Stat 243: Problem Set 5

Eugene Yedvabny

10/22/2014

1

In this ML calculation we are trying to maximize the binomial likelihood parametrized as $\log \frac{p_i}{1-p_i} = X_i^T \beta$. Since we need p_i for $\mathcal{L}$ calculation, we invert the above formulat to yield $p_i = \frac{e^{X_i^T \beta}}{1+e^{X_i^T \beta}}$ The ML calculation can thus be expressed (in R) as `L<-prod(dbinom(y,n,exp(X%*%beta)/(1+exp(X%*%beta))))`

```
load("ps5prob1.Rda")
pred <- X%*%beta
p <- exp(pred)/(1+exp(pred))
dens <- dbinom(y,n,p)
L <- prod(dens)
cat("Likelihood is",L)
```

```
Likelihood is 0
```

Well, as expected, the result is 0. The likelihood function is a product of probability densities, which are by definition less than 1. Thus each new factor reduces the net product until it becomes negligibly small and exceeds the lower bound of the a double representation (the last non-zero value is 4.940656e-324 after 392nd factor).

The solution to the underflow issue is to compute *log-likelihood* instead of the actual likelihood. A log of a product is a sum of logs of the factors, so the exponential decay (which quickly underflows) becomes a higher-precision linear decay.

```
log_L <- sum(log(dens))
cat("log-Likelihood is",log_L)
```

```
log-Likelihood is -1862.331
```

The calculation should then be performed as a maximization of the log (or, more common, minimization of the negative log) in order to find more likely β for this model.

2

The sum of $1 + \sum^{10000} 10^{-16}$ is a rational number and has an *exact* answer - 1.000000000001 - so the ideal precision is **12 decimal points**. This is less than the usual 16 decimal point of a floating-point double, assuming the calculation is done correctly, we should get an exact answer.

```
options(digits = 16)
tiny_nums = rep(1e-16,10000)
x <- c(1,tiny_nums)
sum(x)
```

```
## [1] 1.000000000001
```

w00t! We have a very precise sum. Let's see how Python fares.

```
import numpy as np
x = np.repeat(1e-16,10001)
x[0] = 1
print "%.14f"% np.sum(x)
```

```
## 1.00000000000100
```

There's a bit of a wrangling with formatting, but the end result is the same- high precision sum to the right precision. Iterative summation, however, does not fare so well.

```
sum <- 0
for(i in 1:10001){
  sum <- sum + x[i]
}
sum
```

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

```
sum2 <- 0
for(i in 10001:1){
  sum2 <- sum2 + x[i]
}
sum2
```

```
## [1] 1.000000000001
```

Addition starting with 1 returns just a 1 with no decimal points. Summation starting with 1e-16 returns the right precision. This is because the machine epsilon is 1.11e-16, so $1 + 1 * 10^{-16}$ is rounded to just 1. But $1 * 10^{-16} + 1 * 10^{-16}$ has the same precision (1 decimal point) and the summation is exact. By the time a 1 is added to the cumulative sum of 1e-16, the second summand is greater than the epsilon, so the addition does not lose any precision either. The correct answer in this case has a precision of all 12 decimal points, while the wrong answer has 0.

```
import numpy as np
x = np.repeat(1e-16,10001)
x[0] = 1

sum = 0
for i in xrange(10001):
  sum = sum + x[i]
print "%.14f"%sum

sum2 = 0
for i in xrange(10000,-1,-1):
  sum2 = sum2 + x[i]
print "%.14f"%sum2
```

```
## 1.00000000000000
## 1.00000000000100
```

It seem python suffers from the same problem in iterative summation. Looking over the sum() source code (aside: R source is well layed out but it is in *dire* need of some better alignment. Code folding was the only way to figure out what's going on) it appears that every element of the provided array is coerced into one type (REAL by default), checked against max and min possible double... and then just summed up iteratively. So perhaps the crux is that the summation is done at full precision in C. There doesn't appear to be any reordering of the elements.