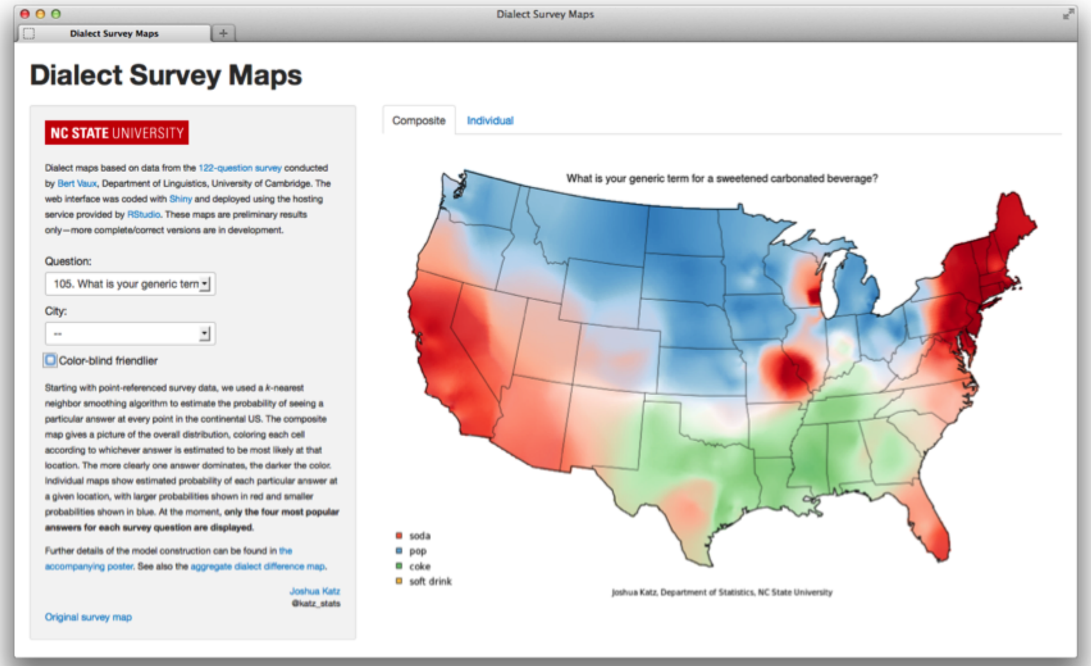
Dialect Survey Maps

NC STATE UNIVERSITY

Dialect maps based on data from the 122-question survey conducted by Bert Vaux, Department of Linguistics, University of Cambridge. The web interface was coded with Shiny and deployed using the hosting service provided by RStudio. These maps are preliminary results only—more complete/correct versions are in development.

Question: 105. What is your generic term

City: --

Color-blind friendlier

Starting with point-referenced survey data, we used a *k*-nearest neighbor smoothing algorithm to estimate the probability of seeing a particular answer at every point in the continental US. The composite map gives a picture of the overall distribution, coloring each cell according to whichever answer is estimated to be most likely at that location. The more clearly one answer dominates, the darker the color. Individual maps show estimated probability of each particular answer at a given location, with larger probabilities shown in red and smaller probabilities shown in blue. At the moment, **only the four most popular answers for each survey question are displayed.**

Further details of the model construction can be found in the accompanying poster. See also the aggregate dialect difference map.

Joshua Katz
@katz_stats

Original survey map

What is your generic term for a sweetened carbonated beverage?

Figure 1: `http://spark.rstudio.com/jkatz/SurveyMaps/`

Amazon EC2, Big Data and High-Performance Computing

Overview

2013 has been an exciting year for the field of Statistics and Big Data, with the release of the new R version 3.0.0. We discuss a few topics in this area, providing toy examples and supporting code for configuring and using Amazon's EC2 Computing Cloud. There are other ways to get the job done, of course. But we found it helpful to build the infrastructure on Amazon from scratch, and hope others might find it useful, too.

Introduction

The term "recent advances" should be placed in context. Most of the fundamental computer science research beneath the technologies discussed here took place long ago. Still, innovation and software development of specific interest to statisticians and data scientists is one of the most important and impactful areas of R&D today. Let's say it together: "Yes, we are sexy!"

This note offers a high-level introduction to some of the recent changes of the R software environment (R Core Team, 2013b) as of versions $\geq 3.0.0$ relating to high-performance computing. Specifically, updated indexing of vectors addresses a substantial size limitation on native R objects under versions $\leq 2.15.3$. Native R objects are still limited to available memory (RAM), however, and many Big Data problems demand memory exceeding RAM on even the best-equipped modern hardware. To help address this problem, we very briefly discuss package **bigmemory** (Kane and Emerson, 2013). Finally, we present the elegant framework for parallel computing using package **foreach** (Weston and Revolution Analytics, 2012).

Toy code examples are provided and were run on Amazon's Elastic Compute Cloud (EC2) running Ubuntu Linux. This isn't necessary, of course, so why do it? Because EC2 is relatively easy to use and scalable. Within a matter of minutes, anyone can request and create a cluster of instances that communicate with each other with low latency. A basic "how-to" is provided as supplementary material available online.

R Version 3.0.0 and Big Data

The ability to handle vectors of length greater than 2^{31} elements is arguably the most significant improvement provided by R versions $\geq$ 3.0.0. R versions $\leq$ 2.15.3 were unable to create such long vectors, as shown in the first code example:

```
> x <- raw(2^31 - 1)

> length(x)
[1] 2147483647
> object.size(x)       # Just over 2 GB
2147483688 bytes

> x <- raw(2^31)
Error in raw(2^31) : vector size specified is too large
```

In contrast, given sufficient RAM, R versions $\geq$ 3.0.0 have no trouble with this same example:

```
> x <- raw(2^31)

> length(x)
[1] 2147483648
```

The introduction of long vectors clearly has made R a more appealing choice for tackling Big Data problems, but there are still important qualifications. Although we technically have the ability to create large objects in R, we are practically limited by available RAM (about 23 GB on our EC2 instance):

```
> x <- raw(2^36)       # This would be about 64 GB
Error in raw(2^36) : vector size specified is too large
```

Unfortunately, there are still pitfalls even if an R object fits in available RAM. As discussed in Emerson and Kane (2012) and elsewhere, most non-trivial operations require the creation of multiple copies of objects, causing memory overhead (even if transient). When a single copy occupies a substantial proportion of RAM, the memory overhead rapidly becomes prohibitive. We provide a simple example, given in R Core Team (2013a), of copying overhead associated from the simple act of increasing the length of a long vector by one element. Here, although the object `x` occupies about 2 GB (2050.2 MB) of RAM, adding one element to the vector triggers the creation of a temporary copy; the peak memory usage during this operation has doubled to over 4 GB. An aside: `gc()` is R's "garbage collector" and can help study and manage some aspects of memory consumption.

```
> x <- raw(2^31 - 1)

> gc(reset=TRUE)[2,6]
2050.2
> x[2^31] <- as.raw(1)
> gc()[2,6]
4099.4
```

In contrast, modifying any one of the existing 2^{31} entries of this new vector x does not create extra copies of the vector:

```
> gc(reset=TRUE)[2,6]
2050.2
> x[2^31 - 1] <- as.raw(2)
> gc()[2,6]
2052.6
```

The memory overhead associated with almost any non-trivial operation is virtually unavoidable with native R objects. R Core recommends working with data sets that occupy at most 10-20% of available RAM in order to avoid such difficulties. Use of databases or other alternatives can help insulate the user from such problems, at least for the purpose of data management and basic manipulations. For an expanded discussion of these options, see Kane, Emerson, and Weston (2013) or an excellent page on CRAN: http://cran.r-project.org/web/views/HighPerformanceComputing.html.

In situations where the dataset is too large or the computations are too intensive, we may still need other options to be able to work efficiently in R. We will now discuss some of these options – **bigmemory**, **foreach**, and Amazon's EC2 – and demonstrate their roles in working with Big Data.

Big Data via bigmemory

The **bigmemory** family of packages enables the creation of matrices that exceed available RAM and, in fact, can be as large as the available hard-drive space. Here, we demonstrate the creation of a 100 GB matrix of single-byte "char" integers. All examples (except for one specifically noted) in this section run immediately without any lags. We use very basic toy examples here.

```
> library(bigmemory)
> N <- 10^9             # A billion rows
> K <- 100              # A hundred columns
> x <- big.matrix(N, K, type="char",
+                 backingfile="big.bin",
+                 descriptorfile="big.desc")
```

The created object x belongs to the big.matrix class. It has an associated memory-mapped "backing" file, big.bin, which resides on the hard drive and persists even after the R session is closed. The object can then be loaded back into R upon relaunch using the attach.big.matrix function. In the terminal, we can see the filebacking for the created object as well as the associated "descriptor" file, big.desc:

```
ubuntu@ip-10-170-20-92:/mnt/test$ ls -als
total 12
4 drwxr-xr-x 2 ubuntu ubuntu         4096 May 27 15:42 .
4 drwxr-xr-x 4 root   root           4096 May 27 15:32 ..
0 -rw-rw-r-- 1 ubuntu ubuntu 100000000000 May 27 15:42 big.bin
4 -rw-rw-r-- 1 ubuntu ubuntu          461 May 27 15:42 big.desc
```

We can now run some basic substitution and extraction operations on x as we would with a regular R object. The operating system manages available RAM and actively-used portions of the memory-mapped file with amazing speed and efficiency.

```
> dim(x)
[1] 1e+09 1e+02

> options(bigmemory.typecast.warning=FALSE)   # Avoids a warning message
> x[1:2, 1:2] <- 1:4         # A trivial assignment with regular matrix syntax
> x[, ncol(x)] <- 1          # Another assignment (takes 2 seconds, more work!)
> x[c(1:2, nrow(x)), c(1:2, ncol(x))]       # A 3x3 R matrix is returned
     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    1
[3,]    0    0    1
```

In general, R functions that operate on matrices will not work on a `big.matrix`, but it is easy to extract subsets into a native R object in RAM as done in the example above. Packages **biganalytics**, **bigtabulate**, **bigalgebra**, and **synchronicity** offer more advanced functionality (references omitted for brevity, but available online), including k-means clustering, linear and generalized linear models, and more. A further advantage of the `big.matrix` data structure is its support for shared memory, which naturally lends itself to parallel computing (Kane, Emerson, and Weston, 2013). The following section discusses the **foreach** package which can be used in conjunction with **bigmemory** for parallel computing.

Parallel Programming via foreach

If multiple processor cores are at your disposal, then a large computational task might be completed more efficiently by making use of parallel computing. The **foreach** package extends the capabilities of standard looping constructs by delegating subtasks to multiple cores, if available, and collating results as work is returned. The framework provides the flexibility of using a variety of parallel transport mechanisms (multicore, snow, MPI, . . .) without requiring code modification. Details are provided in Kane, Emerson, and Weston (2013).

We demonstrate the use of **foreach** on an EC2 cluster with a small example. We use the **doSNOW** library (Revolution Analytics, 2012) to manage the cluster from R. Our goal is to compute the column sums of a 4 by 250,000 matrix filled with integers ranging from 1 to 1,000,000.

```
>  x <- matrix(1:1e6, nrow=4)
```

Our first attempt uses one 4-core EC2 Quadruple Cluster Compute instance, taking 821 seconds:

```
> library(doSNOW)
> machines <- rep("localhost", each = 4)
> cl <- makeCluster(machines, type = "SOCK")
> registerDoSNOW(cl)
> system.time({
+   y <- foreach(j=1:ncol(x), .combine=c) %dopar% { return(sum(x[,j])) }
+ })

   user  system elapsed
249.395  18.929 820.651

> stopCluster(cl)
```

Note that this may be slower than running the same code using 4 cores on your own machine, because Amazon's compute nodes are far from being state-of-the-art. However, Amazon's strength lies in its scalability. With minimal effort, we can request a second Quadruple Cluster Compute instance, designate this second instance as a slave, and run the same code to take advantage of the 4 cores available in the master instance as well as the 4 cores available on the slave instance (identified in `/vol/nodelist` in our example):

```
ubuntu@ip-10-170-20-92:/mnt/test$ cat /vol/nodelist
ec2-174-129-178-209.compute-1.amazonaws.com
```

Back in R:

```
> machines <- readLines("/vol/nodelist")  # Get our slave node information
> machines <- rep(c("localhost", machines), each = 4)
> cl <- makeCluster(machines, type = "SOCK")
> registerDoSNOW(cl)
> system.time({
+   y <- foreach(j=1:ncol(x), .combine=c) %dopar% { return(sum(x[,j])) }
+ })

   user  system elapsed
513.124  21.849 668.685
```

Although we have doubled our computing resources, we have not halved the runtime (reduced only from 821 seconds to 669 seconds). The amount of speed-up is limited by the overhead of communication between machines over a network.

These two approaches shown above are far from optimal – parallel programming isn't trivial and doesn't always provide speed gains! R's native `apply` function (which uses a sequential loop on one core) easily beats both of the above solutions, requiring only 1.3 seconds.

```
> system.time({
+   y <- apply(x, 2, sum)
+ })

   user  system elapsed
  1.308   0.000   1.304
```

In the parallel `foreach` loop above, the entire computation is divided into 250,000 trivial subtasks which are then assigned to the available processor cores. The resulting communication overhead causes the computation to be extremely slow – far slower than the sequential solution.

Fortunately, we can further reduce the runtime by making use of the **itertools** package (Weston and Wickham, 2010), which offers a smarter way to delegate subtasks to individual cores to minimize the communication overhead. Combining **itertools** with **foreach** enables us to take full advantage of the available computing resources. Again working with all 8 cores, we can now beat the sequential `apply` solution, cutting the runtime down to below one second. Instead of creating 250,000 subtasks, **itertools** recognizes that there are 8 cores and so creates 8 subtasks for efficient execution and reduced communication overhead:

```
> system.time({
+   y <- foreach(j=isplitIndices(ncol(x), chunks=length(machines)),
+                .combine=c) %dopar% { return(sum(x[,j])) }
+ })

   user  system elapsed
  0.120   0.072   0.878
```

Amazon's EC2 is highly scalable in that we can request as many instances as needed, paying by the machine-hour. In this way, an iterative task that would require days to finish via a sequential loop can be rushed to completion by increasing the number of instances in the cluster.

Finally, we note that it is easy to create shared file systems on EC2 instances. Thus, if a task requires operations on a large data set (or produces results that need to be combined into a large matrix), the memory-mapped files of **bigmemory** can be stored on the shared disk space and accessed simultaneously by each of the worker processes without incurring the cost of copies. We can then make use of **foreach** and the shared-memory capabilities inherent in **bigmemory** to work with the data from any instance in the cluster. A more in-depth discussion of combining **foreach** with **bigmemory** is given in Kane, Emerson, and Weston (2013).

Jay Emerson and Xiaofei Wang
Yale University

Supplementary Material

We provide a friendly "how-to" on setting up Amazon EC2 instances and computing clusters. All examples shown in this paper were run on Amazon EC2 instances following this procedure. For more information on this and materials covered in this paper, please visit `http://www.stat.yale.edu/~jay/EC2`. We'll try to keep it updated. It currently installs both **shiny** and **FastRWeb**; these packages provide interactive web applications and CGI scripting with R (references omitted here). Topics for another day!

References

1. Emerson, J. W. and Kane, M. J. (2012). Don't drown in the data, *Significance*, **9**, 38–39.
2. Kane, M. J. and Emerson, J. W. (2013). **bigmemory**: *Manage Massive Matrices with Shared Memory and Memory-Mapped Files*, R package version 4.4.3,
   Available from: http://CRAN.R-project.org/package=bigmemory
3. Kane, M. J., Emerson, J. W., and Weston S. B. (2013, to appear). Scalable Strategies for Computing with Massive Data, *Journal of Statistical Software*.
4. R Core Team (2013). Changes in R 3.0.0, *R News*,
   Available from: http://cran.r-project.org/src/base/NEWS
5. R Core Team (2013). **R**: *A Language and Environment for Statistical Computing*,
   Available from: http://www.R-project.org/
6. Revolution Analytics (2012). **doSNOW**: *Foreach parallel adaptor for the snow package*, R package version 1.0.6, Available from: http://CRAN.R-project.org/package=doSNOW
7. Weston, S. B. and Wickham, H. (2010). **itertools**: *Iterator Tools*, R package version 0.1-1, Available from: http://CRAN.R-project.org/package=itertools
8. Weston, S. B. and Revolution Analytics (2012). **foreach**: *Foreach Looping Construct for R*, R package version 1.4.0, Available from: http://CRAN.R-project.org/package=foreach