 2.83 4.32 3.39 2.32 2.69 0.83 2.39
     3.65 1.85 3.83 1.22 1.48
[17] 2.28 4.00 2.28 1.88
> gpa[1,1:2]
  HS.GPA College.GPA
1   3.04         3.1
> gpa[1,c(1,2)]
  HS.GPA College.GPA
1   3.04         3.1
```

Questions:
- How can you access only the first row of a data.frame?
- What does `gpa[,-2]` return?

There are times when you would like to access parts of a data set based on some condition. For example, suppose you would like to view observations where the high school GPA was less than 2.5:

```
> gpa$HS.GPA<2.5
 [1] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
     TRUE  TRUE FALSE  TRUE
[14] FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE

> gpa[gpa$HS.GPA<2.5, ]
   HS.GPA College.GPA
2    2.35         2.3
4    2.05         1.9
8    2.32         2.6
10   0.83         1.6
11   2.39         2.0
13   1.85         2.3
15   1.22         1.8
16   1.48         1.4
17   2.28         2.0
19   2.28         2.2
20   1.88         1.6

> sum(gpa$HS.GPA<2.5)
[1] 11
```

The `gpa$HS.GPA<2.5` part performs the logical comparison of “Is a high school GPA < 2.5?” A TRUE or FALSE is produced for each entry. Using the resulting vector, we can pull out those observations from `gpa` that satisfy the condition. Also, note that R treats the `TRUE` and `FALSE` values as 1's and 0's, respectively, when working with a mathematical function. This is helpful to determine how often a condition is satisfied.

The `ifelse()` function performs a similar logical comparison:

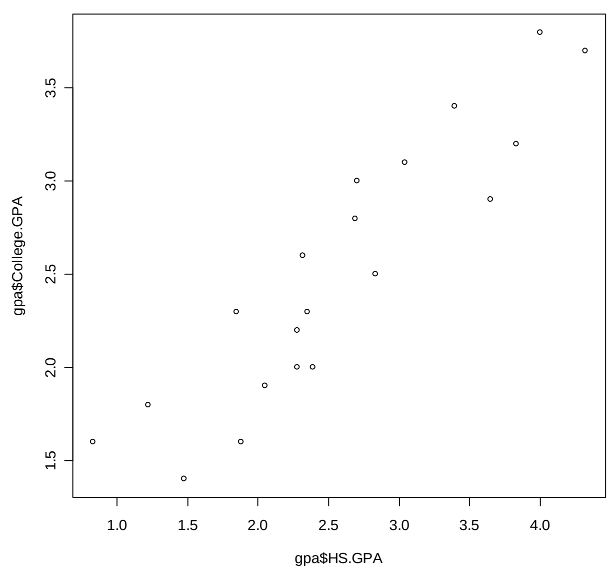
```
> #If then else - note that "&" means "and"
> test.cond<-ifelse(test = gpa$HS.GPA<2.5 &
    gpa$College.GPA<2.5, yes = 1, no = 0)
> sum(test.cond)
[1] 10

> #If then else - note that "or" means "and"
> test.cond<-ifelse(test = gpa$HS.GPA<2.5 |
    gpa$College.GPA<2.5, yes = 1, no = 0)
> sum(test.cond)
[1] 11
```

The `ifelse()` function is useful for more complicated resulting values from the comparison.

Scatter plot

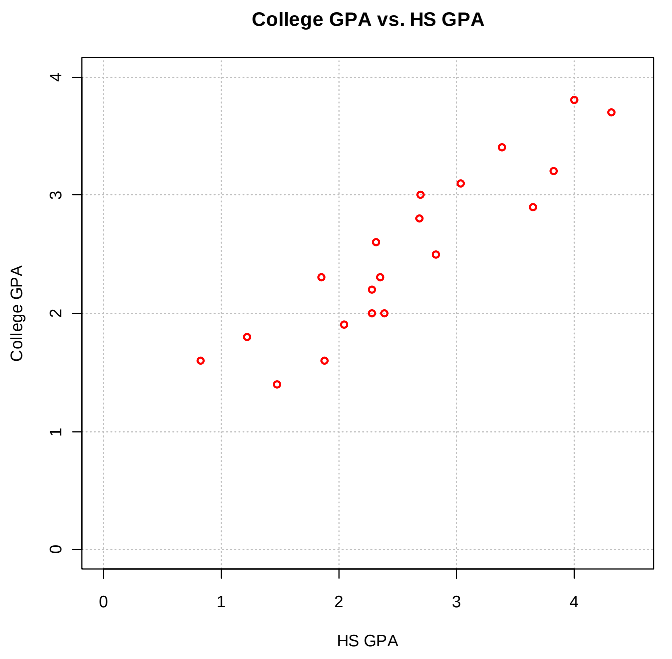
Below is a simple scatter plot of the data created by the `plot()` function. This plot is created in an *R Graphics* window and then copied into Word:

```
> plot(x = gpa$HS.GPA, y = gpa$College.GPA)
```



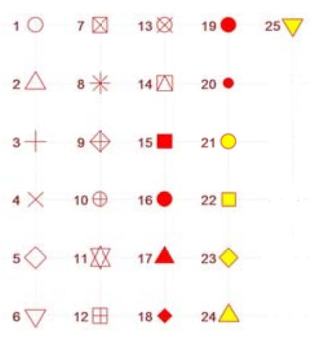
Including optional arguments makes the plot look much better:

```
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab =
    "HS GPA", ylab = "College GPA", main = "College
    GPA vs. HS GPA", xlim = c(0,4.5), ylim =
    c(0,4.5), col = "red", pch = 1, cex = 1.0, lwd = 2.0,
    panel.first = grid(col = "gray", lty = "dotted"))
```



Descriptions of the optional arguments:

- `x` = and `y` = specify what is plotted on the x-axis and y-axis, respectively.
- `xlab` = and `ylab` = specify the x-axis and y-axis labels, respectively.
- `main` = specifies the main title of the plot.
- `xlim` = and `ylim` = specify the x-axis and y-axis limits, respectively. Notice the use of the `c()` function.
- `col` = specifies the color of the plotting points. Run the `colors()` function to see what possible colors can be used. Also, you can see these colors at http://research.stowers-institute.org/efg/R/Color/Chart/index.htm.
- `pch` = specifies the plotting characters. Below is a list of possible characters.



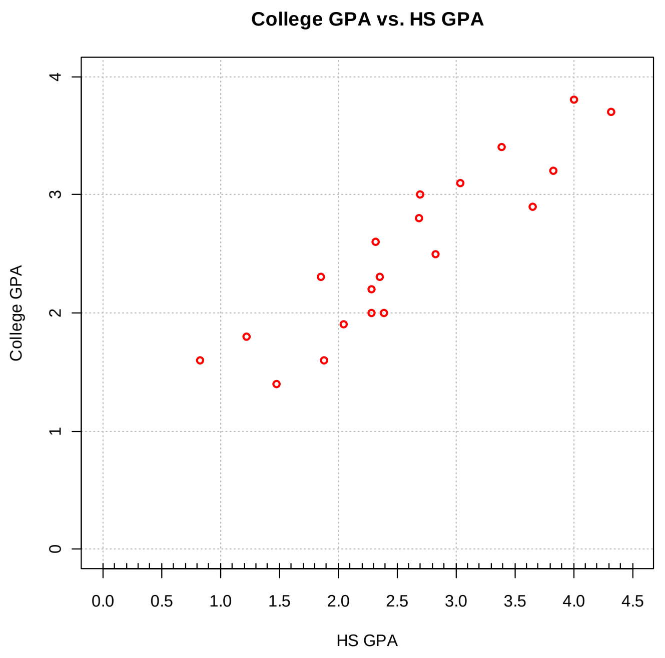
- `cex` = specifies the magnification level of the plotting characters, where 1.0 is the default. A value of 1.5 means 50% larger than the default, and a value of 0.5 means 50% smaller than the default.
- `lwd` = specifies the thickness of plotting points or lines, where 1.0 is the default.
- `panel.first = grid()` specifies that grid lines are plotted. The line types are: 1 = solid, 2 = dashed, 3 = dotted, 4 = dotdash, 5 = longdash, 6 = twodash. The corresponding words "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash" can be given as well. These line type specifications are used in other functions too (including `plot()`) with the `lty` argument.
- The `par()` function's Help contains more information about the different plotting options!

The plot is easily imported into Word. First, make sure the R Graphics window is the current window in R and then select FILE > COPY TO THE CLIPBOARD > AS A METAFILE. Select the PASTE button in Word to import it.

To obtain specific x-axis or y-axis tick marks on a plot, use the `axis()` function. For example,

```
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS
    GPA", ylab = "College GPA", main = "College GPA vs. HS
    GPA", xlim = c(0,4.5), ylim = c(0,4.0), col = "red",
    pch = 1, cex = 1.0, lwd = 2, panel.first=grid(col =
```

```
    "gray", lty = "dotted"), xaxt = "n")
> #Major tick marks for x-axis
> axis(side = 1, at = seq(from = 0, to = 4.5, by = 0.5))
> #Minor tick marks for x-axis
> axis(side = 1, at = seq(from = 0, to = 4.5, by = 0.1),
    tck = 0.01, labels = FALSE)
```



Notice the use of `xaxt = "n"` in the `plot()` function. This specifies that no tick marks are to be drawn on the x-axis.

Fitting the model

The `lm()` function fits linear regression models:

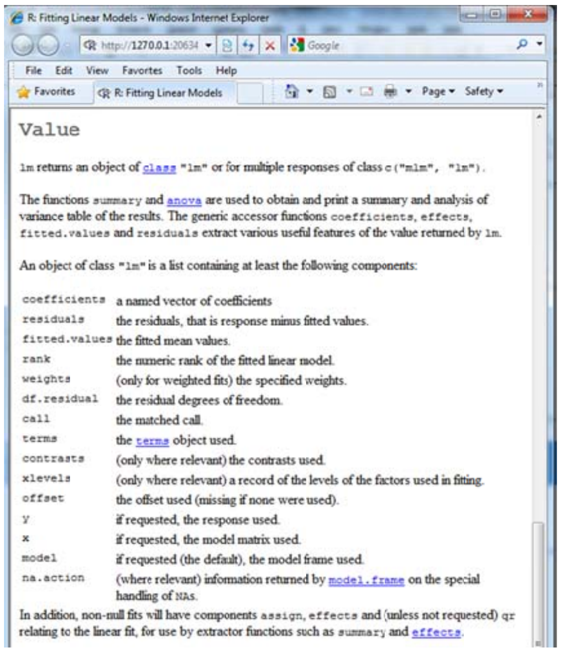
```
> mod.fit<-lm(formula = College.GPA ~ HS.GPA, data = gpa)

> #A very brief look of what is inside of mod.fit
> mod.fit

Call:
lm(formula = College.GPA ~ HS.GPA, data = gpa)

Coefficients:
(Intercept)       HS.GPA
     0.7076       0.6997
```

The ~ symbol separates the dependent and independent variables within the `formula` argument. If there were multiple independent variables, the + symbol would be used to separate them.

The results are stored in an object called `mod.fit`. By running the `mod.fit` object name only at a command prompt, R prints a some information about what is inside of it. To obtain a more thorough listing, use the `names()` function:

```
> names(mod.fit)
 [1] "coefficients"  "residuals"     "effects"       "rank"
     "fitted.values"
 [6] "assign"        "qr"            "df.residual"
     "xlevels"       "call"
[11] "terms"         "model"
```

The `mod.fit` object is referred to as a *list* in R's terminology. Lists provide a general way to link a number of other items together under one object. The linked items do not need to be the same size or type, so lists are often used as the object returned from running more complex functions. A summary of what each item represents within this list is given in the help web page for `lm()`:



Value

`lm` returns an object of class `"lm"` or for multiple responses of class `c("mlm", "lm")`.

The functions `summary` and anova are used to obtain and print a summary and analysis of variance table of the results. The generic accessor functions `coefficients`, `effects`, `fitted.values` and `residuals` extract various useful features of the value returned by `lm`.

An object of class `"lm"` is a list containing at least the following components:

`coefficients`	a named vector of coefficients
`residuals`	the residuals, that is response minus fitted values.
`fitted.values`	the fitted mean values.
`rank`	the numeric rank of the fitted linear model.
`weights`	(only for weighted fits) the specified weights.
`df.residual`	the residual degrees of freedom.
`call`	the matched call.
`terms`	the `terms` object used.
`contrasts`	(only where relevant) the contrasts used.
`xlevels`	(only where relevant) a record of the levels of the factors used in fitting.
`offset`	the offset used (missing if none were used).
`y`	if requested, the response used.
`x`	if requested, the model matrix used.
`model`	if requested (the default), the model frame used.
`na.action`	(where relevant) information returned by `model.frame` on the special handling of `NA`s.

In addition, non-null fits will have components `assign`, `effects` and (unless not requested) `qr` relating to the linear fit, for use by extractor functions such as `summary` and `effects`.

To access part of the list, use the syntax `<list>$<component>`. This is the same syntax used with a data.frame, because a data.frame is a special type of list (each component is a vector of the same length). Below are a couple of examples with the `mod.fit` object:

```
> mod.fit$coefficients
(Intercept)      HS.GPA
  0.7075776   0.6996584

> mod.fit$residuals
          1           2           3           4           5           6           7           8
 0.26546091 -0.05177482  0.40334475 -0.24187731 -0.18761083 -0.03010181  0.32058048  0.26921493
          9          10          11          12          13          14          15          16
 0.21034134  0.31170591 -0.37976115 -0.36133070  0.29805437 -0.18726921  0.23883914 -0.34307203
         17          18          19          20
-0.30279873  0.29378887 -0.10279873 -0.42293538
```

We can combine some of these items together into one data.frame to summarize the model's fit:

```
> save.fit<-data.frame(gpa, College.GPA.hat =
     round(mod.fit$fitted.values,2), residuals =
     round(mod.fit$residuals,2))

> head(save.fit)
  HS.GPA College.GPA College.GPA.hat residuals
1   3.04         3.1            2.83      0.27
2   2.35         2.3            2.35     -0.05
3   2.70         3.0            2.60      0.40
4   2.05         1.9            2.14     -0.24
5   2.83         2.5            2.69     -0.19
6   4.32         3.7            3.73     -0.03
```

The `summary()` function can be used with the `mod.fit` object to summarize the list's contents:

```
> summary(object = mod.fit)

Call:
lm(formula = College.GPA ~ HS.GPA, data = gpa)

Residuals:
     Min       1Q   Median       3Q      Max
-0.42294 -0.25711 -0.04094  0.27536  0.40334

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.70758    0.19941   3.548  0.00230 **
HS.GPA       0.69966    0.07319   9.559 1.78e-08 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 `
' 1

Residual standard error: 0.297 on 18 degrees of freedom
Multiple R-Squared: 0.8354,     Adjusted R-squared: 0.8263
F-statistic: 91.38 on 1 and 18 DF,  p-value: 1.779e-08
```

Notice the different results that we received here from what we received earlier with `summary(gpa)`! We will discuss soon why the same function produces different results.

The estimated regression model is

$$\widehat{\text{College.GPA}} = 0.70758 + 0.69966\text{HS.GPA}.$$

What if there was a categorical independent variable? R automatically creates indicator variables to represent it in a model, where the "set first level equal to 0" type of

coding is performed (SAS does "set last level equal to 0"). Below is a quick example:

```
> where.live<-c("with parents", "dorm", "off-campus")
> x<-rep(x = where.live, each = 7)
> gpa2<-data.frame(gpa, where.live = x[-21])
> head(gpa2)
  HS.GPA College.GPA   where.live
1   3.04         3.1 with parents
2   2.35         2.3 with parents
3   2.70         3.0 with parents
4   2.05         1.9 with parents
5   2.83         2.5 with parents
6   4.32         3.7 with parents

> levels(gpa2$where.live)
[1] "dorm"         "off-campus"   "with parents"
> contrasts(gpa2$where.live)
             off-campus with parents
dorm                  0            0
off-campus            1            0
with parents          0            1

> mod.fit2<-lm(formula = College.GPA ~ HS.GPA + where.live,
    data = gpa2)
> summary(mod.fit2)

Call:
lm(formula = College.GPA ~ HS.GPA + where.live, data =
gpa2)

Residuals:
     Min       1Q   Median       3Q      Max
-0.40615 -0.25755 -0.02649  0.24466  0.45214

Coefficients:
                       Estimate Std. Error t value Pr(>|t|)
(Intercept)             0.80244    0.23009   3.487  0.00304 **
HS.GPA                  0.67101    0.07953   8.437 2.76e-07 ***
where.liveoff-campus   -0.13862    0.17062  -0.812  0.42847
where.livewith parents  0.05806    0.16594   0.350  0.73096
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3033 on 16 degrees of freedom
Multiple R-squared: 0.8475,     Adjusted R-squared: 0.8189
F-statistic: 29.64 on 3 and 16 DF,  p-value: 9.084e-07
```

R uses the ordering given by `levels()` (this will be alphabetical unless specified otherwise) to decide what level to make the base level ("dorm").

If a categorical independent variable is coded as number, you need to specify it is categorical within `lm()`. This is done by using `factor(<variable>)` in the `formula` argument. For example, suppose `gpa2$where.live` had the levels of 1, 2 and 3. The formula argument would be:

```
formula = College.GPA ~ HS.GPA + factor(where.live)
```

The gpa.R program provides an example.

Transformations of independent variables can be included within the `formula` argument. For some transformations, the `I()` function needs to be used to tell R how to interpret the transformation. For example, suppose we would like to have HS.GPA and HS.GPA^2 in the model. The `formula` argument would be:

```
formula = College.GPA ~ HS.GPA + I(HS.GPA^2)
```

The reason for this extra function is because a `formula` argument like

```
formula = Y ~ (X1 + X2)^2
```

is the syntax for R to estimate:

$$E(Y) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$$

Object oriented language

Every object in R has an *attribute* called a *class*. You can view them by using the `attributes()` or `class()` functions:

```
> class(gpa)
[1] "data.frame"
> class(gpa$HS.GPA)
[1] "numeric"
> class(lm)
[1] "function"
> class(mod.fit)
[1] "lm"
```

R is often referred to as an objected oriented language because *generic* functions, like `summary()`, provide different results depending on an object's class. When a generic function is invoked, it first checks for the class of the object. R then looks for a *method* function with the name format `<generic function>.<class name>`.

Examples for `summary()`:

- `summary(mod.fit)` – The function `summary.lm()` summarizes the regression model fit.
- `summary(gpa)` – The function `summary.data.frame()` summarizes the data.frame's contents.
- `summary.default()` – R attempts to run this function if there is no method function for a class.

There are many generic functions! For example, `plot()` is a generic function (try `plot(mod.fit)` to see what happens!). We will also see other generic functions like `predict()` later.

Why is R set-up like this?

> The purpose of generic functions is to use a familiar language set with any object. For example, we frequently want to summarize data or a model, `summary()`; to plot data, `plot()`; and to find predictions, `predict()`; so it is convenient to use the same language set no matter the application.

Understanding generic and method functions may be one of the most difficult parts for new R users. However, it is important to know the basics for these functions now in order to locate the correct help for a function. For example, suppose you want help on what `summary()` does with an object created by `lm()`. Do not examine the help for the generic function itself – `help(summary)`. Instead, examine the help for the method function – `help(summary.lm)`.

To show all method functions associated with a class, use `methods(class = <class>)`. The method functions associated with the *lm* class are:

```
> methods(class = lm)
```

```
 [1] add1.lm*           alias.lm*          anova.lm
     case.names.lm*     confint.lm*
     cooks.distance.lm*
```

<OUTPUT EDITED>

```
 [31] rstudent.lm        simulate.lm*       summary.lm
      variable.names.lm* vcov.lm*

   Non-visible functions are asterisked
```

To show all method functions for a generic function, use `methods(generic.function = <generic function>)`. Below are the method functions associated with `summary()`:

```
> methods(generic.function = summary)
 [1] summary.aov               summary.aovlist
     summary.aspell*           summary.connection
     summary.data.frame
```

<OUTPUT EDITED>

```
 [26] summary.stepfun         summary.stl*
      summary.table           summary.tukeysmooth*

   Non-visible functions are asterisked
```

Below are a few examples of using generic functions with `mod.fit`:

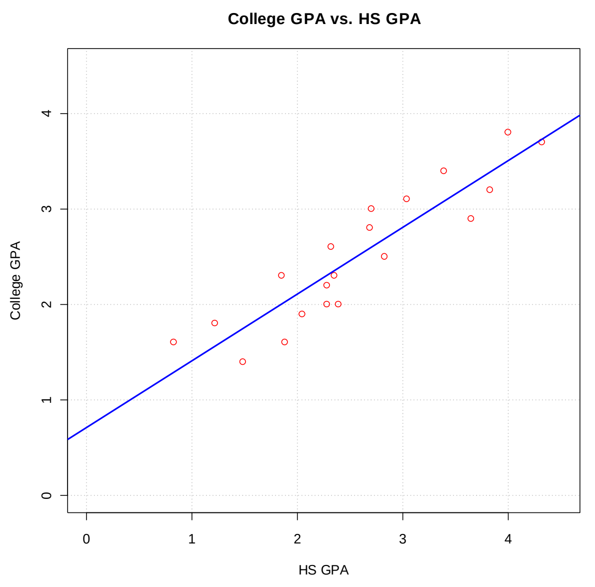
```
> anova(object = mod.fit)
Analysis of Variance Table

Response: College.GPA
          Df Sum Sq Mean Sq F value    Pr(>F)
HS.GPA     1 8.0615  8.0615  91.379 1.779e-08 ***
```

```
Residuals 18 1.5880  0.0882
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘
’ 1

> vcov(object = mod.fit)
           (Intercept)        HS.GPA
(Intercept)   0.03976606 -0.013762181
HS.GPA       -0.01376218  0.005357019

> confint(object = mod.fit, level = 0.95)
               2.5 %    97.5 %
(Intercept) 0.2886238 1.1265315
HS.GPA      0.5458884 0.8534283

> AIC(object = mod.fit)
[1] 12.09198

> residuals(object = mod.fit)
          1           2           3           4           5           6 
 0.26546091 -0.05177482  0.40334475 -0.24187731 -0.18761083 -0.03010181 
          7           8           9          10          11          12 
 0.32058048  0.26921493  0.21034134  0.31170591 -0.37976115 -0.36133070 
         13          14          15          16          17          18 
 0.29805437 -0.18726921  0.23883914 -0.34307203 -0.30279873  0.29378887 
         19          20 
-0.10279873 -0.42293538 

> rstudent(model = mod.fit)
         1          2          3          4          5          6          7 
 0.9195704 -0.1742267  1.4343816 -0.8357009 -0.6386997 -0.1127359  1.1415092 
         8          9         10         11         12         13         14 
 0.9281934  0.7170342  1.2145035 -1.3420370 -1.3244262  1.0499732 -0.6720106 
        15         16         17         18         19         20 
 0.8717594 -1.2519781 -1.0518399  1.0945298 -0.3472377 -1.5389689 
```

Estimating the response

Plot the model on the scatter plot:

```
> #Open a new graphics window
> win.graph(width = 6, height = 6, pointsize = 10)

> #Same scatter plot as before
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS
   GPA", ylab = "College GPA", main = "College GPA vs.
   HS GPA", xlim = c(0,4.5), ylim = c(0,4.5), col =
   "red", pch = 1, cex = 1.0, panel.first=grid(col =
   "gray", lty = "dotted"))

> #Puts the line y = a + bx on the plot
> abline(a = mod.fit$coefficients[1], b =
   mod.fit$coefficients[2], lty = 1, col = "blue",
   lwd = 2)
```

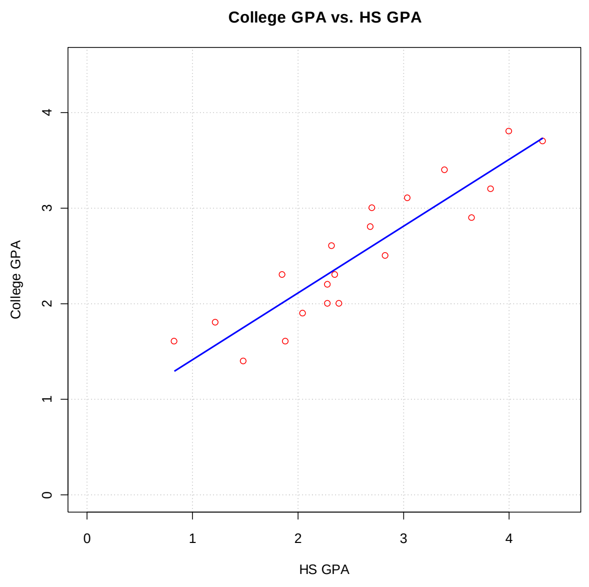


What is a problem with the above plot?

New plot:

```
> win.graph(width = 6, height = 6, pointsize = 10)

> #Same scatter plot as before
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS
      GPA", ylab = "College GPA", main = "College GPA vs.
      HS GPA", xlim = c(0,4.5), ylim = c(0,4.5), col =
     "red", pch = 1, cex = 1.0, panel.first=grid(col =
     "gray", lty = "dotted"))

> #Draw a line from (x0, y0) to (x1, y1)
```

```
> segments(x0 = min(gpa$HS.GPA), y0 =
   mod.fit$coefficients[1]  +  mod.fit$coefficients[2]  *
   min(gpa$HS.GPA), x1 = max(gpa$HS.GPA), y1 =
   mod.fit$coefficients[1]  +   mod.fit$coefficients[2]  *
   max(gpa$HS.GPA), lty = "solid", col = "blue", lwd = 2)
```



The `predict()` function finds point estimates, confidence intervals for the mean response, and prediction intervals for the response variable:

```
> pred.data<-data.frame(HS.GPA = c(2, 3, 4))
> predict(object = mod.fit, newdata = pred.data)
```

```
       1        2        3 
2.106894 2.806553 3.506211 

> predict(object = mod.fit, newdata = pred.data, se.fit =
   TRUE, interval = "confidence", level = 0.95)
$fit
       fit      lwr      upr
1 2.106894 1.942197 2.271591
2 2.806553 2.652079 2.961026
3 3.506211 3.245655 3.766767

$se.fit
         1          2          3 
0.07839267 0.07352648 0.12401980 

$df
[1] 18

$residual.scale
[1] 0.2970191

> save.pred<-predict(object = mod.fit, newdata = pred.data,
   se.fit = TRUE, interval = "confidence", level = 0.95)
> names(save.pred)
[1] "fit"            "se.fit"         "df"
   "residual.scale"
> save.pred$fit
       fit      lwr      upr
1 2.106894 1.942197 2.271591
2 2.806553 2.652079 2.961026
3 3.506211 3.245655 3.766767
```

Viewing function code

Typing a function name, like `lm`, and invoking it at a command prompt gives the actual code used by a function! This is useful when you want to know more about how a function works or if you want to create your own function by modifying the original version. Sometimes, there will be code within the function like .C or .Fortran. These are calls outside of R to a C or Fortran program. The code within these programs can still be viewed, but they need to be obtained from CRAN.

For new R users, the code within functions can be difficult to understand. The following steps are helpful to interpret the code:

1) Copy and paste the function code into a program editor to view it with syntax highlighting.
2) Set values for the function's arguments.
3) Run the code line-by-line to see what it does!

We will see an example of this soon.

Writing your own functions

When the same code is run for different analyses, it is helpful to write a function for it. Below is a function written to estimate a regression model and construct a scatter plot with the estimated model:

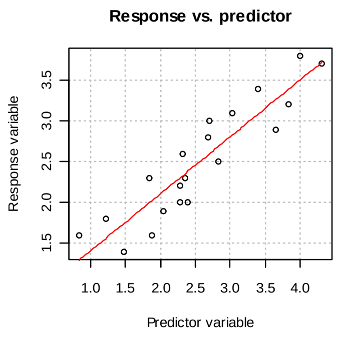
```
my.reg.func<-function(x, y, data) {

    #Fit the simple linear regression model and save the
       results in mod.fit
    mod.fit<-lm(formula = y ~ x, data = data)

    #Open a new graphics window
    win.graph(width = 6, height = 6, pointsize = 10)

    #Same scatter plot
    plot(x = x, y = y, xlab = "x", ylab = "y", main = "y
      vs. x", col = "red", pch = 1, cex = 1.0,
      panel.first=grid(col = "gray", lty = "dotted"))

    #Draw a line from (x0, y0) to (x1, y1)
    segments(x0 = min(x), y0 = mod.fit$coefficients[1] +
      mod.fit$coefficients[2]*min(x), x1 = max(x), y1 =
      mod.fit$coefficients[1] + mod.fit$coefficients[2] *
      max(x), lty = 1, col = "blue", lwd = 2)

    #This is the object returned
    mod.fit
  }

#Run the function and save the results
save.it<-my.reg.func(x = gpa$HS.GPA, y = gpa$College.GPA,
  data = gpa)
```

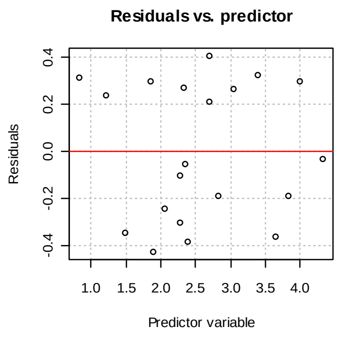
If this was the first time that you saw the code within the function, it might not be clear what it does (especially if

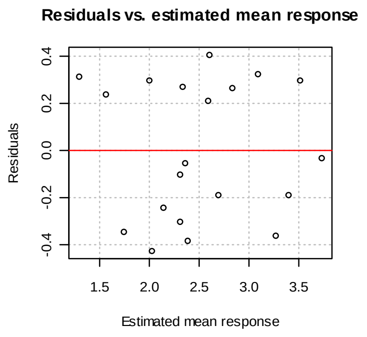
the comments were not given). Following the steps given on page 62 would enable you to figure it out.

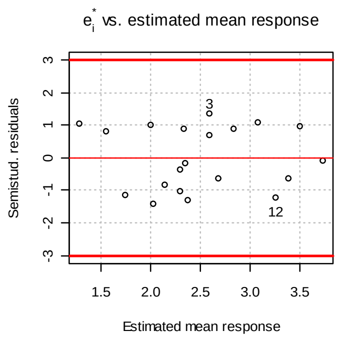
I created the next function for a regression course. The function automates the process of examining diagnostic tools for a simple linear regression model. You can see its code in the file examine.model.simple.R. This code can be run as before or the `source()` function can be used to run it. Below is an example:

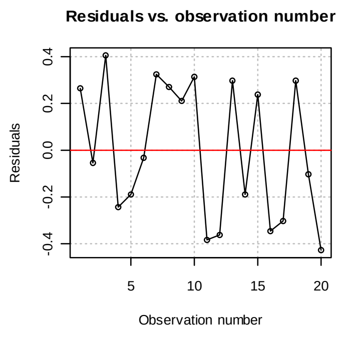
```
> source("C:\\chris\\unl\\Dropbox\\NEW\\workshop\\
   Gallup\\examine.model.simple.R")
> save.it<-examine.model.simple(mod.fit.obj = mod.fit,
   const.var.test = TRUE, boxcox.find = TRUE)
> names(save.it)
[1] "sum.data"        "semi.stud.resid" "levene"       "bp"
[5] "lambda.hat"

> save.it$sum.data
       Y              X        
 Min.   :1.400   Min.   :0.830  
 1st Qu.:1.975   1st Qu.:2.007  
 Median :2.400   Median :2.370  
 Mean   :2.505   Mean   :2.569  
 3rd Qu.:3.025   3rd Qu.:3.127  
 Max.   :3.800   Max.   :4.320  

> save.it$levene
Levene's Test for Homogeneity of Variance (center = median)
      Df F value Pr(>F)
group  1  0.0766 0.7851
      18
```

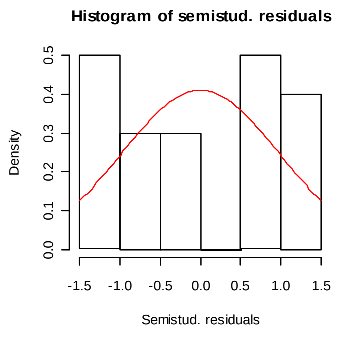


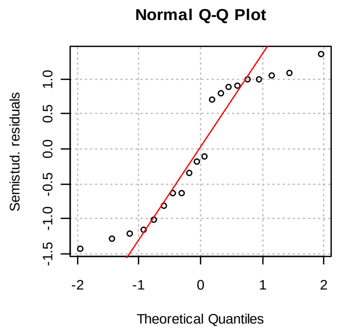


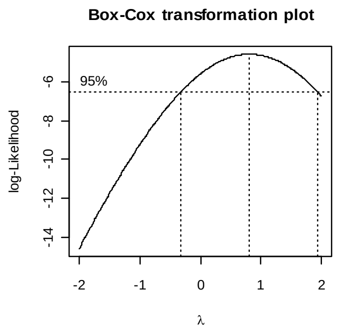












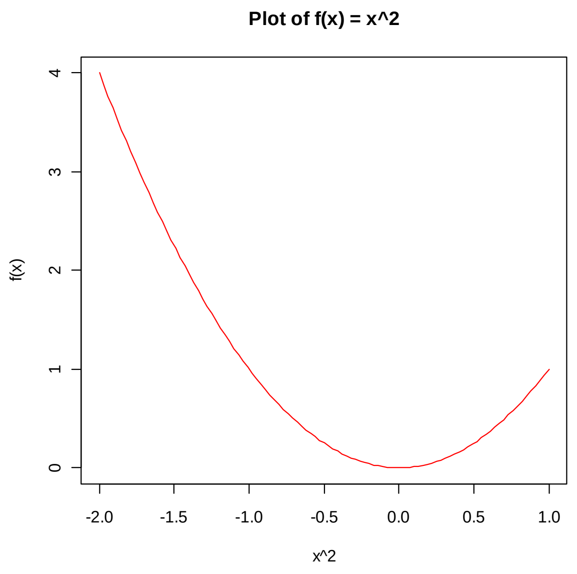
IV. Graphics

"Traditional" R plots are created using functions from the graphics package. This package is installed in R by default, and its functions are always available for use. The functions within it should be able to satisfy the majority of your needs. The best way to start learning about R graphics is with this package, because many of its basics can be applied to other packages. These other packages, like lattice and ggplot2, produce most of the same plots, but they can also produce more sophisticated plots.

Curves

The `curve()` function draws mathematical functions, like f(x), of one variable on a plot (see curve.R). Below is an example with f(x) = x^2:

```
> curve(expr = x^2, from = -2, to = 1, n = 101, main =
   "Plot of f(x) = x^2", ylab = "f(x)", xlab = "x^2", col =
   "red")
```

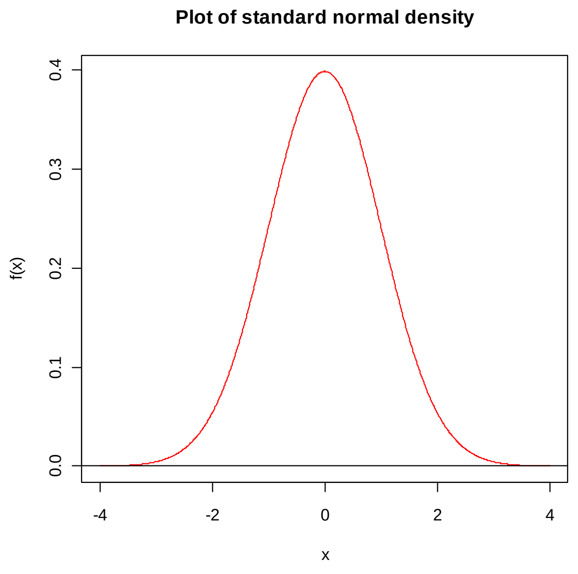


Notes:

- The mathematical equation given in the `expr` argument must vary over the letter x.
- By default, f(x) is evaluated at `n = 101` equally spaced x values in the range given. A larger value for n can produce a smoother curve.

Below is how the standard normal density is plotted:

```
> curve(expr = dnorm(x = x, mean = 0, sd = 1), from = -4,
    to = 4, n = 1000, main = "Plot of standard normal
    density", ylab = "f(x)", xlab = "x", col = "red")
> abline(h = 0) #horizontal line at 0
```

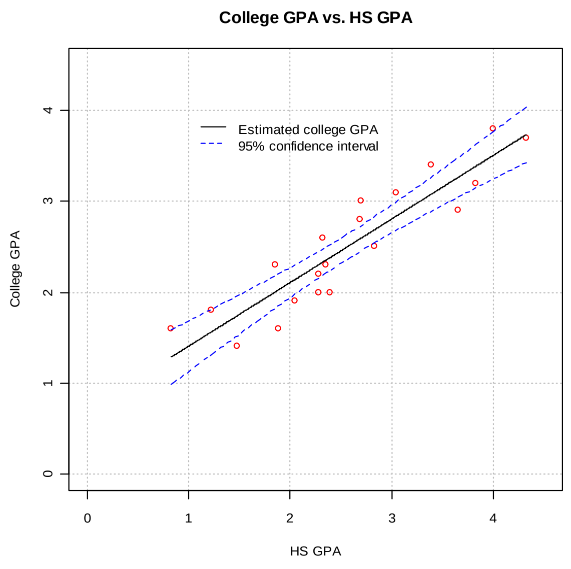


What is your favorite probability distribution for a continuous random variable? Plot the density with the curve function!

The `curve()` function is useful for adding curves to another plot. The code below shows how to add an

estimated regression model and confidence interval bands to a scatter plot with the gpa data:

```
> plot(x = gpa$HS.GPA, y = gpa$College.GPA, xlab = "HS
    GPA", ylab = "College GPA", main = "College GPA vs. HS
    GPA", xlim = c(0,4.5), ylim = c(0,4.5), col = "red",
    pch = 1, cex = 1.0, panel.first = grid(col = "gray",
    lty = "dotted"))
> curve(expr = predict(object = mod.fit, newdata =
    data.frame(HS.GPA = x)), from = min(gpa$HS.GPA),
    to = max(gpa$HS.GPA), add = TRUE, n = 1000)
> curve(expr = predict(object = mod.fit, newdata =
    data.frame(HS.GPA = x), se.fit = TRUE, interval =
    "confidence", level = 0.95)$fit[,2], from =
    min(gpa$HS.GPA), to = max(gpa$HS.GPA), add = TRUE,
    col = "red", lty = "dashed")
> curve(expr = predict(object = mod.fit, newdata =
    data.frame(HS.GPA = x), se.fit = TRUE, interval =
    "confidence", level = 0.95)$fit[,3], from =
    min(gpa$HS.GPA), to = max(gpa$HS.GPA), add = TRUE,
    col = "red", lty = "dashed")
> legend(x = 1, y = 4, legend = c("Estimated college GPA",
    "95% confidence interval"), lty = c("solid", "dashed"),
    col = c("black", "red"), bty = "n")
> #identify(x = gpa$HS.GPA, y = gpa$College.GPA)
```



Two new functions are included in the above code:

- `legend()` – The legend is placed at (x,y) = (1,4) on the plot. Alternatively, you can interactively specify the legend location with `locator(1)`:

```
> legend(locator(1), legend = c("Estimated college
    GPA", "95% confidence interval"), lty = c("solid",
    "dashed"), col = c("black", "red"), bty = "n")
```

After running the code, left-click on the location in the plot for the legend.

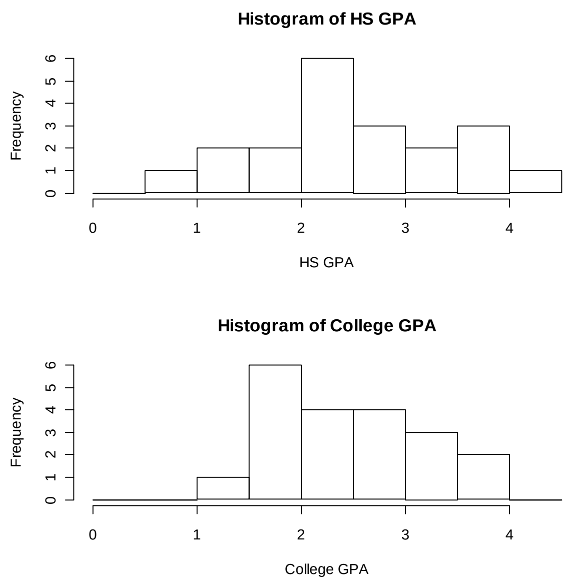
- `identify()` – This function is used to interactively label points on a plot. After running the uncommented code given above, left click on points in the plot, which are then identified with an observation number. To end identifying points, right click and select stop.

Question: How would you add the prediction interval bands to the plot?

Histograms

The `hist()` function plots histograms. The code below shows how to include two histograms in one R Graphics window:

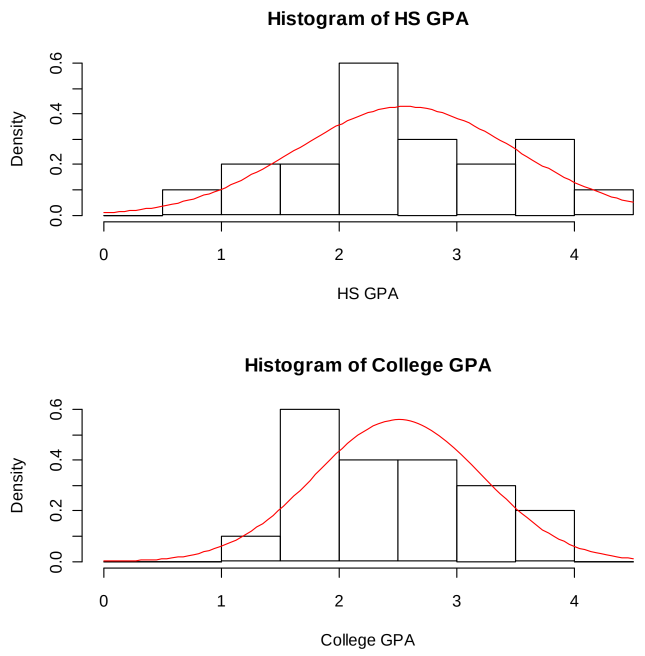
```
> par(mfrow = c(2,1)) #Two rows and one column of plots
> hist(x = gpa$HS.GPA, xlab = "HS GPA", main = "Histogram
    of HS GPA", breaks = c(0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5,
    4, 4.5))
> hist(x = gpa$College.GPA, xlab = "College GPA", main =
    "Histogram of College GPA", breaks = seq(from = 0, to =
    4.5, by = 0.5))
```



If you do not specify the `breaks` argument, R will choose the histogram classes for you. Usually, R's choice will work well. I chose the classes here to make sure that each histogram has the same classes. The use of both the `c()` function and the `seq()` function was done only for demonstration purposes.

We can combine the `hist()` function with the `curve()` function to produce a histogram with a probability density function overlay:

```
> hist(x = gpa$HS.GPA, xlab = "HS GPA", main = "Histogram
    of HS GPA", breaks = c(0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5,
    4, 4.5), freq = FALSE)
> curve(expr = dnorm(x = x, mean = mean(gpa$HS.GPA), sd =
    sd(gpa$HS.GPA)), col = "red", add = TRUE)
> hist(x = gpa$College.GPA, xlab = "College GPA", main =
    "Histogram of College GPA", breaks = seq(from = 0, to =
    4.5, by = 0.5), freq = FALSE)
> curve(expr = dnorm(x = x, mean = mean(gpa$College.GPA),
    sd = sd(gpa$College.GPA)), col = "red", add = TRUE)
```

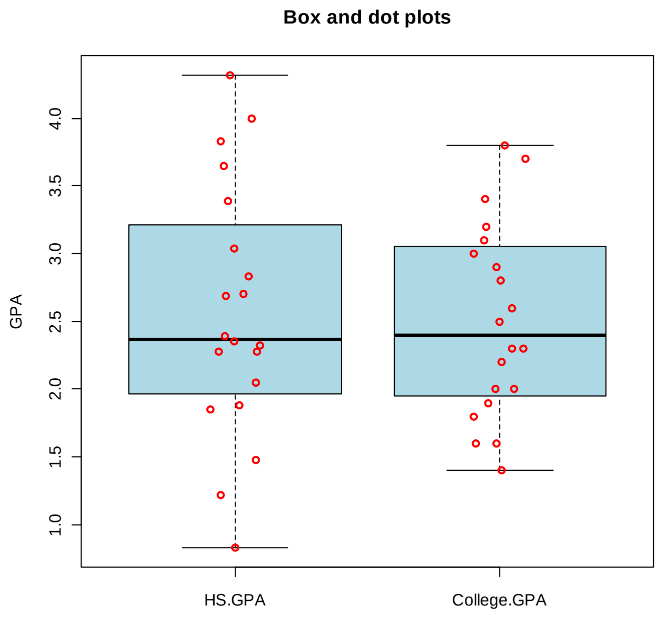


The `freq = FALSE` argument value in `hist()` leads to a rescaling of the y-axis for the histogram bars so that the density overlay can be performed.

Box and dot plots

Box plots are produced by `boxplot()`, and dot plots are produced by `stripchart()`:

```
> par(mfrow = c(1,1))
> boxplot(x = gpa, col = "lightblue", main = "Box and dot
    plots", ylab = "GPA",  xlab = "")
> stripchart(x = gpa, lwd = 2, col = "red", method =
    "jitter", vertical = TRUE, pch = 1, add = TRUE)
```



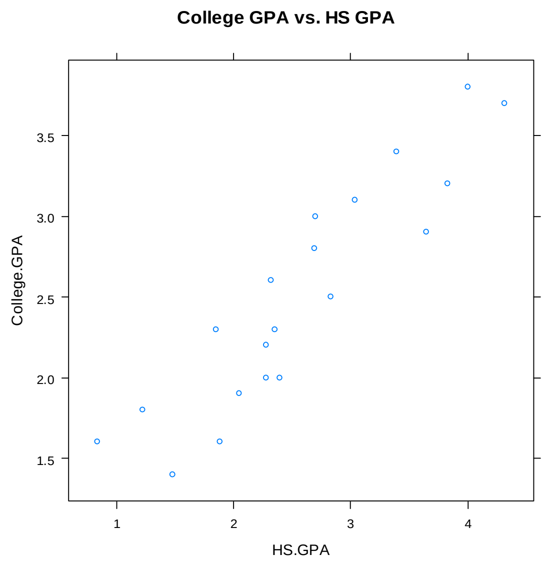
Please see the program for another way to create this plot when the data are organized as:

```
> head(HS.college)
  school  gpa
1     HS 3.04
2     HS 2.35
3     HS 2.70
4     HS 2.05
5     HS 2.83
6     HS 4.32

>   tail(HS.college)
    school gpa
35 College 1.8
36 College 1.4
37 College 2.0
38 College 3.8
39 College 2.2
40 College 1.6
```

lattice package

The lattice package produces many of the same plots as the graphics package. The package is installed by default within R, but you still need to run `library(package = lattice)` to make its functions available for use. Below is an example with the `xyplot()` function and the gpa data.frame.

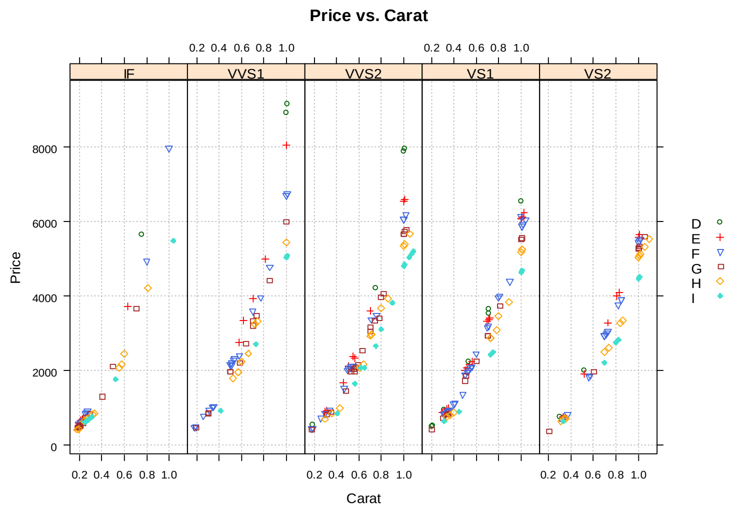
```
> library(package = lattice)
> xyplot(x = College.GPA ~ HS.GPA, data = gpa, main =
   "College GPA vs. HS GPA")
```



An advantage of the lattice package is that co-plots (often referred to as Trellis graphics) can be produced. These plots allow you to plot multivariate data by conditioning on variable values. For example, below is how I produced a scatter plot of diamond prices versus carat size, where I condition on diamond color and use plotting points corresponding to the diamond clarity.

```
> library(package = RODBC)
> z<-odbcConnectExcel(xls.file = "C:\\chris\\unl\\Dropbox\\
   NEW\\workshop\\Gallup\\diamond.xls")
> diamond<-sqlFetch(channel = z, sqtable = "Set1")
> close(z)

> #Change order of the levels of clarity
> diamond$clarity<-factor(x = diamond$clarity, levels =
   c("IF", "VVS1", "VVS2", "VS1", "VS2"))
> levels(x = diamond$clarity)
[1] "IF"   "VVS1" "VVS2" "VS1"  "VS2"

> head(diamond)
  carat color clarity    price
1  0.30     D     VS2 745.9184
2  0.30     E     VS1 865.0820
3  0.30     G    VVS1 865.0820
4  0.30     G     VS1 721.8565
5  0.31     D     VS1 940.1322
6  0.31     E     VS1 890.8626

> library(package = lattice)
> trellis.device(theme = "col.whitebg")

> win.graph(width = 10, height = 7, pointsize = 12)
> xyplot(x = price ~ carat | clarity, data = diamond,
   layout = c(5,1), groups = color, main = "Price vs.
   Carat", auto.key = list(points = TRUE, space =
   "right"), xlab = "Carat", ylab = "Price", panel =
    function(x, y,  ...)
```

```
      { panel.grid(h = -1, v = -1, col = "grey", lwd = 1,
          lty = "dotted")
        panel.xyplot(x, y,  ...)
      }
   )
```

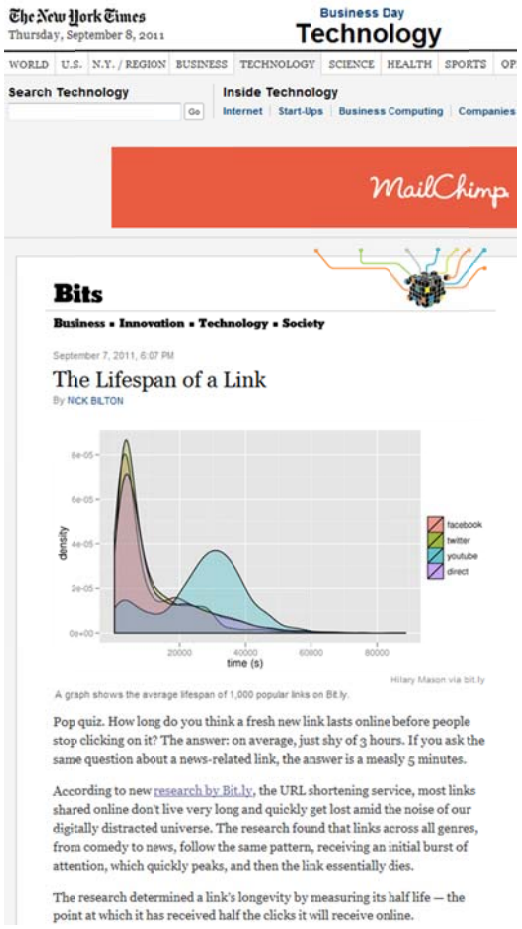


The `layout` argument gives the number of columns for the plot and then the number of rows, which is a different order from how one normally specifies the dimension of a matrix.

A disadvantage of the lattice package is that the code can be less readable to a new R user, especially for more complicated plots.

ggplot2 package

The ggplot2 package is a much newer package for plotting. It currently is at version 0.8.9. I have very little experience with the package, but it seems to be attracting new users. There is even a book on the package (ggplot2: Elegant Graphics for Data Analysis) written by its author. A recent plot created by the package appeared in the New York Times (http://bits.blogs.nytimes.com/2011/09/07/the-lifespan-of-a-link):



The New York Times
Thursday, September 8, 2011

Business Day
Technology

WORLD | U.S. | N.Y. / REGION | BUSINESS | TECHNOLOGY | SCIENCE | HEALTH | SPORTS | OP

Search Technology [Go]

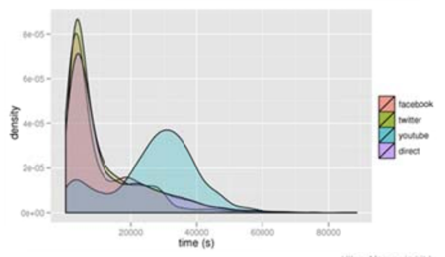
Inside Technology
Internet | Start-Ups | Business Computing | Companies

Bits
Business ▪ Innovation ▪ Technology ▪ Society

September 7, 2011, 6:07 PM

The Lifespan of a Link

By NICK BILTON



Hilary Mason via bit.ly

A graph shows the average lifespan of 1,000 popular links on Bit.ly.

Pop quiz. How long do you think a fresh new link lasts online before people stop clicking on it? The answer: on average, just shy of 3 hours. If you ask the same question about a news-related link, the answer is a measly 5 minutes.

According to new research by Bit.ly, the URL shortening service, most links shared online don't live very long and quickly get lost amid the noise of our digitally distracted universe. The research found that links across all genres, from comedy to news, follow the same pattern, receiving an initial burst of attention, which quickly peaks, and then the link essentially dies.

The research determined a link's longevity by measuring its half life — the point at which it has received half the clicks it will receive online.

Resources

R graphics gallery: http://addictedtor.free.fr/graphiques/

R Graphics (2nd edition) book and corresponding website: http://www.stat.auckland.ac.nz/~paul/RG2e

V. Logistic regression

Bilder and Loughin (*Chance*, 1998) estimated the probability of success for an NFL placekick through a logistic regression model. Their final model was

$$\text{logit}(\hat{\pi}) = 4.4984 - 0.3306\text{change} + 1.2592\text{pat} + 2.8778\text{wind} - 0.0807\text{distance} - 0.0907\text{distance}\times\text{wind}$$

where

- change is a 1 for a "lead-change" placekick and 0 otherwise
- pat is a 1 for a point-after-touchdown and 0 for a field goal
- wind is a 1 for "windy" conditions (>15 MPH at kickoff) and 0 otherwise.
- distance is the distance of the placekick in yards

The corresponding code for this example is in the placekick.R file.

Reading in data

The data is in the comma delimited file placekick.csv. The response variable is named “good” where a 1 is a success and a 0 is a failure.

```
> placekick<-read.csv("C:\\chris\\unl\\Dropbox\\NEW\\
   workshop\\Gallup\\placekick.csv")
> head(placekick)
  distance change pat wind good
1       21      1   0    0    1
2       21      0   0    0    1
3       20      0   1    0    1
4       28      0   0    0    1
5       20      0   1    0    1
6       25      0   0    0    1
```

Each observation can be viewed as a Bernoulli trial.

Fitting the model

The `glm()` function fits generalized linear models. The family argument within the function specifies the type of generalized linear model. Below is the code used to fit the model:

```
> mod.fit<-glm(formula = good ~ change + distance + pat +
    wind + distance:wind, data = placekick, family =
    binomial(link = logit))
> summary(mod.fit)

Call:
glm(formula = good ~ change + pat + wind + distance +
distance:wind, family = binomial(link = logit), data =
placekick)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.8839   0.1775   0.1775   0.4679   1.7098

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept)    4.49835    0.48163   9.340  < 2e-16 ***
change        -0.33056    0.19444  -1.700  0.08913 .
pat            1.25916    0.38707   3.253  0.00114 **
wind           2.87783    1.78593   1.611  0.10709
distance      -0.08074    0.01143  -7.064 1.62e-12 ***
wind:distance -0.09074    0.04569  -1.986  0.04701 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘
’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 1013.43  on 1424  degrees of freedom
Residual deviance:  751.27  on 1419  degrees of freedom
AIC: 763.27
```

```
Number of Fisher Scoring iterations: 6

> names(mod.fit)
 [1] "coefficients"   "residuals"     "fitted.values"
 [4] "effects"        "R"             "rank"
 [7] "qr"             "family"        "linear.predictors"
[10] "deviance"       "aic"           "null.deviance"
[13] "iter"           "weights"       "prior.weights"
[16] "df.residual"    "df.null"       "y"
[19] "converged"      "boundary"      "model"
[22] "call"           "formula"       "terms"
[25] "data"           "offset"        "control"
[28] "method"         "contrasts"     "xlevels"
```

Notes:

- The `formula` and `data` arguments are in the same format as for the `lm()` function.
- To include the interaction between distance and wind, I used `distance:wind`. Alternatively, I could have also used

  ```
  formula = good ~ change + pat + distance*wind
  ```

  or

  ```
  formula = good ~ change + pat + (distance + wind)^2
  ```

- The `names()` function shows what is within `mod.fit`. For example, `mod.fit$coefficients` gives the parameter estimates in a vector.

Data used for logistic regression often comes in a binomial form. For example, there are 7 successes out of 8 trials when change = 1, pat = 0, wind = 0, and distance = 20. Below is how you can convert the data to a binomial format and then estimate the model:

```
> set1<-aggregate(formula = good ~ change + pat + wind +
    distance, data = placekick, FUN = sum)
> head(set1)
  change pat wind distance good
1      0   0    0       18    1
2      1   0    0       18    1
3      0   0    0       19    3
4      1   0    0       19    4
5      0   0    0       20   15
6      1   0    0       20    7

> set2<-aggregate(formula = good ~ change + pat + wind +
    distance, data = placekick, FUN = length)
> head(set2)
  change pat wind distance good
1      0   0    0       18    1
2      1   0    0       18    2
3      0   0    0       19    3
4      1   0    0       19    4
5      0   0    0       20   15
6      1   0    0       20    8

> placekick.bin<-data.frame(set1[,-5], success = set1$good,
    trials = set2$good, proportion = round(set1$good /
   set2$good, 4))
> head(placekick.bin)
  change pat wind distance success trials proportion
1      0   0    0       18       1      1      1.000
2      1   0    0       18       1      2      0.500
3      0   0    0       19       3      3      1.000
4      1   0    0       19       4      4      1.000
5      0   0    0       20      15     15      1.000
6      1   0    0       20       7      8      0.875
```

```
> #Estimate the model with the binomial form of the data
> mod.fit.bin<-glm(formula = success/trials ~ change + pat
    + wind + distance + distance:wind, data =
    placekick.bin, weight = trials, family = binomial(link
    = logit))
>  summary(mod.fit.bin)

Call:
glm(formula = success/trials ~ change + pat + wind +
  distance + distance:wind, family = binomial(link =
  logit), data = placekick.bin, weights = trials)

Deviance  Residuals:
    Min       1Q    Median       3Q       Max
-2.2386  -0.5836    0.1965   0.8736    2.2822

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept)    4.49835    0.48163   9.340  < 2e-16 ***
change        -0.33056    0.19445  -1.700  0.08914 .
pat            1.25916    0.38714   3.252  0.00114 **
wind           2.87783    1.78643   1.611  0.10719
distance      -0.08074    0.01143  -7.064 1.62e-12 ***
wind:distance -0.09074    0.04570  -1.986  0.04706 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘
’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 376.01  on 118  degrees of freedom
Residual deviance: 113.86  on 113  degrees of freedom
AIC: 260.69

Number of Fisher Scoring iterations: 5
```

The estimated model is the same as before. Note how the response was specified in the `formula` argument and how the number of trials was specified in the `weight` argument.

Estimating the response

The estimated model can be also written as

$$\hat{\pi} = \frac{\exp(4.50 - 0.33\text{change} - 0.08\text{distance} - 126\text{pat} + 2.88\text{wind} - 0.09\text{distance} \times \text{wind})}{1 + \exp(4.50 - 0.33\text{change} - 0.08\text{distance} - 126\text{pat} + 2.88\text{wind} - 0.09\text{distance} \times \text{wind})}$$

Using this form, we can estimate the probability of success for a placekick with change = 1, pat = 0, wind = 0, and distance = 30. Below are a few different ways to perform these calculations in R:

```
> beta.hat<-mod.fit.bin$coefficients
> beta.hat
  (Intercept)       change          pat         wind     distance
   4.49835224  -0.33055778   1.25916109   2.87783050  -0.08073996
wind:distance
  -0.09074258
> exp(beta.hat[1] + beta.hat[2]*1 + beta.hat[5]*30) /
    (1 + exp(beta.hat[1] + beta.hat[2]*1 + beta.hat[5]*30))
(Intercept)
  0.8513964
> plogis(q = beta.hat[1] + beta.hat[2]*1 + beta.hat[5]*30)
(Intercept)
  0.8513964
> as.numeric(plogis(q = beta.hat[1] + beta.hat[2]*1 +
    beta.hat[5]*30)) #Removes label
[1] 0.8513964

> predict(object = mod.fit.bin, newdata =
    data.frame(change = 1, pat = 0, wind = 0, distance =
    30), type = "response")
        1
0.8513964

> save.lp<-predict(object = mod.fit.bin, newdata =
    data.frame(change = 1, pat = 0, wind = 0, distance =
    30), type = "link")
```

```
> save.lp
       1
1.745596
> plogis(q = save.lp)
        1
0.8513964
```

Notes:

- The `plogis()` function evaluates the cumulative distribution function for a logistic random variable with location parameter = 0 and scale parameter = 1.
- `as.numeric()` removes unnecessary labels left over from beta.hat.
- The `predict()` function works in the same manner as with simple linear regression. The `type` argument is new, and this specifies the estimation for π (`type = “response”`) or logit(π) (`type = “link”`).

A 95% confidence interval for π is

$$\frac{e^{\hat{\beta}_0+\hat{\beta}_1x_1+\cdots+\hat{\beta}_px_p \pm Z_{1-\alpha/2}\sqrt{\widehat{\text{Var}}(\hat{\beta}_0+\hat{\beta}_1x_1+\cdots+\hat{\beta}_px_p)}}}{1+e^{\hat{\beta}_0+\hat{\beta}_1x_1+\cdots+\hat{\beta}_px_p \pm Z_{1-\alpha/2}\sqrt{\widehat{\text{Var}}(\hat{\beta}_0+\hat{\beta}_1x_1+\cdots+\hat{\beta}_px_p)}}}$$

R does not calculate this interval with the `predict()` function. To find the interval, you need to calculate an interval for the linear predictor $\beta_0 + \beta_1x_1 + \cdots + \beta_px_p$ (i.e., logit(π)) first. The exponential function is then used to find the interval for π:

```
> save.lp<-predict(object = mod.fit.bin, newdata =
    data.frame(change = 1, pat = 0, wind = 0, distance =
    30), type = "link", se = TRUE)
> save.lp
$fit
       1
1.745596

$se.fit
[1] 0.1895555

$residual.scale
[1] 1

> alpha<-0.05
> lower.lp<-save.lp$fit-qnorm(p = 1-alpha/2)*save.lp$se.fit
> upper.lp<-save.lp$fit+qnorm(p = 1-alpha/2)*save.lp$se.fit
> lower.pi<-plogis(q = lower.lp)
> upper.pi<-plogis(q = upper.lp)
> data.frame(lower.pi, upper.pi)
   lower.pi  upper.pi
1 0.7980375 0.8925558
```

Below is my function for these calculations:

```
> ci.pi<-function(newdata, mod.fit.obj, alpha){
    save.lp<-predict(object = mod.fit.obj, newdata =
      newdata, type = "link", se = TRUE)
    lower.lp<-save.lp$fit-qnorm(1-alpha/2)*save.lp$se.fit
    upper.lp<-save.lp$fit+qnorm(1-alpha/2)*save.lp$se.fit
    lower.pi<-plogis(q = lower.lp)
    upper.pi<-plogis(q = upper.lp)
    list(pi.hat = plogis(save.lp$fit), lower = lower.pi,
      upper = upper.pi)
  }

> ci.pi(newdata = data.frame(change = 1, pat = 0, wind = 0,
    distance = c(30, 40)), mod.fit.obj = mod.fit.bin, alpha
    = 0.05)
$pi.hat
```

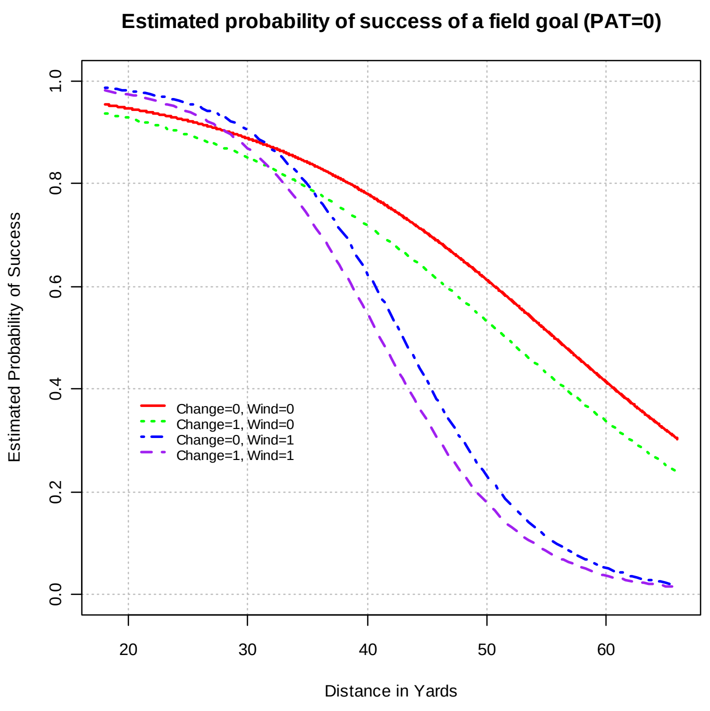
```
        1         2
0.8513964 0.7187352

$lower
        1         2
0.7980375 0.6541806

$upper
        1         2
0.8925558 0.7753770
```

Notice the use of the `list()` function at the end of the function.

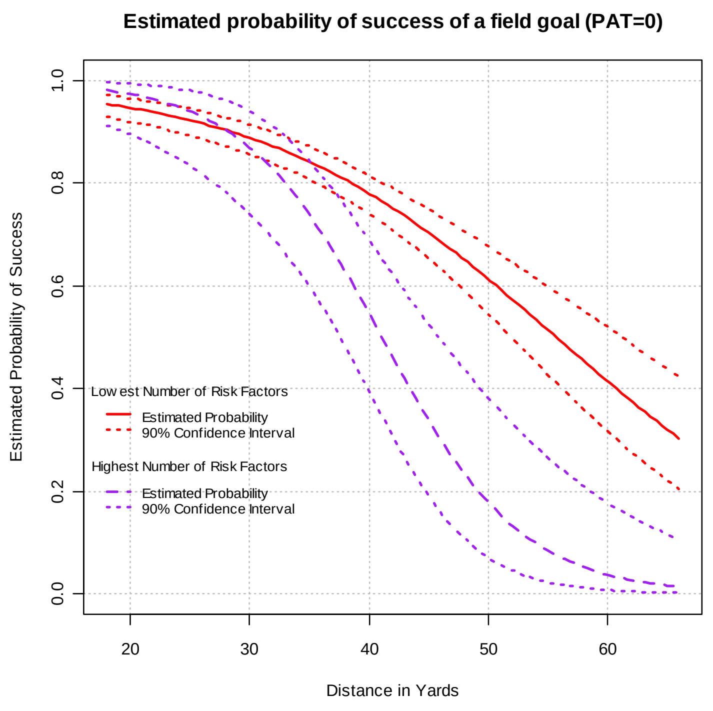
Because there is only one "continuous" independent variable, we can construct a plot of the estimated probability of success versus the distance:

```
> #Dummy plot of the estimated proportion of success at
     each distance
> plot(x = placekick.bin$distance, y =
    placekick.bin$proportion, xlab="Distance in Yards",
    ylab="Estimated Probability of Success", type="n",
    panel.first=grid(col = "gray", lty = "dotted"), main =
    "Estimated probability of success of a field goal
    (PAT=0)")

> #Put estimated logistic regression model on the plot –
    change=0, wind=0
> curve(expr = plogis(beta.hat[1] + beta.hat[5]*x), lwd=2,
    col = "red", add = TRUE, n = 1000)
> #Another way to do the same curve as above
> #curve(expr = predict(object = mod.fit.bin, newdata =
     data.frame(change = 1, pat = 0, wind = 0, distance =
     x), type = "response"), lwd=2, col = "red", add =
     TRUE, n = 1000)

> #Put estimated logistic regression model on the plot –
```

```
    change=1, wind=0
> curve(expr = plogis(beta.hat[1] + beta.hat[2]*1 +
    beta.hat[5]*x), lty=3, lwd=2, col = "green", add =
    TRUE)

> #Put estimated logistic regression model on the plot –
    change=0, wind=1
> curve(expr = plogis(beta.hat[1] + beta.hat[4]*1 +
    beta.hat[5]*x + beta.hat[6]*1*x), lty=4, lwd=2, col =
    "blue", add = TRUE)

> #Put estimated logistic regression model on the plot –
    change=1, wind=1
> curve(expr = plogis(beta.hat[1] + beta.hat[2]*1 +
    beta.hat[4]*1 + beta.hat[5]*x + beta.hat[6]*1*x),
    lty=2, lwd=2, col = "purple", add = TRUE)

> names1<-c("Change=0, Wind=0", "Change=1, Wind=0",
    "Change=0, Wind=1", "Change=1, Wind=1")
> legend(locator(1), legend = names1, lty = c(1,3,4,2), col
    = c("red","green","blue","purple"), bty="n", cex=0.75,
    lwd=2)
```



The placekick.R program contains the code for this plot:



Object oriented language

Objects resulting from `glm()` have the following classes:

```
> class(mod.fit.bin)
[1] "glm" "lm"
```

When using generic functions, R looks for a method function corresponding to the glm class. If a function does not exist, R looks for a method function corresponding to the lm class.

Below are the method functions for glm class objects:

```
 > methods(class = glm)
 [1] add1.glm*           anova.glm          confint.glm*
 [4] cooks.distance.glm* deviance.glm*      drop1.glm*
 [7] effects.glm*        extractAIC.glm*    family.glm*
[10] formula.glm*        influence.glm*     logLik.glm*
[13] model.frame.glm     nobs.glm*          predict.glm
[16] print.glm           residuals.glm      rstandard.glm
[19] rstudent.glm        summary.glm        vcov.glm*
[22] weights.glm*

   Non-visible functions are asterisked
```

What do the following generic functions calculate?

- `vcov()`
- `anova()`
- `confint()`
- `deviance()`

The car package gives some useful additions to these method functions. For example, below are the results from the `Anova()` function:

```
> library(package = car)
> Anova(mod = mod.fit)
Analysis of Deviance Table (Type II tests)

Response: good
              LR Chisq Df Pr(>Chisq)
change           2.863  1  0.0906281 .
pat             11.224  1  0.0008074 ***
wind             2.646  1  0.1038115
distance        73.185  1  < 2.2e-16 ***
wind:distance    5.415  1  0.0199610 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘
’ 1
```

This function performs likelihood ratio tests to determine the importance of an independent variable given all of the other variables are in the model.

The `anova()` function (stats package) can be used in a similar manner to test a full model vs. a reduced model:

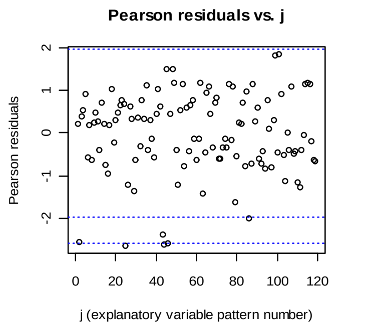
```
> mod.fit.bin.reduced<-glm(formula = success/trials ~
    change + pat, data = placekick.bin, weight = trials,
    family = binomial(link = logit))
> anova(mod.fit.bin.reduced, mod.fit.bin, test = "Chisq")
Analysis of Deviance Table

Model 1: success/trials ~ change + pat
Model 2: success/trials ~ change + pat + wind + distance +
distance:wind
  Resid. Df Resid. Dev Df Deviance P(>|Chi|)
1       116     194.33
2       113     113.86  3   80.475 < 2.2e-16 ***
```

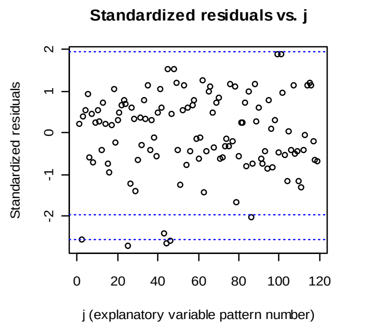
```
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘
’ 1
```

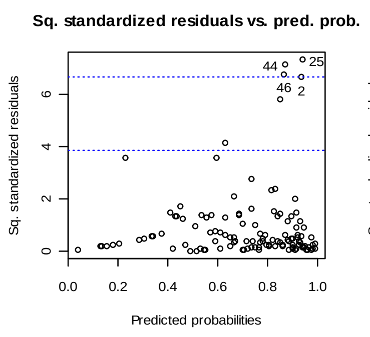
Writing your own functions

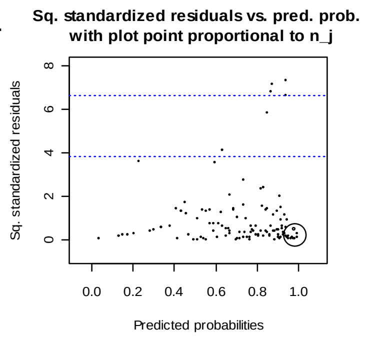
I created the next function for a categorical data analysis course. The function automates the process of examining diagnostic tools for a logistic regression model. You can see its code in the file examine.model.logistic.reg.R. This code can be run as before or the `source()` function can be used to run it. Below is an example:

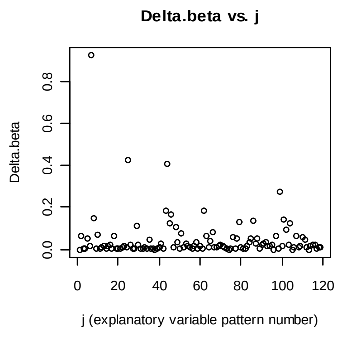
```
> #The examine.model() function is in this program:
> source("C:\\chris\\unl\\Dropbox\\NEW\\workshop\\
    Gallup\\examine.model.logistic.reg.R")

> examine.model(mod.fit.obj = mod.fit.bin)
The Pearson statistic is 104.8678 with p-value = 0.6949
The G^2 is 113.858 with p-value = 0.4597

>   names(save.it)
[1] "h"              "pearson"        "sq.stand.resid"
    "delta.beta"     "pear.stat"
[6] "dev"
```

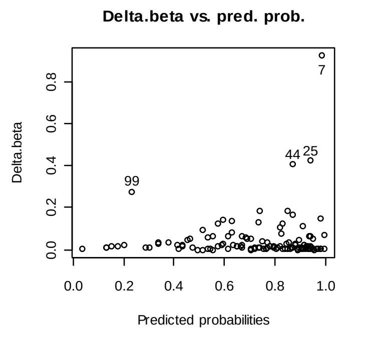


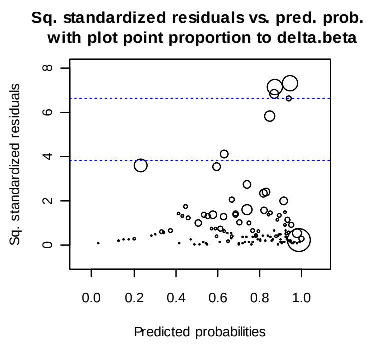












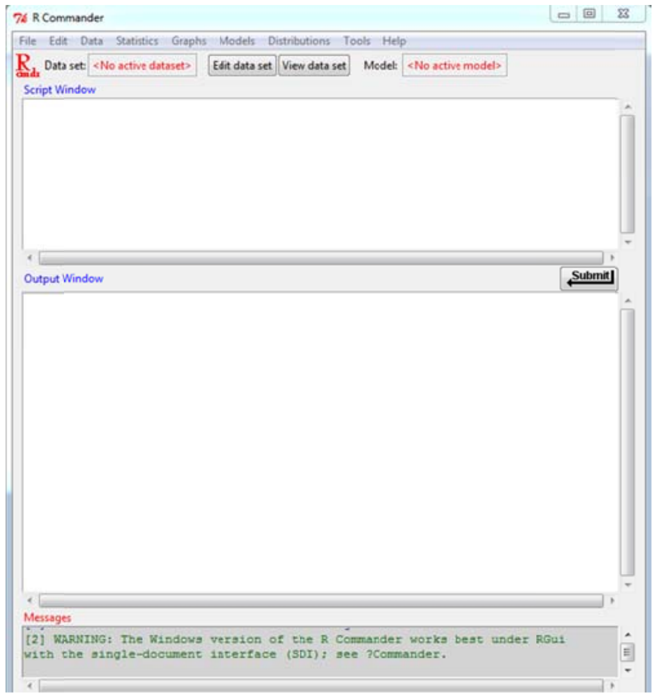
$X^2 = 104.87$ (0.6949), $G^2 = 113.86$ (0.4597)

VI. Additional topics

R Commander

Are there any point-and-click ways to produce plots or output? Yes, the Rcmdr (short for "R Commander") package can for many statistical methods. This package does not come with the initial installation of R, so you will need to install it. Once the package is installed, use `library(package = Rcmdr)` to start it. The first time that you run this code, R will ask if you want to install a number of other packages. Select yes, but note that this may take some time to install all of them.

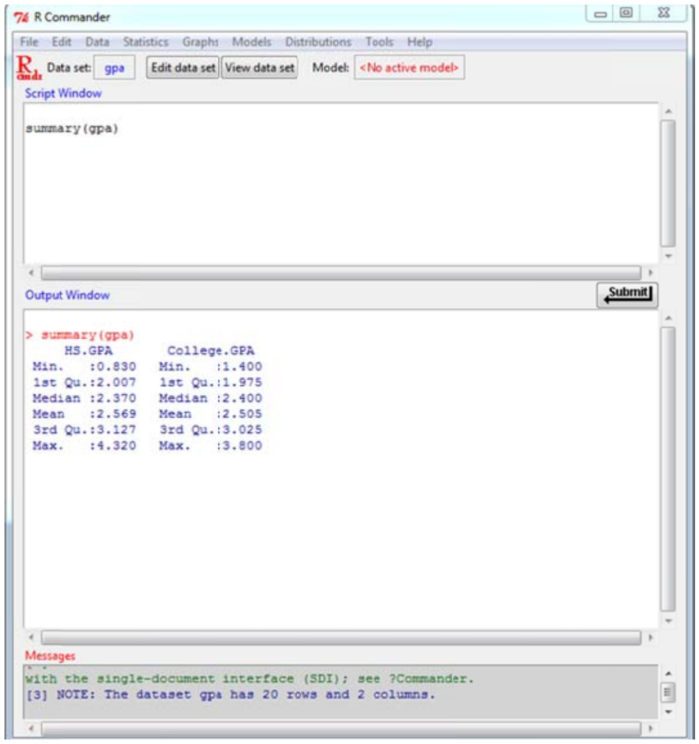
Below is the R Commander window:



```
[2] WARNING: The Windows version of the R Commander works best under RGui
with the single-document interface (SDI); see ?Commander.
```

To begin using R Commander, specify the data set of interest. Because gpa was created earlier, I chose this data set by selecting DATA > ACTIVE DATA SET > SELECT ACTIVE DATA SET. Now, "Data set: gpa" is shown toward the top of the R Commander window.

One of the nice things about R Commander is that it can help with learning R code. For example, select STATISTICS > SUMMARIES > ACTIVE DATA SET to find summary statistics:



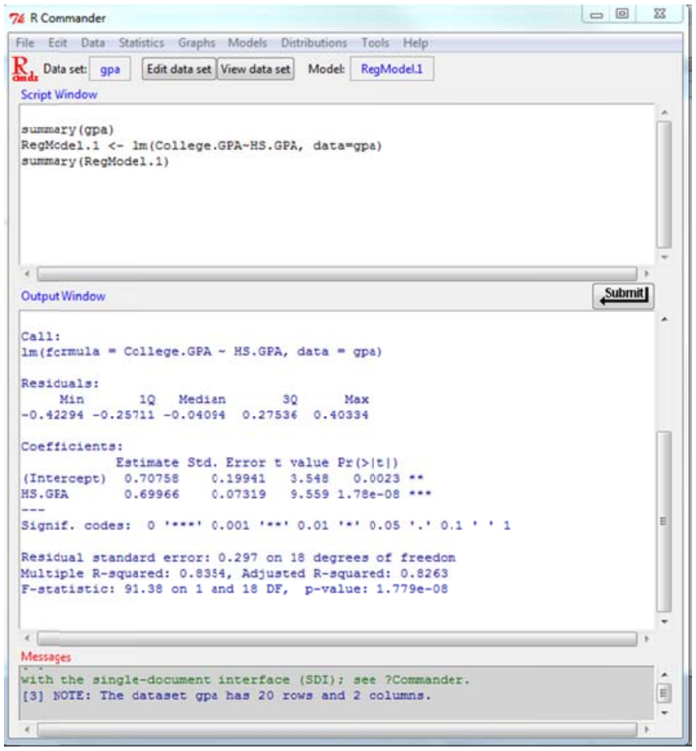
```
summary(gpa)
```

```
> summary(gpa)
     HS.GPA       College.GPA   
 Min.   :0.830   Min.   :1.400  
 1st Qu.:2.007   1st Qu.:1.975  
 Median :2.370   Median :2.400  
 Mean   :2.569   Mean   :2.505  
 3rd Qu.:3.127   3rd Qu.:3.025  
 Max.   :4.320   Max.   :3.800  
```

```
with the single-document interface (SDI); see ?Commander.
[3] NOTE: The dataset gpa has 20 rows and 2 columns.
```

The Script Window logs the R code that performs the calculations. We see here that the `summary()` function is used just like we did earlier with this data set. To save this code, select FILE > SAVE SCRIPT.

To estimate the simple linear regression model, select STATISTICS > FIT MODEL > LINEAR REGRESSION. Next, choose the response and explanatory variables and select OK when completed:



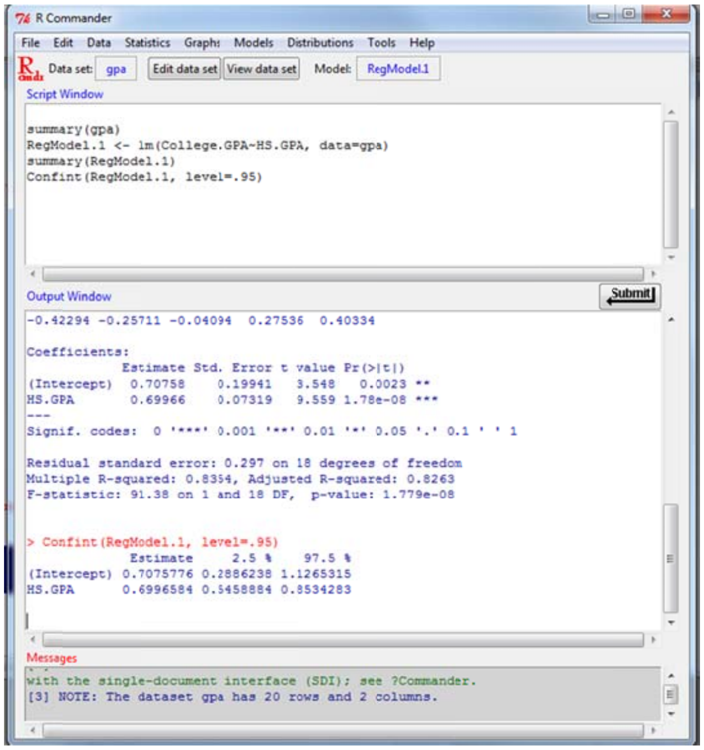
```
summary(gpa)
RegModel.1 <- lm(College.GPA~HS.GPA, data=gpa)
summary(RegModel.1)
```

```
Call:
lm(formula = College.GPA ~ HS.GPA, data = gpa)

Residuals:
     Min       1Q   Median       3Q      Max 
-0.42294 -0.25711 -0.04094  0.27536  0.40334 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  0.70758    0.19941   3.548   0.0023 ** 
HS.GPA       0.69966    0.07319   9.559 1.78e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.297 on 18 degrees of freedom
Multiple R-squared: 0.8354, Adjusted R-squared: 0.8263 
F-statistic: 91.38 on 1 and 18 DF,  p-value: 1.779e-08 
```

```
with the single-document interface (SDI); see ?Commander.
[3] NOTE: The dataset gpa has 20 rows and 2 columns.
```

The author of the package, John Fox, has created many of his own functions to perform calculations. This may lead to functions being given in the Script Window that are different from what we have used before. For example, we can create confidence intervals for the

model parameters by selecting MODELS > CONFIDENCE INTERVALS and then OK. The `Confint()` function is used rather than the `confint()` function:



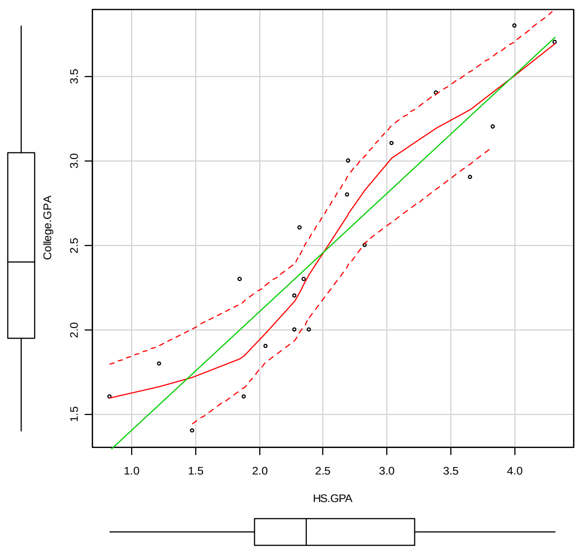
```
summary(gpa)
RegModel.1 <- lm(College.GPA~HS.GPA, data=gpa)
summary(RegModel.1)
Confint(RegModel.1, level=.95)
```

```
-0.42294 -0.25711 -0.04094  0.27536  0.40334 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  0.70758    0.19941   3.548   0.0023 ** 
HS.GPA       0.69966    0.07319   9.559 1.78e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.297 on 18 degrees of freedom
Multiple R-squared: 0.8354, Adjusted R-squared: 0.8263 
F-statistic: 91.38 on 1 and 18 DF,  p-value: 1.779e-08 


> Confint(RegModel.1, level=.95)
             Estimate     2.5 %    97.5 %
(Intercept) 0.7075776 0.2886238 1.1265315
HS.GPA      0.6996584 0.5458884 0.8534283
```

```
with the single-document interface (SDI); see ?Commander.
[3] NOTE: The dataset gpa has 20 rows and 2 columns.
```

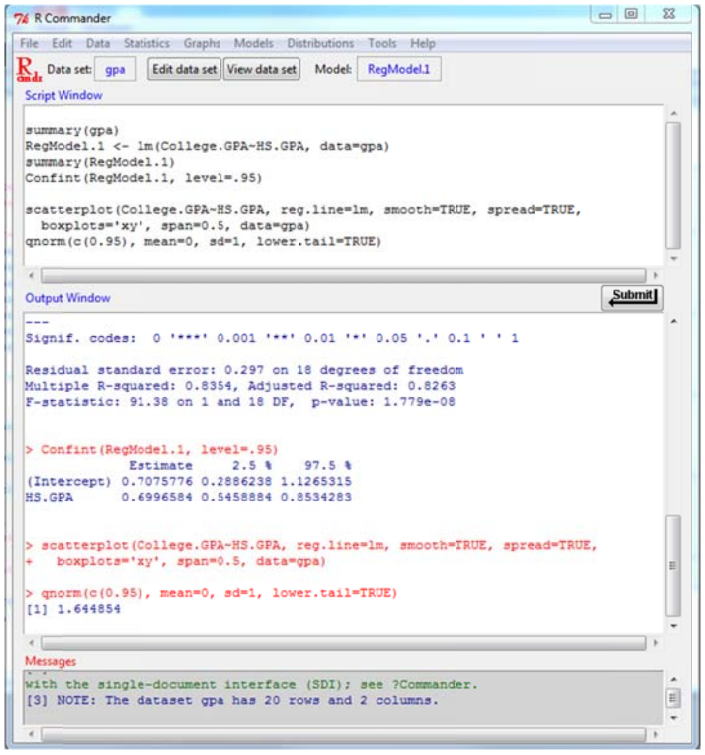
To construct a scatter plot, select GRAPHS > SCATTER PLOT. After selecting what goes on the x and y-axis and using the defaults, R Commander produces the following code and plot.

```
scatterplot(College.GPA~HS.GPA, reg.line=lm, smooth=TRUE,
   spread=TRUE, boxplots='xy', span=0.5, data=gpa)
```



R Commander also provides a quick way to find quantiles and probabilities for particular distributions. For

example, the 0.95 quantile from a standard normal is calculated by selecting DISTRIBUTIONS > CONTINUOUS DISTRIBUTIONS > NORMAL DISTRIBUTION > NORMAL QUANTILES. Type in 0.95 in the probabilities box and then select OK:



```
summary(gpa)
RegModel.1 <- lm(College.GPA~HS.GPA, data=gpa)
summary(RegModel.1)
Confint(RegModel.1, level=.95)

scatterplot(College.GPA~HS.GPA, reg.line=lm, smooth=TRUE, spread=TRUE,
  boxplots='xy', span=0.5, data=gpa)
qnorm(c(0.95), mean=0, sd=1, lower.tail=TRUE)
```

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.297 on 18 degrees of freedom
Multiple R-squared: 0.8354, Adjusted R-squared: 0.8263
F-statistic: 91.38 on 1 and 18 DF,  p-value: 1.779e-08


> Confint(RegModel.1, level=.95)
             Estimate     2.5 %    97.5 %
(Intercept) 0.7075776 0.2886238 1.1265315
HS.GPA      0.6996584 0.5458884 0.8534283


> scatterplot(College.GPA~HS.GPA, reg.line=lm, smooth=TRUE, spread=TRUE,
+   boxplots='xy', span=0.5, data=gpa)

> qnorm(c(0.95), mean=0, sd=1, lower.tail=TRUE)
[1] 1.644854
```

```
with the single-document interface (SDI); see ?Commander.
[3] NOTE: The dataset gpa has 20 rows and 2 columns.
```

Explore the menus on your own to examine the resources available! Note that HELP > INTRODUCTION TO THE R COMMANDER within R Commander opens a PDF file giving an introduction to the package.

Contingency tables

Contingency tables in R are created by using the `array()` function rather using a data.frame. The next example shows how this is done with a 2$\times$2 contingency table. The Bird.R program contains the code.

A question of interest for many basketball fans is whether or not the outcome for a second free throw attempt is dependent on the outcome for the first attempt. Below is a contingency table summarizing Larry Bird’s first and second free throw attempts during the 1980-1 and 1981-2 NBA seasons (Wardrop, 1995):

		Second		
		Made	Missed	Total
First	Made	251	34	285
	Missed	48	5	53
	Total	299	39	338

Below is how to create the contingency table:

```
> n.table<-array(data = c(251, 48, 34, 5), dim = c(2,2),
    dimnames = list(First = c("made", "missed"), Second =
    c("made", "missed")))
> n.table
        Second
First     made missed
  made     251     34
  missed    48      5

> class(n.table)
```

```
[1] "matrix"
> n.table[1,1]
[1] 251
> n.table[1,]
  made missed
   251     34

> #Estimated odds ratio
> theta.hat<-n.table[1,1]*n.table[2,2] /
    (n.table[1,2]*n.table[2,1])
> theta.hat
[1] 0.7689951
> 1/theta.hat
[1] 1.300398
```

Notes:

- Counts are entered in the `data` argument by columns within the contingency table.
- To name the rows and columns, the `dimnames()` function is used with the `list()` function.
- Parts of the contingency table can be accessed in the same manner as with a data.frame. This enables the calculation of quantities like an odds ratio.

The Pearson chi-square test for independence is performed using the `chisq.test()` function:

```
> ind.test<-chisq.test(n.table, correct = FALSE)
> ind.test

        Pearson's Chi-squared test

data:  n.table
X-squared = 0.2727, df = 1, p-value = 0.6015

> names(ind.test)
```

```
[1] "statistic" "parameter" "p.value"   "method"
    "data.name" "observed"
[7] "expected"  "residuals" "stdres"

> #just p-value
> ind.test$p.value
[1] 0.6015021

> #Exact test
> chisq.test(n.table, correct = FALSE, simulate.p.value =
    TRUE, B = 1000)

    Pearson's Chi-squared test with simulated p-value
      (based on 1000 replicates)

data:  n.table
X-squared = 0.2727, df = NA, p-value = 0.6843
```

Notes:

- The `correct = FALSE` argument prevents the Yates continuity correction from being applied.
- The results from `chisq.test()` are given in a list.
- An exact form of the test is performed through specifying `simulate.p.value = TRUE` where the number of permutations taken is specified in the `B` argument.

Below is the code to obtain a data.frame form of the data:

```
> bird.df<-as.data.frame(as.table(n.table))
> bird.df
   First Second Freq
1   made   made  251
2 missed   made   48
3   made missed   34
```

```
4 missed missed    5
```

To fit a loglinear model to the data, we can use the `glm()` function:

```
> mod.fit<-glm(formula = Freq ~ First + Second, data =
    bird.df, family = poisson(link = log))
> summary(mod.fit)

Call:
glm(formula = Freq ~ First + Second, family = poisson(link
= log), data = bird.df)

Deviance Residuals:
      1        2        3        4
-0.0703   0.1623   0.1934  -0.4659

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)   5.52989    0.06241   88.61   <2e-16 ***
Firstmissed  -1.68220    0.14959  -11.25   <2e-16 ***
Secondmissed -2.03688    0.17025  -11.96   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘
’ 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 402.05553  on 3  degrees of freedom
Residual deviance:   0.28575  on 1  degrees of freedom
AIC: 28.212

Number of Fisher Scoring iterations: 3

> #LRT for independence
> 1 - pchisq(q = mod.fit$deviance, df =
    mod.fit$df.residual)
[1] 0.5929559
```

What if the data was not in a contingency table format already? For example, it may be in the form:

	First	Second
1	Made	Made
2	Made	Missed
⋮		
338	Made	Made

The `table()` and `xtabs()` functions calculate the necessary counts for the contingency table (i.e., perform a “crosstab”). Alternatively, the `chisq.test()` works directly with the data in this format too. Please see the program for the code.

More data management

Below are some useful functions demonstrated by small examples. The data_management.R program contains the code.

1)`cbind()` – combine data by columns

```
> x<-1:10
> y<-c(rep(x = 1, times = 5), rep(x = 6, times = 5))
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> y
 [1] 1 1 1 1 1 6 6 6 6 6

> cbind(x, y)
       x y
 [1,]  1 1
 [2,]  2 1
 [3,]  3 1
 [4,]  4 1
 [5,]  5 1
 [6,]  6 6
 [7,]  7 6
 [8,]  8 6
 [9,]  9 6
[10,] 10 6

> data.frame(x, y)
    x y
1   1 1
2   2 1
3   3 1
4   4 1
5   5 1
6   6 6
7   7 6
8   8 6
9   9 6
```

```
10 10 6

> class(cbind(x, y))
[1] "matrix"
> class(data.frame(x, y))
[1] "data.frame"
```

When the two vectors are not of the same length, R may try to *recycle* the smaller vector's contents:

```
> #Examples of recycling
> cbind(x, y[-10])
       x
 [1,]  1 1
 [2,]  2 1
 [3,]  3 1
 [4,]  4 1
 [5,]  5 1
 [6,]  6 6
 [7,]  7 6
 [8,]  8 6
 [9,]  9 6
[10,] 10 1

Warning message:
In cbind(x, y[-10]) :
  number of rows of result is not a multiple of vector
length (arg 2)

> data.frame(x, y[-10])
Error in data.frame(x, y[-10]) :
  arguments imply differing number of rows: 10, 9

> cbind(x, 1)
       x
 [1,]  1 1
 [2,]  2 1
 [3,]  3 1
 [4,]  4 1
 [5,]  5 1
 [6,]  6 1
```

```
 [7,]  7 1
 [8,]  8 1
 [9,]  9 1
[10,] 10 1

> #data.frame(x, 1) #similar to cbind(x, 1)
```

2)`rbind()` – combine data by rows

```
> rbind(x, y)
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
x    1    2    3    4    5    6    7    8    9    10
y    1    1    1    1    1    6    6    6    6     6
```

3)`merge()` – merge data.frames by unique values

```
> set1<-data.frame(name1 = c("a", "b", "c", "d", "e",
    "f"), response1 = c(1, 2, 3, 4, 5, 6))
> set2<-data.frame(name2 = c("a", "a", "b", "c", "d",
    "e"), response2 = c(10, 11, 20, 30, 40, 50))

> set1
  name1 response1
1     a         1
2     b         2
3     c         3
4     d         4
5     e         5
6     f         6

> set2
  name2 response2
1     a        10
2     a        11
3     b        20
4     c        30
5     d        40
6     e        50

> merge(x = set1, y = set2, by.x = "name1", by.y =
```

```
   "name2", all = TRUE)
  name1 response1 response2
1     a         1        10
2     a         1        11
3     b         2        20
4     c         3        30
5     d         4        40
6     e         5        50
7     f         6        NA

> merge(x = set1, y = set2, by.x = "name1", by.y =
    "name2", all = FALSE)
  name1 response1 response2
1     a         1        10
2     a         1        11
3     b         2        20
4     c         3        30
5     d         4        40
6     e         5        50
```

The `all` argument specifies whether or not non-matching rows are included in the resulting data.frame. The default is `all = FALSE`.

4)`expand.grid()` – Find all possible combinations of elements in vectors

```
> x<-1:3
> y<-c("a", "b")

> expand.grid(x, y)
  Var1 Var2
1    1    a
2    2    a
3    3    a
4    1    b
5    2    b
6    3    b

> expand.grid(y, x)
```

```
  Var1 Var2
1    a    1
2    b    1
3    a    2
4    b    2
5    a    3
6    b    3
```

The first argument value is varied over the fastest.

5) `sort()` and `order()` – used to sort the elements of a vector or data.frame

Sorting a vector:

```
> #Example 1
> x<-c("b", "c", 1)
> x
[1] "b" "c" "1"
> sort(x)
[1] "1" "b" "c"
> class(x)
[1] "character"
```

Sorting a data.frame by one variable:

```
> #Example 2
> set1<-data.frame(ID = c(3, 2, 1), response = c(10,
   20, 15))
> set1
  ID response
1  3       10
2  2       20
3  1       15
> sort(set1) #Does not work
Error in `[.data.frame`(x, order(x, na.last = na.last,
decreasing = decreasing)) :
  undefined columns selected
```

```
> order(set1$ID)
[1] 3 2 1
> set1[order(set1$ID),]
  ID response
3  1       15
2  2       20
1  3       10
```

The `order()` function provides the row indexes to use with data.frame.

Sorting a data.frame by two variables:

```
> #Example 3
> set1<-data.frame(ID = c(2, 2, 1), response1 = c(20,
    10, 15), response2 = c(20, 40, 18))
> set1
  ID response1 response2
1  2        20        20
2  2        10        40
3  1        15        18
> set1[order(set1$ID),]
  ID response1 response2
3  1        15        18
1  2        20        20
2  2        10        40
> set1[order(set1$ID, set1$response1),]
  ID response1 response2
3  1        15        18
2  2        10        40
1  2        20        20
```

6) `rev()` – reverse the order of items in a vector

```
> x<-1:10
> rev(x)
 [1] 10  9  8  7  6  5  4  3  2  1
```

7) `reshape()` – Useful for transforming longitudinal data from a “long” to a “wide” format and vice versa.

```
> set1<-data.frame(ID.name = c("subject1", "subject2",
   "subject3"), ID.number = c(1, 2, 3), age = c(19, 16,
   21), time1 = c(1, 0 ,0), time2 = c(0, 0, 1))
> set1
   ID.name ID.number age time1 time2
1 subject1         1  19     1     0
2 subject2         2  16     0     0
3 subject3         3  21     0     1

> #Long format
> set2<-reshape(data = set1, idvar = "ID.name", varying
    = c("time1", "time2"), v.names = "response",
    direction = "long", drop = "ID.number")
> set2
            ID.name age time response
subject1.1 subject1  19    1        1
subject2.1 subject2  16    1        0
subject3.1 subject3  21    1        0
subject1.2 subject1  19    2        0
subject2.2 subject2  16    2        0
subject3.2 subject3  21    2        1

> row.names(set2)<-NULL
> set2
   ID.name age time response
1 subject1  19    1        1
2 subject2  16    1        0
3 subject3  21    1        0
4 subject1  19    2        0
5 subject2  16    2        0
6 subject3  21    2        1

> #Back to wide format
> set3<-reshape(data = set2, timevar = "time", idvar =
    "ID.name", direction = "wide")
> set3
   ID.name age.1 response.1 age.2 response.2
```

```
1 subject1    19          1    19          0
2 subject2    16          0    16          0
3 subject3    21          0    21          1
```

Miscellaneous

Below is a list of items that did not fit elsewhere (see miscellaneous.R for code):

- The RExcel package allows for Excel to use R functions.
- The `traceback()` function is useful to help diagnose code errors.
- The `for()` function is the most commonly used function for loops:

```
> #Create a 3x2 matrix of observed normal random
   variables
>  set.seed(1929)
> set1<-matrix(data = rnorm(n = 6, mean = 0, sd = 1),
   nrow = 3, ncol = 2)
> set1
           [,1]        [,2]
[1,]   0.1102744 -1.06010197
[2,] -0.5237226   0.29005040
[3,] -0.1333107   0.03343786

> save.it<-numeric(3) #Initialize vector to save results
> save.it
[1] 0 0 0

> for (i in 1:3) {
   save.it[i]<-mean(set1[i,])
  }

> save.it
[1] -0.47491379 -0.11683612 -0.04993643
```

- The `apply()` function performs many of the same calculations as `for()`, but much more efficiently:

```
> #Apply a function by row (MARGIN = 1)
> apply(X = set1, MARGIN = 1, FUN = mean)
[1] -0.47491379 -0.11683612 -0.04993643

> #Apply a function by column (MARGIN = 2)
> apply(X = set1, MARGIN = 2, FUN = mean)
[1] -0.1822530 -0.2455379
```

- `search()` displays the *search path* for R. For example, suppose two packages contain functions with the same name and both packages are loaded into R. R will run the function from the package that appears first in the search path.
- The boot package is the main package used for the bootstrap. The package is installed by default within R, but you still need to use `library(package = boot)` to make its functions available for use.
- The multcomp package is useful for performing hypothesis tests involving multiple parameters. The package also provides ways to control the overall familywise error rate for multiple tests.
- `setwd()` function sets the “working directory” for all R files that you read in or write out. For example,

```
> setwd(dir = "C:\\chris\\unl\\Dropbox\\NEW\\workshop\\
   Gallup\\")
> gpa<-read.table(file = "gpa.txt", header = TRUE, sep =
   "")
> head(gpa)
  HS.GPA College.GPA
1   3.04         3.1
2   2.35         2.3
3   2.70         3.0
```

```
4   2.05         1.9
5   2.83         2.5
6   4.32         3.7
```

This is helpful when there is a long folder structure and/or a need to read in or write out many times.

VII. Index of R terms

VIII. Index of R functions