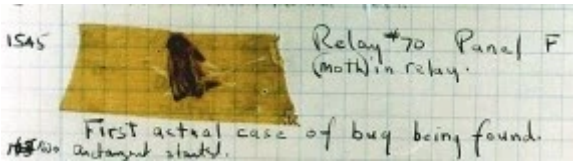
Win-Vector Blog

The Win-Vector LLC data science blog

My favorite R bug

May 23, 2015 John Mount Expository Writing, Practical Data Science, Pragmatic Data Science, Pragmatic Machine Learning, Programming, Rants, Statistics, Tutorials debugging, R

In this note am going to recount "my favorite R bug." It isn't a bug in R. It is a bug in some code I wrote in R. I call it my favorite bug, as it is easy to commit and (thanks to R's overly helpful nature) takes longer than it should to find.



The original problem I was working on was generating a training set as a subset of a simple data frame.

```
# read our data
tf <- read.table('tf.csv.gz',header=TRUE,sep=',')
print(summary(tf))
```

```
##        x                y
##  Min.   :-0.05075   Mode :logical
##  1st Qu.:-0.01739   FALSE:37110
##  Median : 0.01406   TRUE :2943
##  Mean   : 0.00000   NA's :0
##  3rd Qu.: 0.01406
##  Max.   : 0.01406
```

```
# Set our random seed to our last state for
# reproducibility.  I initially did not set the
# seed, I was just using R version 3.2.0 (2015-04-16) -- "Full of Ingredients"
```

```
# on OSX 10.10.3
# But once I started seeing the effect, I saved the state for
# reproducibility.
.Random.seed = readRDS('Random.seed')

# For my application tf was a data frame with a modeling
# variable x (floating point) and an outcome y (logical).
# I wanted a training sample that was non-degenerate
# (has variation in both x and y) and I thought I would
# find such a sample by using rbinom(nrow(tf),1,0.5)
# to pick random training sets and then inspect I had
# a nice training set (and had left at least one row out
# for test)
goodTrainingSample <- function(selection) {
  (sum(selection)>0) && (sum(selection)<nrow(tf)) &&
    (max(tf$x[selection])>min(tf$x[selection])) &&
    (max(tf$y[selection])>min(tf$y[selection]))
}

# run my selection
sel <- rbinom(nrow(tf),1,0.5)
summary(sel)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.0000  0.0000  0.4987  1.0000  1.0000
```

```
sum(sel)
```

```
## [1] 19974
```

Now I used rbinom(nrow(tf),1,0.5) (which gives a sample that should be about half the data) instead of sample.int(nrow(tf),floor(nrow(tf)/2)) because I had been intending to build a model on the training data and then score that on the hold-out data. So I thought referring to the test set as !sel instead of setdiff(seq_len(nrow(tf)),sel) would be convenient.

```
# and it turns out to not be a good training set
print(goodTrainingSample(sel))
```

```
## [1] FALSE
```

```
# one thing that failed is y is a constant on this subset
print(max(tf$y[sel])>min(tf$y[sel]))
```

```
## [1] FALSE
```

```
print(summary(tf[sel,]))
```

```
##        x               y
##  Min.   :0.01406   Mode :logical
##  1st Qu.:0.01406   FALSE:19974
##  Median :0.01406   NA's :0
##  Mean   :0.01406
##  3rd Qu.:0.01406
##  Max.   :0.01406
```

```
# Whoops! everything is constant on the subset!

# okay no, problem that is why we figured we might have to
# generate and test multiple times.
```

But wait, lets bound the odds of failing. Even missing the "y varies" condition is so unlikely we should not expect see that happen. Y is true 2943 times. So the odds of missing all the true values when we are picking each row with 50/50 probability is exactly 2^(-2943). Or about one chance in 10^885 of happening.

We have a bug. Here is some excellent advice on debugging:

> *"Finding your bug is a process of confirming the many things that you believe are true — until you find one which is not true." —Norm Matloff*

We saved the state of the pseudo random number generator, as it would be treacherous to try and debug someting it is involved with without first having saved its state. But that doesn't mean we are accusing the pseudo random number generator (though one does wonder, it is common for some poor pseudo random generators to alternate the lower bit in some situations). Lets instead work through our example carefully. Other people have used R and our code is new, so we really want to look at our own assumptions and actions. Our big assumption was that we called rbinom() correctly and got a usable selection. We even called summary(sel) to check that sel was near

50/50. But wait- that summary doesn't look quite right. You can sum() logicals, but they have a slightly different summary.

```
str(sel)
```

```
##  int [1:40053] 1 1 0 1 1 0 0 1 1 1 ...
```

Aha! sel is an array if integers, not a logical. That makes sense it represents how many successes you get in 1 trial for each row. So using it to sample doesn't give us a sample of 19974 rows, but instead 19974 copies of the first row. But what about the zeros?

```
tf[c(0,0,0),]
```

```
## [1] x y
## <0 rows> (or 0-length row.names)
```

Ah, yet another gift from R's irregular bracket operator. I admit, I messed up and gave a vector of integers where I meant to give a vector of logicals. However, R didn't help me by signaling the problem, even though many of my indices were invalid. Instead of throwing an exception, or warning, or returning NA, it just does nothing (which delayed our finding our own mistake).

The fix is to calculate sel as one of:

Binomial done right.

```
sel <- rbinom(nrow(tf),1,0.5)>0
test <- !sel
summary(sel)
```

```
##    Mode   FALSE    TRUE    NA's
## logical   19860   20193       0
```

```
summary(test)
```

```
##    Mode   FALSE    TRUE    NA's
## logical   20193   19860       0
```

Cutting a uniform sample.

```
sel <- runif(nrow(tf))>=0.5
test <- !sel
summary(sel)
```

```
##    Mode   FALSE    TRUE    NA's
## logical   20061   19992       0
```

```
summary(test)
```

```
##    Mode   FALSE    TRUE    NA's
## logical   19992   20061       0
```

Or, set of integers.

```
sel <- sample.int(nrow(tf),floor(nrow(tf)/2))
test <- setdiff(seq_len(nrow(tf)),sel)
summary(sel)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       2   10030   20030   20050   30100   40050
```

```
summary(test)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       1   10000   20020   20000   29980   40050
```

Wait, does that last example say that sel and test have the same max (40050) and therefore share an element? They were supposed to be disjoint.

```
max(sel)
```

```
## [1] 40053
```

```
max(test)
```

```
## [1] 40051
```

```
str(sel)
```

```
##  int [1:20026] 23276 3586 32407 33656 21518 14269 22146 25252 12882 4564 ...
```

```
str(test)
```

```
##  int [1:20027] 1 3 4 5 8 12 14 15 17 18 ...
```

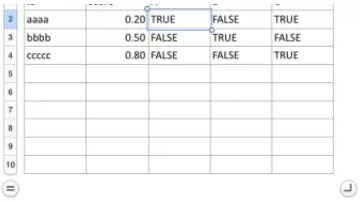
Oh it is just summary() displaying our numbers to only four significant figures even though they are in fact integers and without warning us by turning on scientific notation.

Don't get me wrong: I love R and it is my first choice for analysis. But I wish it had simpler to explain semantics (not so many weird cases on the bracket operator), signaled errors much closer to where you make them (cutting down how far you have to look and how many obvious assumptions you have to test when debugging), and was a bit more faithful in how it displayed data (I don't like it claiming a vector integers has a maximum value of 40050, when 40053 is in fact in the list).

One could say "just be more careful and don't write bugs." I am careful, I write few bugs- but I find them quickly because I check a lot of my intermediate results. I write about them as I research new ways to prevent and detect them quickly.

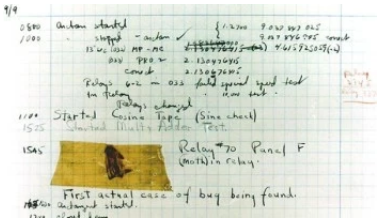
You are going to have to write and debug code to work as a data scientist, just understand time spent debugging is not time spent in analysis. So you want to make bugs hard to write, and easy to find and fix.

(Original knitr source here)



Excel spreadsheets are hard to get right
In "data science"

Using PostgreSQL in R: A quick how-to
In "Coding"



Factors are not first-class citizens in R
In "Computer Science"

May 23, 2015 John Mount Expository Writing, Practical Data Science, Pragmatic Data Science, Pragmatic Machine Learning, Programming, Rants, Statistics, Tutorials debugging, R

4 thoughts on “My favorite R bug”

Reproducibility

May 26, 2015 at 9:37 am

Um … you mention the word “reproducibility” and then start like this:

tf <- read.table('tf.csv.gz',header=TRUE,sep=',')

How on earth are we supposed to reproduce this? I don't have this file and you haven't provided it. Or what's the point of this:

.Random.seed = readRDS('Random.seed')

You read some data from a file that you have but others don't. Are you writing this blog for yourself or what?

jmount

May 26, 2015 at 12:51 pm

Sorry, I should have said explicitly. The data and random seed are in the GitHub repository directory linked to in the article. http://winvector.github.io/binomIssue/ The random seed turned out to be irrelevant once once looks further into the bug.

Reproducibility

May 26, 2015 at 9:40 am

… simple data sets can be given within text using dput or e.g. read.table(text=
“A B C
1 2 3
4 5 6″, sep=” “, header=TRUE)

Why not use these possibilities?

And why read random seed from a file (that nobody has except for you)? You can easily set it using set.seed.

---

jmount

May 26, 2015 at 12:59 pm

I wanted to save the seed I was originally seeing problems with (though that turned out irrelevant, but in debugging you don’t always know what will end up being important). set.seed() with a chosen constant would set to a new repeatable state, which wouldn’t be the state we had already seen. Of course, it later turns out the seed was irrelevant to the problem.

There is nothing wrong with how you entered the data. I just tend to use external files in my work (I wanted to load from a GitHub url- but something interferes with that).

Comments are closed.
