CS 109L NOTES

ARUN DEBRAY
JUNE 3, 2014

Contents

1. Introduction to R: 4/1/14

This class will spend about two weeks on a quick introduction to the R programming language, followed by applications of it to the concepts covered in CS 109 (or the equivalent courses, e.g. Math 151). A good first step is to install the R programming language itself, and then pick a text editor or IDE.

R resembles Python and Matlab more so than Java or C++: it is an interpreted language, and uses a REPL, a read-evaluate-print loop. Basic arithmetic operations are easy, e.g. `2 + 3`, `2 * 3`, and `2 - 5`. Integer division produces a float, e.g. `9 / 4` returns `2.25`, akin to matlab rather than Java; for integer division, use `9 %/% 2`. The modular arithmetic operator is `%%`. There are also lots of nice builtin math functions, e.g. `sqrt(5)`, as well as `cos`, `sin`, `log` (natural logarithm), `exp`, and `pi` (to six decimal places). To do variable assignment, write `<-`; you can use `=`, but it's not recommended (for reasons we'll go over later).

One interesting fact about R is that all things are vectors, even bare integers; the integers seen above are handled as vectors! One can concatenate with the `c()` function, e.g. `c(1, 3, 6, 9)` returns the vector `1 3 6 9`. If this is stored as a variable `x`, then addition, etc., are componentwise: `3 + x` returns 3 plus ever element of `x` as a vector. However, if one attempts to add vectors, life gets more interesting: `c(1, -1) + c(1, 2, 3, 4)` returns `2 1 4 3`. This is the notion of `vector recycling`: the smaller vector is cloned enough times to fit into the larger one. if the dimensions don't fit, there's a warning, but the assignment still goes through. Interestingly, lists aren't nested: `c(0, x)` flattens `x` and then concatenates.

Regular functions as mentioned are vectorized, e.g. `sqrt()` happily accepts lists. But there are other functions which make more sense on vectors: `sum()` and `prod()` take the sum and product of a vector's entries, and `length()`, `min()`, and `max()` do what you might expect. Then, `order()` gives the order of indices necessary to sort the vector into ascending order, and `sort()` sorts a vector. Finally, we have statistics: `mean()`, `sd()` (standard deviation), and `var()` (variance).

For a little history, S is a statistical programming language, and R is an open-source version. But since it's open-source, there are a lot of community additions to the language, especially on the CRAN website (`http://cran.r-project.org//`). One interesting consequence is that there's a built-in `man` page: if one types `?order`, it provides a description of the `order()` function.

One nuance of R is that it's a functional programming language. Functional programming isn't a huge thing at Stanford, but it's now making a resurgence, because it's important in certain paradigms (e.g. parallel computing). In

R, this means that functions are first-class. For example, functions are in scope, and if one enters a function without parentheses, its source is printed.

Instead of just using `c()`, there's nice syntactic sugar for vectors. For example, `1:4` returns `1 2 3 4`. If `n` holds the number 10, then `1:n` returns what you expect. But the colon operator binds very tightly; for example, `1:n+1` is parsed as `(1:n)+1`, which returns the vector of 2 through 11; `1:(n+1)` returns the vector 1 to 11. Pay heed to this: it's a particularly subtle source of some bugs.

A function called `seq()` allows one to create more general arithmetic sequences, and will provide a useful introduction to R's strange argument passing. The default values for its `from` and `to` arguments are 1, so `seq()` returns 1. In Python, there are positional and keyword arguments, and this happens again in R; the default for `seq()` is `from`, then `to`, so `seq(4,7)` returns `4 5 6 7`, but `seq(to=7, from=4)` returns `4 5 6 7`. This is pretty nice, but worryingly, `seq(t=7, f=4)` also returns `4 5 6 7`. It's interesting, but *don't* try this at home. Finally, there's a `by` argument that denotes the step size; for example, `seq(to=7, from=4, by=2)` returns `4 6`. One could also instead pass in `length.out` as the third parameter, which indicates what size vector to return (with even increments).

R is one-indexed. That's unfortunate.

For another example of arguments, `log()` is the natural logarithm; we also have `log10()` and `log2()`. But it's just as easy to call `log(5^10, base=5)`, returning 10.

Another useful function for creating vectors is `rep()`, which replicates lists. For example, `rep(1:4, times=4)` returns `1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4`, and `rep(1:3, each=4)` returns `1 1 1 1 2 2 2 2 3 3 3 3`. Notice the difference!

For types, we have seen that R has integers and floats (but as vectors), but we also have the logical type, with `TRUE` and `FALSE`. By default, R also provides `T` and `F`, which are really easy to overwrite (e.g. `F <- TRUE`), so it's best to use the longer ones. (The shorter ones will be used for brevity sometimes in these notes.) Yet these are *also* vectors, e.g. `length(TRUE)` returns `1`. Thus, one can cons together Booleans: `c(T, F, F, T, T)` returns `TRUE FALSE FALSE TRUE TRUE`.

R doesn't have casting; instead, there's something called coercion. In an integer context, `TRUE` is treated as 1, and `FALSE` as 0. Thus, we can do something like `mean(c(T, F, T, T, F))`, which returns 0.6. This seems kind of absutd, but can be useful for understanding percentages of data.

These Booleans allow for a small conditional: `if(TRUE) 3 else 4`, which (surprise) returns 3. You can throw in other Booleans, such as `if(x == 3) 3 else 4`. Using curly braces, one can make multi-line statements:

```
if(x == 3) {
    3
} else {
    4
}
```

Of coure, you can't use `else` without `if`. Sometimes, mutli-line statements are tricky, because of the interpreter, as in Python.

== also uses vector recycling, so `c(1, 2, 4, 2) == 1:2` returns `TRUE TRUE FALSE TRUE`. We have component-wise and, which is `&`, and a short-circuiting version `x && y` (which returns `TRUE` if one of the components is). (For special operators like `&&`, it's necessary to use backticks to ask for help, e.g. `?‘&&‘`.) The same thing happens with or: `|` and `||`.

We also have a complex type, e.g. `4+2i`, and strings. These are *not* always treated as lists of characters, as

```
> length(c("this", "is", "a", "vector"))
[1] 4
```

. . . which is not what you might expect. Types can be coerced, and if one concatenates a bunch of things that include a string, everything else is coerced into strings, e.g. `3` into `"3"`, `T` into `"TRUE"`.

Indexing into lists in R is very powerful. The standard `x[1]` returns the first element of `x`, but one can also create sublists; for example, if `x <- 1:4`, then `x[c(1, 4)]` returns `1 4`. Indices can be out of order, or repeated, and each time the relevant entry is looked up and put in place. But negative indices indicate removing things; for example, `x[-1]` returns all but the first element! However, R makes a warning for negative subscripts. A zero in an index just doesn't do anything, and `x[0]` returns some type of nothing (e.g. the empty string). But one can also index with Booleans, where a true indicates that something should be in a sublist and false indicates it shouldn't be included. Vector recycling may happen, so `x[c(TRUE, FALSE)]` returns every other element of any-sized list.

This means that one can make logical vectors as list comprehensions, e.g. `y[y %% 2 == 0]`, which does the same thing. Or you could do `y[y %% 3 == 1]`. Or you can choose `y[y > 67]` or `y[y <= 51]`. Notice how much more powerful this is than for other programming languages.

Even more interestingly, sublists are left-hand values, so `y[y %% 4 == 0] <- 0` sets every fourth element of `y` equal to 0, and leaves the rest of `y` in place. This, along with a small amount of functional thinking, allows one to answer interesting questions quickly.

We have already developed enough tools to approximate integrals:

```
> dx <- 0.05
> xs <- seq(from=0, to=10, by=dx)
> sum(xs ^ 2 * dx)
[1] 335.8375
```

This is a Riemann sum approximation of

$$\int_0^{10} x^2\,\mathrm{d}x = \frac{1000}{3} \approx 333.3.$$

2. Lists and Functions: 4/3/14

Vectors are homogeneous in `R`, which means that everything is of a single type, so operations like `c(1:6, TRUE)` coerces `TRUE` to `1`. But there is a heterogeneous data type, called a list. For example, one could obtain a list of the form `list(1, 1:10, c("a", "b", "c"), c(T, F, T), sin)`. Notice that lists can contain functions or even other lists! Then, then individual elements are given with double braces, e.g. `[[1]]` for the first element. Thus, this is how one indexes into a list: querying `mylist[[1]]` returns the first value. But notice that using single-indexing *also* works; for eample, `mylist[c(1, 2, 4)]` returns the sublist of whatever was at the first, second, and fourth indices. This means that `mylist[1]` returns a single-element sublist containing the first element rather than the first element, which is unlikely to be what you intended! There's also a doube-brace indexing convention for vectors, which has a similar meaning, but not the same.

Another piece of weirdness is that periods are perfectly fine within variable names. Sometimes people use this akin to camelCase, but there's no one standard.

Lists can have named elements, which turns them into things that more resemble `structs` in C. For example, one could write `list(vals=1:20, fun=sqrt, name="a list")`, and then one could use indexing like `mylist[[1]]`, but also `mylist[["vals"]]` (notice that we still need double brackets, and quotes), or alternatively, `mylist$vals`. So it makes perfect sense to call `mylist$fun(mylist$vals)`, and this can be useful.

All of the fancy indexing we saw last time for vectors works for lists too, but in the context of sublists and single-indexing (e.g. `mylist[-1]` to remove the first element).

The function `unlist()` takes a list and flattens it. In some sense, a list in `R` has a tree-like structure, and `unlist()` cleans that up. By default, this function coerces the types and returns a vector! There's an option called `recursive`, set to true by default, which recursively flattens the list, so when it's set to false, it returns a list which i one stage shallower (instead of a vector).

Since lists are heterogeneous, some vector operations, such as componentwise addition or multiplication, aren't defined on lists.

Lists are in some sense the central data structure in `R`. For example:

```
> x <- 1:10
> y <- x^2
> plot(x,y)
> abline(lm(y ~ x), col = 'red') # line of best fit, will be covered later
> model <- lm(y ~ x)
> model
#[displays the model; omitted]
> str(model) # structure of the model
```

The point here is that the structure of the model is a list, so one can ask for information about the model, e.g. `model$coefficients` returns a vector of coeffiients.

The `mode()` function returns (as a string) the type of something, e.g. `"list"`, `"numeric"`, `"logical"`, or `"complex"`.

To invoke a script, use `source(`*filename*`)`.

For control flow, we have `if` as mentioned last time, as well as a `for` loop (actually a for-each). It looks like this:

```
nums <- 1:10
for (i in nums) {
    print(i)
}
```

It used to be a huge deal that for-each loops existed (e.g. back before they appeared in Java and C++), but now this is familiar to more people. It prints the numbers in order, which is unlikely to be a surprise. But one can also pass in a list instead of a vector, which does the same thing.

There's also a `while` loop:

```
i <- 0
while( i < 10) {
    print(i)
    i <- i + 1
}
```

But there are in general better ways to loop over things than the `for` and `while` loops, both from a practical standpoint and because `R` is a functional language.

Functions in `R` are anonymous, in the sense that defining a function is much like assigning a value to a variable. For example:

```
myfn <- function(x) {
    return(2 * x)
}
```

Be careful, though; `return()` is a function, not a keyword! That's why the parentheses are necessary. But it's not strictly necessary to use it; if no `return` keyword is specified, then the last value processed returns. This is becoming more common in programming languages, e.g. Scala. So you could rewrite the above code like this:

```
myfn <- function(x) {
    2 * x
}
```

This can actually be reduced to a single-line function, with the follwing syntax:

```
myfn <- function(x) 2 * x
```

Any of these definitions allows us to call `myfn(4)`, which returns 8.

There's a good amount of conciceness: one can calculate the norm $\|\mathbf{x}\|$ of a vector $\mathbf{x}$ as follows:

```
norm <- function(x) sqrt(sum(x^2))
```

But one piece of weirdness is that there was no type-checking. If one passes in arguments of a different type than intended, it causes coercion, but doesn't raise a warning or error. To make a function type-safe, one has to use `stop()`, as in the following:

```
myfn <- function(x) {
    if (!is.numeric(x)) stop("myfn only doubles numbers")
    return (2 * x)
}
```

`is.numeric()` is one of a family of functions, e.g. `is.complex()` and so on. But there's also `as.numeric()`, `as.complex()`, etc., which coerce into the given type.

Functions can have named arguments, such as

```
new.fn <- function(a = 10, b = sin) {
    b(a)
}
```

This has named arguments a and b, with default values, so one can call `new.fn()`, `new.fn(1:5, prod)`, `new.fn(b=prod, 1:5)`.

Let's write an actual function, the good-old factorial.

```
fact <- function(n) {
    is (!is.numeric) stop("Factorial is meaningless on nonnumbers")
    if (n < 0) stop ("Cannot take factorial of negative number")
    if (n %% 1 != 0) stop("Factorial only defined on integers")
    # could be iterative, but we might as well write it recursively
    if (n == 0) return(1)
    n * fact(n-1)
}
```

Of course, R has a built-in factorial function `factorial()`, or you could use `prod(1:n)`. But `factorial()` is vectorized, so it accepts vectors of numbers, and computes the factorial componentwise, but our `fact()` function fails horribly on vectors. The key is that if `n` is a vector, then `n == 0` is also a vector, so `if` isn't happy. Here's one way to vectorize the function:

```
fact.vec <- function(vec) {
    # Not as functional as it could be -- but that's a story for later.
    for (i in 1:length(vec)) {
        vec[i] <- fact(vec[i])
    }
    vec
}
```

Here's a function that computes the cumulative sum of a vector `A`, i.e. a vector `S` such that $\mathtt{S}[i] = \sum_{j=1}^{i} \mathtt{A}[j]$.

```
cumulative.sum <- function(x) {
    # Type-checking omitted
    if (length(x) == 1) return(x)
    # First entry is unchanged
    for(i in 2:length(x)) {
        x[i] <- x[i] + x[i - 1]
    }
}
```

We have a function called `choose()`, which is the binomial coefficient function: `choose(a, b)` returns $\binom{a}{b}$. Both arguments can be vectorized, which makes life a little interesting; what if one calls `choose(5:10, 1:2)`? Ideally, one would get the Cartesian poroduct, but in R, this means vector recycling: beggars can't be choosers. Thus, the result returned is `5 15 28 9 45` (i.e. $\binom{5}{1}$, $\binom{5}{2}$, $\binom{6}{1}$, and so on). It can be implemented recursively, using Pascal's triangle:

```
comb <- function(n, k) {
    if (k == 0 || n == k) return(1)
    comb(n - 1, k) + comb(n-1, k-1)
}
```

There would be a little more work to vectorize it as `choose` does, and that involves making some decisions about how to vectorize.

R is not the language you want to use for lots of matrix math; it has some support for matrices, but not as much as Matlab. Matrices are backed by vectors. Given some vector `mat`, which has `mode()` numeric and `class()` integer, one can write `dim(mat) <- c(5, 6)`. Now, it has class matrix, but still a numeric mode. It's still the same thing with respect to single-indexing like `mat[3]`, but it's also possible to call `mat[3,5]`, which returns the element in the 5th column of the 3rd row. And all of the fancy indexing from last time still works, e.g. `mat <- mat[,-1]`, which drops the last column.

Since a matrix is basically a vector, we can still do things like `mat + 3` or `mat + c(1,1)`, which does the vector recycling that is routine by now. The function `%*%` does matrix multiplication, `det()` calculates the determinant, and `t()` calculates the transpose. Then, `nrow()` and `ncol()` return the number of rows and columns, and one can use the `matrix()` command to initialize a matrix from a vector, e.g. `matrix(1:30, nrow=5, ncol=6, byrow=T)` (rows filled first), which does what we had done by assigning the dimension earlier.

Higher-order tensors certainly exist, and are created by changing the dimension to a vector of length 3 or more, but these won't happen in this class. They have `class()` array.

Exercise 2.1.

```
## 1) Consider the one-player puzzle where you're presented with a 2 by n
##    board of small positive integers, like the following:
##      | 5 | 7 | 1 | 8 | 4 | 8 | 2 | 6 | 2 | 1 | 1 | 5 | 1 |
##      -----------------------------------------------------
##      | 7 | 1 | 3 | 3 | 4 | 9 | 6 | 1 | 5 | 9 | 2 | 3 | 2 |
##    Suppose that you are given an unlimited number of pebbles, and your
##    task is to distribute pebbles across the board (with at most one pebble
##    per square) subject to the constraint that you can't place pebbles in
##    vertically, horizontally, or diagonally adjacent locations. Your score
##    is the sum of all the values inside the squares covered by pebbles.
##
##    Write the function 'max.game.score' which, given a 2 by n matrix,
##    returns the maximum game score that can be achieved by playing the game
##    specified above.

max.game.score <- function(board) {
  ## Returns the maximum game score achievable on 'board'.
}

## > max.game.score(matrix(1:20,nrow=2,ncol=10))
## [1] 60
```

Solution. The idea is that the maximum is given either by placing a pebble in the first column, or in the second column, and then determining what's best after that.

```
max.game.score <- function(board) {
    if (ncol(board) == 0) return(0)
    if (ncol(board) == 1) return(max(board))
    x <- max.game.score(board[,c(-1,-2)])
    y <- max.game.score(board[,-1])

    max(board[1,1] + y, board[2, 1] + y, board[1, 2] + x, board[2, 2] + x)
}
```

3. Factors: 4/8/14

"Does anyone here study frogs? There's always someone."

One minor point about lists is that the `c()` operator (concatenate or cons) can be used to add elements to a list. Specifically, `c(`*list*`,` *item*`)` appends the item to the end of the list, or joins two sublists.

Factors are R's way of keeping track of categorical data. For example, one might study frogs, and want to study lots of different attributes of frogs, including their size, weight, and so on. In Java or C, one might use an `enum` to handle this, but R can track these more powerful. For example, to make a list of colors for frogs:

```
> frag.color <- c("green", "purple", "purple", "mixed", "white", "green", "green", "mixed")
> frog.color <- factor(frag.color)
```

The `factor()` function is rather versatile: it accepts different types (argument `x`, defaults to `character()`, and can be ordered `ordered` argument, defaulting to `is.ordered(x)`). Using `unique(frog)`, one can obtain the different categories (called levels, which default to the labels), which can be used as `factor(froc, levels=unique(frog))` (the `levels` argument can't have repeating entries).

For another example, suppose for some reason you wanted to deal with Roman numerals. Here's a factor in action:

```
> numerals <- c("I", "II", "V", "IV", "II", "I", "I")
> factored.numerals <- factor(numerals, levels=c("I", "II", "III", "IV", "V"), ordered=T)
> factored.numerals
[1] I II V IV II I I
Levels: I < II < III < IV < V
> factored.numerals[1] > factored.numerals[2]
[1] FALSE
> factored.numerals[1] > "III"
[1] FALSE
```

But one could use $1, \ldots, 5$ as the labels, making them look a lot prettier. All the relevant information is still saved, but it's displayed as numbers.

One can use `attributes()` to list the attributes of a factor (or indeed of lots and lots of other things).

Factors don't appear in the wild all that often, but are useful within contexts such as dataframes. For example, there's some test data called `mtcars` in R which has a lot of information about different cars (e.g. fuel efficiency, horsepower, rear axle ratio, and so on). It looks like a matrix, but has `class()` `"data.frame"` and is more like a list than a matrix. When displaying data frames, they might be huge, so using `head()` or `head(mtchars, 10)`, one can display the first six (or however many, as in the second example) rows of data. This also works for other types, too, so we can more easily display large lists, etc. Of course, there's also a `tail()` function; functions in R in general come in little families, which is useful for remembering them.

But as we were saying, data frames look like lists, e.g. one can call things like `mtcars[['mpg']]`, `mtcars$mpg`, or `mtcars[[1]]`, which all do the same thing. In some sense, this is a list of vectors, which may have different types. Using `mtcars[1:5,]`, one can obtain the first five rows as a sub-dataframe (though be careful: `mtcars[1:5]` returns the first five columns). Stuff like `mtcars[1:5, 1:2]` is still possible, as is `mtcars[1:10, c("mpg", "qsec")]`. We can even query by Booleans: `mtcars[mtcars$am == 1]` filters by that condition. Then, the data can be ordered by speed, e.g. `mtcars[order(mtcars$qsec)]`, and so on. This is a powerful way to answer questions about the data, so with indexing magic, one can quickly answer questions such as the average `qsec` time for 4-cylinder cars that get less than 22 MPG, via `mean(mtcars[mtcars$cyl == 4 && mtcars$mpg < 22]$qsec)`. The next interesting thing to do with this is to analyze correlation: one can add a new column to the data frame as `mtchars$speed.disp = mtcars$disp / mtchars$speed`. This makes the correlation a bit more obvious.

This is great, I hear you saying, but how should one create a data frame? The key is the `data.frame` method, as below:

```
frogs <- data.frame(height=c(3,4,2.5), jump=c(10, 2, 4), color=factor(c("g","g","b")), levels=c("g",
    "b"), labels=c("Green", "Blue"))
```

Now we can answer questions like `mean(frogs$height)`.

Often, data frams exist within libraries:

```
> install.packages("spuRs")
# (output trimmed)
> library(spuRs)
# (output trimmed)
# Now, we have a data frame called ufc with data on Upper Flat
# Creek tree data, e.g. species and height.
```

There are also functions called `read.table()` and `write.table()`, which read and write data frames to a file (with lots and lots of options, as with all I/O in R). Finally, recall that we have a "contains" operator, e.g. `1 %in% 1:5` returns `TRUE`.

Moving to a different context, one could talk about state and changing state. One common problem is that one might expect a variable to have one value, but then it's changed. Using the fact that functions are first-class objects, a lot of these issues can be avoided. For example, here's a copy of the `prod()` function that is written functionally:

```
Prod <- function(vec) {
    if (length(vec) == 0) { 1
    } else {
    Prod(vec[-1]) * vec[1]
    }
}
```

There's a similar thing going on for the product of a list:

```
Prod.List <- function(lst) {
    if (length(lst) == 0) return(1)
    Prod.List(lst[-1]) * lst[[1]] # I want the first element itself
}
```

Here's another example of recursive thinking, this time to check whether a vector is sorted.

```
Is.Sorted <- function(vec) {
    if (length(vec) < 2) return(TRUE)
    vec[1] < vec[2] && Is.Sorted(vec[-1])
}
```

For the next example, it's useful to have mapping functions, which exist in the context of a list of vectors (and similar constructions). Specifically, the function `sapply(lst, f)` applies the function `f` to every element of `lst`, and then returns a vector of the results. This is the "little brother" of `lapply`, which returns a list instead of a vector. For example, `sapply(lst, function(vec) vec[2])` returns a vector of the 2nd elements of each list. This illustrates the use of anonymous functions to quickly transform things without having to give them names. Here's another example:

```
> sum(sapply(1:10, function(x) {
+     y <- x ^2 + 3 * x
+     z <- x/3
+ }))
[1] 18.33333
```

A little more interestingly, here's a function to generate the power set $\mathbb{P}(\mathtt{V}) = 2^{\mathtt{V}}$ of a given vector V. For example, $\mathbb{P}(\{1,2,3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$.

```
Powerset <- function(set) { # this is a vector
    # Base case: return empty set, represented as a length-0 numeric vector
    if (length(set) == 0) return numeric(0)
    # General case: pick an element x, and include all subsets of the set minus
    # x along with all of those subsets joined with x
    Power.subset = Powerset(set[-1])
    c(sapply(Power.subset, function(subset) { c(s[1], subset)}), Power.subset)
}
```

Changing `numeric(0)` to `list(numeric(0))` and `sapply` to `lapply` allows one to return a list instead of a vectors.

Notice that `Power.subset` is stored, so that it doesn't have to be computed twice. In a purely functional language, values are immutable and this assignment could be optimized out, then some care does have to be taken to avoid recomputation. This takes it from $O(2^N)$ to $O(N)$, which is *much* nicer.

Here's a mergesort function:

```
# Functions are first-class, so we could have just written this inside Mergesort,
# but this means the interpreter won't have to redigest it every time (in
# particular, all of the recursive calls).

# Merges two sorted vectors
Merge <- function(v1, v2) {
    if (length(v1) == 0) return(v2)
    if (v1[1] < v2[1]) return (c(v1[1], Merge(v1[-1], v2)))
    return (c(v2[1], Merge(v1, v2[-1])))
}

# Something like this. Can you spot the bug?
Mergesort <- function(vec) {
    if (length(vec) <= 1) return(vec)
    l <- length(vec)
    return(Merge(Mergesort(vec[1:l %/% 2]), Mergesort(vec[l %/% 2:l]))) # We want intiger division
        for indices
}
```

4. Functional Thinking: 4/10/14

"We all said, 'Stupid TI86es! Nobody knows how to use them!' It's just like opening Vim for the first time."

Today, we'll have some more exercises about programming with R's functional programming.

Exercise 4.1. Write an `IsSubsequence` function which accepts two vectors and returns whether the former is a subsequence of the latter.

Solution.

```
# Return true if needle is contained in haystack
# > IsSubsequence(c(1, 3, 5), 1:5)
# [1] TRUE
# > IsSubsequence(c(1, 5, 3), 1:5)
# [1] FALSE
# > IsSubsequence(numeric(), 1:5)
# [1] TRUE
# > IsSubsequence(1, numeric())
# [1] FALSE
# > IsSubsequence(numeric(), numeric())
# [1] TRUE

IsSubsequence <- function(needle, haystack) {
    if (length(needle) == 0) return(TRUE)
    if (length(haystack) == 0) return(FALSE)
    # Either the first elements are the same, or the needle is a subsequence of haystack[-1]
    (needle[1] == haystack[1] && IsSubsequence(needle[-1], haystack[-1])) || IsSubsequence(needle,
        haystack[1])
}
```

Exercise 4.2. Write a `QuickSort` function to sort a vector.

Solution.

```
Partition <- function(pivot, vec) {
# Returns a list of two elements, the first all elements before the pivot,
# and the second all elements after the pivot.
    if (length(vec) == 0) return(list(numeric(), numeric()))
    part <- Partiion(pivot, vec[-1])
    if (vec[1] < pivot) return(list(c(part[[1]], vec[1]), part[[2]]))
    list(part[[1]], c(part[[2]], vec[1]))
}

QuickSort <- function(vec) {
    if (length(vec) <= 1) return(vec)
    part <- Partition(vec[1], vec[-1]) # In a better Quicksort, the pivot would be randomized.
    c(QuickSort(part[[1]]), vec[1], QuickSort(part[[2]]))
}
```

One important side point is the notion of lambda calclus and lambda functions. Lambda calculus is a formalization of computation somewhat useful in math and extremely useful in theoretical CS. This is implemented in `R` as the notion of lambda functions, so we can do things like this:

```
> (function (x) 2 + x)(3)
[1] 5
# Functions can return functions!
> (function(x) function(y) x + y)(3)(4)
[1] 7
# Functions can take functions as arguments, and return functions!
# Then, this is called on the squaring function, which gives a function, which
# we call on a number
> (function(f) function(x) f(f(f(x))))(function(y) y^2)(2)
[1] 256
```

This means we can support unevaluated forms of expressions: when one asks `R` for `a^4`, it checks if there's a variable with value `a`, and then plugs it in and evaluates it. One can instead store that as an expression, as `expression(a^4)`, and evaluate it with the `eval()` function (e.g. `eval(expression(a^4))`). A related function is `quote`, which returns its argument unevaluated, so `quote(a^4)` returns `a^4` (which has mode `"call"`, while the mode of an expression is `"expression"` — the differences are beyond the scope of the class), and one can call `eval(quote(a^4))`.

The important point of these functions is that they allow `R` to do symbolic mathematics. For example, there is a function `D` which does symbolic differentiation. Thus, `D(`*expression*`,"x")` symbolically differentiates the expression with respect to `x` (or whatever variable you care to specify). However, beware the following:

```
> D(exp(a)+tanh(a), "a")
[1] 0
```

This is eagerly evaluated, so first, `a` is plugged in, so `D` computes the derivative of a constant. Oops. The proper way to do this is `D(quote(exp(a) + tanh(a)),"a")`. This can be used to compute second or partial derivatives (within expressions, variables aren't evaluated until one asks for them to be), or even Hessians!

Here's an example of this.

```
# Apply the derivative of the expression at the given vector of points
# > TestDeriv(1:10, quote(a^2))
# [1] 2 4 6 8 10 12 14 16 18 20

TestDeriv <- function(points, expr) {
    # can pass a list into eval to simulate a new environment, so we make a
    # new scope in which a = p.
    sapply(points, function(p) eval(D(expr, "a", list(a=p))))
}
```

Then, using this, one can look for minima by looking for zeros, such as by calling

```
sum(sapply(TestDeriv(seq(from=-pi, to=pi by=0.001), quote(x^4 -1)), function(val) {
    abs(val) < 0.0001
}))
```

This incantation provides a way for approximating the number of critical points of a function (the tolerance might need to be better, but that's not the point). One can quickly ask and answer very interesting questions in one line using `sapply` or `lapply` and a bit of thought.

Note that `sapply` and `lapply` can accept more general functions than those we've used so far. For example,

```
> sapply(c("one", "two", "three"), function(str) 4)
  one   two three
    4     4     4
```

So this is a named vector. But it's still of class `"numeric"` (as well as `mode`), so vectors can have named entries! These are accessed as a vector by `names(`*vect*`)`.

Now, we get interesting subtleties: if `vec` is a named vector, then `vec[1]` returns a named subvector, and `[[1]]` returns an unnamed subvector (single element in this case). However, in the specific case we had above, one can get an unnmaed vector by calling

```
> sapply(c("one", "two", "three"), function(str) 4, USE.NAMES=F)
[1] 4 4 4
```

There are various other functions in the `apply` family. For example, `replicate(`*n*`,`*f*`)` is equivalent to `sapply(1:`*n*`,`*f*`)`, and slightly faster. There's also the patriarch function `apply`, which is even able to map over matrices! Note that `sapply` called on a matrix returns a vector without dimension.

`apply(mat, MARGIN, f)` applies the function `f` to the columns of the matrix specified by `MARGIN` (i.e. 1 for rows, 2 for columns, and so on; see the documentation).

Now, we can use `R` to answer questions from the 109 homework! The useful functions are `factorial(n)`, `choose(m, n)` (which returns $\binom{m}{n}$), and `combn(vec, n)`, which returns a matrix holding all combinations of `n` elements from `vec`. To get a list instead, use `combn(vec, n, simplify=FALSE)`. But `combn()` also accepts a function parameter, which is applied to all combinations in the vector. For example, to compute the number of combinations that are even, one can call `sum(combn(1:5, 3, function(vec) sum(vec) %% 2 == 0))`. This is useful for problems that seem to require clever counting, e.g. the problem on the first 109 pset asking how many possible committees can be made out of seven people, four men and three women, two each of whom won't work with each other, by the string `sum(combn(1:7, 4, functioN(vec) !((1 %in% vec) && (2 %in% vec))))`.

Similar to `combn()` is `permn`, though one first is required to call `install.packages("combinat")` and then `library(combinat)` to access it for some reason. This doesn't simplify to a matrix by default.

There are a couple more useful functions, called `Map`, `Filter`, and `Reduce`. These are bread (but maybe not the butter) of functional programming. These are very similar to things we've seen before, and are common higher-order constructs in functional languages.

- `Filter()` takes as its first argument a function that returns a boolean, and then a list, and filters out all the things that the function returns false on. For example, `Filter(function(x) x %% 2 == 0, 1:10)` returns `2 4 6 8 10`.
- `Map()` works like `apply()`, i.e. `Map(function(x) 2 * x, 1:10)` returns a list of the function applied to every element of the vector.
- `Reduce()` accepts a binary function, an initial value, and a list. Then, it successively combines the elements of the list (or vector). For example, `Reduce(function(x,y) x + y, 1:10, 0)` sums up the numbers between 1 and 10.

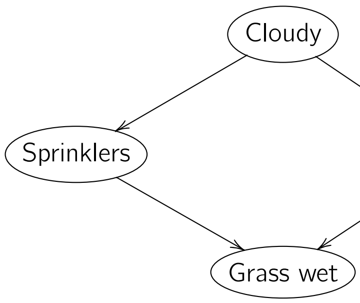
MapReduce is basically a fancy version of applications of these functions, though in a more generalized setting.

5. Spam Filtering, Causal Variables, and Machine Learning: 4/15/14

"Today we're going to talk about spam, which feels really 1990s... when was the last time spam negatively impacted your life?"

But spam still accounts for 90% of emails sent, so it's still worthwhile to understnsd the statistics behind spam filtering. Today's reference is a handout at `http://www.stanford.edu/class/cs109l/unrestricted/handouts/02_spam.pdf`.

A causal network is a collection of variables which depend on each other in a manner represented by a directed graph, e.g.



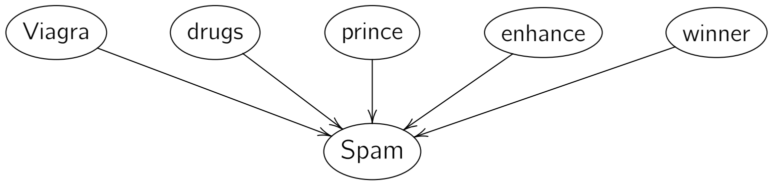
Now, we can ask (and answer) questions such as the value of $P(\text{Cloudy} \mid \text{Grass wet})$. This evaluates to a bunch of conditional probabilities. The general scheme is known as a Markov blanket. To simplify nottio, let R be the event that it has rained, C be that it's cloudy, G that the grass is wet, and S be the that the sprinklers were run. Then, we are asking things like $P(G \mid S, R)$.

The "chain rule" of conditional probability, resolving a joint probability into conditional ones via

$$P(A, B, C) = P(A \mid B, C)P(B \mid C)P(C),$$

is very useful in causal networks, because it resolves joint probabilities into things that might be conditionally independent. It also works along with Bayes' Rule: $P(C \mid G) = P(G \mid C)P(C)/P(G)$.

One can use this to make judgements about whether an email is spam: there's a causal network that looks like this:



Using R, we can simulate this and build a spamfilter. For the data, call `install.packages("kernlab")`. For example, here's a function that accepts the `spam` data frame from the `kernlab` package and factors the first 48 columns into binary random variables into 'yes' and 'no' corresponding to whether a given word does or doesn't exist.

```
RstructureSpamDataset <- function(spam) {
    factored.cols <- lapply(spam[,1:48], function(col) {
        factor(col == 0, levels=c(T< F), labels=c("no", "yes"))
    })
    cbind(data.frame(factored.cols), type=spam$type)
}
```

There's a little nuance in that `sapply` doesn't strip names, even though it returns an atomic, numeric vector.

The goal is to decide whether something is spam or not. Let S be the event that an email is spam, and $\mathbf{X}$ be the joint set of all features $X_1, \ldots, X_n$, Which of $P(\mathbf{X}, S)$ and $P(\mathbf{X}, !S)$ is more likely (i.e. which one better explains the hypothesis)? Well, $P(\mathbf{X}, S) = P(\mathbf{X} \mid S)P(S)$. Now, it;s standard to make something known as the *naïve Bayes assumption*: that for any i and j, $P(X_i, X_j \mid S) = P(X_i \mid S)P(X_j \mid S)$, or in words, that the words in an email are conditionally independent given that an email is (or isn't spam). Then, the computation just involves counting up how often a word appears as spam or non-spam.

The next step is to obtain training and testing data, e.g. as follows:

```
ChooseTrainingAndTesting <- function(dataset, k) {
    ## Chooses a random subset of the dataset to use for training, leaving
    ## the rest for testing.
    ##
    ## Args:
    ##   dataset - a data frame to split into training and testing data frames
    ##   k - the number of rows to include in the training data set
    ## Returns:
    ##   A list with elements $training and $testing, each a data frame with
    ##   a disjoint subset of rows taken randomly from 'dataset'.

    # The sample() function is all sorts of useful here.
    idx <- sample(1:nrow(dataset), k)
    list(training=dataset[idx,], testing[-idx,])
}
```

Next, we want to compute the conditional probabilities $P(X_i \mid !S)$.

```
venNonSpam <- function(training) {
    ## Computes conditional probabilities of the form P(Xi | type = "nonspam").
    ##
    ## Args:
    ##   training - data frame with training data.
    ## Returns:
    ##   A matrix with 2 rows and 48 columns, one per observed variable
    ##   from the training data frame. The first row gives the conditional
    ##   probabilities P(Xi = "no" | type = "nonspam") for each of the 48
    ##   observed variables. The second row gives conditional probabilities
    ##   P(Xi = "yes" | type = "nonspam"). Note that all columns should sum
    ##   to one.
    ## Details:
    ##   This function uses Laplace smoothing for conditional probabilities.
    ##   You can implement Laplace smoothing by imagining, for each of
    ##   the 48 columns, that we see two additional datapoints ("no" and "yes").

    ## Choose the subset of the data frame which contains only "nonspam"
    ## rows, and keep only the first 48 columns.
    subset <- training[training$type == "nonspam", 1:48]

    ## Compute, for each of the 48 columns in the subset above, the probability
    ## of the value being "no". Remember to use Laplace smoothing
    ## (would just be mean(col == "no") otherwise)

    # Good shorthand for filtering part of a list and then counting it.
    pNo <- sapply(subset, function(col) mean(c(col == "no", TRUE, FALSE)))

    return(rbind(no = pNo, yes = 1 - pNo))
}
```

Then, there would be a very similar `GivenSpam` function which dredges up the analogous training data for spam.

Finally, there must be a prediction function. One tricky aspect is that, since this multiplies together a lot of small numbers, the floating-point values might underflow and become zero. Thus, it makes much more sense to add their logs, avoiding this issue.

```
Predict <- function(features, p.nonspam, p.spam, nonspam.probs, spam.probs) {
    ## Predicts whether the feature vector (with 48 observed variables) represents "spam"
    ##  or "nonspam" by choosing argmax P(row | type) * P(type).
    ##
    ## Args:
    ##   features - vector with 48 values (one per observed variable)
    ##   p.nonspam - P(type = "nonspam")
    ##   p.spam - P(type = "spam")
    ##   nonspam.probs - conditional probabilities of the form P(Xi | type = "nonspam")
    ##   spam.probs - conditional probabilities of the form P(Xi | type = "spam")
    ## Returns:
    ##   "spam" or "nonspam"
    p.nonspam <- log(p.nonspam)
    p.spam <- log(p.spam)
    nonspam.probs <- log(nonspam.probs)

    # We can abridge this, because for x, y positive, x > y iff log x > log y.
    nonspam <- p.nonspam + sum((features == "yes") * nonspam.probs[2,]) + sum((features == "no") *
        nonspam.probs[1,])
    spam <- p.nonspam + sum((features == "yes") * spam.probs[2,]) + sum((features == "no") * spam.
        probs[1,])
    if (nonspam > spam) return("nonspam")
    else return("spam")
}
```

Now, the testing accuracy is in `result$performance` (where `result` is given by the whole filter); the example provided in class had about 86% accuracy.

6. Markov Models: 4/22/14

"Does anyne know, in the show Pinky and the Brain, *was Pinky male, or was it left deliberately ambiguous?"*

Today's question is: given a sequence drawn from $\{-1, 1\}$, how can we predict the next element? Basically, we have random varibles $X_1, \ldots, X_n$; then, what is $P(X_{n+1} = 1 \mid X_1, \ldots, X_n)$? The X_i are known as prior information, and can be summarized in the vector $\mathbf{X}$ (i.e. we want to know $P(X_{n+1} \mid \mathbf{X})$).

This class, we'll make something known as the Markov assumption, which is huge in many different fields (e.g. signal processing): that $P(X_{t+1} \mid \mathbf{X}) = P(X_{t+1} \mid X_t)$. In other words, we're going to assume that $P(X_{t+1} = 1 \mid X_t = 1) = \theta$ and $P(X_{t+1} = -1 \mid X_t = -1) = \theta$. θ is the probability that the state doesn't change; correspondingly, the probability that the state switches (no matter the value of X_{t-1}) is $1 - \theta$. Series of random variables that work like this are called Markov chains, and they work rather well for flight dynamics or orbital dynamics in space (e.g. position and velocity determine next values of position and velocity with healthy accuracy).

First, in order to reason about this, we'll need a way to estimate integrals. The idea behind the Midpoint rule is that the area under a curve (the integral) on the interval $[a, b]$ can be approximated by modeling the region as k rectangles, with width Δx, so the area is

$$A \approx \sum_{i=1}^{k} f\left(a + k\Delta x + \frac{\Delta x}{2}\right) \Delta x.$$

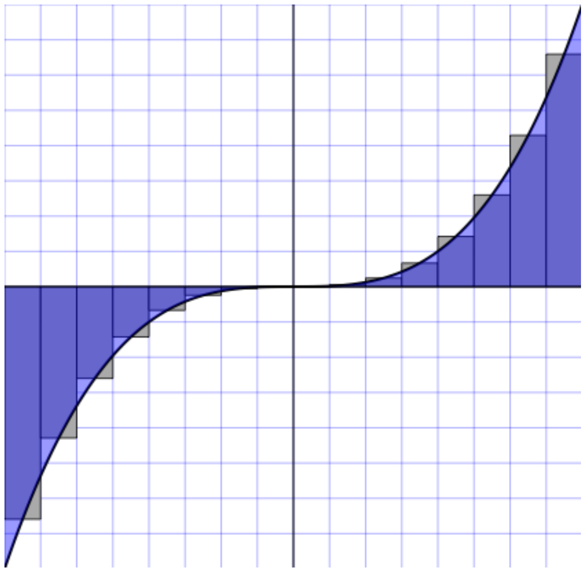


Figure 1. Example of the Midpoint rule: dividing an area into rectangles. Source: `http://en.wikipedia.org/wiki/Rectangle_method`.

```
MidpointRule <- function(f, min, max, k, ...) {
    ## Numerically integrates a function using the midpoint rule.
    ## Args:
    ##   f - the function to integrate
    ##   min - low bound for integration
    ##   max - high bound for integration
    ##   k - the number of points to evaulate the function at
    ##   ... - any additional arguments to the function being integrated
    dx <- (max - min) / k
    xs <- seq(from=(min + dx / 2),to=(max-dx/2),by=dx)
    sum(sapply(xs, function(c) f(c, ...) * dx))
}
```

Now, this function applied to $f(x) = x^2$ on $[0, 1]$ returns 0.333325, which is pretty good (the actual value is 1/3).

Wait, why did we even need to deal with integrals? We need to know what the value of θ is. Sometimes, to understand $P(E)$, we write it as $P(E) = P(EF) + P(E\neg F)$. If F is a discrete random variable on a set S, rather than a distribution, then instead this becomes

$$P(E) = \sum_{i \in S} P(E, F = i).$$

When F is a continuous random variable on a set S, this sum becomes an integral

$$P(E) = \int_S P(E, F = i)\,\mathrm{d}i.$$

Thus, returning to the Markov model, we can integrate against all possible values of θ. Suppose that we start with $P(X_1 = 1) = P(X_1 = -1) = 1/2$, so that we are starting somewhere.

$$P(X_{n+1} = 1 \mid \mathbf{X}) = \int_0^1 P(X_{n+1} = 1, \theta = t \mid \mathbf{X})\,\mathrm{d}t.$$

This allows us to reason about the value of θ (the probability of not moving) based on the Chain rule:

$$\begin{aligned} &= \int_0^1 P(X_{n+1} = 1 \mid \theta = t, \mathbf{X}) P(\theta = t \mid X)\,\mathrm{d}t \\ &= \int_0^1 P(X_{n+1} = 1 \mid \theta = t, X_n) \frac{P(\mathbf{X} \mid \theta = t) P(\theta = t)}{P(\mathbf{X})}\,\mathrm{d}t. \end{aligned}$$

Notice how the Markov assumption makes this something we can actually begin to compute. Now, we take a prior distribution for θ that suggests it's likely to be near 1/2, 0, and 1. Specifically, let

$$f(\theta) = \theta^{-1/2}(1-\theta)^{-1/2}\left(0.2 + \exp\left(-100\left(\theta - \frac{1}{2}\right)^2\right)\right).$$

This can be straightforwardly be implemented in R:

```
UPrior <- function(theta) {
    ## The unnormalized prior density function for the +1/-1 model. The valid
    ## range for theta is (0, 1) with the endpoints not included.
    ## Returns:
    ##  f(theta)
    theta^(-1/2) * (1 - theta)^(-1/2) * (0.2 + exp(-100 * (theta - 0.5)^2))
}

PriorNorm <- function(k) {
    ## Computes the normalizing constant for the prior density function.
    ## Args:
    ##   k - number of points to use for integration
    MidpointRule(UPrior, 0, 1, k)
}

Likelihood <- function(theta, x) {
    ## The likelihood function. This is the probability of seeing x given theta.
    ## P(X | theta) = P(X_1)P(X_2 | X_2)P(X_3 | X_2) ...
    ## Args:
    ##   theta - distribution parameter
    ##   x - the data points (+1/-1 values)

    ChangeCount <- function(x) {
        ## Find the count of changes in the data sequence. Returns the number of
        ## times the value changes from one time to the next.
        sum(x[-1] != x[-length(x)])
    }
    n <- length(x)
    c <- ChangeCount(x)
    .5 * (1-theta) ^ c * theta ^ (n - c - 1)
}

UPost <- function(theta, x) {
    ## The unnormalized posterior density.
    ## Args:
    ##   theta - prior distribution parameter
    ##   x - the data points (+1/-1 values)

    ## P(X, theta) = P(X | theta) * P(theta)
    likelihood(theta, x) * UPrior(theta)
}
```

Now, we can write the finlal prediction function, comming from the integral we computed above.

```
Predict <- function(x, k) {
    ## Predicts the next data point. Passed the previous data points and the
    ## number of points to use for integrations using the midpoint rule.
    ## Integrates the prediction for the next data point given theta with respect
    ## to the unnormalized posterior distribution of theta. Since the posterior
    ## is unnormalized, this results in unnormalized probabilities of the next
    ## point being +1 and -1, which are normalized with respect to the total
    ## unnormalized probability.
    ##
    ## Args:
    ##   x - the data points (+1/-1 values)
    ##   k - the number of points to use for integration
    ## Returns:
    ##   A list with fields $prediction and $probability. The $prediction field
    ##   is the value (+1/-1) predicted for the next data point (whichever has
    ##   higher probability of being the next data point). The $probability field
    ##   is the probability given to that prediction.
    u.prob.same <- MidpointRule(function(theta, x) theta * UPost(theta, x), 0, 1, k, x)
    u.prob.diff <- MidpointRule(function(theta, x) (1-theta) * UPost(theta, x), 0, 1, k, x)
    if (u.prob.same > u.prob.diff)
        return (list(prediction=x[length(x)], probability=u.prob.same / c))
    return (list(prediction=x[length(x)], probability=u.prob.diff / c))
}
```

7. Plotting: 4/29/14

"R has a man's version of color: there's blue and green, but no salmon."

One cool language feature in R is the `ifelse)` operator, which functions as a ternary operator.

The basic plotting function in R is called `plot()`. This is similar to OpenGL in that it's a state machine, which is somewhat non-functional. One can plot one list against another as `plot(vec2, vec1)`, the results of which are in Figure 2. As with most plotting paradigms, there are lots of extra arguments to specify arguments, e.g.

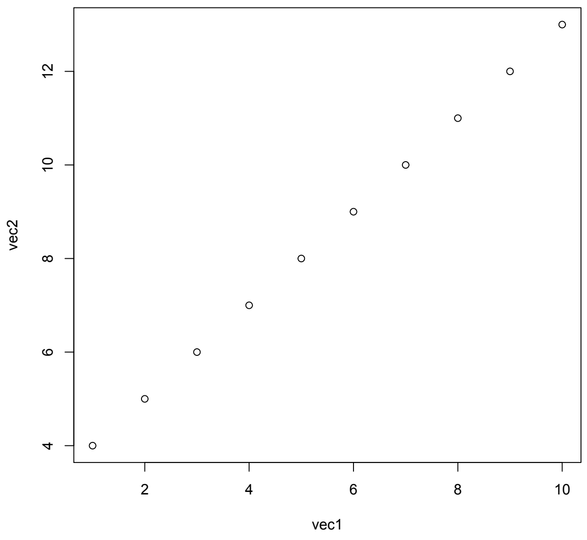


Figure 2. The result of a call to plot two vectors against each other.

`plot(vec1, vec2, col="teal",pch="-")` (where `pch` is the plot marker), etc. `lty` means the line type (e.g. `"dotted"` or `"dashed"`), and `type`, which indicates histogram, scatter, line (`"l"`), `main="Title goes here"`, `sub=}Subtitle"`, and so on.

One can plot curves by calling `plot` and `seq` in unison with each other, but it's also possible to call `curve(fn, from=a, to=b)` to plot a function on an interval. But using `add=TRUE`, one can plot two different curves on the same plot.

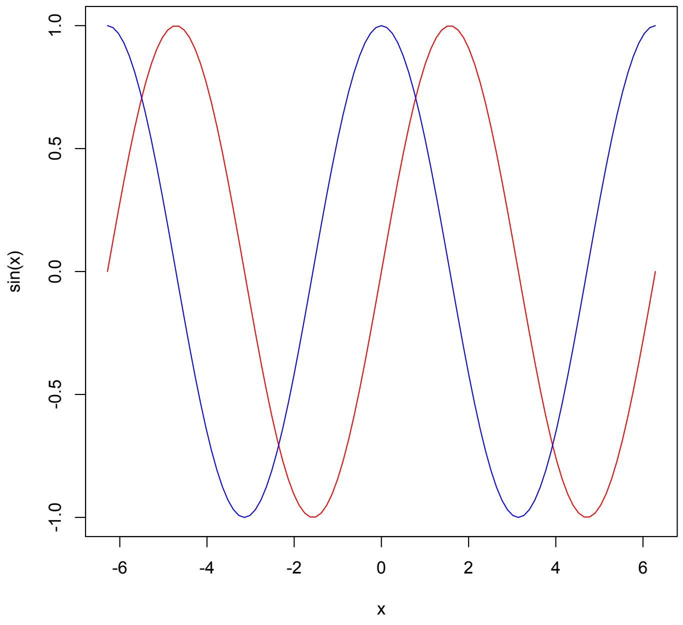


Figure 3. Plotting $f(x) = \sin x$ and $g(x) = \cos x$ on $[-2\pi, 2\pi]$.

Then, `abline` allows one to draw lines: parameter `h` means a horisontal line at that value, and `v` a vertical line (it's vectorized, so providing both draws both lines, and providing vectors draws all of the asked-for lines). The `text(x, y, str)` function draws a string at a given location. There are lots and lots of different functions for plotting things, e.g. `title` just sets the title for a plot, `hist` to make a histogram of a single vector, etc. `jitter()` takes data and adds some small randomness, and `rug()` builds a one-dimensional rug plot of some given data.

The `runif(n, min=a, max=b)` function samples a list of length `n` from a uniform distribution on the specified interval. This can be used to generate other distributions.

```
RBinom <- function(n, num.trials, p) {
    ## Returns n samples from Bin(num.trials, p)
    sapply(1:n, function(x) sum(runif(num.trials, min=0, max=1) < p))
}
```

Of course, it turns out that `rbinom(n, size, prob)` is a builtin, as is the case for a few other distributions. But each distribution for which `r___` is defined has a family of others. For example, `dbinom(x, size, prob)`, which is the PMF/PDF of the distribution, and `pbinom(q, size, prob)` is the cumulative density function. `qbinom` is the quantile: it returns the value of `q` that you need to pass into `pbinom` to get a given probability. For example, how far over do you need to go such that the CDF is 0.9? This makes a lot more sense with continuous distributions. These families exist for `unif` and `binom`, as seen, but also `pois` (Poisson), `exp` (exponential), and `norm` (normal), e.g. `qnorm`, `rpois`, etc. Then, `qnorm` is like a reverse lookup in the standard normal table.

It's pretty convenient that these functions exist, because it's not always obvious how to write these functions (it'll be done on the last day of CS 109). But one great use is Monte Carlo simulation. Here's an example.

```
# uniform * binomial with n = 100, p is the uniform sample.
GenSamples <- function(n) {
    sapply(1:n, function() {
        p = runif(1)
        n.samp <- rbinom(1,100,p)
        p * b.samp
    })
}
```

This makes it easy to estimate the expected value of such a distribution, which isn't difficult, but requires thinking. Certainly, this makes it much easier to compute the variance.

In a nice lingusitic coincidence, one can use Monte Carlo simulation to solve the Monty Hall problem,[1] where one has a 1/3 chance of picking a car behind a door (as opposed to a goat), but then the host opens one door without revealing

[1]Relevant xkcd: `https://xkcd.com/1282/`. For reference, on the original game show, contestants were allowed to keep the goat.

yours and asks if you want to switch. This is a common example of nonintuitive probability; the better strategy is always to switch (which can be seen if one supposes there are millions of doors, rather than 3; the point is that switching is assuming that your original choice is wrong, which is true with probability 2/3). But don't trust us!

```
MontyHall <- function(s) {
    prize.door <- sample(1:3, 1)
    door.chosen <- sample(1:3, 1)
    monty.options <- (1:3)[-c(prize.door, door.chosen)] # always at least one option
    if (length(monty.options) == 1) <- opened.door <- monty.options
    else opened.door <- sample(monty.options, 1)
    if (s) final.choice <- (1:3)[-c(opened,door, door.chosen)]
    else final.choice <- door.chosen

    final.choice == prize.door
}
# and now...
> mean(sapply(1:100, function(x) MontyHall(T)))
[1] 0.73
> mean(sapply(1:100, function(x) MontyHall(F)))
[1] 0.37
```

I hope this is compelling evidence. This is also useful for other counterinutitive problems, e.g. the birthday paradox.

8. The Exponential, Poisson, and Beta Distributions: 5/6/14

"I haven't actually used a two-stroke lawnmower; I've just seen them as gifs on the Internet."

Recall that the beta distribution is a distribution with a prior, e.g. if one has an unfair coin, how would one estimate the probability of heads from the record of past flips?[2] In the same way R had support for the the other distributions we saw, it supports the beta distribution as `dpeta`, `pbeta`, `qbeta`, and `rbeta`. These accept the priors as arguments, e.g. `dbeta(x, 1, 1)` assumes one head and one tail has been seen. This assumes Laplace smoothing, so `beta(x, 1, 11)` assumes that no heads, but 10 tails, have been seen. This does what you'd expect from statistics, so it gets sharper as

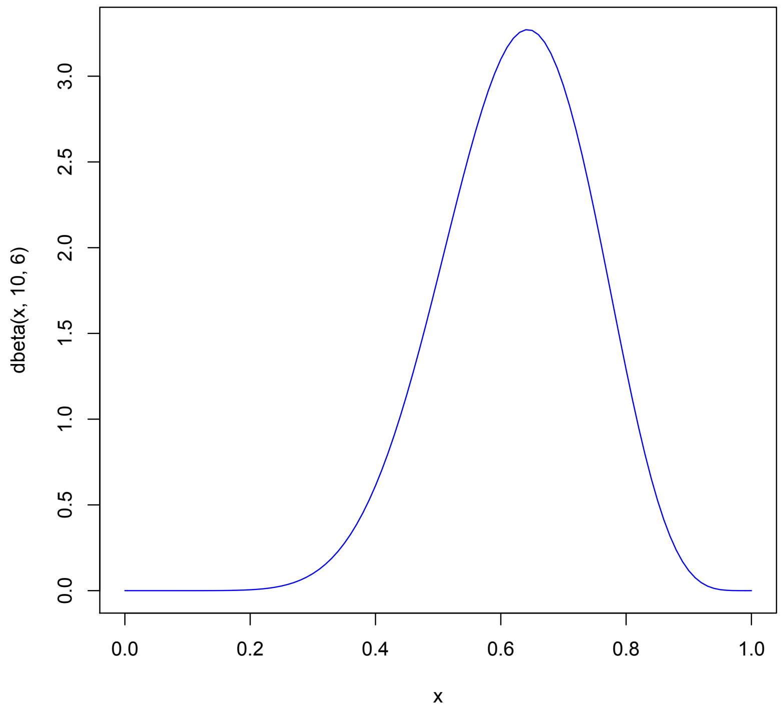


Figure 4. The beta distribution with priors $(10, 6)$.

the priors get larger, as the probability can be better estimated.

[2]Apparently it's extremely difficult to produce a coin that's actually weighted so as to be unfair, I guess unless you're Persei Diaconis.

```
# Anomate the Beta distribution changing as the priors get better

# There's no good functional way to write this.
AnimateBeta <- function(a=1, b=1, p=runif(1), frames=200, sleep=.1) {
    beta.params <- c(a, b)
    plot.beta <- function(a, b) curve(dbeta(x,a,b), from=0, to=1, col="blue")
    for (i in 1:frames) {
        plot.beta(beta.params[1], beta.params[2])
        flip <- sample(1:2, 1, prob=c(p, 1-p))
        # Unfortunately, R lacks a postfix addition operator
        beta.params[flip] <- beta.params[flip] + 1
        Sys.sleep(sleep)
    }
    # See how well the distribution calculates the actual parameter
    abline(v=p, col="red")
}
```

This animation shows the beta distribution converging as the number of priors is better known. The PDF looks like a Dirac delta – its total integral is 1, but its maximum value gets larger and larger as it gets more and more concentrated around a point.

Moving on to the exponential distribution, whose PDF is $f(x) = \lambda e^{-\lambda x}$, the same functions `dexp`, `pexp`, and so on, all still hold. This models how long it takes something to happen (e.g. how long between successive emails). Its PDF looks like Figure 5.

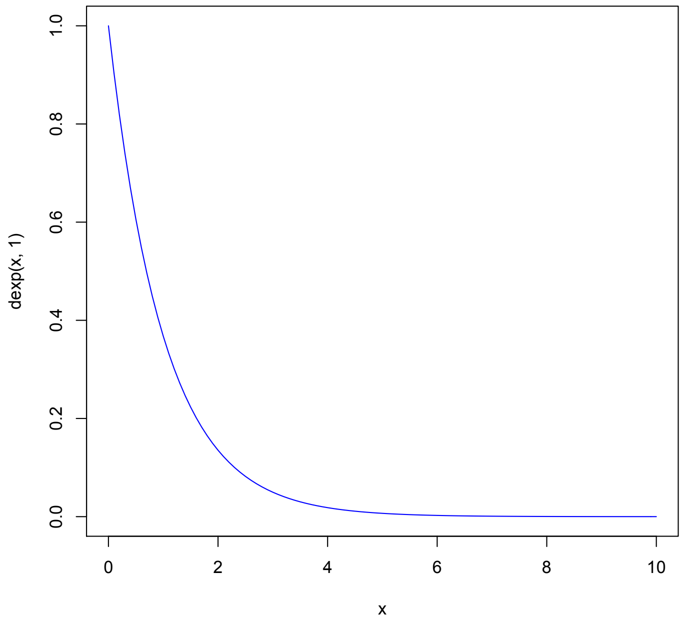


Figure 5. The exponential distribution with parameter $\lambda = 1$.

This is useful for, for example answering questions such as the following; if there are 30 people in line, with a average rate of 3 per minute, what's the probability that we get through the line in under 10 minutes? Each person is a sample from the exponential distribution. But calculating the sum of exponential classes involves taking convolutions, which is complicated, so just simulate it as `sum(rexp(30, rate=3))`. Thus, the full command is `mean(sapply(1:1000, function(i) sum(repx(30, rate=3)) < 10))`, approximating to about 0.52.

Another problem we can solve is that if a two-stroke lawnmower could fail in time $A \sim \text{Exp}(1/2)$ (where time is measured in years, so this happens on average once every two years), but the blades break in time $B \sim \text{Exp}(1/3)$, then what's the average time it takes for the first component to break? The command to execute is

```
mean(sapply(1:10000, function(i) min(rexp(1, 1/2), rexp(1, 1/3))))
```

and it evaluates to about 1.2. (It's also possible to make this more vectorized, if slightly less elegant, by avoiding the call to `min`) — at least in theory. I'm getting the wrong answer for some reason (calculating which one comes first gives 2.34). One can also calculate the variance with `var()`.

Definition. A Poisson process is a set of random variables $N(t)$ for $t \geq 0$, representing the number of things that have happened up to time t, and such that:

(1) At most one thing happens between time t and time $t+1$.
(2) The rate at which things happen, λ is constant (sometimes this definition is called a homogeneous Poisson process). Thus, $N(t+k) - N(t) \sim \text{Poi}(\lambda)$.
(3) $N(0) = 0$, corresponding to the true beginning of the process.
(4) The number of things that happen in disjoint intervals is independent.

This has a number of consequences, e.g. the distribution of events within a given interval is uniform!

Now, one can use R to simulate a Poisson process:

```
> N <- rpois(1, 100)
> times <- runif(N, 0, 1000)> times <- sort(times)
# step plot
> plot(times, 1:N, type="s")
> differences <- times[-1] - times[-N]
> plot(density(differences))
```

Then, the plots are Figure 6.

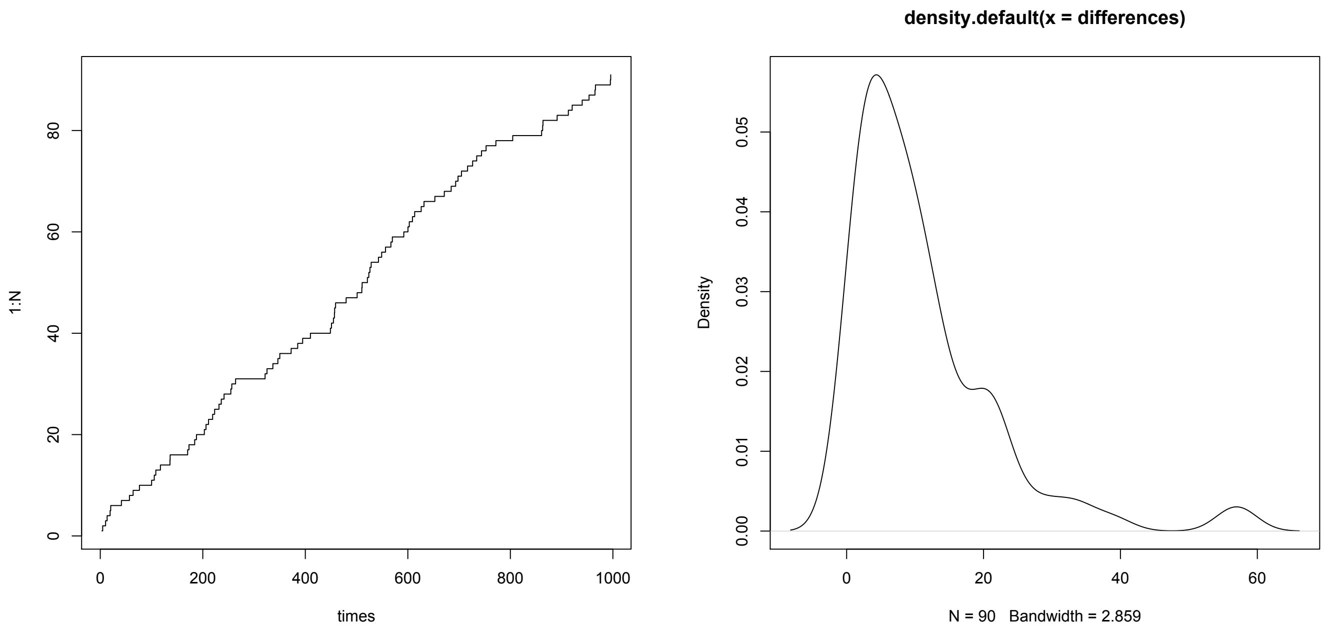


Figure 6. Left: the accumulation $N(t)$ of a Poisson process. Right: the density function of the differences in this process. This will approach the exponential distribution in the limit.

Let T_k be the arrival time of the k^{th} thing, so that $P(T_k > t) = P(N(t) < k)$, so $P(T_1 > t) = P(N(t) < 1) = P(N(t) = 0) = P(N(t) - N(0) = 0)$. Thus, the appxoximation with the exponential distribution is $e^{-\lambda t}(\lambda t)^0/0!$, or $e^{-\lambda t}$. The point is, the assumptions we placed on the Poisson process make these nice properties fall out.

9. Monte Carlo Integration and `ggplot2`: 5/13/14

"D-small could be my rapper name."

Consider a function $f(x)$ on the interval $[1,5]$, where the goal is to compute $\int_1^5 f(x)$. We'll also suppose that we have a good idea of the maximum value M of f (which isn't true in general, but is necessary for this method of estimation).

Choose random points in the box $[1,5] \times [0,M]$, and let X be the probability that a given point lies under the curve, so that $E[X] = P(X) = (1/4M)\int_1^5 f(x)$, so

$$\int_1^5 f(x) \approx 4M\overline{X},$$

where $\overline{X}$ is the sample mean. This is one form of Monte Carlo integration.

Suppose $Y \sim \text{Uni}(a, b)$, so that $E[Y] = (a+b)/2$, but

$$E[Y^2 + Y] = \int_a^b P(Y = y)(y^2 + y)\,\mathrm{d}y.$$

This is because, for general functions g, since Y is uniformly distributed, then

$$E[g(Y)] = \int_a^b P(Y = y)g(y)\,\mathrm{d}y = P(Y = y)\int_a^b g(y)\,\mathrm{d}y.$$

Thus, we can turn this around and use it to compute integrals:

$$\int_a^b g(y)\,\mathrm{d}y = (b-a)E[g(Y)].$$

Now, what does this look like in R?

```
> ys <- runif(1000, min=1, max=5)
> mean(ys)
[1] 3.054048
```

Thus, the value of the integral $\int_1^4 y\,\mathrm{d}y \approx 12$, which is in fact the exact value. Now, let's try that with $\int_1^5 \sin(x)\,\mathrm{d}x \approx 0.25$:

```
> ys <- runif(1000, min=1, max=5)
> mean(ys) * 4
[1] 0.1141142 # not correct... we need more samples
> ys <- runif(100000, min=1, max=5)
> mean(ys) * 4
[1] 0.251086
```

Thus, it's actually pretty accurate if the sample size is large enough.

The choice of Y as uniformly distributed is actually a bit arbitrary, and this can be used to compute semidefinite (that is, improper) integrals. To estimate things such as $\int_1^\infty (1/x^3)\,\mathrm{d}x$, the standard uniform distribution doesn't work. Now, let $Y \sim \text{Exp}(1/10)$; thus,

$$E\left[\frac{1}{Y^3}\right] = \int_1^\infty P(Y = y)\frac{\mathrm{d}y}{y^3}$$
$$\implies E\left[\frac{1}{Y^3 P(Y = y)}\right] = \int_1^\infty \frac{1}{y^3}\,\mathrm{d}y.$$

Now, in R:

```
# Convenience functions to only take x >= 1
rexpbdd <- function() {
    x <- rexp(1, .1)
    while (x < 1) x <- rexp(1, .1)
    x
}

dexpbdd <- function(x) {
    dexp(x, 0.1) / 1 - pexp(1, 0.1)
}
```

For example, `dexpbdd(10)` always is slightly larger than `dexp(10, 0.1)`. But back to the integration:

```
> ys <- sapply(1:10000, function(x) rexpbdd)
> mean(1 / (ys^3 * dexpbdd(ys)))
[1] 0.5039706
```

A little more head-scratching shows that the actual value of the integral is $1/2$, which is pretty impressive, especially becuse the Monte Carlo method is just about as simple even if the function to be integrated is absolutely ridiculous, e.g. $\int_1^\infty \log(x)\sin(x)/x^3\,\mathrm{d}x$.

Moving on to plotting with `qplot()` and `ggplot2`, install this with `install.packages("ggplot2")`. For example, here's a session with `qplot`.

```
> library(ggplot2)
> data(diamonds) # comes with ggplot2
# Notice that these fields aren't global, just within the diamonds data frame
> qplot(carat, price, data=diamonds, colour=color)
```

Notice the British spelling; this is common in ggplot2, but is a frequent source of mistakes for American users!

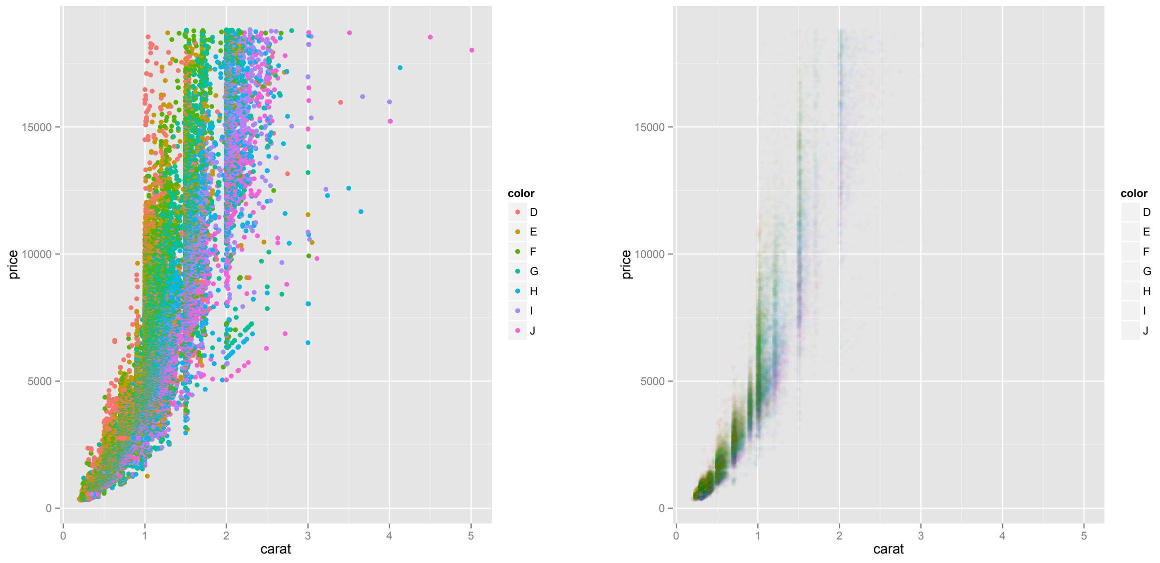


Figure 7. Left: result of plotting price and weight of diamonds against each other. Notice the spikes at 1 or 2 carats — people aren't willing to call a diamond 0.98 carats; they might as well round up just a little. Right: using the alpah channel to see the density of the data. Notice how significantly this differs from the first impression of the data.

These huge plots can be slimmed a bit by taking small samples from the data. Another solution is to use transparency, by setting the apha channel, e.g. `qplot(carat, price, data=diamonds, alpha=I(1/100))`, so 100 points are needed to make a pixel black.

Another paramter to `qplot` is `geom=c("point","smooth")`, which draws a line of best fit and a confidence interval. There are lots of other parameters.

It's possible to make bar-like plots aganst discrete data, though using `jitter` makes it easier to see things. See Figure 8 for an example, with `qplot(color, price / carat, data=diamonds, colour=color, geom="jitter", alpha=I(1/10))}`.

10. The Central Limit Theorem: 5/20/14

"Does it seem like I know what I'm doing sometimes?"

let's look at how the Central Limit theorem applies when we take some samples of samples.

```
> library(ggplot2)
> y <- lapply(1:100, function(i) rbinom(100, 100, .64))
> ybar <- sapply(y, mean)
> ybarbar <- mean(ybar)
> ybarbar
[1] 64.0321
> var(ybar)
[1] 0.2313501 # makes sense: 0.64 * 0.26
```

The beauty is that there's no preconditions on the distribution we used (maybe way out there in measure theory, but not here); the averages are distributed normally. One can plot these and see what happens.

```
GenerateSamples <- function(n) {
    # Distributions with similar means and such
    # Note: rnorm is noticeably the slowest (because no explicit integration formula)
    options <- c(function() rnorm(1, mean=60, sd=7),
                 function() rexp(1, rate=1/70),
                 function() rpois(1, lambda=70),
                 function() runif(1, min=50, max=100),
                 function() rbinom(1, 100, .64))
    sapply(1:n, function(i) options[[sample(1:5, 1)]]())
}
```

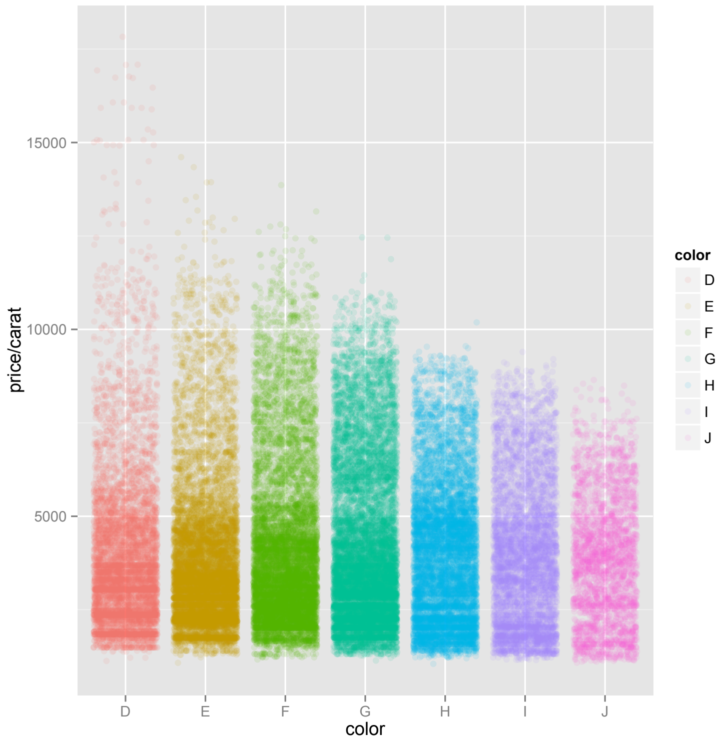


Figure 8. Result of plotting the color of a diaomond to its price per carat (an indicator of quality). Since colors measure quality, the results aren't too surprising — but notice how easy this was, and how pretty the plot is.

`ggplot2` is the industry standard; books have been written about it, etc.

```
> y <- GenerateSamples(10000)
qplot(y, geom="density")
```

The resulting graph is in Figure 9.

```
> y <- lapply(1:1000, function(i) GenerateSamples(10))
> ybar <- sapply(y, mean)
> qplot(ybar, geom="density")
```

This plot looks more normally distributed, but it has a fat tail.

The Central Limit theorem is pretty awesome, and the reason that anyone ought to care about the normal distribution.

There's another distribution called the Cauchy distribution, whose probability density function is

$$f(x) = \frac{1}{\pi\gamma(1 + ((x - x_0)/\gamma)^2)}.$$

Here, x_0 affects the center, and γ is the scale parameter, which changes how closely the mass is distributed to the mean. This suggests that its moments might be well-behaved, but they're really not.

```
# We're assuming the scale parameter is 1 and x_0 = 0, to make life simpler.
cauchy.smaples <- c(1.8, -3.1, 2.3, 1.1, 11.9)
Cauchy.Likelihood <- function(theta, x) {
    prod(1/ (pi * (1 + (x-theta)/2))) # accounts for multiple samples
}

# Now, we can do something like this, a really cool language feature:
> Cauchy.Likelihood <- Vectorize(Cauchy.Likelihood, vectorize.args="theta")
> Cauchy.Likelihood(c(1,4,7), cauchy.samples)
[1] 3.436973e-07 4.689140e-09 5.470806e-11
```

Remember that Newton's method could be useful here, to calculate the derivative of the Cauchy likelihood function and set it equal to zero (the goal here being to numerically approximate the maximum likelihood of the Cauchy distribution,

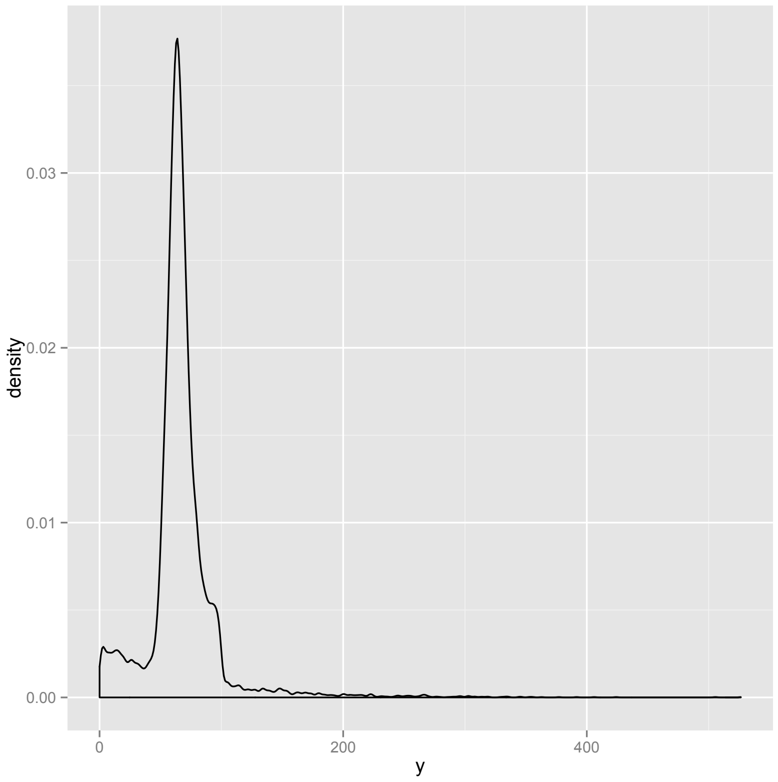


Figure 9. Several common distributions stacked on top of each other.

since its hard to do analytically). And, conveniently enough, R offers the capacity for symbolic differentiation. But log-likeihood is probably a lot easier to handle.

So let's implement Newton's method in R:

```
# derivative of log-likelihood
cauchyLLDeriv <- function(theta, x) {
    return(sum(2 * (x - theta) / (1 + (x - theta)^2)))
}

# second derivative
CauchyLLDeriv2 <- function(theta, x) {
    d <- 1 + (x -theta)^2
    return(sum((4 * (x - theta)^2)/ d^2 - (2/d)))
}

# n is the number of iterations
# theta0 is te initial guess
NewtonsMethod <- function(x, deriv, deriv2, n=10, theta0=mean(x)) {
    theta <- theta0

    for (i in 1:n) {
        theta <- theta - deriv(theta, x) / deriv2(theta, x)
        # This function is more like Python's print function, more versatile than R's
        cat("At iter ", i, theta, "CL is ", Cauchy.Likelihood(theta, x), "\n")
    }
}
```

There's an error in the above function, but the idea is right, and the correct version shows that $\theta \approx 1.6978$, which is quite far from the scale parameter.

12. Cool Things: 6/3/14

The point of today's lecture is to demonstrate some neat stuff about R, including how it can do the CS 109 project in very little code.

First, basic I/O. We've seen `readline()`, but not much else. This accepts a string to use as the prompt. However, there's another function called `scan()`, which reads data from a file (which can be a URL!) or some text, and then reads in data of a certain type. The type is chosen with the argument `what = double()` (a numeric vector of length zero, so a clever way of making a strongly typed null value). One can also control the number of things to read in, via argument `n`, which defaults to EOF (or end of string), `sep` controls the separator, which is a space by default. The argument `quiet=T` signifies to not report however much was scanned (which is default). If parsing fails, then an error is reported.

Of course, the most common parsing/reading function is `read.table`, which reads in text as a data frame from a file. Correspondingly, `dump` takes a vector of names of objects and then writes them to a file. This is useful to save data, and using `source(`*filename*`)`, the objects can be recovered later, in another session.

Models The syntax `x ~ y` means that `y` is modeled by `x`. This statement has class `"formula"`, and is a nice way of storing a relationship between two variables (represented in `R`, of course, by numeric vectors). The `lm` function (for linear model) accepts a formula (and optionally a data frame); for example, passing it `kn(new ~ old)` for housing price data (`data(home)` in the `usingR` library) allows it to calculate the correlation and provide a lot of information: the best-fit line, standard error, r^2, and so on.

In `ggplot2`, one can superimpose plots with `+`, so calling something like `qplot(old, nw, data=home) + stat_function(fun=function(x) coef(m)[1] + coef(m)[2])`, where `coef()` called on an `lm` object provides the coefficients of the linear relationship. This draws the data and its best-fit line. There's a `smooth` operator in `qplot` that does something similar, though this is just syntactic sugar for a call to `lm`.

This can be used to build up more complicated relationships, e.g. `new ~ old + I(old^2)`. Now, `lm` provides coefficients for each term `old` and `old^2`, so it's possible to feed it into `stat_function` and see the relationship. In this specific case, it doesn't do much, because the resulting coefficient is very small, but it's still interesting. The `I()` call tells `lm` to handle the two of them separately. This quadratic model does seem to fit better, but perhaps there's a bit of overfitting going on. But it allows one to try out lots of different models easily.

Then, there's a more general statistical/modeling concept called a generalized linear model, or a GLM. Basically, anything in the exponential family of distributions (all of the well-known ones) can be used to create these models. The `R` function `glm` is used for this; it solves generalized linear models, and thus addresses things like linear and logistic regression. The important point is that `glm` is how one does logistic regression in `R`.

Thus, the CS 109 programming assignment really just boils down to

```
# Reads in a file and puts the data into a frame
FileToDataFrame <- function(file) {
    x <- scan(file, what = " ", n = -1, sep = "\n")
    # some amount of processing

    data <- data.frame(vars, response = responses)
    return(data)
}

library(e1071) # Naive Bayes classifier

train <- FileToDataFrame("simple-train.txt")
test <- FileToDataFrame("simple-test.txt")

m <- glm(response ~ ., family = binomial("logit"), data = train)
prediction <- predict(m, test, type = "response")
prediction <- prediction > .5
mean(prediction == test[[response]])

m2 <- naiveBayes(response ~ ., data = train)
# the meat, right here.
prediction <- predict(m2, test, type = "raw")
prediction <- prediction[, 2] > prediction[, 1]
mean(prediction == test[[response]])
```

This is very nicely concise; you spend more time reading in the files than actually doing the machine learning. Contrast this with the languages most people use for their 109 programming assignment...