Efficiency: How to Make R Work Better

Lecture 6

Kevin McAlister

University of Michigan

February 24, 2017

Basics of Computing with R

How Does R Work?

- R is a statistical computing language.
- We assess R by what it can do with data.
  - How much data can I analyze?
  - How fast can I analyze data?
  - How accurate are the algorithms used by R?
- In general, we don't worry about the third item.
- R packages stored in CRAN are generally peer reviewed. We trust them.

How Does R Work?

- What about quantity and speed?
- R is an object oriented vector language that stores objects in working memory (RAM).
- Data stored in RAM is operated on by the CPU.
- Reading and writing to RAM is much faster than read/write to disk.
- Hard storage is not generally utilized by R.
- This can create issues with **BIG** data.
- What is big? In short, anything bigger than your RAM capacity.

How Much RAM does my machine have?

- Take a second to look at how much RAM your machine has. Different OS's require different steps, so look up how to do it on Google if you aren't sure.
- This is the maximum amount of data that can be in working memory at once.
- If we exceed this limit, the machine will grind to a halt.
- Most OS will have a swap partition. This is a buffer against using all your RAM. Reads and writes to disk.
- Needless to say, this is not great.

Computing Challenges

- The computing challenges that we will encounter are generally of two types:
  1. Algorithmic Complexity (Speed)
  2. Reading/Writing (Size)
- The two are not always separate.
- Example. Run a linear regression on 100,000 data points or logistic regression on 100,000 data points.
- Linear regression must store two objects: residuals and coefficients. Linear regression is a two step procedure.
- Logistic regression requires many more objects: Hessian, Coefficients, Residuals, Optimization Error, etc.
- Size of data is the same, but size of output is different. Logistic regression is algorithmically more complex than linear regression.

Efficiency of Code

- Today, we're going to start by thinking about making code more efficient.
- In particular, we're going to think about what happens when we're pushing R to it's limit.
- Big point: There is no magic efficiency gain that occurs by switching high-level languages.
- R and Python both have the same bottleneck issues.
- Python is nowhere near as nice in terms of statistical computing as R.

Why should you care?

- In general, we're not going to be writing new algorithms for handling data.
- Applying algorithms to analyze data relevant to research interests.
- Time is money (or citations).
- As problems get more complex, finite time solutions are key.

Classes and Operations in R

- R is a vector based language.
- Almost every object in R is treated as a vector.
- Each item is stored with a location.
- We reference locations using `[]`.
- There are a number of classes of objects in R.
- These classes are different ways to combine data in meaningful ways.
- Singular items are generally either characters or numeric.
- `class(2)` vs. `class("2")`

Classes and Operations in R

- Every singular object is also stored as a vector. (i.e. if `a <- 2` then `a[1]` equals 2.
- Multiple object can be stored together in a variety of ways:
  1. `c(1,2,3)` stores a vector.
  2. `matrix(c(1,"2",3))` stores a column vector.
  3. `matrix(ncol = 3, nrow = 1, c(1,2,3))` stores a row vector.
  4. 2-dimensional matrices
  5. k-dimensional vectors
- These structures utilize the typical vector notation in R.
- A data frame is just a named matrix. It allows for $ notation.

Classes and Operations in R

- There is a second type which character vectors can take. The characters can be stored as factors, which are numeric representations of the different characters that exist in a vector.
- Factors have their place. But, in general, factors are bad. They require extra storage and are clunky to utilize in algorithms. When possible, just store characters as character vectors.
- Aside from the matrix based storage objects, we can also store objects as lists.
- A list is a collection of objects that are stored in a serial manner.
- `aa <- list(matrix(ncol = 2, nrow = 2, c(1,0,0,1)),c(1,2,3),"Squirrel")`
- `aa[[1]]` returns the 2×2 matrix.
- Lists can also be nested. An element in a list can also be another list. For example, `aa[[1]][[1]]`.

Classes and Operations in R

- When do we use lists instead of matrices?
- You'll know.
- When it doesn't make sense to store things in a rectangular structure.
- When the number of items to store for each "row" is not the same.
- Letting R communicate with other applications (JSON).

Classes and Operations in R

- Given a set of data containers, R performs algorithms related to data analysis.
- At the base, R performs operations on one "piece" of data at a time.
- Numeric operations only utilize a series of sums and multiplication.
- One at a time operations require $O(CN)$ where C is the number of operations on one datum.
- For any operation, there are 3 steps:
  1. Access datum
  2. Operate on datum
  3. Write results of operation
- Operations are generally optimized. Call lower level languages for these operations.
- The read and write steps are generally the computational bottlenecks.

Classes and Operations in R

- One approach to performing operations over a data container is a for loop.

```
for(i in 1:n){
    for(j in 1:p){
        ...
    }
}
```

- For loops explicitly read one item from RAM, operate on it, and write it back to RAM for each element.
- R optimizes this process by batch reading and writing.
- Vectorization
- Sometimes for loops are the only approach (i.e. dependent iterative processes).
- However, most of the time, there are better options.

Apply in R

- R optimizes the read/write steps by utilizing batch read/write.
- Set up the problem in such a way that all the data can be operated on such that there is one big read step and one big write step.
- If we are working with a vector-like data container, apply functions carry out this process.
- Given a k-dimensional vector container, `apply()` is a wrapper that applies a function over a set of elements in the vector container.
- `apply(x,over,fun,...)`
- `x` is the vector container.
- `fun` is the function to apply.
- `over` tells `apply()` over which index to apply the function.

Apply in R

- For example, if we have a $n \times k$ matrix and we want to sum over the rows: `apply(x,1,sum)`
- For k-dimensional arrays, 1 applies to the first dimension, 2 the second, so on and so forth. For matrices, 1 is always the rows and 2 is always the columns.
- In a 3D array, `c(1,2)` applies the function over each row/column pair. Calculates the sum for `[1,1]` across each page of the array.
- Apply beats a for loop any day. The read/write step is optimized for the data structure.

Apply in R

- There are a number of variants of `apply()` in R that have different uses based on data type being passed in and how we want the apply function to pass over the data.
- In my experience, there are two important ones: `tapply()` and `lapply()`.
- `tapply(x,group,fun,...)` is used when we want to evaluate a function over groups within the data set.
- `x` is the data which is operated upon.
- `fun` is the function being applied to the data.
- `group` is the vector that gives the groupings for each of the data points.
- For example:

```
xx <- rnorm(100,0,1)
gg <- sample(c(0,1),100,replace = T)
tapply(xx,gg,mean)
```

- This snippet will return the mean of `xx` where `gg = 0` and `gg = 1`.
- Can be extended to include more complex functions.

Apply in R

- `lapply()` functions like apply. However, it operates on each element in a list.
- `lapply(x,fun,...)`
- `x` is a list.
- `fun` is the function to be applied to each element in the list.

```
xx <- list(1,4,9)
s.root <- lapply(xx,sqrt)
```

- This function will return the square root of each element in the list as a list.
- `s.root[[1]]` returns the square root of 1.

Making Code More Efficient

- Apply functions answer one aspect of making R code more efficient.
- But, what about other parts?
- Reading in data.
- Summarizing many columns of a data set.
- Creating new columns.
- Subsetting data.
- The list goes on and on and on. R does a pretty good job, but there are still a number of inefficiencies that can be corrected by different implementations than base R.

Introduction to data.table

data.table in R

- While R is great for data sets that are relatively small, it can struggle when data sets get large.
- R is limited by working memory, so using this memory efficiently is key to squeezing every ounce of computing power out of R.
- `library(data.table)` is a package developed for R that addresses a number of inefficiencies in the read/write portion of algorithms.
- R reads data in a high level manner. In other words, R doesn't always operate on the binary level. This is good for ease of development, but bad for efficiency of reading and writing.
- `data.table` approaches algorithms on the binary data, allowing the read/write steps to be much faster.
- Syntactically, data.table is a little different than base R. However, the syntax is more in line with other data oriented languages.

Clocking Functions

- One useful way to compare functions is to compare the amount of time needed to complete the function.
- One way to achieve this is to use `system.time(fun())`
- System time tells us how much time is needed to complete whatever is inside the parentheses.
- There are other more fine-tuned options:
  1. `library(microbenchmark)` runs a function many times and gives the distribution of time needed.
  2. `library(profr); library(ggplot2)` profiles R code. It returns system times needed for each constituent call within a function call. Good for understanding bottlenecks within pre-made functions.

Installing data.table

- Let's start by installing the most up to date version of data.table.
- We can install from CRAN, but it is rarely the most up to date version.
- Install from GitHub:

```
install.packages("data.table", type = "source",
    repos = "http://Rdatatable.github.io/data.table")
library(data.table)
```

- Correction (You can use the Github version later): Install from CRAN for now. There is a bug in the most recent release.
`install.packages("data.table")`
- The package is well maintained and often throws warnings talking about code changes.

Fast and Friendly Read

- On Canvas, there is a data set called chicrimes.csv.
- Let's start by using data.table to load it up.
- Native R function `read.csv()`.
- One of the biggest contributions of data.table is that it improves the read time for relatively big files.
- `fread("file.path")`
- This data set is pretty large.
- Let's compare `read.csv()` to `fread()`

```
system.time(fread("file.path"))
system.time(read.csv("file.path"))
```

- `fread()` is much faster. The difference is much more apparent when the data set gets larger.

Fast and Friendly Read

- Why is it so much faster?
- Binary read vs. R base processing
- 98 percent of the time needed to read a file using `read.csv()` is used processing the file to fit into R's data frame format.
- Converting characters to factors.
- Pre-copying the data for manipulations.
- Lazy programming.
- Even if you don't need to utilize the data.table functions, `fread()` should still be a part of your day to day R usage.
- Scans for delimiters, so no need to understand data structure beforehand.

data.table Syntax

- Store the data as an object.

  ```
  dats <- fread("file.path")
  ```

- Another benefit of data.table is that it suppresses print statements.
- Type `dats`. How does this differ from what prints from a data frame?
- Data tables inherit all the characteristics of data frames. Pass a data table into a function that takes a data frame and nothing changes.
- Like a data frame, data tables have rows and columns. Rows are observations and columns are variables. Reference a specific column with $.
- Try referencing a data table with square brackets (i.e. dats[1,2]). What happens? Now try dats[1,3]. See?

data.table Syntax

- data.table has its own syntax. This syntax is more in line with database languages, like SQL.
- data.table excels at functions operations which require subsetting or aggregation.
- Assume that we have a data table names `dats`, then data.table syntax has the following structure:
  `dats[i,j,by]`
  or in SQL terms:
  `dats[WHERE,SELECT,GROUP BY]`
- It may seem unintuitive at first, but most of the data munging that we do in R is related to this type of operation.
- We'll review each argument one by one.

The i argument

- The first argument in the data.table syntax is the WHERE argument. We can also refer to this as the subset argument.
- Given a data.table, dats, we can return the first row by typing:

```
dats[1]
```

- The WHERE argument can also take logical arguments.
- Working with the Chicago crime data, let's subset the data.table to only crimes that were not domestic violence:

```
non.domestic <- dats[Domestic == "false"]
```

- Note a couple of things. First, when we reference column names, we don't have to utilize $. Second, note that we return another data.table with all the columns. This is equivalent to the subset commend in R.

Logical Review

- A quick review of logicals in R:
  - == is equals.
  - >= is greater than or equal. <= is less than or equal.
  - & is used between two logical statements and returns TRUE if all are TRUE.
  - | is used between two logical statements and returns TRUE if any are TRUE.
  - %in% returns true if the element is in the set: 1 %in% c(1,2,3)
- There are also a set of logical wrappers in R:
  - is.na() returns TRUE is the element is NA
  - is.nan() returns TRUE if the element is NaN (An Operation that returns Infinity)
  - is.numeric(), is.character(), etc.

The j argument

- The next argument is the SELECT argument. This is where we tell data.table what to return.
- This argument is very flexible.
- This argument can be used to add or remove columns from the data table (:=). This is much faster than adding columns to data frames because the columns are added by reference. This means that no copy is created in working memory!
- If we do not need to use the i argument, we can just leave it blank (i.e. dats[,j])

The j argument

```
#Add A Column With 1 if Domestic, 0 if Not
is.domestic <- function(x){
  if(x == "true"){
    return(1)
  }else{
    return(0)
  }
}
#Vectorize the function
is.domestic <- Vectorize(is.domestic, "x")
#Add the Column
dats <- dats[,new.col := is.domestic(Domestic)]
#Remove the Column
dats <- dats[,new.col := NULL]
```

Removing Data from R

- If we want to remove data from R, then we need to use the `rm(obj)` function.
- `obj` is the object to remove from R.
- Remove `dats` from your R session.
- For larger data, we need to take one extra step and do garbage collection.
- Following a `rm()` call, we should run `gc()`.
- `gc()` removes any cached data stored behind the scenes.
- `gc()` runs at certain points in your R sessions, but this forces a call and cleans the memory behind your R session.
- This helps with memory management

The j argument

- Let's read in a new data set, bigdats.csv. There should be 2,000,100 rows and 5 columns. The first column is the customer ID, the second is the treatment assignment for experiment 1, the third is treatment assignment for experiment 2, the fourth is amount spent, and the fifth is number of items bought. Each row is a customer day. So, if a customer visited the site on multiple days, then she has multiple rows in this table.
- Use fread to read in the data, `dats <- fread("file.path")`

The j argument

- The j argument can be used to create new summary tables that aggregate over the table.
- For example, lets find the mean and standard deviation of customer day spending for all customers that received treatment 1:

```
spending.dats <- [TREAT1 == 1,
    list(mean.spent = mean(SPENT),
    sd.spent = sd(SPENT))]
spending.dats
```

- Note that we call a list of functions in j. This allows us to call multiple functions at once. We can name the columns of the resulting table by naming the elements of the list.
- The resulting table is returned as another data table.

The by argument

- The by argument is the GROUP BY argument. This allows us to call functions in the j argument by groups. This is analogous to a `tapply()` call.
- Let's calculate mean spending for treatment 1 by treatment 2 assignment:

```
spending.dats <- [,list(mean.spent = mean(SPENT)),
    by = list(TREAT1, TREAT2)]
```

- Like j, by can take a list of grouping variables and will return j for each unique combination of the grouping variables.
- This is one of the biggest benefits of data.table. Fast aggregation and summarizing of large tables.

An example

- The current table's level of observation is customer day. Let's aggregate the table so that each row's level of observation is just customer.
- Even if a customer visits the site, they are unlikely to spend any money. Thus, this data set has a lot of zeroes. Let's also create a new column that indicates whether or not the customer spent any money during the experiment.
- Let's also create a new column that tells us the number of days each customer visited the site over the course of the experiment.

An example

- The j argument has a special argument, `.N`. `.N` is passed into the argument list and returns the number of rows that are involved in each step of aggregation. In this case, it will return the number of times a customer visited the site over the course of the experiment.
- Each customer receives the same treatment assignment each time she visits the website.
- By SID, use data.table to create the aggregate table. For treatments, we can use `mean(TREAT1)`. For spending, use `sum(SPENT)`. Same for N.ITEMS.
- For the new column, you can just use `sign(SPENT)`.

An example

```
new.dats <- dats[,list(TREAT1 = mean(TREAT1),
    TREAT2 = mean(TREAT2),
    SPENT = sum(SPENT),
    N.ITEMS = sum(N.ITEMS),
    N.VISITS = .N),
    by = SID]
new.dats <- new.dats[,DID.SPEND := sign(SPENT)]
tables()
```

- All of that only took two lines of code.
- It was really fast. Imagine how long it would take if we were using base R.

Let's Do Some Analysis

- Using the new table, let's determine if the treatments have an effect on spending metrics.
- We'll run simple z tests since the sample size is so large.
- Start with raw amount spent. Recall that a z test:

$$z = \frac{mean(SPENT_{T=1}) - mean(SPENT_{T=0})}{\sqrt{\frac{var(SPENT_{T=1})}{N_{T=1}} + \frac{var(SPENT_{T=0})}{N_{T=0}}}}$$

- Use data.table to calculate the constituent parts of the z statistic and calculate it. Is the difference significant at the .05 level?
- Now repeat this for the second experiment. Is the difference significant?
- Using data.table to do this is going to beat the pants off of R's built in t-test function.

Let's Do Some Analysis

```
spend.vals <- new.dats[, list(means = mean(SPENT),
    vars = var(SPENT), .N),
    by = TREAT1]
(spend.vals$means[1] - spend.vals$means[2])/
    sqrt((spend.vals$vars[1]/spend.vals$N[1]) +
    (spend.vals$vars[2]/spend.vals$N[2]))
#28.70174
spend.vals <- new.dats[, list(means = mean(SPENT),
    vars = var(SPENT), .N),
    by = TREAT2]
(spend.vals$means[1] - spend.vals$means[2])/
    sqrt((spend.vals$vars[1]/spend.vals$N[1]) +
    (spend.vals$vars[2]/spend.vals$N[2]))
#.9591
```

Let's Do Some Analysis

- Another question that we can ask about this data is whether or not the treatment assignment is clean. In order for causal claims to be made, we need to be sure that the treatment assignment method is not compromised. In other words, knowing a person's treatment assignment for the first experiment should give us no information about the treatment assignment they receive in the second experiment.
- We can use a chi-square test of independence to check this.
- Start by finding the number of people in each possible combination of treatment assignments.
- Using these values, create a matrix of the counts and pass this to `chisq.test()` in R.
- Are the treatment assignments independent?

Let's Do Some Analysis

```
counts <- new.dats[,.N,by = list(TREAT1, TREAT2)]
setkey(counts,TREAT1,TREAT2)
count.mat <- matrix(ncol = 2, nrow = 2, byrow = T, counts$N)
chisq.test(count.mat)
#p < .05, so dependent treatment assignments
```

Keys and Hashing

- In the previous example, I utilized the function `setkey(dt, keys)`.
- `setkey()` creates a key for our table. On the surface, it simply sorts the data.
- In a few weeks, we'll discuss this more in depth.
- A key is a column in a table over which the data is indexed. Keys are mostly utilized when joining tables that have a common element.
- Keying the join by column allows us to join tables in $O(log(n))$ instead of $O(n^2)$.
- Very important when building a data set from many sources.

data.table

- data.table is the second most discussed R package on Stack Exchange.
- Many useful functions that improve R's performance for inconveniently sized data sets.
- There are many more utilities that we didn't talk about today.
- This is a package that I highly recommend. If you work with larger data sets, data.table can save you large amounts of time.
- It is definitely worth reading the intro vignettes that come with package. Just search data.table in R on the Google.
- Used in conjunction with parallel chunking methods (our next topic), big data problems become pretty damn simple in R.