R Bootcamp

Sunday, September 21, 2008

Outline

- 9-10am "Breakfast"
- 10am-1pm R Basics, Part 1
- 1-2:15pm Lunch with Kekona
- 2:15-5pm R Basics, Part 2 and more cool stuff.

We will be emphasizing what you need to know to become familiar with *R*.

You may feel overwhelmed, but it is OK. Once you have heard this stuff once, you will be able to pick it up again in your classes.

You will revisit the more advanced stuff (mentioned briefly here) in Statistics 202A.

Your Host Today
Ryan Rosario
ryan@stat.ucla.edu

- Third year Ph.D. student (M.S. student in Computer Science).
- Specializing in web data mining, (social) network analysis/ complexity, machine learning, computing and other applications to Computer Science.
- Advisor: Jan de Leeuw
- Graduated from *UCLA*, 2006, Mathematics of Computation, Statistics, B.S.
- Born and raised in Southern California (Thousand Oaks, CA).

¿Y tú? Now it's Your Turn!

- Your degree objective: Ph.D. or M.S.
- Your previous institution(s).
- Your previous major(s).
- Your research interests (if you know).
- Your advisor (only a few of you have one already)
- Hometown/State/Country

What is *R*?

- "*R* is a language and environment for *statistical computing* and graphics" developed by John Chambers and friends.
- "*R* provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is ***highly extensible***."

Why *R*?

- It can be installed on Mac, Windows, Linux (SAS cannot run on Mac), Sun.
- Simple, yet professional graphics (…with practice)
- Its command-line model is very flexible and very extensible.
- It is a **huge** project with a lot of contributed packages.
- It can interface with other open-source and commercial software.
- All of this is FREE.

Why *R*?

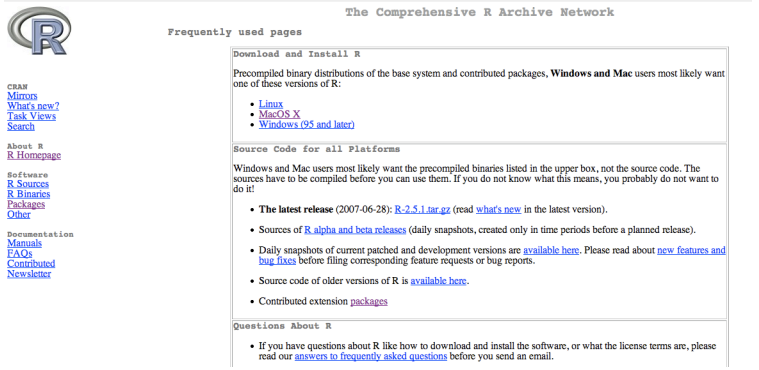
- widely used by Statistics departments all over the world, and usage is growing daily.
- can interface with faster languages like C and FORTRAN, and faster tools like MATLAB.
- Oh yeah, industry is beginning to recognize it as a required skill (read "must know MATLAB, *S* or *R*")
- Other universities (UC Santa Barbara and the university in South Central LA) use *S*+.
- Most *S* code can be run in *R* and vice-verse.
- It is FREE

Why *R*?

- Common trend here (and elsewhere)
  - We are pretty big on open-source software:
    - researchers contribute and benefit
    - easy access to enhance functionality
    - continually updated (sometimes daily)
    - free
  - We do not use Stata, SAS, SPSS, MATLAB, Mathematica (in general).
  - Other open-source projects we have used:
    - GRASS GIS, SAGE, GRETL, Python, MySQL, Processing, Octave, LaTeX, NeoOffice.

How to get R at Home

- Visit **`http://cran.r-project.org`**, or the UCLA mirror at **`http://cran.stat.ucla.edu`**



Your Statistics Account

Your Department account (login/password) allows you to:

- login to (nearly) all Mac systems and servers and use software on the computer.
- store your files and have access to them from any system, or from home.
- make a web page and access email.
- print
- access the Department network from off-campus (VPN)
- José will talk more about this later in the week.

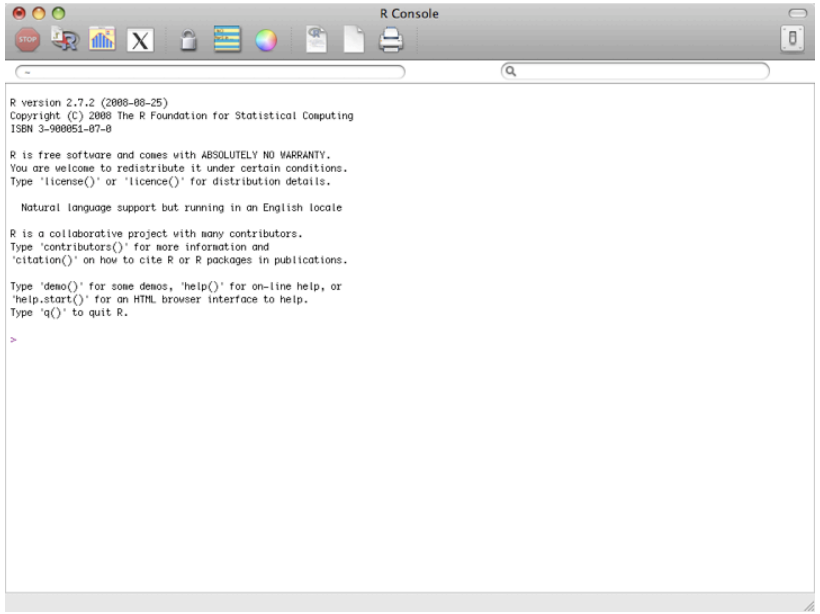
A First Time for Everything

There are two ways to start *R*

1. (Best!) Click on the *R* icon in the dock:

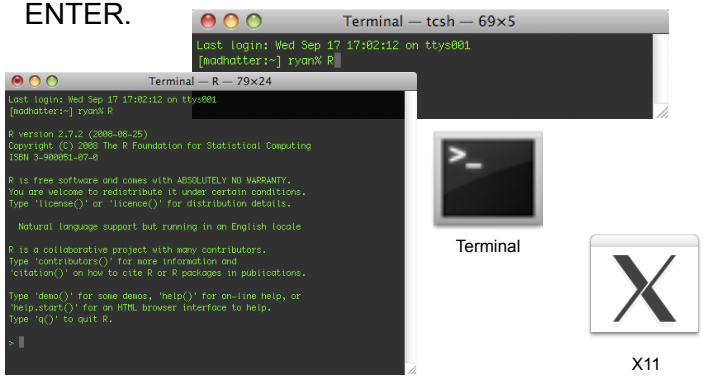
The Dock is the strip at the bottom of the screen with a bunch of icons.

<u>Only works if you are sitting at a machine.</u>



A First Time for Everything

2. From the Terminal (or X11) on your **local account**, or from the **computing servers**, type `R` and hit ENTER.



   – The servers are used for 202A, not now.

Tangent: What are these Servers?

Statistics has 3 servers available for computing (R, Python etc.) if you need a dedicated machine:

`neolab.stat.ucla.edu`
`lab-compute.stat.ucla.edu`
`compute.stat.ucla.edu`

Use your network username/password to login to them.

NEVER use **`login.stat.ucla.edu`** or your *network account* for intense computations. **ALWAYS** use a *local account*, or a computing server.

A First Time for Everything

The > symbol is the R command prompt. First off, *R* can be used as a bloated calculator.

```
> 5+7
[1] 12

> log(exp(sqrt(sin(exp(pi)))))
[1] 1.106619e-08
```

The basic arithmetic operators are +, -, * and /. To get the remainder of dividing a by b, use the modulo, which is %%. Division using / is standard division, not integer division.

Each output is a scalar: a 1 x 1 vector. The [1] means that what is printed is the first element of the returned vector.

It is important to note that arithmetic is *elementwise* when working with vectors and matrices.

Order of Operations

"When in doubt, use parentheses!" `()`

Parentheses clarify precedence in expressions.

$$\frac{4}{\frac{5}{2}} = \frac{8}{5}$$

```
> 4/5/2
[1] 0.4

> 4/(5/2)
[1] 1.6
```


Order of Operations

(1) **P**arentheses `()`
(2) **E**xponents `^` `**`
(3) **M**ultiplication `*`
(4) **D**ivision `/`
(5) **A**ddition `+`
(6) **S**ubtraction `-`

PEMDAS: **P**lease **E**xcuse **M**y **D**ear **A**unt **S**ally

Uh Oh: Unbalanced `()`

If you use open a term with `(`, you must close it with `)`. If you don't, *R* won't know what to do; *R* will not give an error.

```
> 2+(5/(2/4)
+
```

The + sign on the next line means that R is waiting for more input. *It usually means you are missing a `)`, `]`, or `}` (we'll get to the last two later).* If you hit enter again, another + will appear. Argh! To break out of this, hit ESC followed by ENTER.

Warning: *R* will automatically add a closing parenthesis, bracket, or quote when you type the opening in the Mac GUI.

Variables and Assignments

In order to something with a calculation, we need to be able to reference it later. We want to store the output (***value***) of an ***expression*** into a ***variable***. This is called ***assignment***. We assign a value to a variable using `<-`, or `=`.

```
> x = sqrt(25)
> x
[1] 5
> y <- 4
> z <- x^2 + y^2
> x^2 + y^2 -> z
```

Assignment can also use = instead of <-, but <- is preferred. Assignment can also be done the other way -> but it is not recommended.

Variable names must start with a letter and cannot contain the colon (`:`), semicolon (`;`), hyphen (`-`) or comma (`,`).

Objects

A ***variable*** is a type of ***object*** in *R*. Basically, everything in R is an object. (*R* is object-oriented for you CS people.)

When we store an object in *R*, it is stored in your ***workspace***. The workspace is like a virtual folder that only R knows about.

We can display what is currently in the workspace using the functions **`objects()`** or **`ls()`**. They are both identical.

```
> ls()
[1] "x" "y" "z"
```

The output from **`ls()`** is actually a vector of length 3 and can be assigned to a variable.

Removing Objects

To remove a variable from the workspace, use the functions **`remove()`** or **`rm()`**.

If you are a Unix user, you should notice a pattern…

```
> remove(x)
> ls()
[1] "y" "z"
> rm(z)
> objects()
[1] "y"
```


Clearing the Workspace

Sometimes we want to remove ALL of the objects in a workspace. Sometimes this is a good debugging step.

The `remove()` and `rm()` functions take a vector of object names and removes all objects that have names stored in that vector.

```
rm(list=ls())
```

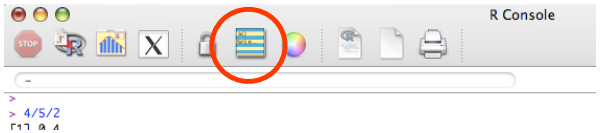
Remember that `ls()` returns a vector. Here I pass that vector into the function as a parameter. `list=` tells `rm()` that I am passing it a vector of names (that it calls list) instead of a single object name. list is the name of the parameter that we are passing to `rm`. ***This is not to be confused with a list data type!***

Reviewing Previous Commands

1. You can use the UP and DOWN arrows on the keyboard to scroll through your previous commands, one at a time. This works in the GUI, and is *supposed* to work in the Terminal.
2. You can display the history by typing `history()`

   -OR-

   If you are using the GUI version of R (not using the terminal) then you can display your entire command history by clicking on the circled button below.



Reviewing Previous Commands

Before quitting *R* today, you will want to save your command history as a reference.

```
savehistory(file="history.log")
```

If we only include a filename (like above), the file is saved in the ***current working directory***.

R routinely saves your history into the file `.RHistory`. This is how *R* is able to display your history on the screen.

The Working Directory

If we specify a filename without a path, like `myfile.txt`, *R* will look in the current working directory for the file. This is true for both importing files and exporting files.

To display the current working directory, type `getwd()`.

```
> getwd()
[1] "/Users/ryan"
```

If I write a PDF file, save my history, or try to read data using only the filename etc., files will be read from or written to the current working directory, `/Users/ryan`

The Working Directory

I do not want files read from or written to my home directory, so I change it to something else, say `/Users/ryan/Desktop/R_bootcamp`

To change the current working directory, use `setwd()`.

```
> setwd("/Users/ryan/Desktop/R_bootcamp")
```

The new directory must already exist for this to work.

You should use a separate directory for each assignment or project!!!

The Working Directory

You can override the use of the working directory by passing the full path to the file you want to read/write.

```
save(x,file="/Users/ryan/Documents/x.Rdata")
```

Will save the file `x.Rdata` in the directory `/Users/ryan/Documents/`, NOT in the current working directory.

The Working Directory

Finally, we can list the files in the current directory by using the command `dir()`.

```
> dir()
[1] "history.png"    "thermal.R"
```

Unix buffs take note! In *R* `ls()` lists the objects in the workspace, `dir()` lists the files in the current working directory.

R Sessions

Each time we start *R,* we start a new R session. When we quit R, the session is destroyed (a geeky term).

When we quit R, we are asked whether or not we want to save the current workspace. You should answer "yes," however, there is a better way to save your workspace that ensures that it is indeed saved. We will visit this in a bit. If you save your workspace using this default method, it is stored as the file `.RData` in **the directory in which R was called from.**

When we enter R again, this default workspace will be restored.

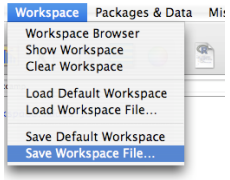
Do not quit R right now.

Saving the Workspace

Saving your workspace allows you to store all of your work in one nice little package. You can then take this workspace file with you to another computer and work on it without having to reimport all of your data and run any code again.

```
save.image(file="myworkspace.Rdata")
```

IMPORTANT: While R is running, everything is stored in memory. If R crashes, you lose your work. So save your work, and save it often.



We can reload the workspace by using the load function

```
load("myworkspace.Rdata")
```

The variables are restored using the same name they were saved as.

Saving the Workspace

Here is proof that this works…

```
> save.image(file="myworkspace.Rdata")
> ls()
[1] "x" "y" "z"
> rm(list=ls())
> ls()
character(0)
> load("myworkspace.Rdata")
> ls()
[1] "x" "y" "z"
```

Saving Objects

We can also save just certain objects. This is good to use when regenerating the object takes a long time or is a pain to do (parameter estimates from a simulation perhaps).

```
save(x,file="x.Rdata")
```

The first parameter is the object that you want to save. The second is the name of the file that we want to save the object to.

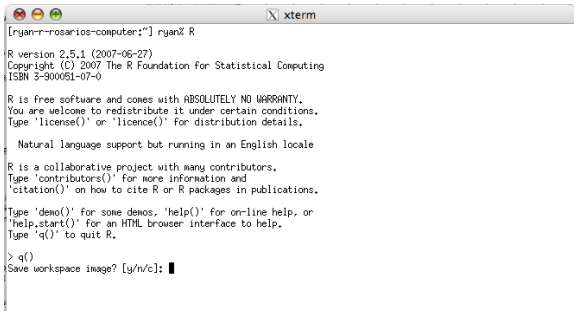
We can load the object by using the load function

```
load("x.Rdata")
```

The variable is restored using the same name it was saved as.

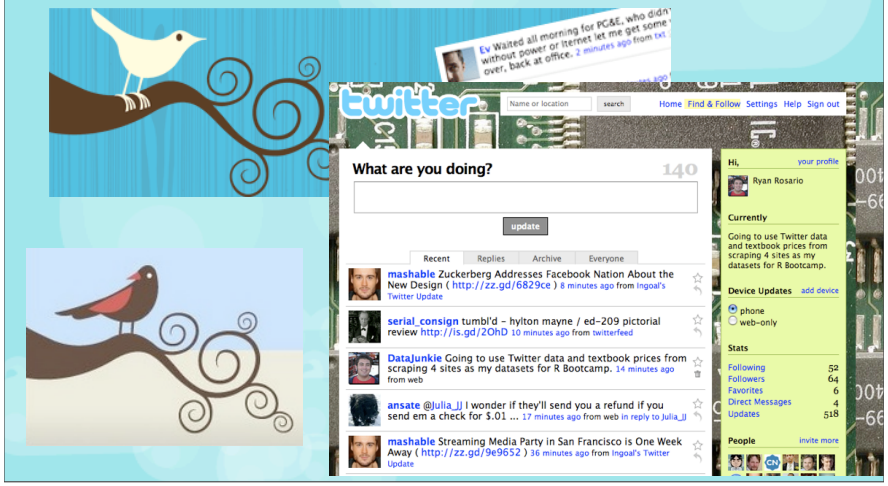
Ending the Fun: Quitting R

To quit R, type `q()`.



You are asked if you want to save your work. If you answer `y`, `save.image` is called. You are taken back to the Unix prompt, or the Finder.

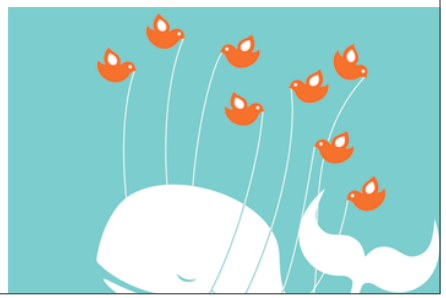
Our First Dataset



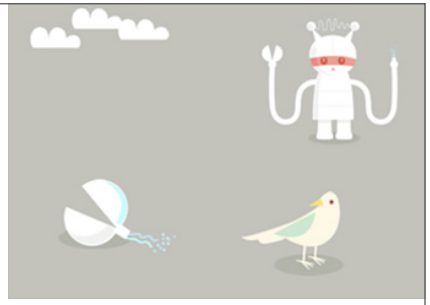
What is Twitter?

Twitter is a *microblogging* platform that allows users to broadcast status updates (called tweets) to the public. Tweets usually describe what a person is currently doing, thinking, or feeling. Sometimes users may engage in conversation with other users, or ask questions to the "Twitterverse." Others post updates about their websites or blogs.

All of a user's updates are displayed on the user's *timeline* and on the *public tiimeline.*



About Twitter



When user *A* encounters a user whose updates are interesting to him or her, *A follows* user *B*. That is, *A* is now subscribed to *B*'s updates. Each time *B* posts a tweet, the tweet will show up in *A*'s friend timeline. *A* will always be kept up to date on *B*'s whereabouts, doings, feelings, etc.

User *B* is notified of the new follower and can follow the person back, or just ignore them.

About Twitter

In the Twitterverse, it is customary to follow someone back if they follow you. That is, when *B* is notified of the new follower, it is typical for *B* to reciprocate and follow *A*.

Users can protect their updates so that only his or her followers can see his/her tweets. This ensures a level of privacy, but is not commonly used.

Spammers have started taking advantage of this culture on microblogging platforms.

Twitter Spam = Twam

Spammers commonly participate in a "mass following." They use web crawlers that visit thousands of random people. The spammer's hope is that these victims will reciprocate and follow the spammer. Once the spammer has a significant number of followers/victims, the spammer will flood the follower's timeline with annoying updates about irrelevant topics.

Twitter-educated users will visit the spammer's profile and not follow it or choose to block the spammer. Typically, the number of people that a spammer follows ("following") is much, much greater than the number of users that follow the spammer ("followers").

Twitter: What's the Point?

Blogs give a first person account of events and personal opinions and can sometimes be seen as more credible than mainstream media. Microblogs are similar but are much more immediate and directly to the point. They are easier to broadcast, and less time consuming to read.

Metrolink Crash on Twitter


Adriennevh: **metrolink** crash in CHatsworth
6 days ago · Reply · View Tweet


RodrigoMx: BREAKING -- A **Metrolink** train has overturned in Chatsworth, Cali. -- Many ambulances are being dispatched
6 days ago · Reply · View Tweet


Metrolink: Ventura Co. Line train 111 has been involved in a collision near the Chatsworth station. More info to follow.
6 days ago · Reply · View Tweet


KeriDeHerrera: Wow. **Metrolink** on fire in Chatsworth, CA. On tv right now. Can't find link yet online.
6 days ago · Reply · View Tweet


leftyshields: retweeting: @**Metrolink** "Ventura Co. Line train 111 has been involved in a collision near the Chatsworth station." :(Thats MY train
6 days ago · Reply · View Tweet


Metrolink: @LAFD the train collision incident involves **Metrolink**, not MTA. Media inq. should call PIO at (213) 452-6855
6 days ago · Reply · View Tweet

Metrolink Crash on Twitter


tylerdranguet: So I just heard the news about the **Metrolink** that's on fire. Thank god it wasn't our train.
6 days ago · Reply · View Tweet


DaveMora: Scary to know i take that **metrolink** route once a week to oxnard.
6 days ago · Reply · View Tweet


DaveMora: Trajic train train accident near my home. **Metrolink** vs freight train head on in chatsworth,california
6 days ago · Reply · View Tweet


Metrolink: Vent. Co. Line trains leaving Los Angeles are only operating as far as Van Nuys station at this time.
6 days ago · Reply · View Tweet


Metrolink: Vent. Co. Line train 111 has been involved in a collision with a freight train near the Chatsworth station.
6 days ago · Reply · View Tweet


tarync: Grandma was in the **Metrolink** accident.
6 days ago · Reply · View Tweet

Metrolink Crash on Twitter


Paul_Kuzma: Rob & Deo r safe from Chatsworth **Metrolink** Train crash. She would've been on it, but got off early today! Waiting 2 hear from @sbcollester.
6 days ago · Reply · View Tweet


trevorcarpenter: Kinda feeling weird tonight. A family friend was one of the victims in the tragic LA **metrolink** train crash yesterday.
5 days ago · Reply · View Tweet


britsilverstein: Reading that one of the valedictorians of my high school was one of the victims of the **Metrolink** Crash really is haunting. R.I.P. Atul.
4 days ago · Reply · View Tweet


zhuli: cmc 09 student (younger bro of a friend from cmc08) died in the la **metrolink** crash. Very upsetting, waiting for metronorth now, bad karma.
4 days ago · Reply · View Tweet


MissMotorMouth: **Metrolink** crash: Teen trainspotters quoted as experts in this tragedy now have made a tribute to the engineer responsible http://is.gd/2CgF (expand)
4 days ago · Reply · View Tweet


Adriennevh: I can't bring myself to watch news of the **metrolink** crash anymore now that I know Ron died there. Now it is personal.
4 days ago · Reply · View Tweet

Lifestreaming

Microblogging is part of a larger social media phenomenon called *lifestreaming*.

Lifestreaming is an aggregation of social information for display (and analysis). They are records of what you are doing in any sense of the phrase: Facebook feed, Twitter timeline, YouTube favorites, Pandora and Last.FM playlists and favorite songs. They all say unique things about you. Sites like FriendFeed can aggregate this information for you.

Internally, your search history on Google, your YouTube history and Amazon browsing and purchasing histories are also forms of lifestreaming used by companies for marketing research.

Loading the Data

There are several ways to load data into R. For now, we will keep it simple and import all of the data. Later on I will describe the other methods.

```
source("http://www.stat.ucla.edu/~rosario/boot08/twitter.R")
```

This file automatically creates all of the data files for you and dumps objects into your workspace. Use `ls()` to list these new objects. You should see the following

```
> source("http://www.stat.ucla.edu/~rosario/boot08/twitter.R")
> ls()
[1] "protected"      "sna.stats"      "spammer.victim"
[4] "usernames"
```


Introducing our Data

`usernames`: a vector containing the usernames for 5000 Twitter users.

```
> length(usernames)
[1] 5000
```

`sna.stats`: a data frame containing 2 variables where each row represents a user, and each variable represents each of the 4 statistics collected for the user.

```
> dim(sna.stats)  #returns dimensions of matrix.
[1] 5000    4
> ncol(sna.stats) #number of columns (one per statistic)
[1] 4
> nrow(sna.stats) #number of rows (one per user)
[1] 5000
```

(`dim`, `ncol` and `nrow` only work on `data.frame`s and matrices)

Introducing our Data

`spammer.victim`: a data frame containing two variables, one indicating whether or not the user is a known spammer, and the other indicating whether or not the user is a known victim of a spammer (some spammer follows this user).

`protected`: a vector of length 5000. Each entry in the vector indicates whether or not the user has "protected" their updates. If an update is protected, only his or her friends can see them. The public cannot.

In these two vectors, entries are either **`TRUE`** or **`FALSE`** (and in some cases **`NA`**).

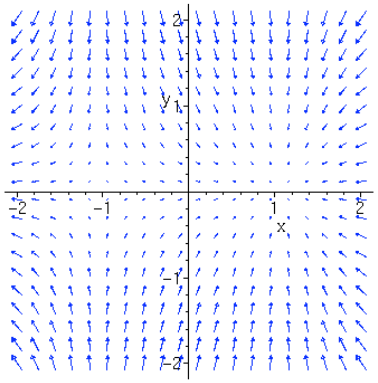
Objects

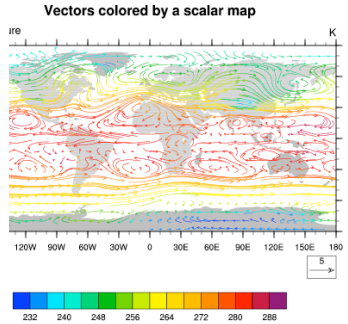
Everything in your workspace is an object. We can see what kind of object each is by using the **`class()`** function.

```
> class(usernames)
[1] "character"
> class(sna.stats)
[1] "data.frame"
> class(protected)
[1] "logical"
> class(spammer.victim)
[1] "data.frame"
```

`logical` and **`character`** are types of vectors. **`numeric`** is another common type.

...we will return to Twitter in a bit. First, let's learn more about *R*.





Vectors

Vectors

A ***vector*** is a collection of values of the same type grouped together in one container. In *R*, we have 3 basic value types:

`numeric`:	**real numbers**
`complex`:	complex numbers*
`integer`:	integers (subset of numeric)
`character`:	**individual characters, or strings**
`logical`:	**`TRUE` or `FALSE`**
`raw`:	raw bytes*

Vectors can contain the value **`NA`** for missing values.

Vectors consist of only one type of data; this type is called the **`mode`** of the vector and can be queried with the **`mode`** function.

Vectors

```
> mode(usernames)
[1] "character"

> mode(sna.stats)          #kind of works for data frames.
[1] "list"               #data.frame is a list of vectors.

> mode(protected)
[1] "logical"

> mode(spammer.victim)
[1] "list"

> mode(spammer.victim[,1])
[1] "logical"
```

Data Structures

As statisticians, we often work with rectangular data. That is, each row is an observation, and each column is a variable. The data structures in *R* reflect this practice.

`vectors` are ordered collections of primitive elements.

`matrices` and `arrays` are rectangular collections of primitive elements (same type) having 2 dimensions.

`factors` are categorical variables.

`data.frame` is a 2 dimensional data table consisting of columns whose elements are of the same type (a "dataset" if you will).

Displaying Vectors

When displaying a vector that has more elements than can be displayed on the screen, *R* will wrap the output to the next line a print a number in brackets `[]` indicating the index of the first element on that line.

```
> usernames[1:50]
 [1] "jeffrandall"    "SarahHotnites"  "bullit"         "toast737"       "bradnoble"
 [6] "rogercaplan"    "tomhamby"       "gandho"         "mshkh"          "readyAIMfire"
[11] "dugpark"        "wileyAU"        "Laura_Bunt"     "davekuo"        "sfearthquakes"
[16] "nexneo"         "jgordonduncan"  "DOTYUNG"        "cfanini"        "kimberlyaberry"
[21] "BoxWrench"      "amy_runner"     "torpedoman69"   "szafranek"      "Gina100189"
[26] "adamkaump"      "Vicfred"        "ezavl"          "absurdities"    "fabfinds"
[31] "tmntwg"         "Deansy"         "fangjie"        "ataboy"         "Hydrocoptic"
[36] "LilNdnPriya"    "TheTop40Charts" "michaeltwofish" "KurTing"        "tetumame"
[41] "chiraggupta"    "vespagrrl"      "matmjuarez"     "gabrielfiorini" "amcalpine"
[46] "thiagojbg"      "radiodemon"     "SouthernOracle" "everyplace"     "hurphendale"
```


Working with Vectors

Many types have have functions you can use to ask an object "are you of type…?"

```
> is.vector(usernames)
[1] TRUE
> is.vector(protected)
[1] TRUE
> is.vector(sna.stats)
[1] FALSE   ##it's a data.frame
```

This function is an example of one that returns a `logical`.

Aside: `mode` vs. `class`

Let `x` be a 5000x4 matrix.

`class(x)` returns an object's type:

```
> class(x)
[1] "matrix"      #or data.frame, numerical, etc.
```

`mode(x)` returns the data type of the elements contained as entries of `x`.

```
> mode(x)
[1] "numeric"     #or logical, numerical, character
```

Aside: Objects

We can find the length of a object which is simply the number of elements it contains. For vectors, this is simple. For other object types (particularly **matrix** and **data.frame**), the length may not return something useful.

```
> length(usernames)  #trivial for vector
[1] 5000

> length(x)          #matrix
[1] 20000            #returns mxn

> length(spammer.victim)  #data.frame
[1] 4             #returns number of variables.
```

All objects have a **length**, **class** and **mode**. For vectors, the **class** is just its **mode**.

Creating *Empty* Vectors

We can create an empty vector `v` several ways.

```
v <- NULL

v <- c()      #creates a concatenation of 0 values.

v <- vector(length=100,mode="numeric")
```

But, usually we create a vector from values.

Creating Vectors

We can create a vector by concatenating values using the `c()` function.

```
v <- c(1,2,3)          #numeric

v <-c("celery","lettuce","pepper") #character

v <- c(TRUE,TRUE,FALSE) #logical
```


Creating Vectors of Integers

Often vectors of integers are used to create indices into other vectors, or matrices. We can create integer vectors as follows

```
> v <- 1:100    #a vector containing integers 1-100.
> class(v)
[1] "numeric"

> v <- rep(1,5) #repeat 1 five times.
> v
[1] 1 1 1 1 1

> x <- 4:6       #integers 4 thru 6
> y = rep(x,1:3) #repeat elements of  x: 1, 2 and 3 times.
> y
[1] 4 5 5 6 6 6
```

Vectors of Mode `numeric`

We can also construct vectors of mode `numeric` using the `seq` (for sequence) function.

```
> x = seq(0,3,len=100)
> #sequence of 100 evenly spaced values from 0 to 3

> x = seq(0,10,by=2)
> #sequence of values from 0 to 10, difference of 2
> #between values

> x
[1] 0 2 4 6 8 10
```


More Fun with Sequences

We can build more advanced sequences that may be helpful. (**Honquan Xu** likes to use these)

```
> rep(9:11,3)   #repeat vector (9,10,11) three times.
[1] 9 10 11 9 10 11 9 10 11

> rep(9:11,c(3,3,3))    #Method a
> #Repeat each of the three elements three times.
[1] 9 9 9 10 10 10 11 11 11

> #Can also be written…       Method b
> rep(9:11,rep(3,3))
```

Do you see the connection between methods a and b?

Naming Elements of a Vector

We can name each element of a vector while creating it, or after creating it.

Naming Elements during Creation

```
> x <- c("A" = 4.0, "B" = 3.0, "C" = 2.0)
```

Naming Elements after Creation using the `names` Function

```
> x = c(4.0,3.0,2.0)
> names(x) <- c("A","B","C")

> names(x)
[1] "A" "B" "C"
```


Printing Vectors with Named Elements

R prints vectors containing named elements differently. Instead of printing the index (i.e. `[1]`), it prints the name of each element above the element.

```
> x <- 1:26
> names(x) <- letters
> x
 a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
```

(`letters` is a global vector containing the alphabet)

Operations on Vectors

Many operations in *R* work on entire vectors since the vector is the basic object in *R*.

Suppose each variable below is a vector of length 100.

```
> x <- runif(100)
> z <- sqrt(x)  #take sqrt of each element of x
> y <- x + 2*z + rep(c(1,2),50)
```


Recycle and Reuse

R will let you add two vectors of differing dimension. I don't recommend this, but it works…if used correctly. *R* will recycle the elements of the shorter vector to match the length of the longer vector.

```
> c(1,2,3) + 2    #no warning (3 is multiple of 1)
[1] 3 4 5

> c(100,200,300,400) + c(1,2,3)  #warning  below
[1] 101 202 303 401

Warning message:
longer object length
      is not a multiple of shorter object length in:
c(100, 200, 300, 400) + c(1, 2, 3)
```


Coercion

Vectors can only contain one element type. If we mix types, *R* will try to convert all of the elements to a common format. It is not recommended doing this intentionally.

```
> x <- c(1:3,"a")  #numbers converted to characters
[1] "1" "2" "3" "a"

#Can be useful, not recommended.
> c(TRUE,TRUE,FALSE)+3
[1] 4 4 3
```


Do-It-Yourself Type Casting

Rather than rely on coercion, we should type cast objects to a common type. Type casting just means converting a variable of one type to another (comparable) type.

```
> as.integer(2/4)  #Integer division
[1] 0
```

Let a correct answer on a multiple choice test be denoted **TRUE**, and let x be the vector of scored responses (logical)

```
> score <- sum(as.numeric(x)) #coercion is ok here.
```

Vector Functions

R lets us perform important data management operations on vectors like sorting and finding unique values in a vector.

```
> sort(x)
[1] -0.916999756 -0.497566155  0.003459213  0.191734263

> y <- c(1,1,2,2,2)
> unique(y)
[1] 1 2

> duplicated(y)
[1] FALSE TRUE FALSE TRUE TRUE
```


Concatenating Vectors

We know we can concatenate values using `c(…)` but we can also concatenate vectors with `c(…)`

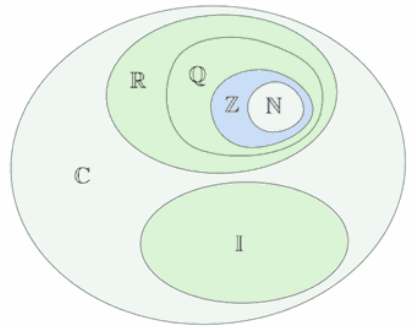
```
> x <- c(1,2,3)
> y <- c(4,5,6)
> z <- c(x,y)       #1 2 3 4 5 6
```

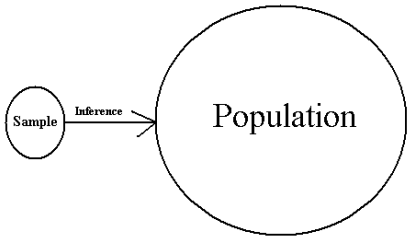
With coercion…

```
> x <- c(TRUE,TRUE,FALSE)
> y <- c(1,2,3)
> z <- c(x,y)        # 2 3 4
```

Append the value 4 to y…

```
> c(y,4)             # 1 2 3 4
```



Subsetting

Subsetting

It is unavoidable; we will always want to pick out certain values from a vector depending on some conditions. How to extract from a vector:

```
usernames[6]        # the sixth element (sixth user)
usernames[6:10]     # elements (users) 6 thru 10

usernames[c(1:5,8,11:15)]
      #elements 1 thru 5, and element 8, and 11 thru 15.
```

Called Indexing by Position

Subsetting

We can also exclude portions of a vector and keep everything else.

```
usernames[-6]       # everything but the sixth element
usernames[-c(6:10)] # everything except elements 6 thru 10
```

We can extract using **1:6**, but to exclude, we use **-c(1:6)**.

We cannot use both extraction and exclusion simultaneously.

```
letters[c(-10,11,43)]     #error
```

Called Exclusion by Position

The Five Rules of Subsetting

We use the brackets [ ] to subset. R has 5 rules for selecting certain elements from a vector.

1. Indexing by position — Done!
2. Indexing by exclusion — Done!
3. Indexing by name
4. Indexing with a logical mask
5. Empty subsetting

Subsetting: Indexing by Name

Recall that we gave names to the elements of a vector.

```
> x <- 1:3
> names(x) <- c("a","b","c")
> x["b"]
b
2

> names(protected) <- usernames
> protected["Mobius"]
 Mobius
FALSE
```


Subsetting: Indexing with a Logical Mask

Using logical masks can make data extraction much faster. Life is difficult without them!!

```
> x <- 6:10
> x >= 7
[1] FALSE TRUE TRUE TRUE TRUE
```

Thus,

```
> x[x >= 7]    #"x such that x >= 7"
[1] 7 8 9 10
```

"The Logical Song": Logical Operators

We use logical operators in subsetting as well as with loops and control statements.

Logical operators include

`<  >  <=  >=  ==  !=`

"It's So Logical": Logical Operators

As humans, we are picky. We often want to select elements that meet more than one condition. Logical operators are the glue that makes this possible.

`&` AND must meet both conditions

```
> x <- 10
> x < 5 & x > 7        #FALSE
> x > 1 & x > 2        #TRUE
```

`|` OR must meet either condition

```
> x<5 | x==10          #TRUE
> x<5 | x>20           #FALSE
> x>5 | x>9            #TRUE
```


It's So Logical: Logical Operators

`!` NOT must *not* meet the condition

```
> x <- 10
> !(x < 5) & !(x > 7)     #FALSE
> !(x < 5) & (x > 7)      #TRUE
> !((x < 5) | (x > 7))    #TRUE

> x <- 3.3
> !is.na(x)    #testing for complete cases.
```

`all(v < 10)` returns **`TRUE`** if all elements of **`v`** are < 10.
`any(v < 10)` returns **`TRUE`** if any elements of **`v`** are < 10.

Are you a good `which` or a bad `which`?

`which` is your best friend! Pass a logical expression to `which` and it returns the indices of the vector where that condition is true.

```
> data <- c(100, 100, NA, 200, 254, NA, 14)
#data with missing values. To remove…
> complete.cases <- which(!is.na(data))
#Use result to subset the original data.

> complete.data <- data[complete.cases]
```

Use it!!!

A Clever Irony: Empty Subsetting

```
> x <- 1:5
> x[]
[1] 1 2 3 4 5
```

Seems pointless, right? Not so fast.

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

Bottom line: empty subsetting is used for blanket assignments.

Subsetting: Common Pitfall

```
> x["a","b"]
Error in x["a", "b"] : incorrect number of
dimensions
```

`x` is a vector! Vector has only 1 dimension. I think you meant:

```
> x[c("a","b"),]
```

```
> x[-"a"] Error in -"a" : Invalid argument
to unary operator
```

Can't do this with names. Need to use **`which`**.

Subsetting for Assignment

An analyst notes that the data from a flight data recorder has some inconsistencies. In particular, some inflight readings for altitude or velocity are 0, or even negative. When we plot these data, these readings show as spikes that obfuscate any trend in the data. Suppose we have data only for the time that the plane is flying (we know these parameters are > 0). Lets fix this…

```
> fdr.velocity<-c(550,600,0,600,600,650,0,-5,600,550,-5)
> fdr.velocity[fdr.velocity <= 0] <- NA
[1] 550 600 NA 600 600 650 NA NA 600 550 NA
```


Sampling

Given a vector (data, sequence, etc.) we can draw a sample from it using the `sample` function.

```
> x <- seq(1,10,0.5)
> sample(x,size=5,replace=TRUE)
[1] 1.0 4.0 7.5 1.5 2.0
```

`replace=TRUE` draws a sample with replacement.

Passing a vector to **`sample`** with no parameters just permutes the vector.

```
> presentation.order <- c("RYAN","MILES","KEKONA","ARIANA")
> sample(presentation.order)
[1] "KEKONA" "RYAN"   "ARIANA" "MILES"
```

Data Frames

A data frame looks like a matrix, but the columns can be of different types.

Data frames have 2 dimensions: the number of observations (rows), and the number of variables (columns).

We have a couple of data frames in this data. In a **`data.frame`**, each column is named and optionally, each row is named.

Data Frames

To display the names of the variables (columns) in a **`data.frame`**, use **`names`**.

```
> names(sna.stats)
[1] "Following" "Followers" "Favorites" "Updates"
```

Use the dollar sign **`$`** to access a particular column/variable in the **`data.frame`**. The result is a vector, so only one a one dimensional index is needed!

```
> sna.stats$Following     #prints all values of the variable.
> sna.stats$Following[1:10] #print the first 10 values.
```

Data frames are similar to matrices. We are getting there!

The Variables in `sna.stats`

Let *A* be the user in question.

`Following` the number of users that *A* follows.

`Followers` the number of users that follow *A*.

`Favorites` the number of tweets that *A* has marked as favorites.

`Updates` the number of updated *A* has sent since signing up.

Helpful Hint...

Commands can be stored in a text file called an R script. We can run the entire R script (say, "`foo.R`") by running the command

```
> source("foo.R")
```

The commands in the file will be run sequentially, and quietly, unless there is an error.

It can also be useful to copy/paste commands from this file onto the R command line.

Use **`#`** to denote comments. Use **`;`** to put more than one command on the same line.

Important: Never write R code in the R text editor. If R crashes, you lose your code! Always use a separate text editor!

Exercise 1

Let's now put this to use; many spammers on Twitter participate in a behavior "mass following." The concept is simple: a spammer follows thousands of users in hopes that those users (victims) naively follow the spammer back. Once a victim follows the spammer, he/she is subscribed to the spammer's updates. Once the spammer is followed by a significant number of users, he floods their timelines with spam.

Use the Following/Follower ratio to develop a way to detect a potential spammer.

Exercise 1

Use the Following/Follower ratio to develop a way to detect a potential spammer.

$$R = \frac{\text{Following}}{\text{Followers}}$$

What does the ratio mean?

If *R > 1*, the user follows more people than people follow the user.

If *R < 1*, the user is followed by more people than the user follows.

Exercise 1

1. First, create a new vector called **`ratio`**.

   Note: Watch out for missing values and division by 0!

2. Plot both a histogram and a boxplot to describe the distribution of **`ratio`**.

   Hint: Take the **`log`** of **`ratio`** instead.

Exercise 1

1. First, create a new vector called **`ratio`**.

   Note: Watch out for missing values and division by 0!

```
> ratio <- sna.stats$Following/sna.stats$Followers
```

But something is wrong. Let's take a look at some basic stats...

```
> summary(ratio)
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.       NA's
-9.900e-09  2.159e-01  4.458e-01  8.250e-01  7.407e-01  1.123e+02  1.000e+00
```

Exercise 1

I did something that other packages do. I coded missing values for **`Following`** using negative numbers.

I coded missing values for **`Followers`** using a huge number (**`9999999999`**).

```
> sna.stats$Following[sna.stats$Following < 0] <- NA
> sna.stats$Followers[sna.stats$Followers == 9999999999] <- NA
```

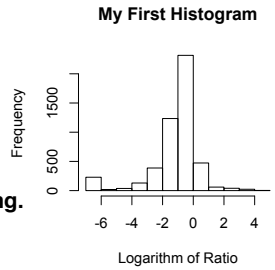

Exercise 1

2. Plot both a histogram and a boxplot to describe the distribution of **`ratio`**.

Hint: Take the **`log`** of **`ratio`** instead.

The **`hist`** command, well, makes a histogram. We can make a histogram by passing the variable name to the function.

```
> hist(log(ratio))
```



Note: add a small number to `ratio` before taking the logarithm, or you will get a warning.

So, how did I get the title and axis labels?

Exercise 1 Aside: Help System

Using the basic options in functions is fine and dandy, but if you want your graphics to look better, or your analysis more fine-tuned, you will need to take advantage of further options. Let's see what we can do with the **`hist`** function. Let's use the help system.

```
> ?hist
> help(hist)
```

Question mark followed by the name of a function brings up the help page for that function. The `help()` function does the same.

You can search the help pages using the **`help.search()`** function.

```
> help.search("poisson")
```


Exercise 1 Aside: Help System

From the help page, we see that the **`xlab`** parameter controls the text label of the x-axis and the **`ylab`** parameter controls the text label for the y-axis.

The **`main`** parameter controls the title of the plot.

```
> hist(log(ratio),xlab="Logarithm of Ratio",
+ ylab="Frequency",main="My First Histogram")
```

To see other graphical parameters, check out **`?par`**

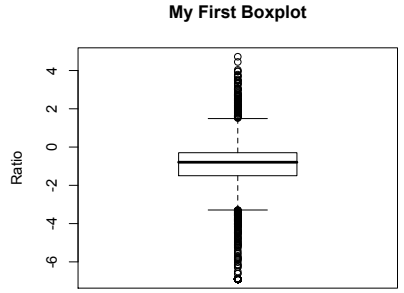
Some people love the help system (myself), others hate it.

Note: the code in these slides is how the code would be displayed on the screen. When you see +, that means the command, as displayed, is broken over multiple lines. DO NOT type the +!

Exercise 1:

We can create a boxplot of the ratio using the **`boxplot`** function.

```
> boxplot(log(ratio),ylab="Ratio")
```

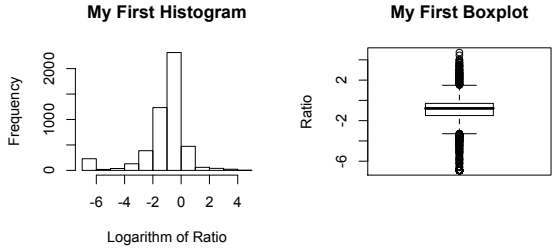


Note: I actually add 0.001 to ratio before taking the log to avoid errors.

Exercise 1: Multiple Plots

We can plot two plots in the same window. To plot both the histogram and the boxplot in the same window:

```
> par(mfrow=c(1,2)) #plot two plots rowwise
> hist(log(ratio),xlab="Logarithm of Ratio",
+ ylab="Frequency",main="My First Histogram")
> boxplot(log(ratio),ylab="Ratio")
```



Exercise 1: Multiple Plots

We can also plot the histogram and boxplot in two *separate* windows.

```
> quartz() #use on Mac
> x11()    #use on Mac, Windows, Linux
```

The graphic will be plotted in whichever window is currently active. To activate a display window, just click on it.

Tangent: Saving Graph for LaTeX

We can save this graph for inclusion in a LaTeX document one of two ways. We use `File > Save` and save the file as a PDF (Mac), or we can write a PDF from the command line

```
> pdf(file="myplot.pdf")          #create a PDF
> par(mfrow=c(1,2))               #plot two plots rowwise
> hist(log(ratio),xlab="Logarithm of Ratio",
+ ylab="Frequency",main="My First Histogram")
> boxplot(log(ratio),ylab="Ratio")
> dev.off()                       #turn off plotting to file.
```

(Brief Demonstration)

Tangent: Saving Graph for LaTeX

Then somewhere in the body of your LaTeX file, you would include the following. This assumes that the .tex file and the PDF file are in the same directory.

```
\begin{figure}[h]
\centering
%Play with the width parameter.
\includegraphics[width=6in]{myplot.pdf}
\end{figure}
```

That comes later in the week...

Exercise 1: On Your Own

3. Using your plots and summary statistics (**`summary`**), devise a method for detecting potential spammers based on following/ follower ratio. You can use other variables as well if you'd like!

How accurate is your method?

Which users are the most likely to be spammers? Do you agree with using just the ratio? What other variables would help?

Finishing Up

Note that we have several objects floating around. Let's combine all of these objects into one and save it to disk.

First, let's combine both data frames (sna.stats and spammer.victim) into <u>one</u> data frame.

```
> dfs <- cbind(sna.stats,spammer.victim)
```

`cbind()` takes two (or more) vectors or data.frames and stacks them next to each as columns.

Finishing Up

(Similarly, **`rbind()`** takes 2 (or more) vectors or data frames and stacks them as rows)

Next, take the vectors **`protected`** and **`ratio`** and slap them onto the newly created data frame.

```
> my.first.dataset <- cbind(dfs,protected,ratio)
```

Finishing Up

Let's use the **usernames** vector to name the rows of our new data frame.

```
> row.names(my.first.dataset) <- usernames
```

Finally, assign names to the columns:

```
> names(my.first.dataset) <- c("Following","Followers",
+ "Favorites","Updates","Spammer","Victim","Protected"
+ "Ratio")
```

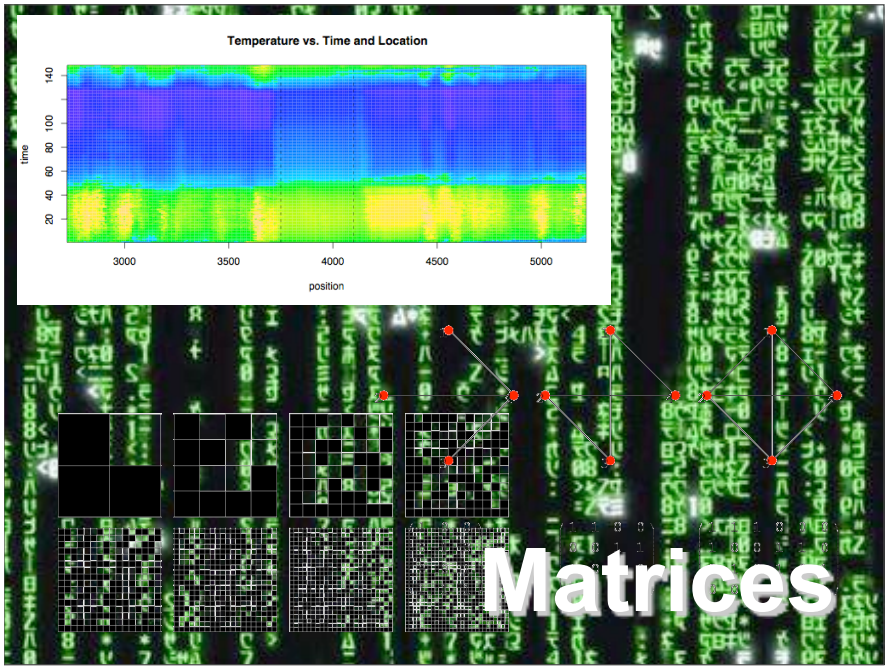

Finishing Up

Now we can save our dataset as a binary file that can later be loaded into R:

```
> save(my.first.dataset,file="mydata.Rdata")
```

OR, we can write our dataset out to disk as an ASCII file using comma-separated values, or some other format:

```
> write.table(my.first.dataset,file="mytable.txt",
+ sep="|")    #Uses pipe as delimiter
> write.csv(my.first.dataset,file="mytable.csv")
  #Uses comma as delimiter (wrapper function)
```

A Twitter Matrix

Now we will take a look at a matrix describing a time series representing the number of updates posted by each user. This data is stored as an R object so we use the **load** function.

```
> load(url("http://www.stat.ucla.edu/~rosario/boot08/data/tweetcounts.Rdata"))
```

The matrix **counts** has dimensions 7522 x 61. Each row represents a user, and each column represents a particular day. Days range from 0 to 60 where 0 represents 4/1/08 and 61 represents 5/31/08.

Let's also load the names of the users that each row represents.

```
> users <- read.table("http://www.stat.ucla.edu/~rosario/boot08/data/
        tweetcountsusers.dat",as.is=TRUE)
```


Matrices

Matrices consist of two dimensions and can be much more powerful than just vectors.

Like vectors, all elements of a matrix must be of the same type. When we print a matrix, R prints column indices and row indices.

```
> A
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```


Creating Matrices from Scratch

We use the **matrix** constructor to create a matrix.

```
#Matrix of random uniform values: 5 rows, 2 columns
A <- matrix(runif(10),5,2)
> A
          [,1]      [,2]
[1,] 0.3778161 0.1205188
[2,] 0.2967549 0.6951926
[3,] 0.4215449 0.2569569
[4,] 0.9628448 0.4448082
[5,] 0.1582352 0.1582504
```


Creating Matrices from Scratch

We don't have to specify both the number of rows *and* columns. We can leave one out, but remember the order that R is expecting the parameters to be in…

```
> A <- matrix(rnorm(10),2) #create a matrix with 2 rows
                           #R figures it has 5 columns

> B <- matrix(rnorm(10),ncol=2)
#create a matrix with 2 cols, R figures out it has 5 rows
```

R expects to see the parameter `nrow` first, so if we leave it out, we must specify `ncol=`

Creating Matrices from Vectors

We can glue together vectors rowwise or columnwise to create a matrix.

```
> x <- c(1,0)
> y <- c(2,2)
> I <- cbind(x,y)
> I
     x y
[1,] 1 2
[2,] 0 2
```

```
> I <- rbind(x,y)
> I
     x y
[1,] 1 0
[2,] 2 2
```


Creating Matrices from Scratch (again)

Suppose we want to hardcode the matrix $\begin{pmatrix} 1 & 1 \\ 2 & 4 \end{pmatrix}$

We can create a matrix from a vector, but it is important to note that R will interpret the values as *columns* unless we specify `byrow=TRUE`.

```
matrix(c(1,1,2,4),2)
     [,1] [,2]
[1,]    1    2
[2,]    1    4
```

Wrong!

```
matrix(c(1,1,2,4),2,byrow=TRUE)
     [,1] [,2]
[1,]    1    1
[2,]    2    4
```

Correct!

R Cool Trick 1,063,199

Oh yeah, and *R* is cool with abbreviating parameter names as long as it can *resolve* what you mean.

```
> A <- matrix(rnorm(10),nr=2,nc=5)
```

This would not work if there there were other parameter names starting with `nr` or `nc`.

Reshaping Matrices

We can change the shape of a matrix, or coerce a vector into a matrix.

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

Subsetting Matrices

Subsetting with matrices is similar to subsetting with vectors except now we have two dimensions (row and column) instead of one.

The `counts` matrix has as rows users, and columns days.

```
> counts[1,]      #First row (user) of counts.
> counts[,1]      #First column (day) of counts.

#upper left corner of counts: first 5 rows,
#first 5 columns
> counts[1:5,1:5]
```

The same subsetting methods can be used with data frames, but it is more convenient to use `$`.

A Bad `which`

Which does not behave as expected on matrices. It treats a matrix like a vector and returns an index in the columns of the matrix.

```
x <- matrix(c(2,4,6,8,10,12,14,16),2)
> which(x == 6)
[1] 3
```

This is kind of useless.

```
> matrix.col<- ceiling(which(x==6)/nrow(x))     #2
> matrix.row <- nrow(x)-which(x==6)%%nrow(x)    #1
```


Naming Rows and Columns of Matrices

We assign names to the rows and columns of a matrix as follows. Note the difference with vectors.

```
> x <- matrix(letters[1:10],5,2)
> rownames(x) <-
+     c("roberts","specter","kennedy","schumer","hatch")

> colnames(x) <- c("var1","var2")
> x
        var1 var2
roberts "a"  "f"
specter "b"  "g"
kennedy "c"  "h"
schumer "d"  "i"
hatch   "e"  "j"
```


Naming Rows and Columns of Matrices

We can assign names to the rows and columns to the object `sna.stats` for our Twitter exercise.

```
> rownames(counts) <- users[,1]
#The vector is actually read in as a data.frame *sigh*
```

If we print out the first 5 rows, we see the following:

```
> counts[1:5,1:5]
           [,1] [,2] [,3] [,4] [,5]
08NTC         1    0    0    0    0
0KissKiss0    0    1    0    0    0
0boy          0    0    1    1    1
1000000       0    0    0    0    0
1000Monkeys   0    3    2    1    0
```

Naming Rows and Columns of Matrices

Eww. What is that `[,1]`, `[,2]` all about?

By default, *R* assigned names to the columns of our matrix to coincide with their column indices. Let's change them by assigning a vector of character strings to `colnames(counts)`.

```
> colnames(counts) <- seq(0, 60)
```

```
> counts[1:5,1:5]
           0 1 2 3 4
08NTC      1 0 0 0 0
0KissKiss0 0 1 0 0 0
0boy       0 0 1 1 1
1000000    0 0 0 0 0
1000Monkeys 0 3 2 1 0
```


Operations

As with vectors, R can operate on entire matrices at once.

There are some special functions that are only valid with matrices.

```
> x%*%y                    #matrix multiplication
> W = t(x)                 #Transpose
> Xinv <- solve(x)         #Invert
> D <- diag(x)             #Extract diag. elements
> D <- diag(1,10,5)
                          #Create 10x5 matrix, 1s on diagonal.
> eigen(x)                 #eigenvalue decomposition
```


Matrices: `apply`

Finally, an operator that makes life a lot easier...

```
> x = matrix(1:50,10)
> dim(x)
[1] 10 5
> apply(x,1,sum)
[1] 105 110 115 120 125 130 135 140 145 150
> apply(x,2,mean)
[1]  5.5 15.5 25.5 35.5 45.5
```


Matrices: `apply`

The function `apply` allows us to operate on rows or columns of a matrix.

In this case, we have taken the data in each column and formed an average with the built-in function `mean`.

The "2" in this call tells R to operate on the data in a column; if we had used "1", it would apply the `mean` function to each row.

In general, `apply` is a kind of vectorized operation, except in this case it works on entire rows or columns of a matrix (actually, it works on arrays, but we're not there yet).

Matrices: **apply**

There are various flavors of this function that work on other R data structures

`lapply`	for lists
`sapply`	similar to **`lapply`**
`tapply`	for ragged arrays

In some cases, these represent a speedup over looping; in others, they are just cleaner from a coding perspective.

Exercise 2

Using our handy matrix, does it appear that Twitter is becoming more popular as times goes by in this time period? That is, does the number of tweets seem to increase as a function of time? What day saw the most tweets? The least?

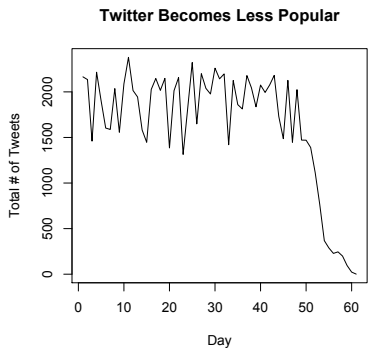
Which user has the highest frequency of tweets over the time period? Which user sends the most tweets over the period?

Can this information be used to detect spammers? How?

Exercise 2

Does it appear that Twitter is becoming more popular as times goes by in this time period?

```
> total.tweets <- apply(counts,2,sum) #same as colSums.
> plot(total.tweets,type="l")
```



Exercise 2

What day saw the most tweets? The least?

```
> total.tweets <- apply(counts,2,sum)
   #same as colSums.
> which.max(total.tweets)
10      # << Name of the column (11th day)
11      #April 11  Search Engine Watch conf?


> which.min(total.tweets)
60
61    #May 31, Twitter down most of the day
      #a dark day (actually, month) in microblogging!
```

Exercise 2

Which user has the highest frequency of tweets over the time period?
Which user has the sends the most tweets over the time period?

```
> mean.tweets <- apply(counts,1,mean)
> which.max(mean.tweets)
lejddfr
   4694   #index! not the actual value
> mean.tweets["lejddfr"]
 lejddfr
41.06557

> total.tweets <- apply(counts,1,sum)
  #same as rowSums()
> which.max(total.tweets)
lejddfr
   4694
> total.tweets["lejddfr"]
lejddfr
   2505
```

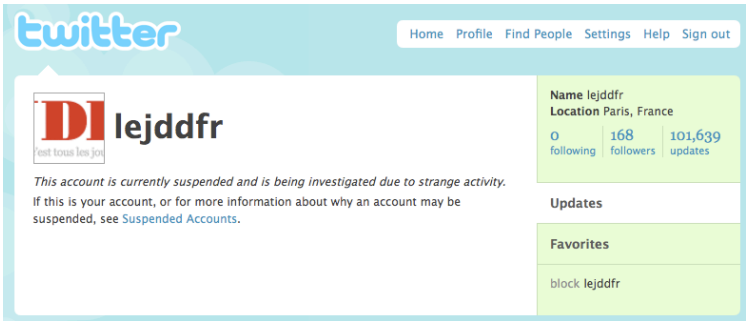

Exercise 2: Discuss

Can this information be used to detect spammers? How?

Exercise 2: Discuss

Can this information be used to detect spammers? How?

Apparently so...



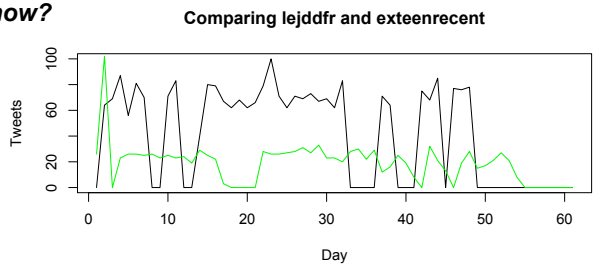
Plot Two Users' # of Daily Tweets over Time

Let's take a look at the two users that sent the most tweets over this 61 day period: **exteenrecent** and **lejddfr**.

First we plot **lejddfr**'s number of tweets per day vs. day.

Then we plot **exteenrecent**'s number of tweets per day vs. day on top of the original plot, and color the line green (`col = 5`).

But ***how?***

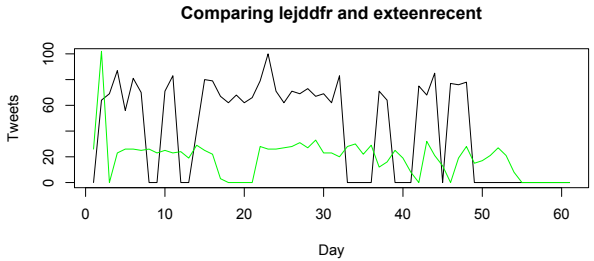


A One Track Mind

R can only render one plot at a time.

To add the second plot on top of the first, use the `lines` function. (Similarly there is also a `points` function).

```
> plot(counts["lejddfr",],type="l")
> lines(counts["exteenrecent",],col="green")
```



amazon.com BARNES&NOBLE BOOKSELLERS BORDERS textbooks.com more choices. lower prices.

Our Last Dataset

About the Data

In September 2008, I was asked to scrape textbook information from **`amazon.com, bn.com, borders.com`** and **`textbooks.com`** so that an analyst could research the current textbook price market.

This data contains the results in a **`data.frame`**. The dataset contains author, book title, department, and the prices for a new copy of the book from Amazon, Barnes & Noble, Borders and textbooks.com.

Loading the Data

The function **`read.csv`** is a wrapper for **`read.table`**; that is, it provides a set of good default values that anticipate what a typical comma-separated file looks like.

`read.table`, however, is a wrapper for an even lower- level function called **`scan`**; this function is better at reading very large data.

R attempts to coerce the column into logical, integer, numeric and then complex; if that all fails, it will store the data as a factor. More on this coming up...

```
the.url <- "http://www.stat.ucla.edu/~rosario/boot08/data/book-shopping.csv"
books <- read.csv(the.url,header=TRUE)
```

`header=TRUE` tells R that the first line read by R contains the variable names.

Let's Investigate...

Trivially, we would expect the average retail price of a book to increase at each retailer as the list price increases.

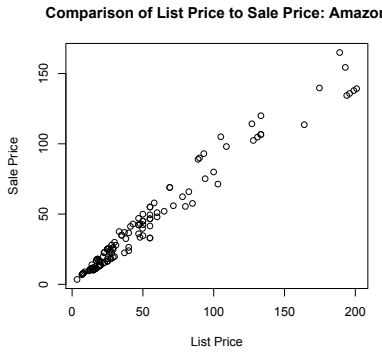
But to what extent? For which retailers is this effect the largest? The smallest?

Scatterplots

First, let's create a scatterplot comparing Amazon's list price (taken as "ground truth") to its retail price for all books.

```
> plot(books$Amazon.List.Price,books$Amazon.Sale.Price)
```

Note: graphical parameters suppressed for brevity.



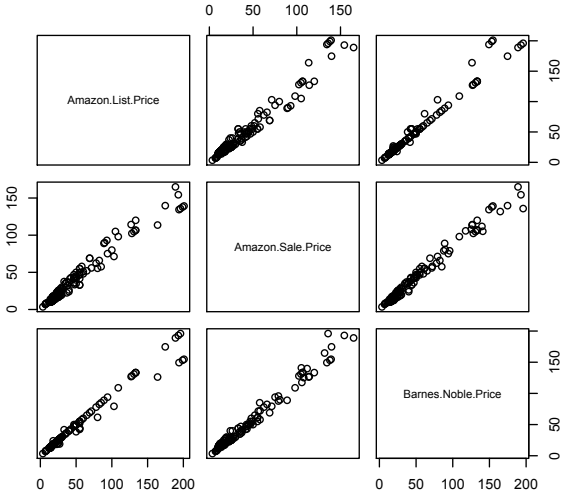
Scatterplots

We can do this for each of the four retailers, but this would become monotonous.

Let's compare all combinations of the list price, Amazon's price and Barnes and Noble's price. We can use the **`pairs`** function to get a *scatterplot matrix*.

```
> pairs(books[,4:6])
```


Scatterplots



Linear Model

The relationship between the list price and Amazon's sale price is strong and linear. Let's fit a linear model of the form:

$$y = \beta_0 + \beta_1 x$$

```
> fit <- lm(books$Amazon.Sale.Price~books$Amazon.List.Price)
```

where **y** is Amazon Sales Price and **x** is the List Price. We estimate the values of β_0 and β_1 .

Linear Model: WTF?

```
lm(y ~ X)
```

The ~ means we are specifying a *formula* defining a relationship. Left of the ~ is the dependent variable.

Right of the ~ are the independent variables. Predictors can be added (multiple regression) or multiplied (interactions). We can also subtract off the intercept (~…-1) if we really want to.

This formula notation can also be used in functions like **plot** and **boxplot**.

Linear Model: WTF?

```
> fit <- lm(books$Amazon.Sale.Price~books$Amazon.List.Price)
> fit

Call:
lm(formula = books$Amazon.Sale.Price ~ books$Amazon.List.Price)

Coefficients:
            (Intercept)  books$Amazon.List.Price
                 2.1802                   0.7742
```

So $\hat{\beta}_0 = 2.1802$ and $\hat{\beta}_1 = 0.7742$.

Linear Model as an Object

What is `fit`? `fit` is an *object* of type `lm` that contains many *attributes*.

```
> attributes(fit)
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

$class
[1] "lm"
```

We can also get more information about the model…

Linear Model as an Object

```
> summary(fit)
Call:
lm(formula = books$Amazon.Sale.Price ~ books$Amazon.List.Price)

Residuals:
     Min       1Q   Median       3Q      Max
-18.5055  -3.6787  -0.6007   2.9765  21.5289

Coefficients:
                       Estimate Std. Error t value Pr(>|t|)
(Intercept)             2.18019    0.81115   2.688  0.00804 **
books$Amazon.List.Price  0.77420    0.01242  62.326  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.839 on 144 degrees of freedom
  (36 observations deleted due to missingness)
Multiple R-squared: 0.9643,   Adjusted R-squared: 0.964
F-statistic:  3885 on 1 and 144 DF,  p-value: < 2.2e-16
```


The `summary` Function.

We can also use the **`summary`** function on objects other than linear models. We can also use it as an exploratory analysis step to analyze the distributions of variables.

```
> summary(books$Borders.Price)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   3.50   16.95   28.98   52.38   68.95  201.00   26.00
```

Here we receive a numerical summary of the temperatures at the very beginning of the study, over all locations.

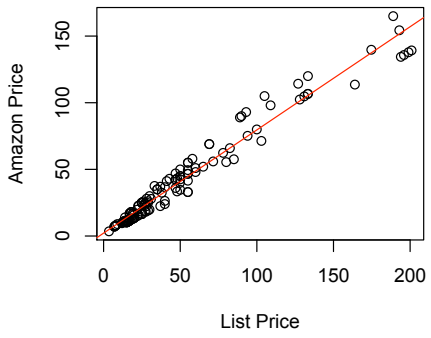
We can also manually find statistics using functions like **`mean`**, **`sd`**, **`median`**, **`quantiles`**, **`var`**, **`cov`**, **`cor`**, **`min`**, **`max`**.

Linear Model as an Object

Using this model, we can predict the response given the predictors.

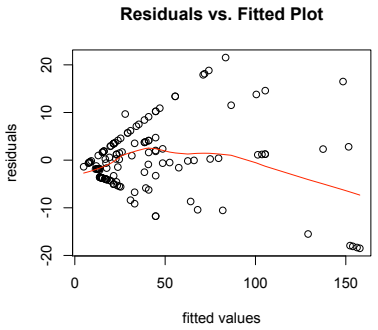
```
> plot(books$Amazon.List.Price,books$Amazon.Sale.Price)
> abline(fit,col="red")
```

How is the fit?



Linear Model as an Object

Using the `residuals` and `fitted` attribute, we can plot the residuals vs. fitted values. Using the `lowess` function, we can overlay a red curve indicating any general trend of the points, **but don't read too much into this curve**.

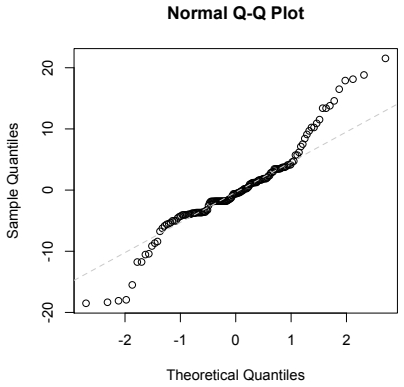



```
> plot(fit$fitted,fit$residuals,xlab="fitted values",
+ ylab="residuals", main="Residuals vs. Fitted Plot")
> lines(lowess(fit$fitted,fit$residuals),col="red")
```

Linear Model as an Object

We can also plot the normal quantile plot of the residuals to verify the assumption of normality in the residuals.

We can do this using the `qqnorm` function. The `qqline` function overlays the line usually associated with this plot.




```
> qqnorm(fit$residuals)
> qqline(fit$residuals,lty=2,col="grey")
```


Exercise 3 (Last One)

Which retailer's average sales prices increase the steepest as list price increases? The shallowest?

Exercise 3 (Last One)

We will be naive and assume the assumptions of the linear model hold. Then, we will just look at the estimate of the regression coefficient.

Retailer	$\hat{\beta}_1$
Amazon	0.7742
Barnes & Noble	0.906
Borders	1.000
Textbooks.com	0.8424

Steepest: Borders Shallowest: Amazon

Putting the Summary in a LaTeX File

Using the `verbatim` environment in LaTeX works fine for output, but for tables we can do better. Using the `xtable` package, we can output the LaTeX that can be used to represent the table.

```
> library(xtable)
 Error in library(xtable) : there is no package called
'xtable'
```

So we need to install it…

```
install.packages("xtable")
```

R will ask you which CRAN repository to use. Choose one in California.

Putting the Summary in a LaTeX File

```
> library(xtable)
```

Let's look at the help file for the `xtable` function first...

```
> help(package="xtable")
```

Note that we can print any *R* object to LaTeX syntax using `xtable`!

```
> xtable(summary(fit))
```

Yields the result on the next slide...

Putting the Summary in a LaTeX File

```
> xtable(summary(fit))

\begin{table}[ht]
\begin{center}
\begin{tabular}{rrrrr}
  \hline
 & Estimate & Std. Error & t value & Pr($>$$|$t$|$) \\
  \hline
(Intercept) & 2.1802 & 0.8112 & 2.69 & 0.0080 \\
  books\$Amazon.List.Price & 0.7742 & 0.0124 & 62.33 & 0.0000 \\
   \hline
\end{tabular}
\end{center}
\end{table}
```

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	2.1802	0.8112	2.69	0.0080
books$Amazon.List.Price	0.7742	0.0124	62.33	0.0000

One Last Data Structure: Factors

A factor represents a categorical variable. When importing data, R attempts to coerce the column into logical, integer, numeric and then complex; if that all fails, it will store the data as a factor.

We can see the different values of a factor using `levels()`.

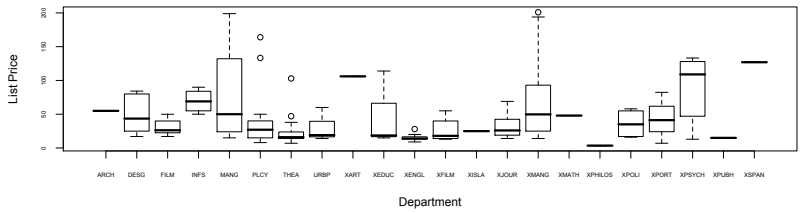
```
> levels(books$Department)
 [1] "ARCH"    "DESG"    "FILM"    "INFS"    "MANG"    "PLCY"
 [7] "THEA"    "URBP"    "XART"    "XEDUC"   "XENGL"   "XFILM"
[13] "XISLA"   "XJOUR"   "XMANG"   "XMATH"   "XPHILOS" "XPOLI"
[19] "XPORT"   "XPSYCH"  "XPUBH"   "XSPAN"
```

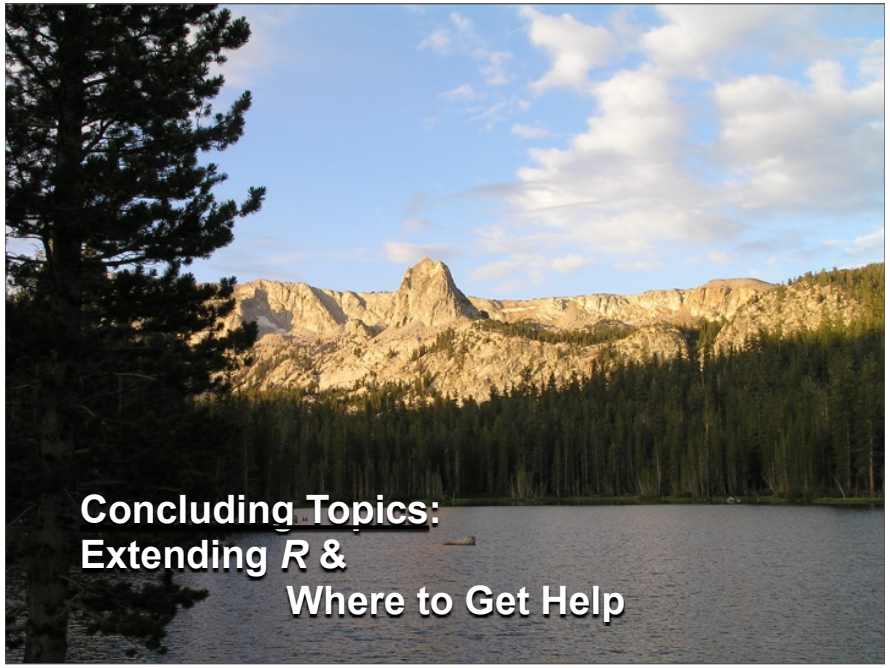
You will revisit this in 202A.

One Last Data Structure: Factors

We can make a boxplot of book list prices by Department using the factor.

```
> boxplot(books$Amazon.List.Price~books$Department)
  #Formula notation.
```



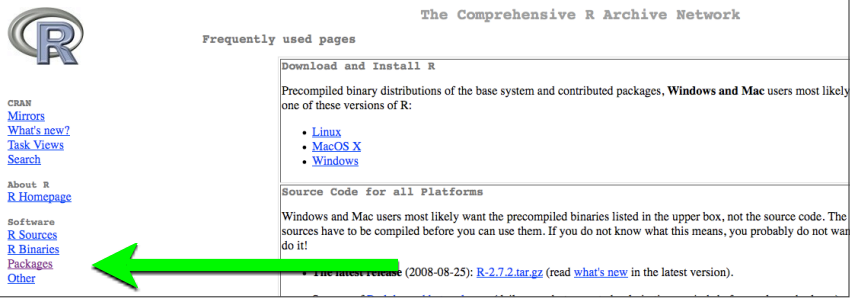


CRAN Package Repository

One can see how R can be extended by visiting the package listing at a CRAN mirror. Our CRAN mirror's URL is

`http://cran.stat.ucla.edu`

Look to the left and click on Packages.



CRAN Package Repository

As of September 20, 2008, there are 1,573 packages in CRAN!

Available Bundles and Packages

Currently, the CRAN package repository features 1542 objects including 1533 packages and 9 bundles containing 40 packages, for a total of 1573 available packages.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

ADaCGH	Analysis of data from aCGH experiments
AER	Applied Econometrics with R
AIS	Tools to look at the data ("Ad Inidicia Spectata")
ALS	multivariate curve resolution alternating least squares (MCR-ALS)
AMORE	A MORE flexible neural network package
ARES	Allelic richness estimation, with extrapolation beyond the sample size
AcceptanceSampling	Creation and evaluation of Acceptance Sampling Plans
AdMit	Adaptive Mixture of Student-t distributions
AdaptFit	Adaptive Semiparametric Regression
AlgDesign	AlgDesign
Amelia	Amelia II: A Program for Missing Data
AnalyzeFMRI	Functions for analysis of fMRI datasets stored in the ANALYZE or NIFTI format
ArDec	Time series autoregressive decomposition
aaMI	Mutual information for protein sequence alignments
abind	Combine multi-dimensional arrays
accuracy	Tools for testing and improving accuracy of statistical results
acepack	ace() and avas() for selecting regression transformations
actuar	Actuarial functions

LaTeX `listing` Environment

"Using the package listings you can add non-formatted text as you would do with **`\begin{verbatim}`** but its main aim is to include the source code of any programming language within your document. It supports highlighting of all the most common languages and it is highly customizable. If you just want to write code within your document the package provides the **`lstlisting`** environment."

```
1 bozo<-function(n,p) {
2 s<-0 # initialize sum
3 for (x in 0:n) s<-s+choose(n,x)*p^x*(1-p)^(n-x)
4 print(cat("1 = ",s)); return(s)
5 }
```

Credit: Jan de Leeuw

Download from CTAN:

`http://www.ctan.org/tex-archive/macros/latex/contrib/listings/`

LaTeX and *R*: Sweave

Users can also directly embed R code into a LaTeX document and not only will LaTeX typeset the document, but the embedded R code will be replaced with the output resulting from executing the code in R!

```
Well, we can now include R in our document.  Here's a simple example
<<two>>=
2 + 2
@
What I actually typed in \verb@foo.Rnw@ was
\begin{tabbing}
\verb@<<two>>=@ \\
\verb@2 + 2@ \\
\verb+@+ \\
\end{tabbing}
This is not \LaTeX.  It is a ``code chunk'' to be processed by \verb@Sweave@.
When \verb@Sweave@ hits such a thing, it processes it, runs R to get the
results, and stuffs (by default) the output in the \LaTeX\ file it is
creating.  The \LaTeX\ between code chunks is copied verbatim (except
for \verb@Sexpr@, about which see below).  Hence to create a Rnw document
you just write plain old \LaTeX\ interspersed with ``code chunks'' which
are plain old R.
```

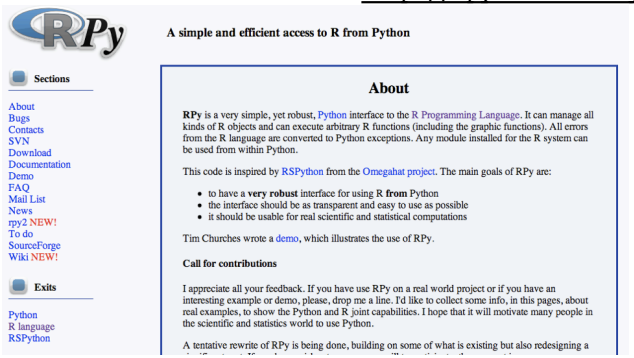
Source: http://www.stat.umn.edu/~charlie/Sweave/foo.Rnw

Sweave is a tool for “literate programming” and “reproducible research.”

`rpy`: *R* from Python!

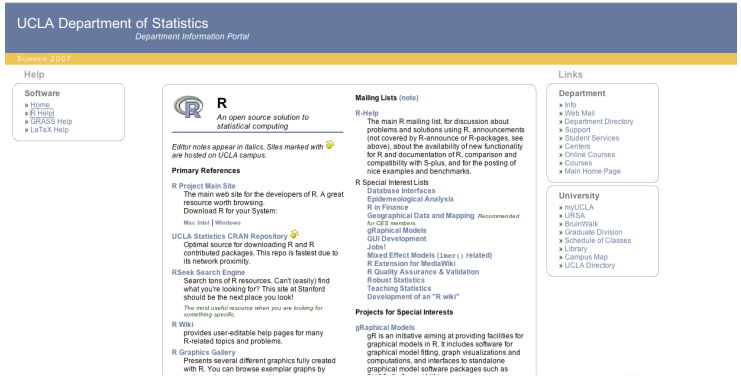
“**`RPy`** is a very simple, yet robust, Python interface to the *R* Programming Language. It can manage all kinds of *R* objects and can execute arbitrary *R* functions. All errors from the *R* language are converted to Python exceptions. Any module installed for the *R* system can be used from within Python.”

`http://rpy.sourceforge.net/`



Department Software Resources

`http://info.stat.ucla.edu/grad`



Rseek.org

`http://www.rseek.org`

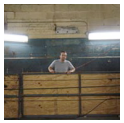
A custom Google search engine for R help. It is AWESOME. Helped me find the answers to all of my questions.

USE IT!!

Statistical Computing at UCLA

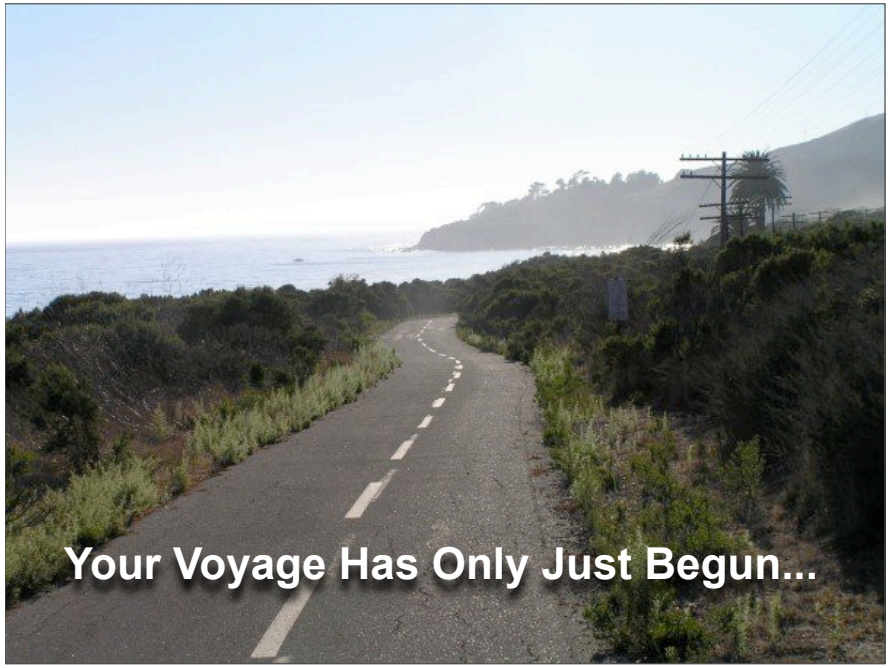
Course titles in quotes are "informal" descriptions of the course. The official course title may differ.

Statistics 202A
"*Data Mining and Statistical Programming*"
Mark Hansen



Statistics 202B
"*Intro to Multivariate Analysis and Matrix Optimization*"
Jan de Leeuw

Statistics 202C,
"*Markov Chain Monte Carlo Methods*"
Chiara Sabatti



Enjoy your 1st Year!
You can contact me at:
ryan@stat.ucla.edu

Thanks for a Great Day!