Lecture 6: Numerical Stability and R Errors

STAT 385 - James Balamuta

June 21, 2016

On the Agenda

1. Setting up Groups
   - Stable Marriage Problem
2. Numerical Stability within R
   - Observing the Variance Estimator
   - When $1 + 1 \neq 2$
3. Errors
   - Common R Errors

Grouping Algorithm

- How to Assign Groups?
  - Instructor picks randomly. Not everyone is happy.
  - Students pick themselves. Someone feels left out.
  - What about an Algorithm?

Stable Marriage Problem

- There is a fantastic TV documentary exploring algorithms on Netflix called: The Secret Rules of Modern Living: Algorithms
- Within it, the details of the Stable Marriage algorithm used in Medical School residency pairings and on Dating Websites is described.
- Video Explanation Link (Queued)

Grouping Today

- Our algorithm will try to optimize the following criteria:
  - Leadership style
  - Schedule
  - Skills
  - Project Interest

Moving along...

Up next, we're going to learn about implementing a **variance estimator**!

Computational Statistics and the Variance Estimator

- **Computational Statistics**, the red-headed step child between statistics and computer science, has worked time and time again to obtain an algorithm for calculating *variance*.
- Yes, **variance.**

Why is the algorithm for variance complicated?

Consider the definitions of **Mean** and the **Variance**:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^2$$

Note that the algorithm for the variance relies upon a version of the "Sum of Squares", e.g.

$$S_{xx} = \sum_{i=1}^{n} (x_i - \bar{x})^2$$

That is:

$$\sigma^2 = \frac{1}{n} S_{xx}$$

Sum of Squares

Sum of Squares provides a measurement of the total variability of a data set by squaring each point and then summing them.

$$\sum_{i=1}^{n} x_i^2$$

More often, we use the **Corrected Sum of Squares**, which compares each data point to the mean of the data set to obtain a deviation and then square it.

$$\sum_{i=1}^{n} (x_i - \bar{x})^2$$

Why do we use the corrected Sum of Squares?

- When we talk about Sum of Squares it will always be the *corrected* form.
- The question for today is: **Why?**

Arithmetic Overflow

Using the initial (uncorrected) Sum of Squares definition is sure to cause an **arithmetic overflow** when working with large numbers, for example:

```
(x = (1.0024e6)^2)                          # Uncorrected
```

```
## [1] 1.004806e+12
```

```
(y = (1.0024e6 - 1.0000156e6)^2) # Corrected
```

```
## [1] 5685363
```

If we were to add to `x`, we would hit *R*'s 32-bit integer limit (see `?integer`):

```
.Machine$integer.max  # Maximum integer in memory
```

```
## [1] 2147483647
```

Arithmetic Overflow - Behind the Scenes

$R > 3.0$, will try to address this behind the scenes by automatically converting the `integer` to a `numeric` with precision:

```
.Machine$double.xmax  # Maximum numeric in memory
```

```
## [1] 1.797693e+308
```

Arithmetic Overflows and Big Data

- Within *Big Data* this problem may be more transparent as the information summarized is larger.
- Thus, you may need to use an external package for very big numbers. I would recommend the following:
  - Rmpfr
  - bit64

Forms of the Variance Estimator

- Two-Pass Algorithm Form:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

- Naive Algorithm Form:

$$\sigma^2 = \frac{\sum\limits_{i=1}^{n} x_i^2 - \left(\sum\limits_{i=1}^{n} x_i\right)^2 / n}{n}$$

Sum of Squares Manipulation for Naive version

I'm opting to simply show the S_{xx} modification instead of working with σ^2 since it just scales the term by $\frac{1}{n}$.

$$\begin{aligned}
S_{xx} &= \sum_{i=1}^{n} (x_i - \bar{x})^2 && \text{Definition} \\
&= \sum_{i=1}^{n} \left(x_i^2 - 2x_i\bar{x} + \bar{x}^2 \right) && \text{Expand the square} \\
&= \sum_{i=1}^{n} x_i^2 - 2\bar{x} \sum_{i=1}^{n} x_i + \bar{x}^2 \sum_{i=1}^{n} 1 && \text{Split Summation} \\
&= \sum_{i=1}^{n} x_i^2 - 2\bar{x} \sum_{i=1}^{n} x_i + \underbrace{n\bar{x}^2}_{\sum_{i=1}^{n} c = n \cdot c} && \text{Separate the summation} \\
&= \sum_{i=1}^{n} x_i^2 - 2\bar{x} \left[n \cdot \frac{1}{n} \right] \sum_{i=1}^{n} x_i + n\bar{x}^2 && \text{Multiple by 1}
\end{aligned}$$

Sum of Squares Manipulation for Naive version - Cont.

$$
\begin{aligned}
S_{xx} &= \sum_{i=1}^{n} x_i^2 - 2\bar{x}\left[n \cdot \frac{1}{n}\right]\sum_{i=1}^{n} x_i + n\bar{x}^2 && \text{Multiple by 1} \\
&= \sum_{i=1}^{n} x_i^2 - 2\bar{x}n \cdot \underbrace{\left[\frac{1}{n}\sum_{i=1}^{n} x_i\right]}_{=\bar{x}} + n\bar{x}^2 && \text{Group terms for mean} \\
&= \sum_{i=1}^{n} x_i^2 - 2\bar{x}n\bar{x} + n\bar{x}^2 && \text{Substitute the mean} \\
&= \sum_{i=1}^{n} x_i^2 - 2n\bar{x}^2 + n\bar{x}^2 && \text{Rearrange terms} \\
&= \sum_{i=1}^{n} x_i^2 - n\bar{x}^2 && \text{Simplify}
\end{aligned}
$$

Implementing Naive Variance

```r
var_naive = function(x){
  n = length(x)                 # Obtain the length
  sum_x = 0                     # Storage for Sum of X
  sum_x2 = 0                    # Storage for Sum of X^2
  for(i in seq_along(x)){ # Calculate sums
    sum_x = sum_x + x[i]
    sum_x2 = sum_x2 + x[i]^2
  }

  # Compute the variance
  v = (sum_x2 - sum_x*sum_x/n)/n
  return(v)
}
```

Implementing Two-Pass Variance

```
var_2p = function(x){
  n  = length(x)                # Length
  mu = 0; v = 0                 # Storage for mean and var

  for(i in seq_along(x)){  # Calculate the Sum for Mean
    mu = mu + x[i]
  }

  mu = mu / n                   # Calculate the Mean

  for(i in seq_along(x)){  # Calculate Sum for Variance
    v = v + (x[i] - mu)*(x[i] - mu)
  }

  v = v/n                       # Calculate Variance
  return(v)                     # Return
}
```

Calculations

```r
set.seed(1234) # Set seed for reproducibility
x = rnorm(2e6, mean = 1e20, sd = 1e12)

(method1 = var_naive(x))
```

```
## [1] 1.318357e+27
```

```r
(method2 = var_2p(x))
```

```
## [1] 1.001425e+24
```

```r
(baser = var(x)*((2e6)-1)/(2e6))
```

```
## [1] 1.001425e+24
```

```r
all.equal(method1, method2)
```

```
## [1] "Mean relative difference: 0.9992404"
```

R's Implementation

R opts to implement this method using a two-pass approach.

- Check out the source here
- There are quite a few papers on this topic going considerably far back. See Algorithms for Computing the Sample Variance: Analysis and Recommendations (1983)

$1 + 1 \neq 2$

Computers in all their infinite wisdom and ability are not perfect. One of the particularly problematic areas of computers is handling **numeric** or **float** data types. Consider:

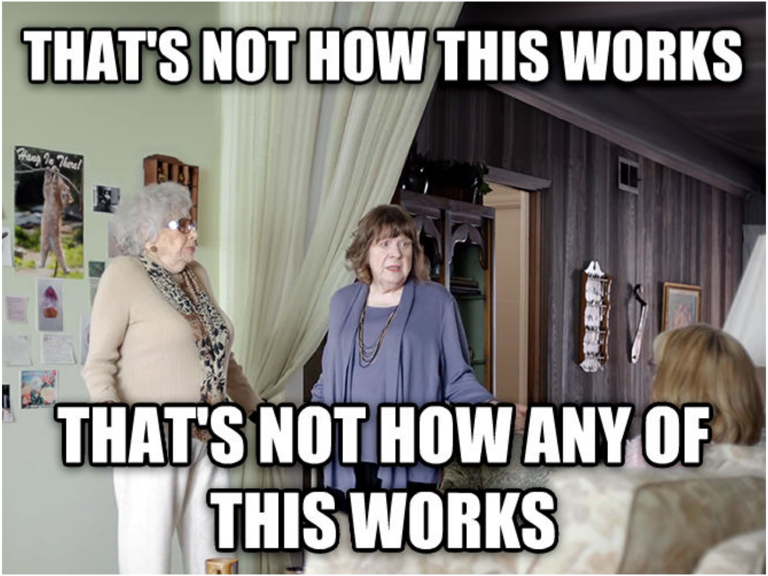
```
x = 0.1
x = x + 0.05
x
```

```
## [1] 0.15
```

```
if(x == 0.15){
  cat("x equals 0.15")
} else {
  cat("x is not equal to 0.15")
}
```

```
## x is not equal to 0.15
```

Why isn't `x` equal to 0.15!?


THAT'S NOT HOW THIS WORKS
Hang In There!
THAT'S NOT HOW ANY OF THIS WORKS

Enter: Numerical Stability

In essence, *R* views the two numbers differnetly due to rounding error during the computation:

```
sprintf("%.20f", 0.15) # Formats Numeric
```

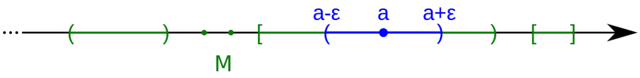
```
## [1] "0.14999999999999999445"
```

```
sprintf("%.20f", x)
```

```
## [1] "0.15000000000000002220"
```

ϵ neighborhood

Specifically, we are hitting a machine tolerance fault given by an ϵ neighborhood.



The value of the ϵ is given by:

```
.Machine$double.eps
```

```
## [1] 2.220446e-16
```

This gives us the ability to compare up to `1e-15` places accurately.

```
sprintf("%.15f", 1 + c(-1,1)*.Machine$double.eps)
```

```
## [1] "1.000000000000000" "1.000000000000000"
```

Discrete Solution Check

To get around rounding error between two objects, we add a tolerance parameter to check whether the value is in the ϵ neighborhood or not.

```
all.equal(x, 0.15, tolerance = 1e-3)
```

```
## [1] TRUE
```

Discrete Solution Check

Since `all.equal` may not strictly return `TRUE` or `FALSE`, it is highly advisable to wrap it in `isTRUE()`, e.g.

```
isTRUE(all.equal(x, 0.15))
```

```
## [1] TRUE
```

Thus, in an `if` statement, you would use:

```
if(isTRUE(all.equal(x, 0.15))){
  cat("In threshold")
} else {
  cat("Out of threshold")
}
```

```
## In threshold
```

Bad Loop

To magnify the issue consider a loop like so:

```
inc_value = 360L / 14L # Value to increment
i = 0                  # Increment storage
while(i != 360){       # Loop
  i = i + inc_value    # Add values
}
```

After 14 iterations, the loop should complete, but it does *not*! In fact, this loop will go onto infinity.

Good Loop

To fix the looping issue, we opt to always stick with *integer*s as counters

```
inc_value = 360 / 14 # Value
i = 0L                # Integer
o = 0                 # Numeric
while(i != 14L){
  o = o + inc_value   # Sum
  i = i + 1L          # Increment loop
}
i
```

```
## [1] 14
```

Summary

- Statistics and Computer Science rely on each greatly in this Brave New World of Data Science.
- When working with big numbers, understand that arithmetic overflow is a reality.
- **Never, ever, ever** use a floating-point representation as an incrementor for a loop.
  - Always use an `integer` for an incrementor and then convert it to a `numeric` within a function.
- This topic **will** come up again when we switch to using `Rcpp`.

Coming up next. . . .

Common *R* errors. . .

Checking for Equality vs. Assignment

The most common error by far that affects programmers is making an assignment when trying to check for equality (and vice versa)

```r
# Assigning in `if`
if(x = 42) { cat("Life!") }
## Error: unexpected '=' in "if(x ="

# Correct
if(x == 42) { cat("Life!") }
```

```r
# Equality Check instead of Assignment
x == 42
## No Error, but prints `TRUE` or `FALSE`

# Correct
x = 42
```

if vectorization usage

As emphasis on vectorization grows, there is a tendency to compare two vectors using the default `if()` instead of `ifelse()`

```
x = 1:5
y = 2:6
if(x > y){ T }
## Warning messages:
## In if (x > y) { : the condition has length > 1 and only

# Correct
ifelse(x > y, T, F)
```

Vector Recycling

Sometimes the length of vectors are not equal or the data does not divide evenly or oddly when perform a vectorized computation.

```
x = 1:5
y = 2:3

x + y
## Warning message:
## In x + y : longer object length is not a multiple
## of shorter object length

# Correct
x = 1:4
y = 2:3
x + y
# Repeats y twice
# 1 + 2, 2 + 3, 3 + 2, 4 + 3
```

Mismatched curly brackets {} or parentheses ()

Often it is ideal to use parentheses for order of operations or curly brackets {}, though this sometimes causes a mismatch.

```
2*(x + y))
## Error: unexpected ')' in "2*(x + y))"

# Corrected
2*((x + y))
```

No Multiplication

When working on computations, sometimes we just “slip” and opt not to write a multiplication sign thinking the interpreter can understand the context.

```
2x+4
## Error: unexpected symbol in "2x"

# Correct
2*x + 4
```

Manual Data Entry

Sometimes it's easier as we'll see next week to manually enter data. The issue with this is sometimes you forget simple things like a `,`.

```
c(1, 2 3, 4)
## Error: unexpected numeric constant in "c(1,2 3"

# Correct
c(1, 2, 3, 4)
```

Strings in character values

At times, there may come a need to place a quotation inside of a string. To do this, requires using an escape character \ or using '' instead.

```
"toad"princess"
## Error: unexpected symbol in ""toad"princess"

# Corrected
"toad\"princess"
'toad"princess'
```

Handling Missing Value Operations

The `NA` character indicates the presence of a missing value. These missing values can play havoc with computations.

```
x = c(1,NA,2)
3 + x
# No Error, but: [1] NA

sum(x)
# No Error, but: [1] NA

# Corrected
1 + na.omit(x)    # Deletes NA
sum(x, na.rm = T) # Removes NA inside function
```

Finiteness of Values

R can have some funky finiteness problems due to how `NA` values are created.

```
x = c(NA,-Inf, Inf ,NaN)
is.na(x)
# No error, but: [1]  TRUE FALSE FALSE  TRUE

is.infinite(x)
# No error, but: [1] FALSE  TRUE  TRUE FALSE

# Correct
is.finite(x)
# [1] FALSE FALSE FALSE FALSE
```

Summary of Errors

- There are many odd errors that *R* can cause.
- StackOverflow is a great community to ask questions about errors.
- When dealing with `NA` values, be on your guard!