New York City R-Meetup

Columbia University
June 3, 2010

Jay Emerson
Yale University
`http://www.stat.yale.edu/~jay/`
Talk resources: `http://www.stat.yale.edu/~jay/Rmeetup/`

Abstract

A crash course in "poor man's" debugging via some non-trivial examples: an inefficiency in **R**'s `kmeans()` function; reshaping Olympic diving scores; working with streaming video.

Contents

1 About this document (recap from March R Meetup)

This document is intended to complement (and not completely duplicate) the talk, and was created using *Sweave* (which is included with **R**). For more information, see Friedrich Leisch's web page:

`http://www.stat.uni-muenchen.de/~leisch/Sweave/`

The short story: you use **R** to flush your "master copy" of the document (a `.Rnw` file) through *Sweave*, producing a `.tex` file which is then processed using LaTeX. In this way, you can include **R** code in your document and, automatically, the *results* of the code. Even plots are easy to integrate. The file (`Rmeetup.Rnw`) used to produce this document (`Rmeetup.pdf`) is available along with other materials related to the Meetup, at

`http://www.stat.yale.edu/~jay/Rmeetup/`

If you are interested in *Sweave* and look at `Rmeetup.Rnw`, please note that I've done a few unusual things because of the nature of this particular presentation. Please read the special comments at the top of the file.

2 Debugging kmeans()

2.1 Background

Have a look at what Wikipedia has to say about the k-means algorithm:

`http://en.wikipedia.org/wiki/K-means_algorithm`

In particular, look at the picture, a "demonstration of the algorithm." Do you buy it? I don't. And the last picture certainly isn't the result of a convergence. So, I'll quickly introduce Lloyd's k-means algorithm (the first and simplest of the clustering algorithms, which is included in **R** but is not the default).

Lloyd's algorithm (Lloyd, 1957) takes a set of observations or cases (think: rows of an nxp matrix, or points in $\mathbb{R}^p$) and clusters them into k groups. It tries to minimize the within-cluster sum of squares

$$\sum_{i=1}^{k} \sum_{x_j \in S_i} (x_j - \mu_i)^2$$

where μ_i is the mean of all the points in cluster S_i. The algorithm proceeds as follows (I'll spare you the formality of the exhaustive notation):

1. Partition the data at random into k sets.
2. Calculate the centroid of each set.
3. Assign each point to the set corresponding to the closest centroid.
4. Repeat the last two steps until nothing is moved around, or until some maximum number of iterations has been reached.

R provides Lloyd's algorithm as an option to `kmeans()`; the default algorithm, by Hartigan and Wong (1979) is much smarter. Like MacQueen's algorithm (MacQueen, 1967), it updates the centroids any time a point is moved; it also makes clever (time-saving) choices in checking for the closest cluster.

There is a problem with **R**'s implementation, however, and the problem arises when considering multiple starting points. I should note that it's generally prudent to consider several different starting points, because the algorithm is guaranteed to converge, but is not guaranteed to coverge to a global optima. This is particularly true for large, high-dimensional problems. I'll start with a simple example (large, not particularly difficult).

2.2 The problem

Let's simulate a data set in $\mathbb{R}^2$ with 4 clusters and then do a cluster analysis via `kmeans()` with 3 randomly selected starting points using Lloyd's algorithm:

```
> N <- 1e+05
> x <- matrix(0, N, 2)
> x[seq(1, N, by = 4), ] <- rnorm(N/2)
> x[seq(2, N, by = 4), ] <- rnorm(N/2, 3, 1)
> x[seq(3, N, by = 4), ] <- rnorm(N/2, -3, 1)
> x[seq(4, N, by = 4), 1] <- rnorm(N/4, 2, 1)
> x[seq(4, N, by = 4), 2] <- rnorm(N/4, -2.5, 1)
> start.kmeans <- proc.time()[3]
> ans.kmeans <- kmeans(x, 4, nstart = 3, iter.max = 10, algorithm = "Lloyd")
> ans.kmeans$centers

        [,1]        [,2]
1  3.0091228  3.01091520
2 -3.0068832 -3.00651790
3  2.0376395 -2.53674463
4 -0.0347932  0.03792785

> end.kmeans <- proc.time()[3]
> end.kmeans - start.kmeans

elapsed
  2.787
```

Figure 1 shows a scatterplot of 10,000 randomly selected points and superimposes the estimated cluster centers from the analysis above. Here, the time consumed is 2.787 seconds (note the *Sweave* code for the previous number in the text). I intentionally limit the algorithm to 10 iterations for reasons that will soon become evident.

We might reasonably want to achieve a speed gain by taking advantage of parallel computing tools:

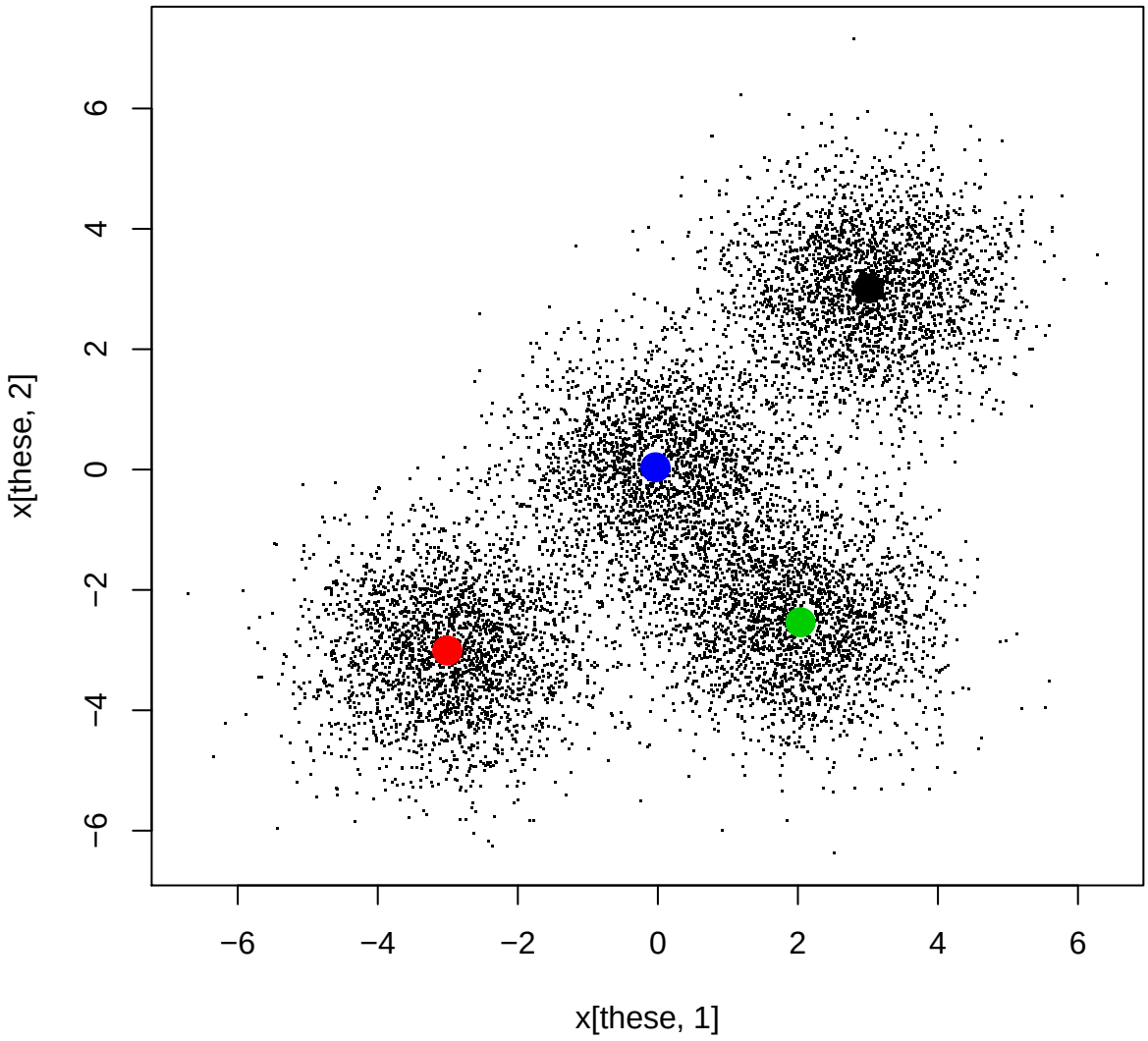

Figure 1: Simulated data for `kmeans()` problem.

```
> library(foreach)
> library(doMC)
> registerDoMC(3)
> start.kmeans <- proc.time()[3]
> ans.kmeans.par <- foreach(i = 1:3) %dopar% {
+     return(kmeans(x, 4, nstart = 1, iter.max = 10, algorithm = "Lloyd"))
+ }
> TSS <- sapply(ans.kmeans.par, function(a) return(sum(a$withinss)))
> ans.kmeans.par <- ans.kmeans.par[[which.min(TSS)]]
> ans.kmeans.par$centers

         [,1]        [,2]
1 -3.00688322 -3.00651790
2  2.03765082 -2.53682040
3  3.00912279  3.01091520
4 -0.03472168  0.03790064

> end.kmeans <- proc.time()[3]
> end.kmeans - start.kmeans

elapsed
  0.199
```

Note that the solution is very similar to the one achieved earlier, although the ordering of the clusters is arbitrary. More importantly, the job only took 0.199 seconds in parallel! Surely this is too good to be true: using 3 processor cores should, at best, taken one third of the time of our first (sequential) run. Is this a problem? It sounds like free lunch. There is no problem with a free lunch once in a while, is there?

2.3 Debugging

Fire up **R**. Try this (I truncate my output):

```
> kmeans

function (x, centers, iter.max = 10, nstart = 1, algorithm = c("Hartigan-Wong",
    "Lloyd", "Forgy", "MacQueen"))
{
    do_one <- function(nmeth) {
        Z <- switch(nmeth, {
            Z <- .Fortran("kmns", as.double(x), as.integer(m),
...
```

This doesn't always work with **R** functions, but sometimes we have a chance to look directly at the code. This is one of those times. I'm going to put this code into a file, `mykmeans.R`, and edit it by hand, inserting `cat()` statements in various places. Here's a clever way to do this, using `sink()` (although this doesn't seem to work in `Sweave`, it will work interactively):

```
> sink("mykmeans.R")
> kmeans
> sink()
```

Now I'll edit the file, changing the function name and adding `cat()` statements. Note that you also have to delete a trailing line: `<environment: namespace:stats>`. After my additions, here's a sample of what part of this file might look like:

```
mykmeans <- function (x, centers, iter.max = 10, nstart = 1,
                      algorithm = c("Hartigan-Wong", "Lloyd",
                                    "Forgy", "MacQueen"))
{
cat("JJJ statement 1: 0 elapsed time.\n")
base <- proc.time()[3]
...
cat("JJJ statement 5:", proc.time()[3]-base, "elapsed time.\n")
    if (nstart >= 2 && !is.null(cn)) {
        best <- sum(Z$wss)
        for (i in 2:nstart) {
            centers <- cn[sample.int(mm, k), , drop = FALSE]
cat("JJJ statement 6:", proc.time()[3]-base, "elapsed time.\n")
            ZZ <- do_one(nmeth)
cat("JJJ statement 7:", proc.time()[3]-base, "elapsed time.\n")
            if ((z <- sum(ZZ$wss)) < best) {
                Z <- ZZ
                best <- z
            }
        }
    }
...
```

We can then repeat our explorations, but using `mykmeans()`:

```
> source("mykmeans.R")
> start.kmeans <- proc.time()[3]
> ans.kmeans <- mykmeans(x, 4, nstart = 3, iter.max = 10, algorithm = "Lloyd")

JJJ statement 1: 0 elapsed time.
JJJ statement 5: 2.424 elapsed time.
JJJ statement 6: 2.425 elapsed time.
JJJ statement 7: 2.52 elapsed time.
JJJ statement 6: 2.52 elapsed time.
JJJ statement 7: 2.563 elapsed time.

> ans.kmeans$centers
```

```
        [,1]        [,2]
1 -0.0347932  0.03792785
2  3.0091228  3.01091520
3  2.0376395 -2.53674463
4 -3.0068832 -3.00651790

> end.kmeans <- proc.time()[3]
> end.kmeans - start.kmeans

elapsed
  2.566
```

Now we're in business: most of the time was consumed before statement 5 (I knew this of course, which is why statement 5 was 5 rather than 2). So let's add a few more statements to figure out where the time is being spent. We'll do this together in the Meetup.

3 Reshaping Olympic diving scores

Package *YaleToolkit* contains a function `whatis()` that is my preferred tool for preliminary examination of data frames. The output is a little wide for this document, but you'll get the idea:

```
> x <- read.csv("Diving2000.csv", header = TRUE, as.is = TRUE)
> library(YaleToolkit)
> whatis(x)

   variable.name      type missing distinct.values precision
1          Event character       0               4        NA
2          Round character       0               3        NA
3          Diver character       0             156        NA
4        Country character       0              42        NA
5           Rank   numeric       0              49       1.0
6         DiveNo   numeric       0               6       1.0
7     Difficulty   numeric       0              20       0.1
8         JScore   numeric       0              21       0.1
9          Judge character       0              25        NA
10      JCountry character       0              21        NA
                  min           max
1             M10mPF         W3mSB
2              Final          Semi
3  ABALLI Jesus-Iory ZHUPINA Olena
4                ARG           ZIM
5                  1            49
6                  1             6
7                1.5           3.8
```

```
8                   0            10
9          ALT Walter  ZAITSEV Oleg
10                AUS           ZIM
```

We can also look at the head of the data set (selected columns):

```
> x[1:14, c(3, 6:9)]
```

```
      Diver DiveNo Difficulty JScore                   Judge
1  XIONG Ni      1        3.1    8.0 RUIZ-PEDREGUERA Rolando
2  XIONG Ni      1        3.1    9.0             GEAR Dennis
3  XIONG Ni      1        3.1    8.5           BOYS Beverley
4  XIONG Ni      1        3.1    8.5           JOHNSON Bente
5  XIONG Ni      1        3.1    8.5         BOUSSARD Michel
6  XIONG Ni      1        3.1    8.5          CALDERON Felix
7  XIONG Ni      1        3.1    8.5              CRUZ Julia
8  XIONG Ni      2        3.0    8.5 RUIZ-PEDREGUERA Rolando
9  XIONG Ni      2        3.0    8.0             GEAR Dennis
10 XIONG Ni      2        3.0    8.0           BOYS Beverley
11 XIONG Ni      2        3.0    9.0           JOHNSON Bente
12 XIONG Ni      2        3.0    8.0         BOUSSARD Michel
13 XIONG Ni      2        3.0    8.0          CALDERON Felix
14 XIONG Ni      2        3.0    8.0              CRUZ Julia
```

The challenge: create a new column containing the average scores of the panel for each dive. Attempt 1:

```
> meancol <- function(scores) {
+     temp <- matrix(scores, length(scores)/7, ncol = 7)
+     means <- apply(temp, 1, mean)
+     ans <- rep(means, 7)
+     return(ans)
+ }
> x$panelmean <- meancol(x$JScore)
```

Did it work?

```
> x[1:14, c(3, 6:9, 11)]
```

```
      Diver DiveNo Difficulty JScore                   Judge panelmean
1  XIONG Ni      1        3.1    8.0 RUIZ-PEDREGUERA Rolando  6.285714
2  XIONG Ni      1        3.1    9.0             GEAR Dennis  6.571429
3  XIONG Ni      1        3.1    8.5           BOYS Beverley  7.071429
4  XIONG Ni      1        3.1    8.5           JOHNSON Bente  7.214286
5  XIONG Ni      1        3.1    8.5         BOUSSARD Michel  7.214286
6  XIONG Ni      1        3.1    8.5          CALDERON Felix  7.000000
7  XIONG Ni      1        3.1    8.5              CRUZ Julia  6.928571
```

```
8  XIONG Ni      2        3.0    8.5 RUIZ-PEDREGUERA Rolando  6.642857
9  XIONG Ni      2        3.0    8.0              GEAR Dennis  6.714286
10 XIONG Ni      2        3.0    8.0            BOYS Beverley  6.857143
11 XIONG Ni      2        3.0    9.0            JOHNSON Bente  6.714286
12 XIONG Ni      2        3.0    8.0          BOUSSARD Michel  7.142857
13 XIONG Ni      2        3.0    8.0           CALDERON Felix  6.785714
14 XIONG Ni      2        3.0    8.0               CRUZ Julia  6.928571
```

Nope! We should have the mean scores repeated 7 times for all of the lines for each dive, and we clearly don't have that. Thus, let's try `browser()`:

```
> meancol <- function(scores) {
+     browser()
+     temp <- matrix(scores, length(scores)/7, ncol = 7)
+     means <- apply(temp, 1, mean)
+     ans <- rep(means, 7)
+     return(ans)
+ }
```

We'll have to play with this together in the Meetup, it isn't really conducive to displaying in a document like this. However, with this new `browser()` line in the function, it gives us the ability to step through the code a line at a time to see if we believe what we have. Here's a snapshot of what happens, though:

```
> x$panelmean <- meancol(x$JScore)

Called from: meancol(x$JScore)

Browse[1]> n

debug: temp <- matrix(scores, length(scores)/7, ncol = 7)

Browse[2]> n

debug: means <- apply(temp, 1, mean)

Browse[2]> ls()

[1] "scores" "temp"

Browse[2]> head(temp)

     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]  8.0  5.5  7.0  6.0  7.5  3.0    7
[2,]  9.0  6.0  7.0  7.0  7.5  2.5    7
[3,]  8.5  7.0  7.0  6.5  7.0  6.5    7
[4,]  8.5  7.5  7.0  7.0  7.0  6.5    7
[5,]  8.5  7.0  7.5  7.0  7.5  6.0    7
[6,]  8.5  7.0  7.0  6.5  7.0  6.0    7
```

```
Browse[2]> scores[1:14]

 [1] 8.0 9.0 8.5 8.5 8.5 8.5 8.5 8.5 8.0 8.0 9.0 8.0 8.0 8.0
```

At this point, we remember that `matrix()` fills in columns, not rows, and here we wanted to fill in row-by-row. So we fix it up and go back to work. We'll finish this in the Meetup.

4 Working with streaming video

I don't really think I'll get here – it's more of a teaser. And I really can't show this in a static document, so you're out of luck. Email me if you're interested. I'll be speaking on this more formally at the Interface conference in Seattle in a few weeks, and then again at UseR in mid-July.

Figure 2: A first video capture

References

[1] John W. Emerson and Walton Green (2007), *YaleToolkit*: Data exploration tools from Yale University. **R** package version 3.1. URL `http://CRAN.R-project.org/package=YaleToolkit`.

[2] J. A. Hartigan and M. A. Wong (1979), "A K-means clustering algorithm." *Applied Statistics*, Vol. 28, 100-108.

[3] Michael J. Kane and John W. Emerson (2010), *bigmemory*: Manage massive matrices with support for shared memory and memory-mapped files. **R** package version 4.2.3. URL `http://CRAN.R-project.org/package=bigmemory`.

[4] J. MacQueen (1967), "Some methods for classification and analysis of multivariate observations." In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, eds L. M. Le Cam & J. Neyman, Vol. 1, pp. 281-297. Berkeley, CA: University of California Press.

[5] S. P. Lloyd (1957), "Least squares quantization in PCM." Technical Note, Bell Laboratories. Published in 1982 in *IEEE Transactions on Information Theory* Vol. 28, 128-137.

[6] R Development Core Team (2009), *R: language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL `http://www.r-project.org/`.

[7] Simon Urbanek (2009), *multicore*: Parallel processing of **R** code on machines with multiple cores or CPUs. **R** package version 0.1-3. URL `http://CRAN.R-project.org/package=multicore`.

[8] Steve Weston and REvolution Computing (2009), *foreach*: Foreach looping construct for **R**. **R** package version 1.3.0. URL `http://CRAN.R-project.org/package=foreach`.

[9] Steve Weston and REvolution Computing (2009), *doMC*: Foreach parallel adaptor for the *multicore* package. **R** package version 1.2.0. URL `http://CRAN.R-project.org/package=doMC`.