REVOLUTION ANALYTICS

Applications - Parallel Clustering

Revolution Analytics

Outline

Cluster Analysis in R

- k-means is a collection of clustering algorithms which partition a set of points into “k” groups
- The `kmeans()` function in R will pick `k` initial cluster centers at random, then assign points to each cluster so that the sums of squares is minimized.
- The function `kmeans()` is part of the `stats` package.
- Shows how to parallelize a complex function

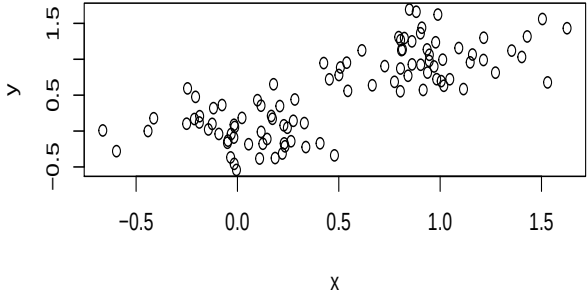
A Small Data Set Example

- We want to identify 5 clusters of points from this small 2-dimensional data set, `X`:

```
# Create artificial data
set.seed(1)
X <- rbind(matrix(rnorm(100, mean = 0, sd = 0.3), ncol = 2), matrix(rnorm(100,
  mean = 1, sd = 0.3), ncol = 2))
colnames(X) <- c("x", "y")
```

Plot

```
plot(X)
```



Cluster analysis with k-means

- From R, we call :

```
# Simple code for clustering
kmeans(X, 5)
```

Cluster analysis with k-means

```
# Create function to plot k-means clustering solution
clusterPlot <- function(x, n = 5, nstart = 1) {
  cl <- kmeans(x, n, nstart = nstart)
  plot(x, col = cl$cluster)
  points(cl$centers, col = 1:n, pch = 8, cex = 2)
}
```

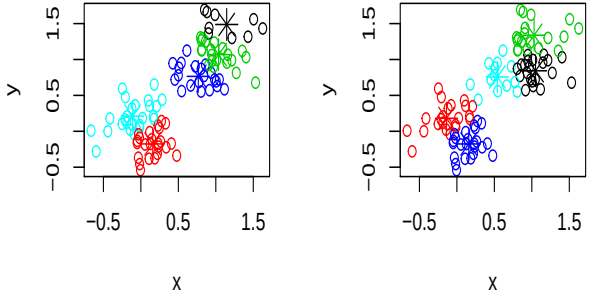
Outline

Two different 5-cluster solutions

- Five arbitrary cluster center points are set, then adjusted iteratively to minimize the total sum of squares of points from center points.
- But we may get different clustering solutions (i.e., different local sums of squares minimums) with different initial center points, as shown next

Clusters vary with starting points

```
par(mfrow = c(1, 2))
clusterPlot(X)
clusterPlot(X)
```



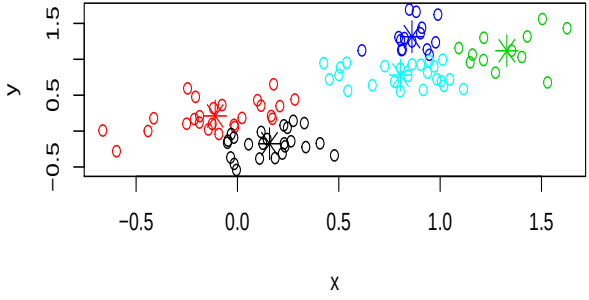

```
par(mfrow = c(1, 1))
```

The Best of 25

- The `kmeans()` function will optionally repeat the procedure, each time randomly selecting initial center points.
- It then computes the minimum sums of squares over each of the procedures, determining the global optimum from the calculated local optimums.
- This graph shows the result of `kmeans(X, 5, nstart=25)` which gives the best solution from 25 potentially different sets of starting values.

The Best of 25

```
clusterPlot(X, n = 5, nstart = 25)
```



Outline

Application for Parallel Computing

Cluster analysis is a good application for parallel computing because:

- For complex problems, you need to calculate k-means for many different starting points for cluster centers
- For large problems, computation of each cluster solution can be time consuming
- Each of the clustering solutions can be calculated independently

Steps for Parallel Computation

- Pass the data to a specified number of computing resources just once.
- Split the work into smaller tasks for passing to each computing resource.
- Combine the results from all the computing resources so the best result is returned.

An Example with `rxExec()`

- Suppose we have a large array Z and we want to classify 10 clusters.
- Because it is a complex problem, we would like to run the clustering procedure 400 times, allowing up to 35 iterations each time, in order to establish a global optimum.

Slow on a Single Processor

- We could do this analysis in R by calling:

```
kmeans(Z, 10, iter.max = 35, nstart = 400)
```

- but it would be slow doing all the calculations on just one processor.

Instead we'll take advantage of all available processors by using `rxExec()`.

Step 1: Create a Master Process

- With `rxExec()`, you can use a variety of sequential or parallel backends.
- We have mostly been using the local parallel compute context, or an explicit parallel backend using `doParallel`.
- You can think of `rxExec()` as the “master process” and the parallel back end as the “workers” - the compute context, however established, is the glue that binds the master to the workers.

```
rxSetComputeContext(RxLocalParallel())
```

Step 2: Define Worker Data and Tasks

- In this example, instead of doing 400 `kmeans()` starting values on one processor, we will do 50 on each of eight processors
- The following call to `rxExec()` does this:

```
numTimes <- 8
results <- rxExec(kmeans, X, centers = 5, iter.max = 35, nstart = 50,
  timesToRun = numTimes)
```

Step 3: Combine the Results

- Now each of the workers has produced information on the "best" 5-cluster set for their sets of starting cluster center points. The returned results object is a list containing these `kmeans` objects.
- Each `kmeans` objects has a `withinss` component which gives the within-cluster sum of squares for each cluster.
- If we calculate the total sums of squares for each of the cluster sets, we know that the smaller will give us our final globally-optimal result.

Step 3 (continued)

```
(sumSSW <- vapply(results, function(x) sum(x$withinss), FUN.VALUE = numeric(1)))
```

```
## [1] 7.062323 7.062323 7.062323 7.062323 7.062323 7.062323 7.062323 7.062323
```

```
results[[which.min(sumSSW)]]
```

```
## K-means clustering with 5 clusters of sizes 25, 12, 24, 24, 15
##
## Cluster means:
##            x          y
## 1 -0.1096832  0.2106891
## 2  1.3290081  1.1185534
## 3  0.1581362 -0.1761590
...
```

Creating rxExec kmeans

- Based on this example, a parallel version of `kmeans()` with all of its functionality can be created.
- It can be used in place of `kmeans()` for faster processing, with the user specifying the number of processors.
- It combines the parallel call to `rxExec()` with the calculation of the best, as shown on the next slide.

kmeansRSR

```
kMeansRSR <- function(x, centers = 5, iter.max = 10, nstart = 1, numTimes = 20) {
  results <- rxExec(FUN = kmeans, x = x, centers = centers, iter.max = iter.max,
    nstart = nstart, elemType = "cores", timesToRun = numTimes)
  sumSSW <- vapply(results, function(x) sum(x$withinss), FUN.VALUE = numeric(1))
  results[[which.min(sumSSW)]]
}
```

Compare kmeans and kMeansRSR

- To compare the speeds of the serial and parallel functions, we can create a sample data set and time the results:

```
# Create 5000 x 50 matrix
nrow <- 5000
ncol <- 50
Z <- matrix(rnorm(nrow * ncol), nrow, ncol)
iter.max <- 35
workers <- 8
```

Time them…

```
nstart <- 800
# Time kmeans
(km1st <- system.time(km1 <- kmeans(Z, 10, iter.max, nstart))[[3]])
```

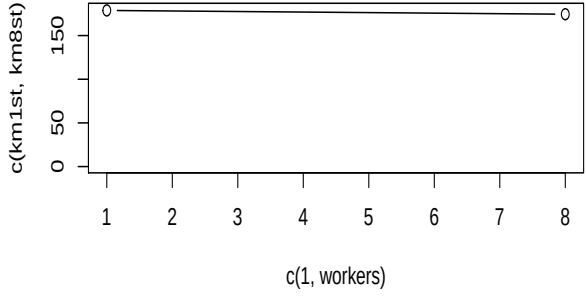
```
## [1] 178.782
```

```
# Time kmeansRSR
(km8st <- system.time(kmrsr <- kMeansRSR(Z, 10, iter.max, nstart = nstart/(2 *
  workers), numTimes = 2 * workers))[[3]])
```

```
## [1] 174.434
```

kmeans Performance Results

```
plot(x = c(1, workers), y = c(km1st, km8st), type = "b", ylim = c(0,
  max(km1st, km8st) + 1))
```



Session Summary

- This presentation has shown an example of how you can write your own faster, parallelized routines using existing R functions and `rxExec()` and // or `foreach()`.

Other Parallelization Strategies

- DoRedis – fault tolerant system that can dynamically allocate workers
- R+Hadoop
- `RHIPE` // `RHadoop` – talk Hadoop without leaving R
- `Segue` – seamlessly talk with Amazon EMR

Course Bibliography

- Revolution Analytics *RevoScaleR Distributed Computing Guide*.
- R vignettes:
  - Weston, S. *Using The foreach Package*
  - Calaway, R. *Using The iterators Package*
- Das, S. and Granger, B. *Financial Application with Parallel R*, Journal of Investment Management, 7(4)
- Schmidberger, M, et.al, (2009). *State of the Art in Parallel Computing in R*, Journal of Statistical Software. 31(1)
- Q. Ethan McCallum and Stephen Weston. (2011) *Parallel R - Data Analysis in the Distributed World*, O'Reilly Media

Bibliography (Continued)

- John Fox (2002), *Bootstrapping Regression Models*, Appendix to An R and S-PLUS Companion to Applied Regression
- Davison, A.C. & D.V. Hinkley (1997). *Bootstrap Methods and their Application*. Cambridge: Cambridge University Press
- Efron, B. & R.J. Tibshirani (1993). *An Introduction to the Bootstrap*. New York: Chapman and Hall.

Questions?

REVOLUTION ANALYTICS

Thank you

Revolution Analytics is the leading commercial provider of software and support for the popular open source R statistics language.

www.revolutionanalytics.com
1.855.GET.REVO
Twitter: @RevolutionR