01 - Computers Don't Do Real Math

BIOS:7330 - Grant Brown

1 A Comforting Illusion

2 Binary Formats

3 Boolean Logic and Bitwise Operations

Section Contents

1 A Comforting Illusion

A Comforting Illusion

- Current computing tools are great: it's much easier than it used to be to be productive quickly when learning a new language.
- Languages which tend to 'take care' of difficult details, or which ship with lots of neat features are often called 'high level languages'.
- These tools tend to have a number of 'layers of abstraction' between the user and the computer hardware.
- Many high level languages are also 'interpreted', rather than compiled languages: https://www.youtube.com/watch?v=JNMy969SjyU

What is R anyway?

R is a self styled 'statistical computing environment', and it comes with an amazing array of tools:

- The base R language; a high level, interpreted, general purpose programming language
- Out-of-the-box support for vectors and matrices, and the associated mathematical operations
- Built in software for many types of statistical models, general purpose optimization tools, and families of functions related to probability distributions
- Several different graphical tools for various types of plotting
- A robust packaging system, and an internet capable package manager.
- Many, many other features

In addition to these built-in features, R has been extended in innumerable ways by user contributed packages.

What is R anyway?

- The internals of R are always evolving. In fact, thanks to the efforts of folks like Dr. Tierney in the statistics department, R now has a bytecode compiler and support for Just-in-time compilation, as mentioned in the video a few slides ago. See This paper
- Like many high level languages, R also provides interfaces to other languages, including compiled ones.
- We'll spend quite a bit of time working with C/C++ in R later in the course

The illusion: Binary Numbers

- Computers represent data using base 2, as a sequence of zeros and ones, each digit is called a "bit"
- Data is organized into blocks which are multiples of 8 bits, known as a Byte
- Bytes fall into larger groupings (Wikipedia)
- The precise meaning of a collection of bits depends on the **data type**, or the binary format with which the computer interprets the data.
- **Question:** If all data on a computer is 'binary', what's the difference between data types we refer to as 'binary data' (such as SAS data sets, .rda files, .xls files, etc.), and 'text data' like CSV files?

Section Contents

2 Binary Formats

Binary Formats: Unsigned Integers

- Unsigned integers describe integers between zero and some maximum representable value dictated by the number of bytes dedicated to storing the number.
- Computer integers are known as 'fixed point' numbers
- Any sequence of binary digits must be interpreted in a particular order, according to its endianness: big or little endian. This is equivalent, when writing digits, to reading from left to right or right to left
- Big endian (left to right) powers of 2: $(2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0)$
- The number represented can be thought of as the dot-product with this sequence of powers of 2. For example, the number "5" could be represented in big endian order as 00000101

Binary Formats: Unsigned Integers

```
for (i in 1:1000){print(intToBits(i)); Sys.sleep(0.25)}
```

Binary Formats: Unsigned Integers

- While it's clear how a computer can represent non-negative integers using unsigned fixed point binary numbers, but it may be less obvious how we do arithmetic with them.
- In principal, this actually works very much like what we'd expect in base-10 arithmetic
- Consider the problem of addition:
  - 5 : (00000101)
  - 11 : (00001011)
- Consider the problem of subtraction (requires care, generaly recommend signed data types):
  - 11 : (00001011)
  - 5 : (00000101)

Example with marbles

Binary Formats: Unsigned Integers

- Before we get to multiplication, there's another operation which we need to mention: shifting
- Right and left shifting is exactly what it sounds like. To shift a binary number to the left:
  - Discard the leftmost digit - this information is lost
  - Add a '0' as the rightmost digit
- Example:
  - $11: \ (00001011)$
  - $11 << 1: \ (00010110) = 11 \cdot 2^1 = 22$
  - $11 << 2: \ (00101100) = 11 \cdot 2^2 = 44$
- In general, $A << B = A \cdot 2^B$

Binary Formats: Unsigned Integers

- This helps motivate an important algorithm for multiplication
- If we think of our binary integers as coefficients multiplied by powers of two, it's clear that we can represent a number x as $\sum_{i=1}^{n}(2^i\alpha_i)$, where α_i is equal to one or zero, depending on whether the corresponding coefficient in x is a one or zero.
- We can see, $Ax = A\sum_{i=1}^{n}(2^i\alpha_i)$
- This reduces multiplication to the following operations, which we know how to do:
  - Multiplication by powers of two (shifting)
  - Multiplication by zero
  - Addition
- The resulting algorithm is very ancient, and is known as peasent multiplication
- Similar techniques are used to define and implement division algorithms

Binary Formats: Unsigned Integers

- While we're describing algorithms, they're not algorithms which you're going to be writing - these are implemented 'in hardware', as circuits in Central Processing Units (CPUs), Graphical Processing Units (GPUs), and other computational hardware.
- We're also glossing over the surrounding computer architecture, which is responsible for many things, including:
  - storing and accessing data in RAM
  - managing peripheral systems like hard drives and networking systems
- Understanding how these tools are constructed is very useful for understanding numerical problems.

Binary Formats: Signed Integers

- A negative integer can be represented as a positive integer, along with an indicator of the sign (a sign-bit)
- Most computer systems use a slighty different representation known as "two's complement", where the sign bit (most significant position) is set to a '1', all other bits are set to their complement, and one is added to the result. (Wikipedia)
- This has the advantage that the usual algorithms for addition and multiplication work without modification
- Example:
  - $11:\ (00001011)$
  - $-11:\ (11110101)$
  - Sum: (00000000)

Binary Formats: ASCII

- ASCII is the most common 'character encoding', a mapping from binary digits to human readable characters.
- A table of values, along with the history and development of ASCII is given here.
- This brings up another numeric representation you should be aware of: hexadecimal
- 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- One byte is mapped to two hex characters, from 00 to FF.

Binary Formats: ASCII - Example

- Consider the following example:

```
writeBin(1:128, con = "out.bin", size = 1)
```

- **Exercise:** Examine the resulting file in a text editor, and read the help page for 'writeBin'.

Binary Formats: Floating Point Numbers

- The world of computer integers is relatively straightforward
  - They're exactly representable, within a given numeric range
  - Computing with integers can be very, very fast, compared to floating point arithmetic
  - If the possible range of numbers is known ahead of time, integer data can be quite small, saving both storage space and the amount of time spent transfering data between components of the computer (often the single biggest bottleneck for computation!)
- On the other hand, most of the mathematics used in the practice of statistics concerns real numbers in some way - we need some way to represent decimal data.
- For performance critical applications, however, keep an eye out for potential integer approximations to the algorithms you're using.

Binary Formats: Floating Point Numbers

- Floating point numbers offer a clever tradeoff between the ability to represent very large numbers, and the ability to represent very small numbers.
- Just like for integers, floating point numbers can be represented by different numbers of bytes - offering a tradeoff between range/precision and size
- Binary floating point numbers are somewhat analogous to scientific notation, consisting of:
  - A signed significand (s), giving a digit string.
  - A signed exponent (e), which is multiplied by the significand
- We then compute $s \cdot 2^e$
- There are a lot more to these constructs, but these two components reveal the underlying tradeoff. (Additional Reading)

Binary Formats: Floating Point Numbers

float

1 bit sign	8 bit exponent	23 bit mantissa

double

1 bit sign	11 bit exponent	51 bit mantissa



Binary Formats: Semi-Reliable Size Conventions

C language data types

Binary data example: MNIST Digits

- One of the first ways in which statisticians often interact with binary data directly is via data sets encoded in a binary format.
- As an example, the MNIST image recognition data which serves as a canonical basis for testing machine learning techniques is stored as binary data: link
- **Exercise:** Figure out how to read in the MNIST data

Section Contents

3 Boolean Logic and Bitwise Operations

Side Note: Boolean Logic, XOR, and Encryption

- Everyone should be familiar with boolean logic, and how it relates to binary data.
- In binary, a '1' is usually taken to indicate a 'yes', or 'true' value, while a '0' is used to indicate a 'no', or 'false' value.
- These can becompared with the usual operations:
  - 1 AND 1 = 1, 1 AND 0 = 0, 0 AND 0 = 0
  - 1 OR 1 = 1, 1 OR 0 = 1, 0 OR 0 = 0
- There are more advanced operations and properties, like associativity, distributivity, and De Morgan's laws.
- Example: A AND (B OR C) = (A AND B) OR (A AND C)
- These tools are extraordinarily important for statistical programming; you will find it helpful to be able to do algebra on your conditional logic.
- This operation can be applied at a bit-wise level. See the R functions starting with 'bitw'.

Side Note: Boolean Logic, XOR, and Encryption

- One important bit-wise operation which you may or may not have heard of is the 'exclusive or', or XOR comparison.
- *A* XOR *B* is equivalent to 'A or B, but not both'. Equivalently, (*A* OR *B*) AND NOT (*A* AND *B*)
  - 1 XOR 1 = 0
  - 1 XOR 0 = 1
  - 0 XOR 0 = 0
- The XOR operation is not always of direct interest to statisticians, but it's incredibly important for encryption
- It has a useful property: it can aggregate/preserve randomness: a random bit XOR'd with a non-random bit is random.

Side Note: Boolean Logic, XOR, and Encryption

- This can be used to implement a form of encryption known as a 'one-time pad'
  1. Convert a text string, X, to a sequence of bits.
  2. Generate a random (p=0.5) sequence of bytes of the same length as that for X, call this sequence K
  3. Compute $E = X$ XOR K
- The string E is now encrypted, and (assuming that the key, K is truly random and unknown), it is impossible to infer the content X, because all possible strings of the same length are equally likely.
- The content, X, can be recovered by E XOR K, if the key is known. Try to think about why this would be true.

More information: Wikipedia

Exercise: One Time Pad

Write a set of R functions:

- 'encrypt(string)', which accepts a text string and returns a list containing an encrypted version of it, as well as the key used to encrypt it.
- 'decrypt(enc)', which accepts a list (as generated by the 'encrypt' function), and returns the decrypted string.

This is a useful programming exercise, and hopefully gives you a bit of an idea about how encryption algorithms work. Data security is an increasingly important part of statistical computing.

Conclusions

- Due to the default choices on powerful, modern computer systems, we often forget that we're not working with real numbers
- Much of the time, your code will 'just work' when translated from your analytic derivations.
- Nevertheless, an understanding of the kinds of things which can go wrong will help you in **many** circumstances.
- **Exercise:** Explore 'overflow' in likelihood evaluation on the density and log scales.
- We'll return to these topics as we encounter problems throughout the course.