Statistics 133: Review

R

Features of R

- Allows custom analysis
- High-level scripting language
- Statistical programming language
- Interactive exploratory data analysis
- Easy to replicate analysis
- Sound numerical methods
- Large Community of contributors

Computation with Expressions

- The R prompt is: >
- At the prompt, type an ***expression***
- Hit the return/enter key
- R ***evaluates*** the expression (performs a ***computation***)
- R returns a value

Parsing Expressions - How does R know what computation to perform?

- It breaks down an expression into parts, called tokens
- From these pieces it can figure out what computation to perform

To parse expressions R uses: white space; atomic tokens (e.g. + / ^ (and []); quotation marks; naming conventions; new line; end of line (i.e. ;); and comment character (#);

Order of operations is what you expect:

- exponentiation first, followed by multiplication and division, then addition and subtraction
- left to right
- parentheses override order

Output and Assignment

- **=** and **<-** are both valid assignment operators

Variables

- Variables have a name and a value
- To access the value we use the name
- Variables allow us to:
  - Store a value without needing to recompute it
  - Write a general expression,. eg. **sqrt(a^2 + b^2)**
  - Reduce redundancy (and mistakes)

Variable Names

Variable names must follow some rules

May not start with a digit or underscore (_)

May contain numbers, characters, and some punctuation - period and underscore are ok, but most others are not

Case-sensitive, so x and X are different

Avoid names that have meaning in R, e.g., function names. If in doubt, check:

Function Style Expressions

When you use a function with a particular set of arguments, you are said to be ***calling*** the function

R ***evaluates*** the function call and returns the output

Be familiar with the following built- in functions

log, exp, sqrt, abs,
mean, sd, median, summary, min, max, sum,
sample, rnorm, runif,
class, length, dim, head, names, is.na
all, any, which
read.table, read.csv, library, load
c, rep, seq, order, sort
histogram, boxplot, dotchart, plot, barchart, mosaicplot, abline, points
sapply, lapply, apply, tapply
substr, nchar, strsplit,
gsub, sub, regexpr, gregexpr, grep

Calling a function

FunctionName(argument, …, argument)

Functions can have one or more inputs
Some arguments are required.
Other arguments are optional. They have default values so you don't have to specify them

Example: rnorm
function (n, mean = 0, sd = 1)

It has 3 arguments: n, mean and sd

mean and sd are optional. – they have default values (0 and 1, respectively)

n must be specified – it has no default

Arguments can be identified by position: **rnorm(3, 0, 5)**

Arguments can be identified by name: **rnorm(n = 3, sd = 5)**

Use position and name: **rnorm(3, sd = 5)**

Simple and Compound Expressions

Simple Expression: **rnorm(3, sd = 5)**
Compound Expression: **mean(rnorm(3))**
Ill-formed Expressions: **mean(rnorm(3)]**

Data Types

R has a number of built-in data types.
The three most basic types are numeric, character, and logical.
A *logical* vector contains values that are either TRUE or FALSE.
You can check the type using the class function.
Another important type is factor. A *factor* vector is a special storage class used for qualitative data. The values are internally stored as integers by each integer corresponds to a *level*, which is a character string

Vectors

Ordered container
Primitive elements of the same type
There's no such thing as a scalar in R, just a vector of length one.

Special Values

NA: The missing value symbol is NA. It stands for "Not Available"
NA can be an element of a vector of any type
NA is different from the character string "NA"
You can check for the presence of NA values using the **is.na()** function.
Other special values are NaN, for "not a number," which typically arises when you try to compute an indeterminate form such as 0/0.
The result of dividing a non-zero number by zero is Inf (or -Inf).
NULL is a special value that denotes an empty vector

Subsetting

Inclusion/Index: BMI of the 10th person in the family

```
> fbmi[10]
[1] 30.04911
```

Exclusion: Ages of all but the first person in the family

```
> fage[-1]
 [1] 33 79 47 27 33 67 52 59 27 55 24 46 48
```

Name: Height of person "j"

```
> fheight["j"]
 j
71
```

Logical: Genders of the family members who weigh more than 250

```
> fgender[fweight > 250]
[1] m f m m m f
Levels: m f
```

Assign values to elements of a vector

In general, the same indexing may be used to *assign* values to elements of a vector.

Subsetting all of the elements of a vector: fweight[ ] = 0 sets all element values in fweight to 0

Logical expressions:

Example: Create a logical expression that identifies the women in the family

```
> fgender == "f"
 [1] FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE    [8] TRUE FALSE FALSE TRUE FALSE
FALSE  TRUE
```

Use this logical expression to subset the vector of fweight

```
> fweight[ fgender == "f"]
[1] 125 105 190 124 166 125
```

Boolean Algebra

Boolean algebra is a mathematical formalization of the truth or falsity of statements.

It has three operations, “not,” “or,” and “and.”

Boolean algebra tells us how to evaluate the truth or falsity of *compound statements* that are built using these operations. For example, if A and B are statements, some compound statements are

A and B

(not A) or B

The compound statement A and B is TRUE only if both A and B are TRUE.

The compound statement A or B is TRUE if *either or both* A or B is TRUE.

In R, we write ! for “not,” & for “and,” and | for “or.” Note: all of these are vectorized!

Two other useful functions that operate on logical vectors are **all** and **any**.

Creating vectors

c() - catenate vectors and values together

: - create a sequence of values 1 apart

seq() – create more complex sequences (arguments: from, to, by and length)

rep() – repeat values in a vector (times and each are two of its arguments)
sort() – sort the values in a vector
order() – provide the order of values

Data Frames

Ordered container of vectors
Vectors must all be the *same length*
Vectors can be *different types*
The data frame gives us a way to collect all of these variables (vectors) into one object.

```
> data.frame(firstName = fnames,
gender = fgender, age = fage, height = fheight, weight = fweight, bmi = fbmi, overWt =
foverWt)
```

dataframeName$vectorName

```
> mean(family$height)
[1] 67.07143
```

Subsetting Data frames

We subset rows and columns of data frames
We subset by **position**, **exclusion**, **logical, name**, and **all**

```
> family[ 10:13, -(3:14)]
   firstName gender
10       Tom      m
11       Ann      f
12       Dan      m
13       Art      m
```

Reading data into R

Many data sets are stored in text files.
The easiest way to read these into R is using either the **read.table** or **read.csv** function, both of which return a data frame.
There are quite a few options that can be changed. Some of the important ones are

file - name or URL
header - are column names at the top of the file?
sep - what divides elements of the table
na.strings - symbol for missing values, like 9999
skip - number of lines at the top of the file to ignore

Lists

Data frames are actually a special kind of *list*.

Unlike a data frame each element can have a different length.

```
> Ingredients = list(cheese = c("Cheddar", "Swiss"),
+                    meat = c("Ham","Turkey", "Bologna"))
> Ingredients
$cheese
[1] "Cheddar" "Swiss"
$meat
[1] "Ham"     "Turkey"  "Bologna"
```

Note that the elements are not associated with one another by position, as they were in a given row of a data frame.

Indexing lists:

Lists can be indexed by name, using $.

They can also be indexed like vectors, using []. The result will be another list of length 1.

```
> Ingredients[2]
$meat
[1] "Ham"     "Turkey"  "Bologna"
> class(Ingredients[2])
[1] "list"
```

To extract individual elements of a list, enclose the index in [[]]. The result will be coerced to a simpler structure, depending on the element.

```
> Ingredients[[2]]
[1] "Ham"     "Turkey"  "Bologna"
> class(Ingredients[[2]])
[1] "character"
```

Matrices and Arrays

Rectangular collection of elements

Dimensions are two, three, or more

Homogeneous primitive elements (e.g. all numeric or all character)

You can create a matrix in R using the **matrix** function.

By default, matrices in R are assigned by *column-major* order.

You can assign them by *row-major* order by setting the byrow argument to TRUE.

Note that the first argument to matrix is a vector, so all elements must be of the same type (numeric, character, or logical).

Assign names to the rows and columns of a matrix:

Find the dimensions of a matrix with dim()

Exchange rows and columns: t()

To index elements of a matrix, use the same five methods of indexing we covered for vectors, but with the first index for rows and the second for columns

Matrices and arrays are actually just stored as vectors with shape information, so our discussions of "vectorized" calculations hold for matrices as well. This is NOT true for lists and data frames.

Apply Functions

Sometimes we want an operation to be applied to each element of a list, to each vector in a data frame, or to individual dimensions of a matrix
R provides the *apply* mechanism to do this.

There are several apply functions:

sapply() and lapply() for lists and data frames
apply() for matrices
tapply() for "tables", i.e. ragged arrays as vectors

With these functions we can avoid looping, and instead we write code that is meaningful in a statistical setting.

For example, a list of rainfall data is constructed such that each element represents the measurements taken at a particular weather station. When we think about studying the average rainfall at each station - we don't think in terms of loops.

```
> lapply(rain, mean)
$st050183
[1] 6.631707

$st050263
[1] 3.798993 …

> sapply(rain, mean)
st050183 st050263 st050712
6.631707 3.798993 5.102299  …
```

Additional arguments. If the function that we wish to apply has additional arguments that need to be specified we do this as follows:

```
> lapply(rain, mean, na.rm = TRUE, trim = 0.1)

> args(lapply)
function (X, FUN, ...)
```

X takes the list object
FUN is the function to apply to each element in X
… allows any number of arguments to be passed to FUN

tapply()

This function is useful to apply a function to subsets of a vector.

```
> x
[1] 1 2 3 4 5 6 7 8 9 10
> v
[1] 1 1 1 0 0 0 1 1 1 0
> tapply(x, v, mean)
0      1
6.25 5.00
```

Creating your own function to apply

```
>sapply(rain, function(x) {
     sum(x > 0)/length(x) }
)
```

apply()

apply(x, 1, sum) for the matrix x, the sum function is applied across the columns so that the row dimension (i.e. dim 1) is preserved.

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

Graphics

Visualization enters every step of the data analysis cycle
- Data cleaning – are there anomalies?
- Exploration
- Model checking
- Reporting results

Plots can uncover structure in data that can't be detected with numerical summaries
Important communication skill

There are two models in R – painter and object-oriented. We will use the painter's model

Painter's model – start with a blank canvas, add/paint on it in multiple passes

The appropriate graphical techniques depend on the kind of data that you are working with. Quantitative data – rug plot, histogram, scatter plot, boxplot; Qualitative data – dotchart, barchart, mosaic plot; One quantitative and one qualitative – side-by=side boxplots

Method of Comparison

Often, we not only want to better understand a distribution, but we want to compare the distribution for subgroups or to compare against another population or standard

Relationships between more than 2 variables

Qualitative information can be conveyed in plots through color, plotting symbol, juxtaposed panels

Making good plots is an iterative process
- Goal is to convey a message as clearly as possible

Graph Construction:

Vocabulary: title, subtitle, axis label, axis, tickmark, tickmark label, plotting region, plotting symbol, legend, marker/reference line, point, text

3 Properties of good graph construction: Data stand out, Facilitate comparison, Information rich

Data Stand Out
Avoid having other graph elements interfere with data

Use visually prominent symbols
Avoid over-plotting, e.g. jitter the values or use transparent plotting symbols

Choosing the Scale of the Axis
- Include all or nearly all of the data
- Fill data region
- Origin need not be on the scale
- Choose a scale that improves resolution

Eliminate superfluous material
- Chart junk – stuff that adds no meaning, e.g. butterflies on top of barplots, background images
- Extra tick marks and grid lines
- Unnecessary text and arrows
- Decimal places beyond the measurement error or the level of difference

Facilitate Comparisons
Put Juxtaposed plots on same scale
Make it easy to distinguish elements of superposed plots (e.g. color)
Choosing the Scale
- Keep scales on x and y axes the same for both plots to facilitate the comparison
- Zoom in to focus on the region that contains the bulk of the data
- These two principles may go counter to one another
- Keep the scale the same throughout the plot (i.e. don't change it mid-axis)

Emphasizes the important difference
Avoid Jiggling the baseline
Comparison: volume, area, height
Banking: Aspect Ratio
- The height/width of the data region was selected to be about 1 so that the trend line is at about 45 degrees.
- The Aspect ratio affects our visual decoding of the rate of change
- The banking to 45 degrees helps us see rate of change
- The ability to effectively judge rate of change allows us to see important patterns in data

Information Rich
Describe what you see in the **Caption.** Captions should be comprehensive, self-contained, draw attention to the important features, describe conclusions

Add context with **Reference Markers** (lines and points) including text
Add **Legends** and **Labels**
Use color and plotting symbols to add more information
Plot the same thing more than once in different ways/scales

Reduce clutter

Good Plot-Making Practice

- Put major conclusions in graphical form
- Provide reference information
- Proof read for clarity and consistency

Graphing is an iterative process

Color Guidelines

Choosing a set of colors which work well together is a challenging task for anyone who does not have an intuitive gift for color

Remember 7-10% of males are red-green colorblind.

Colorfulness

Saturated/colorful colors are hard to look at for a long time.

They tend to produce an after-image effect, which can be distracting.

Luminance: If the size of the areas presented in a graph is important, then the areas should be rendered with colors of similar luminance (brightness).

Lighter colors tend to make areas look larger than darker colors

Data Type and color choice

Qualitative – Choose a **qualitative** scheme that makes it easy to distinguish between categories

Quantitative – Choose a color scheme that implies magnitude. Does the data progress from low to high? Use a **sequential** scheme where light colors are for low values. OR, do both low and high value deserve equal emphasis? Use a **diverging** scheme where light colors represent middle values

Control Flow and Writing Functions

Functions allow us to

Organize our code into tasks
Reuse the same code on different datasets by making the data an *argument* to the function.

Anatomy of a function

function (arguments) body of expressions

Typically we assign the function to a particular name. This should describe what the function does.

myFunction = function (arguments) body

A function without a name is called an "orphan" function. These can be very powerfully used with the apply mechanism.

The keyword *function* tells R that you want to create a function.

The *arguments* to a function are its inputs, which may have default values.

```
> args(median)
function (x, na.rm = FALSE)
```

When you're writing your own function, it's good practice to put the most important arguments first. Often these will not have default values. This allows the user of your function to easily specify the arguments by position, eg. plot(xvec, yvec) rather than plot(x = xvec, y = yvec).

The *body* of the function typically consists of expressions surrounded by curly brackets. Think of these as performing some operations on the input values given by the arguments.

```
{
  expression 1
  expression 2
  return(value)
}
```

The *return* expression hands control back to the caller of the function and returns a given value. If the function returns more than one thing, this is done using a named list,

For example

return(list(total = sum(x), avg = mean(x))).

In the absence of a return expression, a function will return the *last* evaluated expression. This is particularly common if the function is short.

A return expression anywhere in the function will cause the function to return control to the user *immediately*, without evaluating the rest of the function. This is often used in conjunction with if statements, which we'll come to later.

Considerations when writing a function:

- What will the function do?
- What should we call it? (Relate the name to what it does)
- What will be the arguments?
- Which arguments have default values and what are they?
- What (if anything) should the function return?

Flow Control

Flow control structures allow us to control which statements are evaluated and in what order.

In R the primary ones consist of if/else statements and for loops. Expressions can be grouped together using curly braces "{" and "}". A group of expressions is called a *block*.

if/else statements

The basic syntax for an if/else statement is

```
if ( condition ) {
  statements
} else {
  statements
}
```

First, condition is evaluated. If the first element of the result is TRUE then statement1 is evaluated. If the first element of the result is FALSE then statement2 is evaluated. *Only the first element of the result is used.*

If the result is numeric, 0 is treated as FALSE and any other number as TRUE. If the result is not logical or numeric, or if it is NA, you will get an error.

When we discussed Boolean algebra before, we met the operators & (AND) and | (OR).

Recall that these are both *vectorized* operators.

If/else statements, on the other hand, are based on a single, "global" condition. So we often see constructions using any or all to express something related to the whole vector, like

```
if ( any(x < -1 | x > 1) )
  warning("Value(s) in x outside the interval [-1,1]")
```

The result of an if/else statement can be assigned. For example,

```
> if ( any(x <= 0) ) y = log(1+x) else y = log(x)
```

Also, the else clause is optional. Another way to do the above is

```
> if ( any(x <= 0) ) x = 1+x
> y = log(x)
```

Note that this version this changes x as well.

If/else statements can be nested.

```
if (condition1 )
  statement1
else if (condition2)
  statement2
else if (condition3)
  statement3
else
  statement4
```

The conditions are evaluated, in order, until one evaluates to TRUE. Then the associated statement/block is evaluated. The statement in the final else clause is evaluated if none of the conditions evaluates to TRUE.

The **for** statement

Looping is the repeated evaluation of a statement or block of statements.

Much of what is handled using loops in other languages can be more efficiently handled in R using vectorized calculations or one of the apply mechanisms.

However, certain algorithms, such as those requiring recursion, can only be handled by loops.

There are two main looping constructs in R: for and while. We have focused only on the for loop. A *for loop* repeats a statement or block of statements a predefined number of times.

The syntax in R is

```
for ( name in vector ){
  statement
}
```

For each element in vector, the variable name is set to the value of that element and statement is evaluated.

vector often contains integers, but can be any valid type.

Environments and variable scope

R has a special mechanism for allowing you to use the same name in different places in your code and have it refer to different objects. For example, you want to be able to create new variables in your functions and not worry if there are variables with the same name already in the workspace. The solution relies on *environments* and the *variable scoping rules*.

When you call a function, R creates a new workspace containing just the variables defined by the arguments of that function. This collection of variables is called a *frame*. However, R has a way of accessing variables that are not in the frame created by the function.

An *environment* is just a frame (collection of variables) plus a pointer to the next environment to look in. When R doesn't find the variable x in the environment defined by a function, it looks in the next one. The "next environment to look in" is called the parent environment. If R reaches the Global Environment and still can't find the variable, it looks in something called the *search path*. This is a list of additional environments, which is used for packages of functions and user attached data. You can see the search path by typing search().

This helps explain why we can write over built-in objects in R. What we're really doing is creating that object in the Global Environment, and then when we refer to it by name, R finds it here before it finds the predefined one.

Catching errors

1. The function *stop* stops execution of the current expression and prints a specified error message.

```
> showstop <- function(x){
+   if(any(x < 0)) stop("x must be >= 0")
+   return("ok")
+ }
> showstop(1)
[1] "ok"
> showstop(c(-1, 1))
Error in showstop(c(-1, 1)) : x must be >= 0
```

2. The *warnin*g function prints a warning message without stopping the execution of the function.

```
> ratio.warn <- function(x, y){
+   if(any(y == 0))
+     warning("Dividing by zero")
+   return(x/y)
+ }
> ratio.warn(x = 1, y = c(1, 0))
[1]   1 Inf
Warning message:
In ratio.warn(x = 1, y = c(1, 0)) : Dividing by zero
```

Some debugging strategies

1. The traceback function prints the sequence of calls that led to the last error. This can show you where in your function something is going wrong.

It may not even be in the function itself, but in another function that is being called within the original function.

```
> cv <- function(x) sd(x/mean(x))
> cv(0)
Error in var(x, na.rm = na.rm) : missing observations in cov/cor
> traceback()
3: var(x, na.rm = na.rm)
2: sd(x/mean(x))
1: cv(0)
```

2. If you have some idea where the error is occurring, you can use print to check that

key variables are what you think they are.

3. Consider “commenting out” lines of your code where the error might occur, then adding them back in one by one.

4. To step through the function, expression by expression, and be able to print out any variable at each step, use the debug function. Use undebug to turn off debugging.

While in the debugger, you can use the following commands:
'n' (or just return) - Advance to the next step.
'c' - continue to the end of the current context: e.g. to the end of the loop if within a loop or to the end of the function.
'Q' - exit the browser and the current evaluation and return to the top-level prompt.

You can also evaluate any valid R expression. For example, you can type the names of variables to see their current values.

Efficient programming
The first rule of efficient programming in R is to make use of vectorized calculations and the apply mechanisms whenever possible.

You can check how much time it takes to evaluate any expression by wrapping it in system.time(). Units are in seconds.

```
> system.time(normal.samples <- rnorm(1000000))
   user  system elapsed
  0.196   0.013   0.221
```

A systematic way to time every part of a function is to use the Rprof and summaryRprof functions. This can be a handy way to find bottlenecks. The general syntax looks like this:

```
Rprof(“profiling”)
statements
Rprof(NULL)
summaryRprof(“profiling”)
```

R packages
If a particular package is already installed on your system, you can access its contents by typing
```
> library(“nameofpackage”)
```

The authors of the package write documentation for the functions and datasets included in it, which you can read as usual using help().

Simulation

Background:

We can think of probability theory as complimentary to statistical inference. Probability allows us to quantify statements about the chance of an event taking place.

For example - Flip a fair coin

What's the chance it lands heads?
Flip it 4 times, what proportion of heads do you expect?
Will you get exactly that proportion?
What happens when you flip the coin 1000 times?

In 4 flips, we can get 0, 1, 2, 3, or 4 Heads and so the proportion of Heads can be: 0, 0.25, 0.5, 0.75, or 1. We expect the proportion to be 0.5, but a proportion of 0.25 is quite likely:

There are 16 possible ways for 4 tosses to land, e.g. HHHH, HHHT, HHTH, …
Each is equally likely, so the chance of any particular sequence of Hs and Ts is 1/16
So chance of 0.25 proportion is 4/16 : HTTT, THTT, TTHT, TTTH

We can think of the proportion of Heads in 4 flips as a statistic because it summarizes data

Notice that it is a random quantity – it takes on 5 possible values, each with some probability

1,000 Flips

When we flip the coin 1,000 times, we can get a many different possible proportions of Heads, i.e. 0, 0.001, 0.002, 0.003, …, 0.998, 0.999, 1.000
It's highly unlikely that we would get 0 for the proportion – how unlikely?
What does the distribution of the proportion of heads in 1000 flips look like?
With some advanced math tools, we can figure this out, but we can also get a good idea using a **simulation**.
In our simulation we will assume that the chance of Heads is 0.5 and find out what the possible values for the proportion of heads in 1,000 flips looks like
If we were to carry out an experiment with a coin and get a particular proportion, say 0.37, then we could use this simulation study to help us understand the results of our experiment.

Generalization:

A *statistic* is often just a function of a random sample, for example the sample mean, the 95th quantile, or the sample proportion.

Statistics are often used as *estimators* of quantities of interest about the distribution, called *parameters*. Statistics are random variables (since they depend on the sample); parameters are not.

In simple cases, we can study the *sampling distribution* of the statistic analytically. For example, we can prove that under mild conditions the distribution of the sample proportion is close to normal for large sample sizes.

In more complicated cases, we turn to simulation.

The main idea in a simulation study is to replace the mathematical expression for the distribution with a *sample* from that distribution.

We use the sample to *estimate* features of the distribution, such as the behavior of various statistics under repeated sampling from the distribution.

This set of techniques, sometimes called Monte Carlo methods, is very powerful. Statisticians routinely use it to evaluate complicated methods for which exact mathematical results are difficult or impossible to obtain.

The downside: whereas mathematical results are symbolic, in terms of arbitrary parameters and sample size, in a simulation we must specify particular values.

To approximate the *sampling distribution* of the statistic, we repeat the whole experiment B times. The larger B is, the better our approximation will tend to be.

Steps in carrying out a simulation study:

Specify what makes up an individual experiment: sample size, distributions, parameters, statistic of interest.

Write an expression or function to carry out an individual experiment and return the statistic.

Determine what inputs, if any, to vary (e.g. different sample sizes or parameters).

For each combination of inputs, repeat the experiment B times, providing B samples of the statistic.

For each combination of inputs, summarize the *empirical distribution* of the statistic of interest.

State and/or plot the results.

Example: Carry out a simulation study of the median when sampling from the normal distribution. How does it vary with the sample size and with the standard deviation of the normal distribution?

Useful Random Number Generators

sample(x, size, replace = FALSE,
prob = NULL)

Think of an urn with tickets, each ticket marked with a value. Mix up the tickets and draw one at a time from the urn

x = vector with one element for each ticket, values correspond to what is written on the ticket.
size = number of draws to take from the urn
replace = replace the ticket between draws or not.
prob = set of weights for the elements in x (an element might represent more than one ticket)

Standard Probability Distributions:

rnorm(n, mean = 0, sd = 1) – sample from the normal distribution with center = mean and spread = sd

rbinom(n, size, prob), - sample from the binomial distribution with number of trials = size and chance of success = prob

runif(n, min = 0, max = 1) – sample from the uniform distribution on the interval (0, 1)

Other distributions: rexp(), rpois(), rt(), rf() – each has arguments for parameter values relevant to the distribution

R uses a **pseudo random number generator**:

It starts with a **seed** and an **algorithm** (i.e. a function)
The seed is plugged into the algorithm and a number is returned
That number is then plugged into the algorithm and the next number is created

The algorithms are such that the numbers produced behave/look like random values

The Seed

There is one big advantage to pseudo-random number generators: You can reproduce your simulation results by controlling the seed: set.seed() allows you to do this

When researchers publish results from simulation studies, they typically include the random number generator and the seed that was used so that others can verify/replicate their results

Hypertext Markup Language

Tree hierarchy
One root node
Root node has child nodes and each of these can have child nodes and so on
Any node must have one and only one parent

Table in HTML

```
<html>
 <head>
 </head>
 <body>
  <table>
    <tr> <th>A</th>  <th>B</th>  </tr>
    <tr>  <td>1</td><td>25,000</td></tr>
    <tr>  <td>7</td><td>100,000</td></tr>
  </table>
 </body>
</html>
```

Tables are defined with the <table> tag.
A table has rows marked up with the <tr> tag.
Each row is divided into data cells with the <td> tag. (td stands for table data).
A data cell can contain text, images, lists, paragraphs, forms, horizontal rules, tables, etc.
Headings in a table are defined with the <th> tag.

```
<html>
 <head>
 </head>
 <body>
  <table cellpadding="6" border="2">
    <tr> <th>A</th>  <th>B</th>  </tr>
    <tr align="right">  <td>1</td><td>25,000</td></tr>
    <tr align="right">  <td>7</td><td>100,000</td></tr>
  </table>
 </body>
</html>
```

Unordered Lists
Unordered lists have items marked with bullets.

```
<ul>
```

```
 <li>Coffee</li>
 <li>Milk</li>
</ul>
```

Paragraphs, line breaks, images, links, other lists, etc. can be placed in a list

Ordered Lists
Ordered lists have items marked with numbers.

```
<ol>
 <li>Coffee</li>
 <li>Milk</li>
</ol>
```

Paragraphs and Sections

```
<h1>
My BML Report
</h1>
 <h2>Introduction</h2>
   <p>
The BML model is a simple traffic model...
</p>
<p> We studied the BML model behavior for...
</p>
```

Images
The img tag is used to embed images in a Web page

```
 <img src=”images/bml34.png" width ="400"/>
```

The src attribute give the file name for the image
The width attribute is optional
This tag is empty – the start and end tag are collapsed.

Links

```
<a href="http://mae.ucdavis.edu/dsouza/"> D'Souzza</a>  discovered ....
```

<a> is an anchor tag
The content is the text that is “clickable”
The link can be to another place within the document

Element Syntax

Each HTML element has an element name, e.g.

body : the main content of the page
h1 : largest header
p : paragraph
br : line break

The start tag is the name surrounded by angle brackets: <h1>
The end tag is a slash and the name surrounded by angle brackets </h1>
The element content occurs between the star t tag and the end tag
Some HTML elements have no content

Attribute Syntax
Attributes provide additional information to an HTML element.
Attributes always come in name/value pairs like this: name="value"
Attributes are always specified in the start tag of an HTML element.

Well-formed XHTML

Well-formed HTML is called XHTML.
Tag names follow strict rules for matching case
Attribute values must be in quotes
Elements must be properly nested (i.e. you can draw a tree with it)

Example HTML for a Report

```
<html>
 <head>
<link rel="stylesheet" type="text/css" href="bmlStyle.css" />
 </head>
 <body>
  <h1>BML Model Simulation Study</h1>
   <h2>Introduction</h2>
   <p> The BML model is a simple traffic model... </p>
   <h2 class="bml”>Earlier Findings</h2>
   <p>
<a href="http://mae.ucdavis.edu/dsouza/">D'Souzza</a> discovered ....
</p>
<p>
A total traffic jam might look like this
<img src="images/bml34.png" width ="200"/>
</p>
 </body>
</html>
```

Cascading Style Sheet

```
body
{ background-color:#d0e4fe; }
h1
{ color:orange; text-align:center; }
h2.bml
{ color:green; text-align:center; }
p
{ font-family:"Times New Roman”; font-size:20px; }
```

CSS

selector {property: value; }

Selector may be:

HTML tag name	h1 {color: green;}
attribute value for id	#idXYZ {color:blue;}
class	.bmlStyle {font-size: 2em}

UNIX

The UNIX *kernel* is the part of the OS that actually carries out basic tasks.

The UNIX *shell* is the user interface to the kernel. Like flavors of UNIX, there are also many different shells. For this course, it doesn't matter which one you use. The default on the lab computers is called tcsh.

The first thing you need to know about UNIX are how to work with *directories* and *files*. Technically, everything in UNIX is a file, but it's easier to think of directories as you would *folders* on Windows or Mac OS.

Directories are organized in a *tree structure*.

To see the directory you're currently in, type the command **pwd** ("present working directory").

There are two "special" directories: The top level directory, named "/", is called the *root directory*.

Your *home directory*, named "~", contains all your files. For mary, "~" and "/users/mary" mean the same thing.

cd. Means "change directory". Here we move from the home directory to the BMLHW directory.

```
$ pwd
/Users/nolan
$ cd Desktop/BMLHW
$ ls
BMLcode        BMLreport       Stat133HW6.pdf
$ ls -a
.                               BMLcode                         Stat133HW6.pdf
..                              BMLreport
```

ls -a means to show all files, including the hidden files starting with a dot (".").

The two hidden files here are special and exist in every directory. "." refers to the current directory, and ".." refers to the directory above it.

This brings us to the distinction between *relative and absolute path names*. (Think of a path like an address in UNIX, telling you where you are in the directory tree.)

You may have noticed that I typed cd Desktop/BMLHW, rather than **cd /Users/nolan/Desktop/BMLHW**.

The first is the relative path; the second is the absolute path.
To refer to a file, you need to refer to it using a relative or absolute path name.

Example:
$ pwd
/Users/nolan/Desktop/BMLHW

$ ls BMLcode/
analyze.R genCars.R moveCars.R runSim.R

Example:
$ cd ..

Note that file names must be unique within a particular directory, but having,
/Users/nolan/Desktop/genCars.R and
/Users/nolan/Desktop/BMLHW/BMLcode/genCars.R is OK.

Running R in batch
We can start R BATCH jobs from the UNIX command line.
BATCH jobs are useful whenever you: have a long job and want to use the computer for other things in the meantime; want to log out of the machine while the job is running and come back to it later; run the job on a remote machine.

To start a BATCH job, use:
nice R CMD BATCH scriptfile.R outfile.Rout &

(Actually on the lab computers all jobs are "niced" by default, so this isn't strictly necessary.)

Shell command syntax

command -options arg1 arg2

Blanks and – are delimiters
The number of arguments may vary.
An argument comes at the end of the command line.
It's usually the name of a file or some text.
Many commands have default argument and so the syntax is $ command

Example: move/rename a file.
$ mv test.txt newname.txt

This command has two arguments.
It moves the file test.txt to newname.txt

Options
Options come between the command and the arguments.
They tell the command to do something other than its default. They are usually prefaced with one or two hyphens.

$ rmdir unixexamples
rmdir: unixexamples: Directory not empty
$ rm -r unixexamples

Wildcards
You can refer to multiple files at once using *wildcards*. The most common one is the asterisk (*). It stands in for anything (including nothing at all).

$ ls
AGing.txt Bing.xt Gagging.text Going.nxt ing.ext
$ ls G*
Gagging.text Going.nxt
$ ls *.xt
Bing.xt

The question mark (?) is similar, except it can only represent a single character.
$ ls
AGing.txt Bing.xt Gagging.text Going.nxt ing.ext
$ ls ?ing.xt
Bing.xt

Finally, square brackets can be replaced by whatever characters are within those brackets.
$ ls
AGing.txt Bing.xt Gagging.text Going.nxt ing.ext
$ ls [A-G]ing.*
Bing.xt

The wildcards can also be combined.
$ ls *G*
AGing.txt Gagging.text Going.nxt
$ ls *i*.*e*
Gagging.text ing.ext

Redirection & Pipes
Redirection refers to changing the input and output of individual commands/programs.
The "standard input" or STDIN is usually your keyboard.
The "standard output" or STDOUT is usually your terminal (monitor).

As an example, if we type cat at the prompt and hit return, the computer will accept input from us until it hits an end-of-file (EOF) marker, which on most systems is CNTRL-D. Each time we hit return, our input is printed to the terminal.

We can redirect as follows

> redirects STDOUT to a file
< redirects STDIN from a file
>> redirects STDOUT to a file, but appends rather than overwriting.

(There's also a <<, but its use is more advanced than what we'll cover.)
Here are two examples:

```
$ cat > temp.txt
$ sort < temp.txt
```

Compound Expressions & Pipes
The idea behind pipes is that rather than redirecting output to a file, we redirect it into another program.

Another way to say this is that STDOUT of one program is used as STDIN to another program.

A common use of pipes is to view the output of a command through a *pager,* like less. This is particularly useful if the output is very long.

```
$ ls | less
```

Note that the data flows from left to right.
A program through which data is piped is called a *filter*.
We have seen a few filters: head, tail, and wc.

Two more common filters are
sort - sort lines of text files alphabetically
uniq - strip duplicate lines when they follow each other

```
$ cat somenumbers.txt
One
```

Two
One
One
Two
Two
Two
Three
Three
One

$ cat somenumbers.txt | sort | uniq
One
Three
Two

Here is one more useful filter:
grep - print lines matching a pattern (We'll talk about patterns more shortly -- for now just think of the pattern as requiring an exact match.)

$ grep "save" *.R

This functions prints all lines in any file ending with .R which contain the word (pattern) save.

Getting Help
To look at the syntax of any particular UNIX command, type man (for "manual") and then the name of the command.
The two most important parts the SYNOPSIS and DESCRIPTION. These are very much like the "Usage" and "Arguments" in R's help pages.
SYNOPSIS shows you the syntax for a particular command. Bracketed arguments are optional.
DESCRIPTION tells you what all the options do.
Press the space bar to scroll forward through the man page, "b" to go backward, and "q" to exit.

Here are a few more handy commands:

wc -l - count the number of lines in a file
head -x - look at the first x lines of a file (default n=10)
tail - like head, but look at the end of the file;
cp - copy a file
$ cp unixexamples/Bing.xt .

cat - print the contents of a file

Managing a job

To see information about currently running processes, just type top. There are arguments to top that allow you to sort by CPU usage, memory, etc. See man top for more details.

The number at the beginning of the line is called the *process ID*, or PID.

To kill a particular process (stop it from running), type kill PID, substituting the correct PID.

Sometimes you want to see the list of all processes in a non-interactive way. For example, you might want to pipe the results through a filter, as we'll discuss later.

On BSD UNIX systems (like the Apple machines in the lab), ps -aux will list all processes.

On other systems, ps -ef does the trick.

Advantages of Shell Commands

Shell commands give us a programmatic way to work with files and processes (rather than a point-and-click "manual" approach).

- Document: if you need to record what you did
- Reproduce: repeat another time
- Volume: have many many operations to perform
- Speed: need to perform things quickly
- Less error prone: want to reduce mistakes

Text Manipulation

String manipulation functions

substring(text, first, last) – extract a portion of a character sting from text, beginning at first, ending at last

nchar(text) – return the number of characters in a string

strsplit(x, split) – split the string into pieces using split to divide it strsplit(x, “”) – splits into one character pieces

paste(x, y, z, …, sep = " ", collapse = NULL) – paste together character strings separated by one blank

tolower(x) toupper(x) - convert upper-case characters to lower-case, or vice versa. Non-alphabetic characters are left unchanged

Example:

```
> cNames
[1] "Dewitt County"
[2] "Lac qui Parle County"
[3] "St John the Baptist Parish"
[4] "Stone County”
```

How to add a . after St in St John?

```
> test = cNames[3]
> substring(test, 1, 3)
[1] "St "
> substring(test, 1, 3)  ==  "St "
[1] TRUE
> newName =
paste("St. ", substring(test, 4, nchar(test)),  sep ="")
```

Another approach

```
> string = "The Slippery St Frances"
> chars = unlist(strsplit(string, ""))
> chars
 [1] "T" "h" "e" " " "S" "l" "i" "p" "p" "e" "r"
[12] "y" " " "S" "t" " " "F" "r" "a" "n" "c" "e"  "s"
> possible = which(chars == "S")
> possible
[1]  5 14
> substring(string, possible, possible + 2) == "St "
[1] FALSE  TRUE
```

Computational model for pattern matching

Look at each character
Check to see if it is “S”
If it is, then look at the next character(s)

This is the idea behind regular expressions

The **regular expression** “St ” is made up of three **literal** characters. The **regular expression matching engine** does something very similar to what we just did.

Luckily, we don’t actually need to write our own functions for replacement. The R functions gsub() and sub() look for a pattern and replace it within a string with some other text.

The “g” in gsub() refers to global. It changes all the matches, whereas sub() only replaces the first match (in each element – both gsub() and sub are vectorized).

```
> gsub("St ", "St. ", cNames)
[1] "Dewitt County"
[2] "Lac qui Parle County"
[3] "St. John the Baptist Parish"
[4] "Stone County"
```

```
> strings = c("a test", "and one and one is two",  "one two three")
> gsub("one", "1", strings)
[1] "a test"  "and 1 and 1 is two"  "1 two three"
```

```
> sub("one", "1", strings)
[1] "a test"  "and 1 and one is two"  "1 two three"
```

Regular Expressions

Regular expressions give us a powerful way of matching patterns in text data

Most importantly, we do this all *programatically* rather than by hand, so that we can easily reproduce our work if needed.

With regular expressions, we can

- extract pieces of text – e.g., find all links in an HTML document
- create variables from information found in text
- clean and transform text into a uniform format, resolving inconsistencies in format

between files
mine text by treating documents directly as data
"scrape" the web for data

A *regular expression* (aka regex or regexp) is a pattern that describes a set of strings.

This set may be finite or infinite, depending on the particular regexp. We say the regexp "matches" each element of that set.

Syntax:

Literal *characters* are matched only by the character itself.

A ***character class*** is matched by *any* single member of the specified class. For example **[A-Z]** is matched by any capital letter.

Modifiers operate on literal characters, character classes, or combinations of the two. For example ^ is an anchor that indicates the literal must appear at the beginning of the string

Warning

The syntax for regexps is *extremely* concise
It can be overwhelming if you try to read it like you would regular text.
Always break it down into these three components: literals, character classes, modifiers

Equivalent Characters

Concepts of "numbers", "punctuation", and "regular letters" get at the idea of *equivalent characters* or *character classes*.

We can enumerate any collection of characters within [ ]. Example: **[Tt]his**

The character "-" when used within the character class pattern identifies a range. Examples: **[0-9]**, **[A-Za-z]**

If we put a caret (^) as the first character , this indicates that the equivalent characters are the **complement** of the enumerated characters. Example: **[^0-9]**

If we want to include the character "-" in the set of characters to match, put it at the beginning of the character set to avoid confusion. Example: **[-+][0-9]**

Named Equivalence Classes

These can be used in conjunction with other characters, for example **[[:digit:]_]**

grep(pattern, x) It looks for the regular expression in pattern in the character string(s) in x. It returns the *indices* of the elements for which there was a match.

gsub(pattern, replacement, x) Look the regular expression in pattern in x and replace the matching characters with replacement (all occurrences) sub() works the same way but only replaces the first occurrence.

regexpr(pattern, text) returns an integer vector giving the starting position of the first match or *-1* if there is none. The return value has an attribute "match.length", that gives the length of the matched text (or *-1* for no match).

gregexpr(pattern,text)Returns the locations of all occurrences of the pattern in each element of text. The return is a list.

```
> subjectLines
[1] "Re: 90 days" "Fancy rep1!c@ted watches" "It's me"
> grep("[[:alpha:]][[:digit:][:punct:]][[:alpha:]]", subjectLines)
[1] 2 3
```

gregexpr() shows exactly where the pattern was found

To search for the more general pattern of *any* number of digits or punctuation marks between letters, we use

[[:alpha:]][[:digit:][:punct:]]+[[:alpha:]]

The plus sign indicates that members from the second character class (digits and punctuation) may appear *one or more* times.

The plus sign is an example of a *meta character*.

Meta characters

Meta characters that control *how many times* something is repeated

The position of a character in a pattern determines whether it is treated as a meta character.

Examples: [-+*/], [1-9]*

When you want to refer to one of these symbols literally, you need to precede it with a backslash (\). However, this already has a special meaning in R's character strings -- it's

used to indicate control characters like newline (\n).

So to refer to these symbols in R's regular expressions, you need to precede them with *two* backslashes. The characters for which you need to do this are:
. ^ $ + ? ( ) [ ] { } | \
. any character
^ anchor to the beginning of the string
$ anchor to the end of the string
+ one or more times
? 0 or 1 time
* 0 or more times
{} range, e.g. {3} exactly 3, {2,6} two through 6, {5,} five or more
| Or
\ - escape
() group together, e.g. gr(e|a)y

Greedy Matching
Be careful with patterns matching too much. The matching is greedy in that it matches as much as possible.

For example: when trying to remove HTML tags from a document, the regular expression <.*> will match too much but the regular expression <[^>]*> will be just right. Why?

eXtensible Markup Language: XML

Most of the data sets we have seen have been in the form of ASCII tables.
Advantages: easy to read, write, and process; in standard cases, don't need a lot of extra information. But these advantages can quickly disappear.

XML is a standard for *semantic*, *hierarchical* representation of data

Pros: data is self-describing; format separates content from structure; data can be easily merged and exchanged; file is human-readable; file is also easily machine-generated; standards are widely adopted.

Cons: XML documents can be very verbose and hard to read; it's so general that it's hard to develop tools for all cases; files can be quite large due to high amount of redundancy; XML has become quite popular in many scientific fields, and it is standard in many web applications for the exchange and visualization of data.

Example:

```
<Placemark id="217">
 <name>8.2</name>
 <description>
Date: 2008-9-15
Magnitude: 1.5
Depth: 8.2 km
 </description>
 <styleUrl>#ball1-2</styleUrl>
 <Point>
  <coordinates>-147.426, 60.929, 0</coordinates>
 </Point>
</Placemark>
```

Syntax

The basic unit of XML code is called an "element" or "node." It is made up of both *markup* and content. Markup consists of *tags, attributes*, and *comments*.

Well-formed

Tag names are case-sensitive; start and end tags must match exactly.
All XML documents must contain a *root node* containing all the other nodes
No spaces are allowed between the < and the tag name.
Tag names must begin with a letter and contain only alphanumeric characters.
An element must have both an open and closing tag unless it is empty.
An empty element that does not have a closing tag must be of the form

<tagname/>.
Tags must nest properly. (Inner tags must close before outer ones.)
All attributes must appear in quotes in the format: **name = "value"**
Isolated markup characters must be specified via entity references. < is specified by < and > is specified by >.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Edited with XML Spy v2006 (http://www.altova.com) -->
<CATALOG>
	<PLANT>
		<COMMON>Bloodroot</COMMON>
		<BOTANICAL>Sanguinaria canadensis</BOTANICAL>
		<ZONE>4</ZONE>
		<LIGHT>Mostly Shady</LIGHT>
		<PRICE>$2.44</PRICE>
		<AVAILABILITY>031599</AVAILABILITY>
	</PLANT>
	<PLANT>
		<COMMON>Columbine</COMMON>
		<BOTANICAL>Aquilegia canadensis</BOTANICAL>
		<ZONE>3</ZONE>
		<LIGHT>Mostly Shady</LIGHT>
		<PRICE>$9.37</PRICE>
		<AVAILABILITY>030699</AVAILABILITY>
	</PLANT>
	<PLANT>
		<COMMON>Marsh Marigold</COMMON>
		<BOTANICAL>Caltha palustris</BOTANICAL>
		<ZONE>4</ZONE>
		<LIGHT>Mostly Sunny</LIGHT>
		<PRICE>$6.81</PRICE>
		<AVAILABILITY>051799</AVAILABILITY>
	</PLANT>
</CATALOG>
```

Tree Representation

Well-formed XML can be represented in a tree structure. This tree structure is called the Document Object Model or DOM for short.

Tree terminology

There is only one *root or document node* in the tree, and all the other nodes are

contained within it.
We think of these other nodes as being *descendants* of the root node.
We use the language of a family tree to refer to relationships between nodes.
Parents, children, siblings, ancestors, descendants
The *terminal nodes* in a tree are also known as *leaf nodes*. Content always falls in a leaf node.

Handy functions for parsing XML
readHTMLTable: reads an HTML table into R
xmlParse: read an XML file into R
xmlValue: retrieve text content of a node
xmlSize: return the number of child nodes of a node
xmlSApply: applies the function to each child node of a node - a special XML version of sapply that takes an XMLNode object as its primary argument. It iterates over the node's children nodes, invoking the given function.
xmlRoot: extract the root node from the document

The tree structure is represented in R as a list of lists. We can access an element within a node (i.e., a child), using the usual [[ ]] indexing for lists.

```
> ## Look at the first plant node
> oneplant = root[[1]]
> class(oneplant)
[1] "XMLNode"
```

XPath

XPath is an extraction tool designed for locating content in an XML file. It uses the hierarchy of a well-formed XML document to specify the desired chunks to extract. The syntax is similar to but more powerful than the way files are located in a hierarchy of directories in a computer file system.

Example:

```
<Envelope>
  <subject>Reference rates</subject>
  <Sender>
    <name>European Central Bank</name>
 </Sender>
 <Cube>
   <Cube time="2008-04-21">
     <Cube currency="USD" rate="1.5898"/>
    <Cube currency="JPY" rate="164.43"/>
```

```
    <Cube currency="BGN" rate="1.9558"/>
    <Cube currency="CZK" rate="25.091"/>
  </Cube>
  <Cube time="2008-04-17">
    <Cube currency="USD" rate="1.5872"/>
    <Cube currency="JPY" rate="162.74"/>
    <Cube currency="BGN" rate="1.9558"/>
    <Cube currency="CZK" rate="24.975"/>
  </Cube>
 </Cube>
</Envelope>
```

The XPath expression: /Envelope/Sender/name follows the following steps:

1. The first location step identifies the root node, **<Envelope>**.
2. The next location step finds the **<Sender>** child of **<Envelope>**.
3. The third location step identifies **<Sender>**'s child called **<name>**.

/Envelope/Cube/Cube finds all <Cube> elements that have a <Cube> parent and <Envelop> grandparent (which is specified to be the root of the document)

//Cube[@currency = "JPY"] finds all <Cube> elements anywhere in the document such that the node has a currency attribute with a value of JPY

XPath syntax

XPath locates sets of nodes in XML files. An XPath expression is a *location path* that is made up of *location steps*, each of which has three parts: the axis, nodetest, and predicate, that follow the syntax:

axis::nodetest[predicate]

The location step can be thought of as directions from one location (or context) to another. The nodetest is typically a node name that you wish to locate. The predicate filters the qualifying nodes.

In our case, the axis will be either “child”, which is the default and can be dropped, or “descendant-or-self”, which says look anywhere down the tree and is abbreviated by “//”. The nodetest will always be a node name, and the predicate will either be a number, [2] which asks for, e.g., the second node, or an attribute filter, e.g., [@currency = “JPY” or @currency=“USD”]

Functions that take XPath expressions

getNodeSet(xmlTree, xpathExpression) returns a list of XML nodes from xmlTree that

satisfy the XPath expression.

xpathSApply(xmlTree, xpath, function) the function is applied to those nodes in the XML tree that satisfy the Xpath expression. The return value is a vector when possible. xpathApply returns a list.

Example
> xpathSApply(catalog, "/CATALOG/PLANT/BOTANICAL", xmlValue)
[1] "Sanguinaria canadensis" "Aquilegia canadensis" "Caltha palustris"

One more function in XML package
readHTMLTable(doc, header = TRUE, colClasses = NULL, skip.rows = 3, which = 2)

Generating XML
Handy functions for creating XML
newXMLDoc: create a new XML document
newXMLNode: create a new XML Node
saveXML: save the XML tree in a text file
parseXMLAndAdd: parses character string of XML into an XML node and adds it to a tree

Example
```
doc = newXMLDoc()
root = newXMLNode("toplevel", doc = doc)
child1 = newXMLNode("level1", parent = root)
newXMLNode("level2", "This is the content", parent = child1)
saveXML(doc, file = "simple.xml")
```

Note: We only need to store (assign to a variable) nodes that we want to refer to as parents. For the leaf nodes, I just use newXMLNode without assigning. The names of the nodes in R (e.g. root, child1) are *not* part of the resulting XML file.

JavaScript Object Notation: JSON

Features of JSON: text format; lightweight data-interchange; easy for humans to read and write; easy for machines to parse and generate.

J

JSON is built on two structures:

An *unordered collection of comma-separated* name:value pairs

{“lender_id”:”matt”, “loan_count”:23}

An *ordered array of values*

[{“lender_id”:”matt”, “loan_count”:23}, {“lender_id”:”skylar”}, {extra: 2}]

Comparison to XML

JSON is simpler

Not as rich – no attributes, unordered, no schema for describing acceptable format

Compressed JSON and XML not much different in size

Resampling

Resampling is a broad term that refers to techniques which use repeated samples from some empirical distribution. Each sample, treated as a pseudo-data set, is analyzed in some way (perhaps just summarized) and by repeating this process, getting slightly different results each time, we can get some idea of the variability in what we're doing.

Monte Carlo

Monte Carlo Methods are not technically resampling, but they are related and we've done them. For example, we can sample repeatedly from the uniform distribution and used these samples to approximate the distribution of the maximum difference between observed quantiles and their expected values. We then compare this approximate distribution to actual maximum differences from some data thought to follow the uniform to decide whether these data seem to come from the uniform distribution.

The Bootstrap Idea

The original sample approximates the population. If we replicate sample values to build up an approximate population, we can then sample repeatedly from this population. In the height example, we then get many samples of size 10 and treat each as a new pseudo-data set and the average is computed. The distribution of the sample averages now gives some idea of what the sample average could have been if a different set of 10 students were chosen.

Example: Suppose we have height data on 10 students chosen at random from the class, measured in inches:

```
> heights=c(66,68,62,69,65,62,70,72,63,61)
```

Based on these heights, we can estimate the average height of all the students in the class. Our best guess is the average of the sample heights. How accurate is this guess? One way to answer this question, which you might learn in another statistics class, is to make assumptions about the distribution of the sample average (perhaps it follows the normal curve) and do some calculations. The bootstrap method provides a way to approximate the distribution of the sample average without making any such assumptions.

```
> bootAvgs = replicate(1000, mean(sample(heights, size = 10, replace = TRUE)))
```

Cross Validation

Cross validation is a technique for assessing how well the results of some statistical analysis will generalize to an independent data set. Using the same data to create the model as well as assess the model can lead to an overly optimistic view of how well the model will be able to make predictions in an independent data set. The idea of cross validation is to leave some data aside as a test set (to assess the predictions) while the rest of the data is a training set used to make decisions about how to make predictions.