MICHAEL CLARK
CENTER FOR SOCIAL RESEARCH
UNIVERSITY OF NOTRE DAME

INTRODUCTION TO R

A SECOND COURSE

Contents

Preface

Current draft November 28, 2012

This is the second session of a short course providing an introduction to using the R statistical environment with an eye toward applied social science research. Nothing but the previous course knowledge is assumed, however the basic notion of creation and manipulation of objects, getting and using various R packages and similar introductory details from that is required. As a preliminary step, you should go through the first course handout, install R, install a couple packages, and play around with the code provided.

This handout was created in Rstudio using R 2.15.2 .

What You're Getting with R

Let's start as we did in the first course by just chatting a bit about R and what it is capable of. The following is a list chosen just to give a sense of the breadth of R's capabilities.

- Additions and advancements to all standard analyses including many cutting edge, just published techniques
- Structural Equation Modeling
- Data mining/Predictive analytics/Machine learning and related
- Natural language processing and text mining
- Complex survey analysis
- Network analysis and graphs
- Image analysis
- Statistical genetics/bioinformatics/high-throughput screening
- Bayesian inference for a wide variety of analyses
- Reproducible research/Live document updating
- Ability to easily use other statistical packages within the R environment
- Ability to take advantage of other programming languages (e.g. C, Python)
- Mapping and geospatial analysis
- Web capabilities and interactivity
- Great graphing capabilities

Examples of how specific some of the packages can get:

- A package oriented toward vegetation ecologists (vegan)[1].
- A package that will allow one to create interactive Google Visualization charts (googleVis).
- Another that will automate the creation of Mplus syntax files, run them, extract the information from the model results, and display them, all without leaving R (MplusAutomation).
- One that will produce Latex code that will produce highlighted R code, used throughout this document (highlight)...

[1] This actually has some functionality that might be useful to anyone and I have played with it in the past.

and much, much more, and most of the list above has been available for quite some time. Most R packages will be very well developed and many even have their own sizable community devoted to further development. And perhaps best of all, it doesn't cost any more to add functionality. While some of these may seem like something you wouldn't use, it is often the case that packages from other disciplines or utilized for other purposes have functionality one would desire for their own project. Learning the function will then possibly expose one to new ways of thinking about and exploring their own data. In short, one can learn a lot just by using R.

Dealing with Data

Introduction to Dealing With Data

It's pretty well impossible to expect anyone to use a package they cannot get their data into easily, and the novice to R will find starting it up will provide no details on how to go about this. The closest one gets to in the File menu is to load an R workspace, which, while the associated files are *.Rdata files, they are not the data sets people in the social sciences are used to. R can handle typical data formats with ease as other stats packages do, and the following provides detail.

Reading in Text Data

Much of the data one comes across will primarily start as a text file and then be imported into a statistical package. While this is convenient, since any computer will have some program that can read the data, it can tend to be problematic importing it into a statistical package because issues can arise from various sources such as the way the data was recorded, the information relating how the data is to be read[2], the package being used to read in the data, and the various choices and knowledge the user is responsible for. Two common formats are fixed and delimited text files.

R manual

[2] Hopefully becoming less of an issue as standards are employed and requirements are put in place for data storage and access.

Fixed Format

Fixed format tends to throw some students for a loop as they download the data, open it up and are confronted with a wall-o'-text, i.e. just one big block of typically numeric information.

On top of this, a single observation may span more than one row within the file. In general these files are not decipherable just by looking at them, and sadly, codebook quality is highly variable (regardless

Typical view of fixed format data.

```
01234567890123456
789
12345678901234567
890
```

of source), where a poor codebook can make fixed format unreadable.

On the plus side, with the right information, any statistical package can read it in and R is no exception. Furthermore, it can cut down on the tedium necessarily associated with these files. The key information one needs regards how many columns pertain to each variable. In R, the function to read it is read.fwf, and an example is given below. The code on the left will first create and then read the data on the right. When trying this yourself you may find it easier to first create the file as a separate enterprise and then just use the last line replacing the ffdat in last line with something like "C:/wheredafileat/ffdat.txt", as you will normally be dealing with a separate file.

```
1234.56
987.654
12345.6
```

```
ffdat <- tempfile()
cat(file = ffdat, "1234.56", "987.654", "12345.6", sep = "\n")
read.fwf(ffdat, width = c(1, 2, 4))

##   V1 V2     V3
## 1  1 23  4.560
## 2  9 87  0.654
## 3  1 23 45.600
```

A brief summary before going too much further. The first line of the code creates an empty temporary file. The second line updates the file with the numeric information listed with an appropriate line separator. The final line reads this data in and displays it, but while this is fine for demonstration purposes, you'd actually be assigning the data to an object so that it can be used further.

Tab-Delimited Format

Delimited files are the sort of text file one will more likely deal with as often the data that is made publicly available may actually start out in spreadsheet form or is assumed to end there. If you open up a file like tab-delimited, it's more obvious what's going on, unlike what you see in fixed format files. It makes some initial eyeball inspection easier between the post-import data and the original file, and in general things are easier in importing these files to statistical packages. To continue with the above, the tab delimited form of the data might look like the following[3]:

[3] Often the file extension is .dat

```
V1   V2   V3
1    23   4.560
9    87   0.654
1    23   45.600
```

Now we can see what's going on already, although often you will not get a header row of variable names and it will be important for any statistical package you are using to note whether the first line is a

header or the data won't be read correctly[4]. In R, the default function is read.table. So continuing with the previous example, assuming you have the data somewhere on your machine already, the code is simply something like the following:

[4] You might also see some alignment difference between the header and subsequent rows, but this is inconsequential for the purposes of importing the data.

```
mydat <- read.table("C:/myfile.dat", header = TRUE)
```

Comma-separated Values

Yet another format one will be likely to use is one with comma-separated (comma delimited) values[5]. One can think of it as just like the tab delimited type, but with commas separating the values instead of spaces or tabs. I often will use it when transferring data between programs as some programs will have issues readings spaces correctly, whereas the comma is hard to misinterpret. I won't spend much time with this as importing is essentially the same as with read.table, however the appropriate function will be read.csv.

[5] As an aside, if you have MS Excel installed .csv files are typically associated with it.

```
V1,V2,V3
1,23,4.560
9,87,0.654
1,23,45.600
```

Reading in Data from Other Programs

Dealing with data from other packages can be a fairly straightforward procedure compared to text files though typically you may have to specify an option or two to get things the way you want. The foreign package is required here, and it might be a good idea to have this load when you start up R as you will use it often[6]. You will use the functions from the package in the same way as read.table above, but for example, to read a Stata file you'd use the function read.dta and to read an SPSS file you'd use the read.spss function. Going from one statistical package to another will almost certainly result in some changes in the data or loss of information. For example, when reading in an SPSS file you may have to choose to import factors as numeric, Stata data files can hold meta information that other programs don't read etc. This is not to say you will lose vital data, just that some choices will have to be made in the process as different programs deal with data files in different ways and there is no 1-1 matching of file structure.

[6] In your R/etc folder you will find the Rprofile.site file, put the line library(foreign) somewhere in there.

The R-commander Package

The Rcmdr package can be a great help in getting used to the process of importing data for those new to R. Rcmdr is a package that allows one to use a menu system to do analysis and deal with data, and its layout will definitely feel familiar to those coming from some of the other standard stat packages. In the Data menu there are options for importing text, Stata, SPSS, Excel and other files and it is usually straightforward.

Verification

It is always important to verify that the data you now have is what it should be. R isn't going to mistake a two for a five or anything like that, as with any statistical program but there may be issues with the original format that might result in rounding problems, missing values etc. As an example, if 99 represents missing on some variable, it's important that the variable is imported as such and not treated as a numeric value, or the change should be made immediately after import. Programs should be able to read them in appropriately most of the time, but it is a good idea double check.

Checking random observations

A first step to verifying the data can be comparing random observations in the new data with those in the old. Given a data set with 200 observations and 100 variables, the following would pull out five cases and their values on a random selection of five variables.

A simple check for whether two objects are the same can be accomplished via the all.equal function.

```
mydata[sample(1:200, 5), sample(1:100, 5)]
```

If there are specific cases you want to look at you would note the exact and column row numbers, but we will be talking about that and related indexing topics in a bit.

Descriptive Statistics

When importing data from another statistical package, another way to check initial the imported data is to run descriptive statistics on some of the data. The results should be *identical* to those of the source data. One can use the summary function as a first step for specific variables or the whole data set once it is imported.

Summary

Importing data into any statistical package can actually be very difficult at times so don't be surprised if you hit a snag. While there are even programs dealing with this specifically, you shouldn't feel the need to purchase them these days as most statistical programs generally behave well with each other's data. Sometimes the easiest path will involve using multiple packages. I often will save data in a particular statistical package as a text file, which is then easily imported into another package. For example I might go from Excel to text to R. Whatever the format and whatever route you use, make sure to verify the data in some fashion after import.

Data Manipulation & Management

I won't spend a whole lot of time here as one can review the notes for the first course for more detail, but there are a couple things worth mentioning.

Data Formats

There are a variety of formats for dealing with data each of which has their use in analysis. Brief descriptions follow.

Vectors

Single items of information such as a number are *scalars*, but usually we have many that are concatenated into a *vector*[7] that represents some variable of interest. Vectors in R can be of several types, though the elements of vector must all be of the same kind. Unlike the way one typically interacts with a single data set, one can have interactions with multiple vectors from any number of sources. In this sense, one could for example have a data set as one object and use variables from it to predict a vector of values that is a completely separate object. Types or modes of vectors include:

NUMERIC/INTEGER Numeric data is typically stored as a numeric or integer mode object, and we can coerce vectors to be of these types using e.g. using the function as.numeric. It usually will not matter to the analysis how a truly numeric variable is stored.

[7] In math and stats texts these would just be numeric, but for our purposes I'm just distinguishing between one item or several.

```
x = rnorm(10)
```

LOGICAL vector taking on *only* TRUE or FALSE values but which is not the same as a factor or character. One will often use these in their own functions, conditional, statements etc. Note that these can also be treated as though they were numeric taking on values of 0 and 1 for False and True respectively.

```
x2 = x > 1
x2

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE

sum(x2)

## [1] 3

x2 * 2

##  [1] 0 0 0 0 0 2 2 0 2 0
```

CHARACTER A vector of character strings. Note that these must be converted to a factor if they are to be used as a variable in an analysis.

```
as.character(x2)
```

FACTOR In other packages we typically enter such information directly or numerically with labels, which is no different here.

```
as.factor(x2)
# Example from scratch:
x <- rep(c(1:3, 5))
x2 <- factor(x, levels = 1:3, labels = c("a", "b", "c"), ordered = T)
```

On the other hand, one may find that R's default dealing with ordered categorical information that has been imported from other packages can be problematic, and there is often something that comes up in dealing with them. So you may end up seeing things like blanks as factor levels, need to do something with the level ordering for graphical display, convert it to a numeric variable for some purposes etc. These issues don't seem like that big of deal, and they aren't normally, but they will come up (and for those that work with Social Science data everyday they come up all the time) and require additional steps to get things just the way you want. Dealing with factor variables in R may take some getting used to, but generally one can go about it like they do elsewhere.

The default ordering in plots is by alphabetical ordering of labels.

Lists

Lists are simply collections of objects be they single items, several vectors, multiple datasets or even combinations of those. While one

probably doesn't deal with such a thing typically in other stats packages, lists can make iterative operations very easy and are definitely something to utilize in analysis within R. Furthermore most output from analysis will result in a list object, so it is something you'll want to know how to access the elements of. The following shows the creation of a list, ways to extract components of the list and elements of the list component, and using the lapply function and its counterpart sapply[8] on the list components.

[8] Sapply converts the result to a vector or matrix if possible.

```
xlist = list(1:3, 4:6)
xlist

## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 4 5 6

names(xlist) = c("A", "B")
xlist[["A"]]

## [1] 1 2 3

xlist$A

## [1] 1 2 3

xlist$A[2]

## [1] 2

# lapply(x,sum) Not shown
sapply(x, sum)

##  [1]  0.24293 -0.04679 -0.50959  0.95315 -0.20230  1.96167  1.10532
##  [8]  0.03913  1.74018 -1.36667
```

Matrices

Matrices in R are like data sets we usually deal with but are full of vectors only of one *mode*[9], and you may find that various functions may only work with matrices[10]. When you look at the *x* object after running the code, note the row and column designations. You'll see for example [,1] for the first column and [5,] for the fifth row. This is how you extract row and columns from a matrix, e.g. mymatrix [,2] or mymatrix[1:5,]. Use a number before and after the comma to extract specific elements of the matrix. In the following we look at the first and fifth through 8 rows, and all but the first, third and fifth columns.

[9] For simple vectors, mode is the same as class.
[10] The function data.matrix will convert a data frame to a matrix

```
x = matrix(rnorm(100), ncol = 10, byrow = T)
x[c(1, 5:8), -c(1, 3, 5)]
```

Arrays

Researchers sometimes forget or are simply unaware that the there is no restriction of data to one or two dimensions. Arrays can be of any dimension[11], and often the results of some function operation will result in an array object. The first example creates an array of two 2x25 matrices, the second uses the apply function on the iris3 array[12] to get column means (with 20% trim) for each type of flower.

[11] A two dimensional array is the same thing as a matrix, which could be converted to a dataframe as well.
[12] Available in any installation of R.

```
x = array(rnorm(100), dim = c(2, 25, 2))
apply(iris3, c(2, 3), mean, trim = 0.2)

##          Setosa Versicolor Virginica
## Sepal L.   5.00      5.910     6.547
## Sepal W.   3.41      2.797     2.963
## Petal L.   1.46      4.307     5.493
## Petal W.   0.22      1.340     2.023
```

Data Frames

At last we come to the object type perhaps of most interest in applied social science research. Data frames are R objects that we refer to as data sets in other packages, and while they look like matrices whose columns may be vectors of any mode, though in R they are treated more like a list of variables of the same length[13]. As an exercise, rerun any of the previous examples that created a vector, list, matrix or array, and convert them to a data.frame by typing `data.frame(x)`.

It may not be obvious to those first introduced to R, but simply doing something like data.frame(x) just spits out the result of making x a dataframe, it does not automatically change the x object permanently unless a new object is created or the old one overwritten:
x = data.frame(x) #overwrites it
y = data.frame(x) #create a new object

[13] For example, we can extract columns with the dollar sign operator- mydata$var1.

Initial Examination of Data

Let's get to messing around with some data shall we? I will not spend a lot of time with IED as data indexing, subsetting, and summary statistics were covered in the first course and previous section, but this will serve as a brief review. We will read in data regarding high school student test scores on different subjects. The following will pull it in from the UCLA ATS website (variable labels), examine the first few rows, the general structure and obtain some descriptive statistics on the variables.

```
hs1 <- read.table("http://www.ats.ucla.edu/stat/data/hs1.csv", header = T, sep = ",")
# head(hs1) str(hs1) summary(hs1)
```

```
# the describe function is much better however
library(psych)
describe(hs1)
```

```
##          var   n   mean    sd median trimmed   mad min max range  skew
## female     1 200   0.55  0.50    1.0    0.56  0.00   0   1     1 -0.18
## id         2 200 100.50 57.88  100.5  100.50 74.13   1 200   199  0.00
## race       3 198   3.42  1.04    4.0    3.64  0.00   1   4     3 -1.55
## ses        4 200   2.06  0.72    2.0    2.07  1.48   1   3     2 -0.08
## schtyp     5 200   1.16  0.37    1.0    1.07  0.00   1   2     1  1.84
## prgtype*   6 200   1.73  0.84    1.0    1.66  0.00   1   3     2  0.55
## read       7 200  52.23 10.25   50.0   52.03 10.38  28  76    48  0.19
## write      8 200  52.77  9.48   54.0   53.36 11.86  31  67    36 -0.47
## math       9 200  52.65  9.37   52.0   52.23 10.38  33  75    42  0.28
## science   10 195  51.66  9.87   53.0   51.83 11.86  26  74    48 -0.18
## socst     11 200  52.41 10.74   52.0   52.99 13.34  26  71    45 -0.38
## prog      12 200   1.73  0.84    1.0    1.66  0.00   1   3     2  0.55
##          kurtosis   se
## female      -1.98 0.04
## id          -1.22 4.09
## race         0.80 0.07
## ses         -1.10 0.05
## schtyp       1.40 0.03
## prgtype*    -1.37 0.06
## read        -0.66 0.72
## write       -0.78 0.67
## math        -0.69 0.66
## science     -0.59 0.71
## socst       -0.57 0.76
## prog        -1.37 0.06
```

Often we want to get separate information for different groups. Here we'll look at several scores broken down by gender, and a single score broken down by two grouping variables gender and program type.

```
# describe.by(hs1[,7:11], group=hs1$gender)
describe.by(hs1[, 7:11], group = hs1$gender, mat = T)  #different view
```

```
##          item group1 var   n  mean     sd median trimmed    mad min max
## read1       1      0   1  91 52.82 10.507     52   52.82 11.861  31  76
## read2       2      1   1 109 51.73 10.058     50   51.42 10.378  28  76
## write1      3      0   2  91 50.12 10.305     52   50.44 11.861  31  67
## write2      4      1   2 109 54.99  8.134     57   55.57  7.413  35  67
## math1       5      0   3  91 52.95  9.665     52   52.45 10.378  35  75
## math2       6      1   3 109 52.39  9.151     53   52.07 10.378  33  72
## science1    7      0   4  86 52.88 10.754     55   53.26 11.861  26  74
## science2    8      1   4 109 50.70  9.039     50   50.80  8.896  29  69
## socst1      9      0   5  91 51.79 11.334     51   52.45 14.826  26  71
## socst2     10      1   5 109 52.92 10.234     56   53.34  7.413  26  71
##          range     skew kurtosis     se
## read1       45  0.04598  -0.7867 1.1014
## read2       48  0.31898  -0.5456 0.9634
## write1      36 -0.17694  -1.1681 1.0803
## write2      32 -0.58190  -0.5024 0.7791
## math1       40  0.32034  -0.6947 1.0131
## math2       39  0.23145  -0.7569 0.8765
## science1    48 -0.31428  -0.6938 1.1597
## science2    40 -0.12892  -0.5350 0.8657
```

```
## socst1      45 -0.36525  -0.7158 1.1881
## socst2      45 -0.34843  -0.5268 0.9803

# describe.by(hs1$science, group=list(hs1$gender,hs1$prgtype))
```

At this point I'll remind you to check out the first course notes regarding the various means to subset the data and examine different parts of it. Moving on, notice that some of the variables were imported as integers, but would be better off considered as factors with associated labels[14]. Let's go ahead and change them:

[14] This will have other benefits in analysis, for example auto-dummy coding.

```
hs1[,c(1,3:6)] <- lapply(hs1[,c(1,3:6)], as.factor)

#change the labels
levels(hs1$gender) = c("male","female")
levels(hs1$race) = c("Hispanic","Asian","Black","White","Other")
levels(hs1$ses) = c("Low","Med","High")
levels(hs1$schtyp) = c("Public","Private")
levels(hs1$prgtype) = c("Academic","General","Vocational")

#Alternate demo to give a sense of how one can use R
vars <- c("gender","race","ses","schtyp","prgtype")
labs <- list(gender=c("male","female"),
             race=c("Hispanic","Asian","Black","White","Other"),
             ses=c("Low","Med","High"),
             schtyp=c("Public","Private"),
             prgtype=c("Academic","General","Vocational"))

for (i in 1:5){
  hs1[,vars[i]] = factor(hs1[,vars[i]],labels=labs[[i]])
}
str(hs1)

## 'data.frame': 200 obs. of  11 variables:
##  $ gender : Factor w/ 2 levels "male","female": 1 2 1 1 1 1 1 1 1 1 ...
##  $ id     : int  70 121 86 141 172 113 50 11 84 48 ...
##  $ race   : Factor w/ 5 levels "Hispanic","Asian",..: 4 4 4 4 4 4 3 1 4 3 ...
##  $ ses    : Factor w/ 3 levels "Low","Med","High": 1 2 3 3 2 2 2 2 2 2 ...
##  $ schtyp : Factor w/ 2 levels "Public","Private": 1 1 1 1 1 1 1 1 1 1 ...
##  $ prgtype: Factor w/ 3 levels "Academic","General",..: 2 3 2 3 1 1 2 1 2 1 ...
##  $ read   : int  57 68 44 63 47 44 50 34 63 57 ...
##  $ write  : int  52 59 33 44 52 52 59 46 57 55 ...
##  $ math   : int  41 53 54 47 57 51 42 45 54 52 ...
##  $ science: int  47 63 58 53 53 63 53 39 NA 50 ...
##  $ socst  : int  57 61 31 56 61 61 61 36 51 51 ...

# summary(hs1)
```

Let's look at the correlation of the test scores. I use the second argument so that it will only use observations with complete data on all four (case-wise deletion). While not the case here, even moderately sized correlation matrices can get a bit unwieldy without having seven decimal places further cluttering up the view, so I provide an example of rounding also[15].

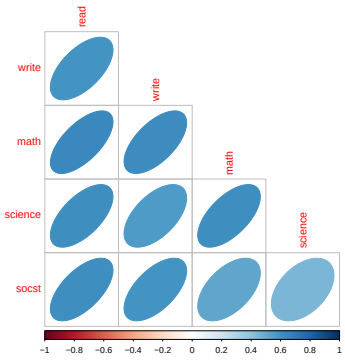
[15] What rounding you will see in output is package dependent.

```
cormat <- cor(hs1[, 7:11], use = "complete")
round(cormat, 2)

##         read write math science socst
## read    1.00  0.60 0.65    0.62  0.62
## write   0.60  1.00 0.62    0.57  0.60
## math    0.65  0.62 1.00    0.62  0.53
## science 0.62  0.57 0.62    1.00  0.45
## socst   0.62  0.60 0.53    0.45  1.00
```

Now we are ready to start examining the data graphically. As preview, examine the correlation matrix just created with the following code.

```
library(corrplot)
corrplot(cormat, "ellipse", type = "lower", diag = F)
```

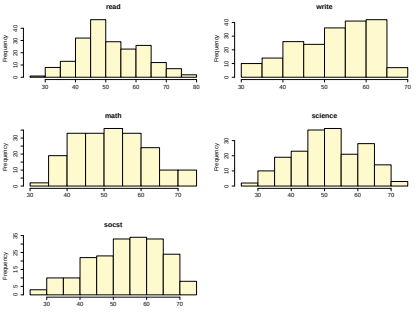


Graphical Techniques

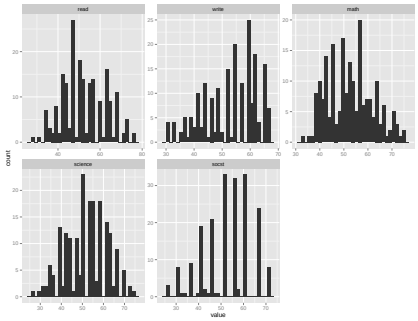
It is an odd time at present in which we have all manner of technologies available to produce any number of visualizations, yet people are beholden to journals still pretending to save ink like it's 1950. This results in researchers still producing graphs that enlighten little more than what could be gained from a single sentence[16]. Good graphs are sufficiently, but not overly, complex and yet done in a way that plays on the readers intuition and clarifies the material presented. This is not an easy task, but in my opinion, R can make the task more easily attainable than other statistics packages, while allowing for possibilities that most would not think of or be able to do in other packages.

While you have a great deal of control with basic R plotting functions, by default they may not look like much. For example, to examine the test scores' distributions we might do something like the following, but rather than write out a line code for every histogram we might want, we'll set the dimensions of the graphics device to be able to hold all five, then populate it with the histogram of each variable. We could have used apply or lapply here to accomplish the same thing in a single line of code, but by default the graphs would not retain the variable names. Note also that there are many, many options we would have control of that aren't tweaked here. In the following I use the sapply function in a way to deal with the columns of interest and within it I create a function on the fly to plot histograms of those variables. The console output left behind is meaningless for our purposes, but refers to contents of objects created by the hist function.

[16] In perhaps one of the more extreme cases, I just recently saw a journal article present a graphic of words from the classic Stroop task in different shades of black and white.



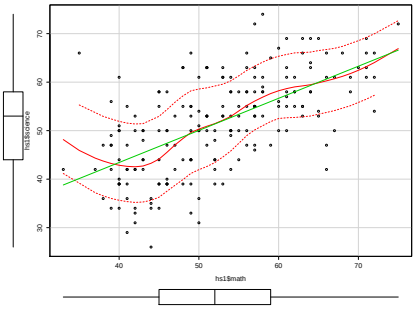
```
par(mfrow = c(3, 2))  #set panels for the graph
sapply(7:11, function(x) hist(hs1[, x], main = colnames(hs1)[x], xlab = "",
    col = "lemonchiffon"))
par(mfrow = c(1, 1))  #reset
```

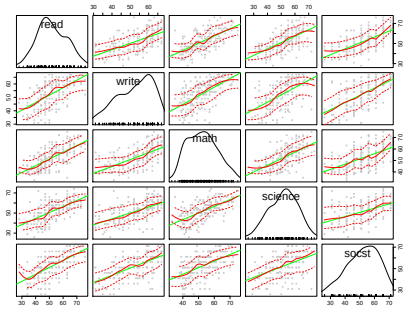


For a comparison, we'll give a preview to ggplot2 (and reshape2) to accomplish the same thing. Note how the data is transformed to 'long' format first. Again, many options available are not noted, this just shows the basics.

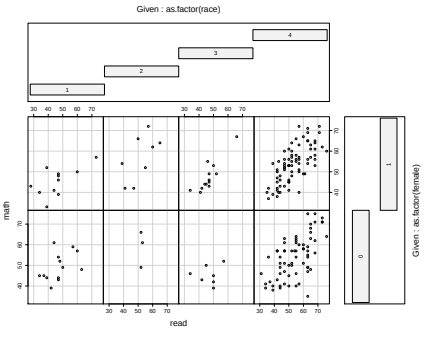
```
library(ggplot2)
library(reshape2)
graphdat = melt(hs1[, 7:11])
head(graphdat)
ggplot(aes(x = value), data = graphdat) + geom_histogram() + facet_wrap(~variable,
    scales = "free")
```

To examine both univariate and bivariate relationships simultaneously, one might instead use the scatterplot matrix capability of the car package. First is an example of a single scatterplot, scatterplot followed by the matrix. In the first, the univariate information is provided in the margins. With the matrix scatterplotMatrix the diagonals give us the univariate distribution, while the off-diagonal scatterplots are supplemented by loess curves to help spot potential curvilinear relationships. The coplot function that follows provides another example that one can use to examine multivariate information in a fairly straightforward manner.



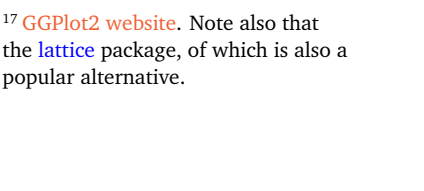


```
library(car)
scatterplot(hs1$math, hs1$science)
scatterplotMatrix(hs1[, 7:11], pch = 19, cex = 0.5, col = c("green", "red",
    "grey80"))
coplot(math ~ read | as.factor(race) * as.factor(gender), data = hs1)
```



A powerful set of graphing tools is provided by the ggplot2 package[17]. Along with associated packages, one can fairly easily reshape the data into a manageable format and produce graphs of good quality quickly.

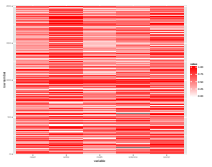
While we won't go into too much detail here, the things that will probably take some getting used to initially are thinking about the creation of your visualization via adding layers that each apply different a different graphical nuance, and also perhaps dealing with the data in 'melted' form (though the latter is not required for plotting). In the following the data is first inspected with several plots to get a sense of its structure to see if there are any notable problems. Afterwords the data is 'melted' which can make plotting easier. To begin, we first create the base plot *g*, which is actually empty. To it we add a plot of the overall means for each test and racial category. Just as a demonstration, we do



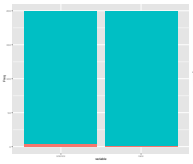
[17] GGPlot2 website. Note also that the lattice package, of which is also a popular alternative.

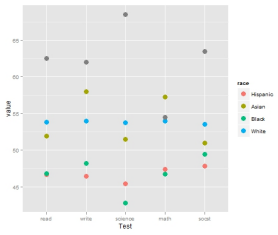
a further breakdown by gender. Finally the results are further conditioned on socio-economic status (note that most of the non-white cells are very low N).

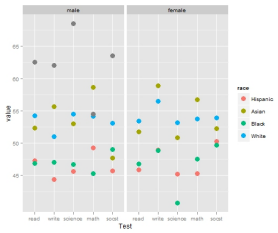
```
library(ggplot2)
library(reshape2)
# visualize the data itself
ggstructure(hs1[, 7:11])
ggfluctuation(xtabs(~race + gender, hs1))
ggmissing(hs1)
# melt the data
hs2 <- melt(hs1, id.vars = c("id", "gender", "race", "ses", "schtyp", "prgtype"),
    measure.vars = c("read", "write", "science", "math", "socst"), variable.name = "Test")
head(hs2)
```

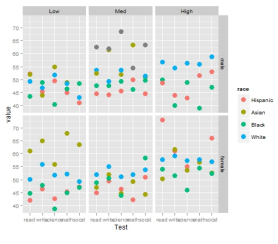


```
# create base plot
g <- ggplot(data = hs2, aes(x = Test, y = value))
# gray dots are missing on race
g + stat_summary(fun.y = "mean", geom = "point")
# facet by gender
g + stat_summary(fun.y = "mean", geom = "point", size = 4) + facet_grid(. ~
    gender)
# add another facet
g + stat_summary(fun.y = "mean", geom = "point", data = hs2, size = 4) + facet_grid(gender ~
    ses)
```









Summary

Once you get the hang of it, great visualizations can be relatively easily produced within the R environment, and there are many packages specific to certain types of graphs, e.g. networks, maps, etc., as well as interactive, dynamic, graphics, web-related functionality etc. In short, one doesn't need a separate program to produce high quality graphics and one has great control over every aspect of the production process. It is definitely worth getting used to R's capabilities and watch as your standard plots soon fall by the wayside.

Analysis

In this section we'll provide a few examples of how to go about getting some statistical results from your data. The basic approach illustrated should get you pretty far in that many of the analyses in R packages work in a similar manner. For regression type analyses you will get the same type of output though it may look a little different.

A Simple Group Comparison

To begin we'll just do a simple group comparison on one of the tests from the previous data. For example, let's see if there is a statistical difference among male and female observations with regard to the reading test from the previous data. We'll use the t.test function[18].

[18] By default a Welch's correction for unequal variances is applied.

```
t.test(read ~ female, hs1)

##
##  Welch Two Sample t-test
##
## data:  read by female
## t = 0.7451, df = 188.5, p-value = 0.4572
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.796  3.977
## sample estimates:
##   mean in group male mean in group female
##                52.82                 51.73
```

As a further exercise in programming, we'll do this for each of the tests using the powerful apply function. Conceptually it's simply doing five t-tests. The code is also simple in the sense it only requires one line and the apply function to pull off. But as we are new to R, I'll break it down a little further by first just calculating a simple mean for each column to show how apply works. So in the following, we *apply*, to the columns (*2*), the mean function (*mean*), with the additional argument to remove missing values (*na.rm=T*). I verify this with the built in colMeans function after.

Other approaches would be more statistically appropriate.

```
apply(hs1[, 7:11], 2, mean, na.rm = T)

##    read   write    math science   socst
##   52.23   52.77   52.65   51.66   52.41

colMeans(hs1[, 7:11], na.rm = T)

##    read   write    math science   socst
##   52.23   52.77   52.65   51.66   52.41
```

Back to the t-test. I do the exact thing we just did but in this case I create a function 'on the fly'. This constructed function only requires one input (*y*), which here will be a single vector pertaining to the columns Reading, Writing etc. test scores. The y itself is fed to the t.test function as the outcome of interest. Once you start getting comfortable with the apply functions[19], you can try even more succinct approaches as in the second example.

[19] It's not certain this is possible. Just as soon as you feel like you've got the hang of it you'll find quirks. Check out the plyr package as an alternative.

```
myt <- function(y) {
    t.test(y ~ gender, data = hs1)
```

```
}
apply(hs1[, 7:11], 2, myt)
# alternate one-liner
# sapply(7:11, function(y) t.test(hs1[,y] ~ hs1$gender), simplify=F )
```

Linear Regression Example

Next we come to the familiar linear regression model, and as we will see, the way the function and its arguments work is very similar to the t.test function before. Here we will use the lm function and the associated object that is created by it.

Creating the Model Object

First we will create the model object, and we'll use a new data set[20]. We'll read this Stata file in directly using the foreign library[21]. It is random sample of 400 elementary schools from the California Department of Education's API 2000 data set. This data file contains a measure of school academic performance as well as other attributes of the elementary schools, such as, class size, enrollment, poverty, etc. After importing it, I suggest you use the str and summary functions we used before to start to get a feel for the data. Perhaps produce a simple graph.

[20] I again use data from UCLA's ATS website, in this case an example they use for Stata which can allow you to compare and contrast. Also it's a reminder to check their website when learning any program as there is a lot of useful examples to get you started.

[21] You can disregard the warning. The district number variable just has an empty label.

```
library(foreign)
regdata <- read.dta("http://www.ats.ucla.edu/stat/stata/webbooks/reg/elemapi2.dta")
```

Here the goal is to predict academic performance (api00) from percentage receiving free meals (meals), percentage of English language learners (ell), and percentage of teachers with emergency credentials (emer). Running the regression is fairly straightforward as follows:

Create a model object with the lm function. The minimal requirement here is the formula specification as was done for the t-test along with the data object name, but if interested you might type ?lm to see what else is possible.

Examine the model object. This will only produce the coefficients.

Summarize it. This produces the output one is used to seeing in articles and other packages.

Summary Results

Let's take a look at the summarized model object...

Using the xtable function (from the package of the same name) on the summary object will produce latex code for easy import of the output into a document in professional looking form.

```
mod1 <- lm(api00 ~ meals + ell + emer, data = regdata)
mod1

##
## Call:
## lm(formula = api00 ~ meals + ell + emer, data = regdata)
##
## Coefficients:
## (Intercept)        meals          ell         emer
##      886.70        -3.16        -0.91        -1.57

summary(mod1)

##
## Call:
## lm(formula = api00 ~ meals + ell + emer, data = regdata)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -185.47  -39.95   -3.66   36.45  178.48
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  886.703      6.260  141.65  < 2e-16 ***
## meals         -3.159      0.150  -21.10  < 2e-16 ***
## ell           -0.910      0.185   -4.93  1.2e-06 ***
## emer          -1.573      0.293   -5.37  1.4e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 57.8 on 396 degrees of freedom
## Multiple R-squared: 0.836,Adjusted R-squared: 0.835
## F-statistic:  673 on 3 and 396 DF,  p-value: <2e-16
```

To begin, you get the actual code that produced the output. This is followed by quantiles of the residuals for initial inspection. Then come the coefficients, and this part is typical of output from other stat packages[22]. You might go ahead and get used to seeing < 2.2e-16; other statistical packages will put zero, but R reminds you that the p-value is just really small. The coefficients and related output is what I sometimes refer to as the *micro* level of the analysis, as we get details about the individual variables. At the *macro* level we see how the model does as a whole, and as you would see in other packages for standard regression output, you get the residual standard error[23], F-statistic, and the R-squared value and its adjusted value. Not surprisingly, all three variables are statistically notable predictors of academic achievement and the model as a whole does well.

[22] the asterisks signify those places where magic was used in the analysis.

[23] In Stata this is labeled Root MSE, in SPSS, the Standard Error of the Estimate.

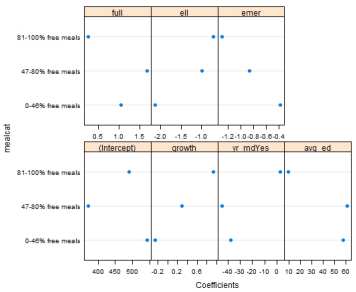
If you use the structure function (str) on the model object, you'll find that there are a lot of things already stored and ready to use for further processing. As an example we can extract the coefficients and fitted values for the model, and at least for some cases, one has access to them in one of two ways:

```
mod1$coef
coef(mod1)
mod1$fitted.values
fitted(mod1)
```

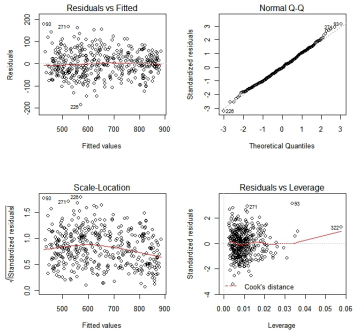
Plotting the Model Object

Having such easy access to the output components is quite handy. Consider the following which allows one to visualize change in coefficients across groups. It uses lmList from the nlme package to run regressions on separate groups, at which point we can extract and plot them to see how they might change across groups. This is not really the best way to go about looking for such things, but does illustrate the sorts of things one can do fairly easily with the model object, and R makes deep exploration of data and models notably easier than other packages. Here we see how a model changes over the ordered categories representing percentage of the school district receiving free meals.

```
library(nlme)
modlist2 <- lmList(api00 ~ growth + yr_rnd + avg_ed + full + ell + emer | mealcat,
    data = regdata, na.action = na.omit)
plot(coef(modlist2), nrow = 2)
```



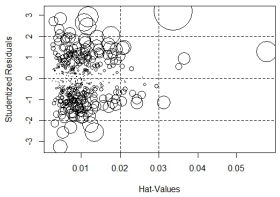
It should also be noted that the *lm* object comes with 'built-in' plots to help one start to visually inspect the residuals for problems. Just using the basic plot function on an *lm* object produces four plots to allow one to inspect how the residuals behave (technically one is making use of the plot.lm function). For inspection of outliers, I generally suggest influencePlot from the car package which produces a single plot that allows one to detect outliers via three different measures (also shown). In general, the lm plots have enhanced versions in the car package along with additional visual diagnostics that are available. Check out crPlots and avPlots functions to obtain component plus residual and added-variable plots.




```
plot(mod1)
influencePlot(mod1)
```

Statistical Examination of the Model

Testing Assumptions. Along with visual inspection of the results, traditional tests regarding model behavior are also available. These include such tests as tests for normality (e.g. shapiro.test) the Breusch-Pagan test for heteroscedasticity (bptest), RESET test (resettest) and production of variance inflation factors (vifvif) to name a few. A good package to get you started is lmtest and car packages.



Model Comparison. Often one is also interested in investigating a comparison of one model to a baseline or subset of it in order to draw attention to the unique characteristics of certain predictors. This too is fairly straightforward in R via the add1, drop1, and anova functions. As an example, we'll rerun the mod1 with only two predictors and then add the third.

```
mod1a <- lm(api00 ~ meals + ell, data = regdata)
# (.) is interpreted as 'what is already there'
add1(mod1a, ~. + emer, , test = "Chisq")

## Single term additions
##
## Model:
## api00 ~ meals + ell
##        Df Sum of Sq     RSS  AIC Pr(>Chi)
## <none>              1420232 3276
## emer    1     96343 1323889 3250  1.2e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As an alternative, we could could have created two model objects, one with and without *emer* and compared them via anova.

Variable Importance/Comparison. Variable importance is a tricky subject that unfortunately is typically carried out poorly, and in some sense is perhaps a misguided endeavor in many settings[24]. Substantive considerations should come first to determine whether something is important at all given the result[25]. Some like to use a standardization of the coefficient but this doesn't come free. If you're going to use statistical significance as an initial indicator of importance, it might seem a bit odd to abandon it when comparing relative importance among a set of predictors. Just because one standardized coefficient is larger than another doesn't mean it is statistically speaking, no more than just looking at two means would give you that information. For standard regression the package relaimpo provides nice functionality for predictor comparison and a metric, the average semi-partial squared, that nicely decomposes the model R-squared into the individual contributions by the predictors. However, better approaches in determining importance would include implementation of things like shrinkage and cross-validation.

Model Exploration. Whatever theoretical model one has come up with, no matter how sound the thinking that went into it, the simple fact is that nature is a lot more clever than we are. Whenever possible, which is almost always, one should incorporate a straightforward model search into their analytical approach. Using theory as a basis for providing viable models and the principle of parsimony to guide the search for the *set* of best models[26], one can examine their relative performance given the data at hand. A straightforward approach

[24] More important would be overall patterns of effects and theory alignment.

[25] As a hint, statistical significance is a fairly poor indicator of such a thing.

[26] One model based on one set of data should never be considered definitive, *especially* in the realm of social science.

for regression is provided by, for example, the stepAIC function in the MASS package, and depending on what package you're using, there may be something built in for such things. Others that come to mind are the BMA (Bayesian Model Averaging) package, and one might explore the CRAN Task Views for Machine Learning for more exploratory approaches that go well beyond the OLS arena.

Prediction

Prediction allows for validation of a model under new circumstances, as well as more exploration in cases where we do not have new data and so will simulate responses at predictor values of interest. One can readily do so with the basic predict function in R. If one does nothing but apply it to the same data with no options, it is akin to producing the fitted values from the model. But the data we feed it can come from anywhere and so it is far more useful in that sense. The following produces a predicted value for a school from the low end of the spectrum of values on the predictors and subsequently from one on the upper end. Given that the predictors were all negatively associated with achievement, what prediction would *you* make about what to expect?

```
# x and y = T save the model and response data frames respectively
mod1 <- lm(api00 ~ meals + ell + emer, data = regdata, x = T, y = T)
basepred <- predict(mod1)
head(cbind(basepred, fitted(mod1)))  #same

##   basepred
## 1    629.1 629.1
## 2    547.1 547.1
## 3    508.2 508.2
## 4    560.5 560.5
## 5    557.8 557.8
## 6    852.4 852.4


# obtain 25th and 75th percentile values of the predictor variables
lowervals <- apply(mod1$x, 2, quantile, 0.25)
lowervals2 <- data.frame(t(lowervals))  #t transposes
uppervals <- apply(mod1$x, 2, quantile, 0.75)
uppervals2 <- data.frame(t(uppervals))

# -1 removes the intercept column
predlow <- predict(mod1, newdata = lowervals2[-1])
predhigh <- predict(mod1, newdata = uppervals2[-1])
c(predlow, predhigh)

##     1     1
## 775.2 526.8
```

Other models will have alternative possibilities for what exactly is predicted (e.g. probabilities), and other packages will often have their

own built in predict functions. As an example from packages about to be mentioned in the next section, a binomial logit model run with the glm function in MASS could provide predictions of probabilities, and the sim function in Zelig will have a variety of possibilities depending the model one chooses, and generally makes this process pretty easy.

Beyond the Ordinary

Most of the time one will likely need more than the standard linear model and lm function. In general the question with R is not 'Can R do x analysis?' but 'Which R package(s) does x analysis?' or simply 'How would I do this in R?'. Recall that R has over 4000 packages, the bulk of which are specific to particular types of analyses. However even the base R installation comes with quite a bit of functionality, akin to what one would expect from typical stat packages, and we will start with an example there.

GLM example

The glm function in R provides functionality to examine generalized linear models. In this example,[27] we will predict the number of fish a team of fisherfolk catch with how many children were in the group (child), how many people were in the group (persons), and whether or not they brought a camper to the park (camper). In the following, the glm function is used to fit a poisson regression model to the data. As one can see, the basic format is the same for lm, complete with summary functions.

[27] This is taken from the UCLA ATS website, so feel free to see it performed in other packages there for comparison.

```
glmdata <- read.dta("http://www.stata-press.com/data/r12/fish.dta")
pois_out <- glm(count ~ child + camper + persons, data = glmdata, family = poisson)
summary(pois_out)

##
## Call:
## glm(formula = count ~ child + camper + persons, family = poisson,
##     data = glmdata)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -6.810  -1.443  -0.906  -0.041  16.142
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -1.9818     0.1523   -13.0   <2e-16 ***
## child        -1.6900     0.0810   -20.9   <2e-16 ***
## camper        0.9309     0.0891    10.4   <2e-16 ***
## persons       1.0913     0.0393    27.8   <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: 2958.4  on 249  degrees of freedom
```

```
## Residual deviance: 1337.1  on 246  degrees of freedom
## AIC: 1682
##
## Number of Fisher Scoring iterations: 6

exp(pois_out$coef)  #exponentiated coefficients

## (Intercept)       child      camper     persons
##      0.1378      0.1845      2.5369      2.9780
```

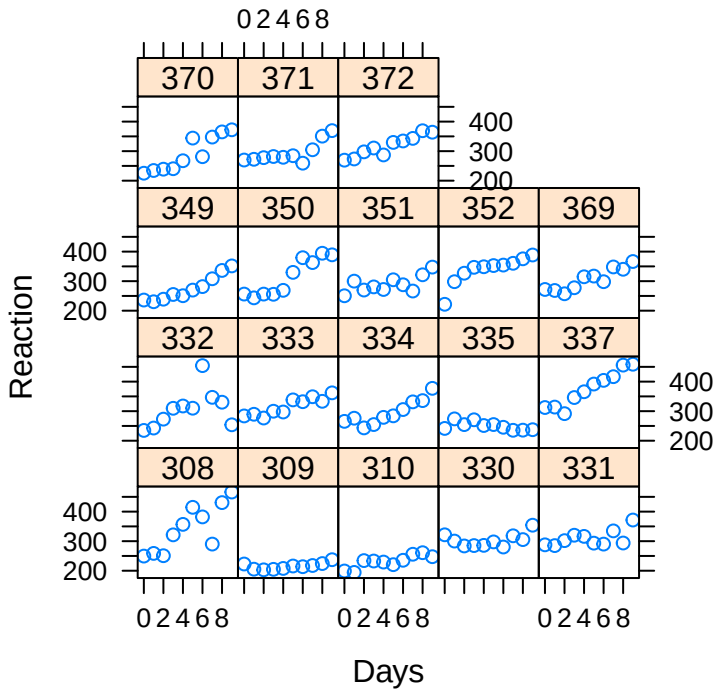
This particular example actually exhibits zero inflation, which can be assessed using a variety of packages. For example, one can use a specific function (zeroinfl) from the pscl package), or keep essentially the same set up as glm using vglm (with *zipoisson*' family) from the VGAM package.

LME example

The following shows an example of a mixed model using the lme4 package and associated lmer function. In this case, it is not just the addition of an argument that is required[28], but a different formula specification to note the random effects. Here is a model looking at reaction times over a period of 10 days in a sleep deprivation study.

[28] Using glmer, one would specify a family just like with the glm function.

```
library(lme4)
data(sleepstudy)
# lmList(Reaction ~ 1|Subject, sleepstudy) #subject means
xyplot(Reaction ~ Days | Subject, sleepstudy, lty = 1)  #Reaction time over days for each subject
```



```
lme_mod_1 <- lmer(Reaction ~ 1 + (1 | Subject), sleepstudy)
# random effect for subject (random intercepts)
lme_mod_1  #note how just printing the lmer model object provides the 'summary' functionality

## Linear mixed model fit by REML
## Formula: Reaction ~ 1 + (1 | Subject)
##    Data: sleepstudy
##   AIC  BIC logLik deviance REMLdev
##  1910 1920   -952     1911    1904
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  Subject  (Intercept) 1278     35.8
##  Residual             1959     44.3
## Number of obs: 180, groups: Subject, 18
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)   298.51       9.05      33

# summary(lme_mod_1)
lme_mod_2 <- lmer(Reaction ~ Days + (1 + Days | Subject), sleepstudy)  # 'growth model'
lme_mod_2

## Linear mixed model fit by REML
## Formula: Reaction ~ Days + (1 + Days | Subject)
##    Data: sleepstudy
##   AIC  BIC logLik deviance REMLdev
##  1756 1775   -872     1752    1744
## Random effects:
##  Groups   Name        Variance Std.Dev. Corr
##  Subject  (Intercept) 612.1    24.74
##           Days         35.1     5.92    0.066
##  Residual             654.9    25.59
## Number of obs: 180, groups: Subject, 18
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)   251.41       6.82    36.8
## Days           10.47       1.55     6.8
##
## Correlation of Fixed Effects:
##      (Intr)
## Days -0.138
```

Using Other Functions & Packages

With R there are often many ways via all the different packages and programming capabilities, each one doing something uniquely to the point where you might use one for one piece of output, another for some graphic etc. The trick though is to just get used to how things are done generally then move to those packages that fulfill specific needs, especially since most of them are are using some of the more basic functions internally.

There are some general packages and associated functions one can use to help ease the social scientist's transition from other commonly

used statistical packages. Examples include Zelig from political scientist Gary King and others. (along with the plm package for panel data), William Revelle's psych package, car and rms and MASS for general routines that are utilized in many social sciences. Those alone will get you through the analyses typically covered in graduate social science courses, but see the Task Views for Social Sciences to find more.

However, one of the fun things about R is gaining experience with techniques more widely used in a variety of other disciplines, and using R regularly will expose one to quite a lot more than they probably would be in packages that have a certain customer in mind (something impossible to do from R's approach). There are great offerings of packages from people from ecology, biology, computer science and so on, so as soon as you get comfortable feel free to explore as they will typically operate in the same way but with different function names and options.

Other Topics

The following section will have certain parts covered in the course time permitting. As the handout will always be available, this section may be added to indefinitely regardless of whether the topics are covered in the course.

Using the Parallel Package

Typical computers made in the past few years have a muti-processor/core setup, but for which much software is not making use of by default, and even mid-range computers today might have up to 8 processors available. This may not mean much to many applied researchers, but will when some analyses or operations end up taking several minutes or possibly hours or even longer with simulation studies. Even so, no sense waiting any longer than you have to in order to get your work done.

With version 2.14 R now has built-in support for parallel processing. This was previously accomplished via other packages such as snow or multicore, and those packages are still useful. However now one can use the parallel package that comes with base R to explicitly take control of computer resource utilization, though there will likely be many packages in the future which will do so implicitly and/or via an argument for a particular function[29].

The key is using this functionality with the 'apply' type of vectorized operations for which one normally does with loops in other statistical

[29] There are a functions in the plyr package with a parallel argument for example.

languages. Once you specify the number of processors to use, it is then merely a function of using new functions like parApply, parLappply etc. in the same way as you have used apply, lapply etc. in basic R programming to get an even faster turnaround on those iterative types of operations. This will make use of all processors on your machine that you specify. For example, a modern desktop pc with Windows[30] might have eight total processors available, so if we wanted to use seven of them for some analysis:

[30] All of this is available on other operating systems, but one will have to change or add some arguments.

```
library(parallel)
cl <- makeCluster(7)
```

...

```
stopCluster(cl)
```

Note the code of interest will go between the creation of the cluster and stopping it, and while there are more arguments one could specify in creating the cluster, the above would be fine in Windows.

In the first course we learned some about loops and iterative operations, and it is with these scenarios where making use of more of your computing power can become noticeable[31]. As an overly simple example, the following just adds 3 to each element of the numbers 1 through 20.

[31] Even on a standard desktop I've seen some operations that take several minutes whittled down to seconds.

```
add3 = function(x) {
    x + 3
}
parSapply(cl, 1:20, add3)
```

Note that more processors to work with doesn't automatically mean a speed gain. Some R functions simply aren't built that way yet (e.g. plots) and in a general sense if you can't feed data to the processors quickly enough you could even see a slowdown. However, if one uses the high performance computing resources typically found on the campuses of research universities, familiarity with the parallel package will be required to take full advantage of those services. Furthermore, for just a single machine the setup is relatively straightforward and takes little more than your normal usage of R.

Using Other Programs

In some cases one may want to do the bulk of their work within R but also use other programs for additional functionality. Typically other stats programs are not going to be as flexible as R, and you might be able to be more efficient by simply calling on other programs from within the R environment. This is especially the case with simulation

studies, where one may want to see how results change over a variety of parameter settings, or similarly, in exploratory endeavors where there may be many models to investigate.

One of the issues to recognize is that all programs have their unique quirks to overcome, so there isn't necessarily a one-size-fits-all approach you could use almost every time for every purpose. However, the concept is consistent, so understanding the basics will allow you to get on your way in such situations. It also helps to know what some R packages are doing when they are utilizing other programs, particularly if you may want to tweak some aspect or customize it for your own. To that end the example at the end of this document comes from Andres Martinez, our data management consultant, that shows how to call on Mplus from within R. Note also that at least in this case, there is a package specifically designed for this called MplusAutomation.

Conclusion

R has several strengths relative to other statistical packages. It makes data cleaning and exploration easier and more efficient. The number of analyses available via contributed packages is unmatched. Furthermore, because of its object-oriented nature, further extensions of analysis and functionality are made far easier. It is hoped that after this and the first set of notes it is fairly clear both what R has to offer and the sorts of things you can expect from it. That said, learning it will take time, perhaps a lot of it, and you'll never get to a point where it won't throw a curve ball at you here and there. However, the more you use it, the more tools you will rapidly add to your analytical toolbox, the more concepts that were fuzzy in the past will become clear, and the more confidence you will gain in your approach.

Best of luck in your endeavors!

Appendix

Demonstration using another program from within R.

Scenario

There are two type of people in a population. People in the first type have a relationship defined by Y=0.5+0.2x. People in the second type you have a relationship defined by Y=0.7x. By using a regression mixture model one assumes that the joint distribution of y and x is multivariate normal.

Question

How does the violation of the normality assumption affects parameter estimation?

Goal

Determine the effect nonnormality has on the parameter estimation/ class enumeration. Does non-normality occur in both classes?

Model

Type 1: Y1 = 05. + 0.2x + nonnormal error
    The first group has a regression weight of 0.2 and an intercept of 0.5 plus an error that is non-normally distributed.
Type 2: Y2 = 0.7x + normal error
    The second group has a regression weight of 0.7 plus an error that is normally distributed.

```
# This is an example of how to run Mplus from within R Source functions:
# datagen (to generate the data and save it in a file)
# mplusin (to build the Mplus input file and run analysis)
# collect (to collect results)
source("C:/csr/examples/r-mplus/ex01/r-mplus-functions.txt")
# Generate data and save it in a file using datagen function:
for (i in 1:3) {
    datagen(ng1 = 3000, ng2 = 3000, rep = i, flnm = "C:/csr/examples/r-mplus/ex01/data")
}
setwd("C:/Program Files/Mplus")  #the working directory usually is the location of mplus.exe
# Create Mplus input files and run analysis:
for (i in 1:3) {
    mplusin(infile = "C:/csr/examples/r-mplus/ex01/data", rep = i, saveloc = "C:/csr/examples/r-mplus/ex01/",
        mpinput = "input.inp")
    # call is a text file that will be piped into mplus.exe:
    call <- "input.inp"
    # out.fln is the ouptut file name'
    out.fln <- paste("C:/csr/examples/r-mplus/ex01/out", i, ".txt", sep = "")
    call <- rbind(call, out.fln)
    # source.fln is the file sourced by mplus.exe. It contains the information
    # in call
    source.fln <- paste("infile.txt")
    write(call, source.fln)
    # Call DOS from R:
    shell("Mplus < infile.txt")
    # In DOS, Mplus calls Mplus.exe The less than (<) in the DOS window pipes
    # what the Mplus.exe is asking for Mplus asks questions infile.txt
    # contains the answer to those questions Some executable files only
    # require a file in a special format, so the less than sign is not needed
    # The structure of infile.txt naturally depends on the executable being
    # used Watch out if multiple simultaneous simulations are being run on the
    # same executable. It is better to copy the executable to the folder and
    # use one exectuable for each set of simulations. shell or system may
    # work, depending on how the executableis set
}
# Compile estimates into a csv file:
collect(est.loc = "C:/csr/examples/r-mplus/ex01/est", nruns = 3, sum.fln = "C:/csr/examples/r-mplus/ex01/results.csv")
collect(est.loc = "C:/csr/examples/r-mplus/ex01/est", nruns = 3, sum.fln = "C:/csr/examples/r-mplus/ex01/results.txt",
    sum.type = ".txt")
# est.loc is the location of the file names without the replication number
# nruns is the number of replications sum.fln is the file name to which
# the estimates will be stored sum.type is the file type that sum.fln is
# (here '.csv' and '.txt')
```

The file above sources "C:/csr/examples/r-mplus/ex01/r-mplusfunctions.txt" in the first line, which starts on the next page. You'd want all the following functions in that r-mplus-functions.txt file.

```
#This is an example of how to run Mplus from within R
#Some notes on Mplus:
#Requires an input file (.inp)
#May require a data file (.dat)
#Example 01
#Generate data for 2 groups with an effect size of 0.2/0.7 and a main effect of 0.5
#The combined data are fit by a simple linear regression
##########
#Generate data
#Function to generate the data and save it in a file:
datagen <- function(ng1, ng2, rep, flnm) {
  #ng1 is the number of observations in group 1
  #ng2 is the number of observations in group 2
  #rep is the replication number
  #flnm is the file name of the dataset
  #initialize the data matrix for each group:
  dat1 <- matrix(NA, ncol=2, nrow=ng1)
  dat2 <- matrix(NA, ncol=2, nrow=ng2)
  #generate x for each group:
  dat1[,1] <- rnorm(ng1)
  dat2[,1] <- rnorm(ng2)
  #generate y for each group:
  dat1[,2] <- dat1[,1]*0.2 + rnorm(ng1, sd=0.98)
  dat2[,2] <- 0.5 + dat2[,1]*0.7 + rnorm(ng2, sd=0.714)
  #combine group1 and group2 data:
  dat <- rbind(dat1, dat2)
  #determine file name:
  file.str <- paste(flnm, rep, ".txt", sep="")
  #flnm determines the file name, including location
  #(e.g. flnm="C:/csr/examples/r-mplus/ex01")
  #Note R uses forward slash instead of backward slash (which Windows uses)
  #rep determines the repetition (e.g. rep=1)
  #".txt" determines the file extension
  #sep="" determine how the different objects are separated
  write.table(dat, file.str, row.names=F, col.names=F)
}
##########
#Build Mplus input file
#Function to build the Mplus input file (also called the control file) to run a regression mixture for two classes:
#If building this function from scratch it is usually best to:
#Start from a working control file
#Paste into R and modify as needed (in R)
mplusin <- function(infile, rep, saveloc, mpinput) {
  #infile is the data file to be analzyed
  #rep is the replication number
  #saveloc is the file location to which the estimates will be written
  #mpinput is the file location to which the Mplus input file will be written
  #define the name of the input file:
  mpmat <- 'title: latent class model assuming cross-sectional data;'
  file <- paste(infile,rep,'.txt',sep='')
  mpmat <- rbind(mpmat, paste('data: file is ', file, ';',sep=''))
  mpmat <- rbind(mpmat, 'variable:')
  mpmat <- rbind(mpmat, '')
  mpmat <- rbind(mpmat, 'names are x y; ')
  mpmat <- rbind(mpmat, 'classes=c(2);')
  mpmat <- rbind(mpmat, '')
  mpmat <- rbind(mpmat, 'usevariables are ')
  mpmat <- rbind(mpmat, 'x y;')
  mpmat <- rbind(mpmat, '')
  mpmat <- rbind(mpmat, '')
  mpmat <- rbind(mpmat, 'analysis: type=mixture;')
```

```
  mpmat <- rbind(mpmat, ' estimator=mlr;')
  mpmat <- rbind(mpmat, '')
  mpmat <- rbind(mpmat, 'model: ')
  mpmat <- rbind(mpmat, '%overall%')
  mpmat <- rbind(mpmat, 'y on x;')
  mpmat <- rbind(mpmat, '%c#1%')
  mpmat <- rbind(mpmat, 'y on x;')
  mpmat <- rbind(mpmat, 'y;')
  mpmat <- rbind(mpmat, '%c#2%')
  mpmat <- rbind(mpmat, 'y on x;')
  mpmat <- rbind(mpmat, 'y;')
  mpmat <- rbind(mpmat, paste('Savedata: results are',saveloc,'est', rep, '.txt;',sep=''))
  write(mpmat,mpinput)
}
##########
#Collecting the results
#Function to collect the results:
collect <- function(est.loc, nruns, sum.fln, sum.type='.csv') {
  #est.loc is the location of the file names without the replication number
  #(e.g., "C:/csr/examples/r-mplus/ex01")
  #nruns is the number of replications
  #sum.fln is the file name (.txt or .csv) to which the estimates will be stored
  #sum.type is the file type of sum.fln (default is .csv)
  results <- NULL #initialize the vector
  for (i in 1:nruns) {
    est.fln <- paste(est.loc, i, ".txt", sep="")
    #not all rows are not of the same length so it is better to read in one line at a time
    #parameter estimates are in line 1, the standard errors are in line 2, model fit is in line 3
    pars <- read.table(est.fln, nrow=1) #the first row contains the parameter estimates, the number of columns may change
    #read.table does not work well when the number of columns is not constant
    ses <- read.table(est.fln, nrow=1, skip=1) #skip the first row, read the next row, which are the standard errors
    model.fit <- read.table(est.fln, nrow=1, skip=2) #skip the first 2 rows, read the next row, which are the model fit statistics
    #Collect the class 1 and class 2 parameters:
    class1 <- c(pars[[1]], pars[[2]], pars[[3]])
    class2 <- c(pars[[4]], pars[[5]], pars[[6]])
    se.class1 <- c(ses[[1]], ses[[2]], ses[[3]])
    se.class2 <- c(ses[[4]], ses[[5]], ses[[6]])
    #Separates the high and low class based on slope (as generated):
    #if the slope for class 1 is smaller than the slope of class2, the EST will have class1
    #parameters first and then class 2 parameters; for each the mean is at the end
    ifelse(class1[2]<class2[2], EST<-c(class1,class2,pars[[7]]), EST<-c(class2,class1,pars[[7]]))
    ifelse(class1[2]<class2[2], SE<-c(se.class1,se.class2,ses[[7]]),
           SE<-c(se.class2,se.class1,ses[[7]]))
    #collect the need output information
    #i is the dataset number
    res <- c(i, EST, SE, model.fit)
    results <- rbind(results, res)
    }
  #assign colnames
  colnames(results) <- c("rep", "c1 Int","c1 Slope","c1 res","c2 int","c2 slope",
                         "c2 resid", "mean", "se c1 Int","se c1 Slope","se c1 resid",
                         "se c2 int", "se c2 Slope","se c2 resid", "se mean",
                         "LL", "LLC", "Free", "AIC", "BIC", "ADJ BIC", "Entropy")
  #write results depending on file type:
  if (sum.type=='.txt'){write.table(results, sum.fln, row.names=F)}
  if (sum.type=='.csv'){write.csv(results, sum.fln, row.names=F)}
}
```

```
# Determine what estimate files are available useful when simulations do
# not converge
estgen <- function(location) {
    setwd(location)
    x <- as.matrix(shell("dir est*.*", intern = T))
    x <- as.matrix(x[6:(dim(x)[1] - 2), ])
    splt1 <- strsplit(c(x), "est")
    x1 <- sapply(splt1, "[", 2)
    splt2 <- strsplit(c(x1), ".txt")
    x2 <- sort(as.numeric(sapply(splt2, "[", 1)))
    return(x2)
}
```