Numerical Stability

James Balamuta

Department of Informatics, Statistics
University of Illinois at Urbana-Champaign

June 29, 2017

Administrative

- HW1 grades should be released tomorrow (Wednesday)
- **HW2 is due on Sunday, July 2nd @ 11:59 PM**
- Extra OH on Thursday in IH 122 from 12 - 1 PM.

On the Agenda

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** given by:

$$\begin{aligned}\sigma^2 &= E\left[(X - E[X])^2\right] \\ &= \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 \; *\end{aligned}$$

*Discrete Case Representation

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.

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

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

History of the Sum of Squares

- **Sum of Squares** (SS) provides a measurement of the total variability of a data set by squaring each point and then summing them.
- As we have seen, SS appears during linear regression and ANOVA with the forms of TSS, FSS, and RSS.

(Uncorrected) Sum of Squares

- In an "uncorrected" ANOVA table, where the intercept is considered a source, we have the **Total Sum of Squares (TSS)** given as:

$$TSS = \sum_{i=1}^{n} y_i^2$$

Table 1: Uncorrected ANOVA Table

Source	DF	SS
Intercept	1	$n\bar{Y}^2$
Fitted	$p-1$	$\sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2$
Residual	$n-p$	$\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$
Total	n	$\sum_{i=1}^{n} y_i^2$

(Corrected) Sum of Squares

- 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.

$$TSS = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

Table 2: Corrected ANOVA Table

Source	DF	SS
Fitted	$p-1$	$\sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2$
Residual	$n-p$	$\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$
Total	$n-1$	$\sum_{i=1}^{n} (y_i - \bar{y})^2$

Why do we use the corrected Sum of Squares?

- In any case, when we talk about Sum of Squares it will always be the *corrected* form.
- The question for today:

Why is this approached preferred computationally?

Calculating Sum of Squares for One Point

Using the Uncorrected vs. the Corrected Sum of Squares definition would yield:

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

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

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

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

Imagine There's More than One Point

Now, consider applying both of the definitions over a sequence of n points and summing the results.

- Which definition might lead to a computational issue?

Arithmetic Overflow

In the case of the uncorrected version, it is sure to cause an **arithmetic overflow** when working with large numbers.

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`
  - `gmp`
  - `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 $1/n$.

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

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

```
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(method2, baser)
```

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

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)

On the Agenda

1. Numerical Stability
   - Variance Estimation
   - Overflows
   - Theory of Estimators
   - Variance Implementations
2. Comparing Numbers
   - Precision
   - Case Study: Decimal Index

$1 + 1 \neq 2$

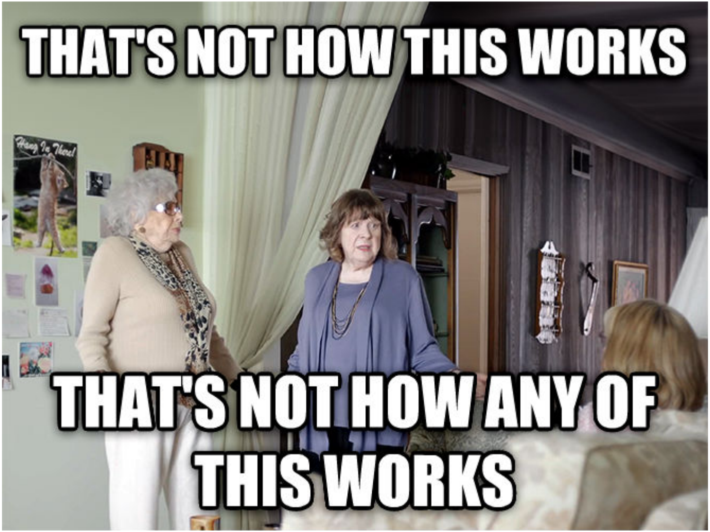
Computers in all their infinite wisdom and ability are not perfect. One of the limiting areas of computing is handling **numeric** or **float** data types.

```
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
```


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
```

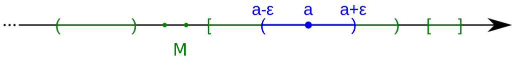
```
## [1] "0.14999999999999999445"
```

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

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

ϵ neighborhood

Rounding error exists due to 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
```

If we disable the ϵ neighborhood, we return to beginning with a problem:

```
all.equal(x, 0.15, tolerance = 0)
```

```
## [1] "Mean relative difference: 1.850372e-16"
```

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 = 360 / 14    # 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` values as counters

```
inc_value = 360 / 14 # Value
i = 0L               # Loop Counter
o = 0                # Result Variable
while(i != 14L) {
  o = o + inc_value  # Sum
  i = i + 1L         # Increment loop
}
i
```

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

Summary

- Statistics and Computer Science rely on each other greatly in this *Brave New World* of Data Science.
- When working with big numbers, understand that **arithmetic overflow** is a reality and must be accounted for.
- **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`.