Data Science Course Notes

Joao Pinelo Silva

September 11, 2015

Calculating / Estimating Memory Requirements

DATA: Data frame with 1,500,000 rows and 120 columns of numeric data and a 64 bits

Formula
Number of rows x Number of columns x 8 bytes

Why 8 bytes? machine/OS is 64 bits = 8 bytes, since there are 8 bits per byte

1,500,000 x 120 x 8 = 1440000000

1440000000 / 2^20 bytes/MB = 1,373.29 MB = 1.34 GB

R Connections

For example opening, reading and or writing files, open URL...

Create File

```
dput(x, file = "dputed_x.R")
```

file() basic arguments:

- description = name of file;
- open = code indicating:
  - "r" read only,
  - ""w writing and initializing a new file,
  - "a" appending,
  - "rb", "wb", "ab" reading, writing, or appending in binary mode (windows)

Subsetting

- [returns object of same class as original - can be used to extract more than one element (one exception)
- [[extract elements of list or df. Can extract a single element. Not necessarily resturn list or df.
- '$ extract elements of list or df by name.

Order: *row*, *Column*

Subsetting vectors

```r
a <- rnorm(20)
b <- rep(NA, 20)
x <- sample(c(a,b), 15)
x
```

```
##  [1] -1.08402667          NA          NA          NA  0.96128191
##  [6]          NA  2.06642638  0.14065123          NA          NA
## [11]  2.35725514          NA  2.11801243 -1.50504664  0.02855351
```

```r
x[1:10]
```

```
##  [1] -1.0840267         NA         NA         NA  0.9612819         NA
##  [7]  2.0664264  0.1406512         NA         NA
```

```r
x[is.na(x)]
```

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

```r
y <- x[!is.na(x)]
y[y > 0]
```

```
## [1] 0.96128191 2.06642638 0.14065123 2.35725514 2.11801243 0.02855351
```

```r
x[x > 0]
```

```
##  [1]         NA         NA         NA 0.96128191         NA 2.06642638
##  [7] 0.14065123         NA         NA 2.35725514         NA 2.11801243
## [13] 0.02855351
```

```r
x[!is.na(x) & x > 0]
```

```
## [1] 0.96128191 2.06642638 0.14065123 2.35725514 2.11801243 0.02855351
```

```r
x[c(3, 5, 7)]
```

```
## [1]        NA 0.9612819 2.0664264
```

```r
x[c(-2, -10, -12, -15)] # same as below, putting minus before 'c'
```

```
##  [1] -1.0840267         NA         NA  0.9612819         NA  2.0664264
##  [7]  0.1406512         NA  2.3572551  2.1180124 -1.5050466
```

```r
x[-c(2, 10, 12, 15)]
```

```
##  [1] -1.0840267         NA         NA  0.9612819         NA  2.0664264
##  [7]  0.1406512         NA  2.3572551  2.1180124 -1.5050466
```

```r
vect <- c(foo = 11, bar = 2, norf = NA)
names(vect)
```

```
## [1] "foo"  "bar"  "norf"
```

```r
vect2 <- c(11, 2, NA)
names(vect2) <- c("foo", "bar", "norf")
vect2
```

```
##  foo  bar norf
##   11    2   NA
```

```r
vect["bar"]
```

```
## bar
##   2
```

```r
vect[c("foo", "bar")]
```

```
## foo bar
##  11   2
```

Subsetting lists

```r
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
name <- "foo"
x[[name]] # computed index for 'foo'
```

```
## [1] 1 2 3 4
```

```r
x$name # element 'name' does not exist in x, therefore NULL below
```

```
## NULL
```

```r
x$foo # element 'foo' does exist in x, and therefore its elements are printed below
```

```
## [1] 1 2 3 4
```

- `[` returns element of list. Cannot use `[[` or `$` to extract multiple elements of list.
- `[[` returns neste elements - the element inside the list element. Can take integer seq.

```r
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
x[[c(1, 3)]]
```

```
## [1] 14
```

```r
x[[1]][[3]]
```

```
## [1] 14
```

```r
x[[c(2, 1)]]
```

```
## [1] 3.14
```

NAs - Select, See, and Eventually Remove

```r
x <- c(1, 2, NA, 4, NA, 6)
NAs <- is.na(x)
NAs
```

```
## [1] FALSE FALSE  TRUE FALSE  TRUE FALSE
```

```r
x[!NAs]
```

```
## [1] 1 2 4 6
```

Removing NAs on combination of several rows/cols
`complete.cases()` - Useful to keep only cases of database with only complete data accross multiple variables and subjects.

```r
x <- c(1, 2, NA, 4)
y <- c("a", "b", "c", NA)
good <- complete.cases(x, y)
good
```

```
## [1]  TRUE  TRUE FALSE FALSE
```

```r
x[good]
```

```
## [1] 1 2
```

```r
y[good]
```

```
## [1] "a" "b"
```

To use in data frames
`good <- complete.cases(df)`, removes rows with NAs in accross all variables.
`df[good, ][1:6, ]`, will contain only rows where there are no NAs

Sequences

```r
1:20
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```r
pi:10 # 10 isn't reached as it would be over
```

```
## [1] 3.141593 4.141593 5.141593 6.141593 7.141593 8.141593 9.141593
```

```r
12:1
```

```
##  [1] 12 11 10  9  8  7  6  5  4  3  2  1
```

`seq()` args: `by`; `length`

```r
seq(1, 20)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```r
seq(1, 5, by = 0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```r
seq(1, 5, length = 30)
```

```
##  [1] 1.000000 1.137931 1.275862 1.413793 1.551724 1.689655 1.827586
##  [8] 1.965517 2.103448 2.241379 2.379310 2.517241 2.655172 2.793103
## [15] 2.931034 3.068966 3.206897 3.344828 3.482759 3.620690 3.758621
## [22] 3.896552 4.034483 4.172414 4.310345 4.448276 4.586207 4.724138
## [29] 4.862069 5.000000
```

```r
my_seq <- 1:10
1:length(my_seq)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq(along.with = my_seq)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq_along(my_seq)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

rep()

```r
rep(0, times = 40)
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [36] 0 0 0 0 0
```

```r
rep(c(1, 2, 3), times = 10)
```

```
##  [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

```r
rep(c(1, 2, 3), each = 10)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
```

paste()

args:
`collapse` refers to separator of elements within a vector
`sep` refers to separator of elements from different vectors

```r
my_name <- c("My", "name", "is", "joao")
paste(my_name, collapse = " ")
```

```
## [1] "My name is joao"
```

```r
paste(c(1:3), c("X", "Y", "Z"), sep = "")
```

```
## [1] "1X" "2Y" "3Z"
```

```r
paste(LETTERS, 1:4, sep = "-")
```

```
##  [1] "A-1" "B-2" "C-3" "D-4" "E-1" "F-2" "G-3" "H-4" "I-1" "J-2" "K-3"
## [12] "L-4" "M-1" "N-2" "O-3" "P-4" "Q-1" "R-2" "S-3" "T-4" "U-1" "V-2"
## [23] "W-3" "X-4" "Y-1" "Z-2"
```

Sample()

```r
y <- rnorm(100)
z <- rep(NA, 100)
my_data <- sample(c(y,z), 10)
my_data
```

```
##  [1]  0.48727006          NA          NA          NA          NA
##  [6]  0.08033121 -0.73255558  1.35262509          NA -1.43628788
```

identical()

```r
vect <- c(foo = 11, bar = 2, norf = NA)
vect2 <- c(11, 2, NA)
names(vect2) <- c("foo", "bar", "norf")
identical(vect, vect2)
```

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

matrix()

```r
my_matrix <- matrix(data = 1:20, nrow = 4, ncol = 5)
```

colnames()

```r
a <- c("John", 34, 65, 55, 45, 1)
b <- c("Mary", 44, 45, 68, 45, 2)
my_data <- data.frame(rbind(a, b)) # without data.frame, my_data would be a matrix and coerce all integ
cnames <- c("patient", "age", "weight", "bp", "rating", "test")
colnames(my_data) <- cnames
my_data
```

```
##   patient age weight bp rating test
## a    John  34     65 55     45    1
## b    Mary  44     45 68     45    2
```

LOGICAL OPERATORS

`&` AND, recycles
`&&` AND, does not recycle, eveluates first element of vector only

`|` OR, recycles
`||` OR, does not recycle, eveluates first element of vector only

`&` has precedence over `|`

isTRUE()

```r
isTRUE(5 > 4)
```

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

identical()

```r
identical('nip', 'nip')
```

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

xor()

Exclusive OR - TRUE if one (and only one) of args is TRUE

```
xor(TRUE, TRUE)
```

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

which()

Takes logical vector as argument and returns indices of the vector that are TRUE

```
ints <- sample(10)
ints
```

```
##  [1]  7  2  1  4  3  5  6 10  8  9
```

```
which(ints > 7)
```

```
## [1]  8  9 10
```

any()

Returns TRUE if one or more of the elements in the logical vector is TRUE

```
ints <- sample(10)
ints
```

```
##  [1]  1 10  4  7  5  8  6  2  3  9
```

```
any(ints > 5)
```

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

all()

Returns TRUE if one or more of the elements in the logical vector is TRUE

```
ints <- sample(10)
ints
```

```
##  [1]  4  2  1  9  5  7  6  3  8 10
```

```
all(ints > 0)
```

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

Create a function that takes another function as argument

```
  evaluate <- function(func, dat){
  result <- func(dat)
  result
}
```

Anonimous Function

Function that is defined but not named

```
  evaluate <- function(func, dat){ # create function `evaluate`
  result <- func(dat)
  result
  }

evaluate(function(x){x+1}, 6)  # anonymous function inside `evaluate` function

# Example:
evaluate(function(x){x[1]}, c(8, 4, 0)) # Returns '8', the first element of vector 8,4,0
evaluate(function(x){x[length(x)]}, c(8, 4, 0)) # Returns '0', the last element of the vector
```

Non-evaluated example

```
lapply(unique_vals, function(elem) elem[2])
```

Elipsis = ... Allows an infinite munber of arguments to be passed into a function.

When creating a function, all arguments created after an ellipses MUST have default values. Normally `...` is the last argument. But there are exceptions, such as in `paste()`

Elipsis inside function

```
telegram <- function(...){
  paste("START", ..., "STOP")
}
```

Unpacking function args

```
mad_libs <- function(...){
  # Do your argument unpacking here!
  args <- list(...)
  place <- args[["place"]]
  adjective <- args[["adjective"]]
  noun <- args[["noun"]]

  # Don't modify any code below this comment.
  # Notice the variables you'll need to create in order for the code below to
  # be functional!
  paste("News from", place, "today where", adjective, "students took to the streets in protest of the n
}
```

Create Binary Operator

```
"%p%" <- function(left, right){
  paste(left, right, sep = " ")
}
"Hello" %p% "world!"
```

```
## [1] "Hello world!"
```

lapply()

Takes a list and a function, and applies the funcion to each list member. On a datatable for example, it applies the function to each column. `lapply(data, fun)` # data = datatable for example and fun some function such as class(). write fun name only, no parenthesis.
lapply() returns a list - hence the *l*

```
a <- c("John", 34, 65, 55, 45, 1)
b <- c("Mary", 44, 45, 68, 45, 2)
my_data <- data.frame(rbind(a, b)) # without data.frame, my_data would be a matrix and coerce all integ
cnames <- c("patient", "age", "weight", "bp", "rating", "test")
colnames(my_data) <- cnames
my_data$age <- c(100, 50)
my_data
```

```
##   patient age weight bp rating test
## a    John 100     65 55     45    1
## b    Mary  50     45 68     45    2
```

```
lapply(my_data, class) # lapply returns list
```

```
## $patient
## [1] "factor"
##
## $age
## [1] "numeric"
##
## $weight
## [1] "factor"
##
## $bp
## [1] "factor"
##
## $rating
## [1] "factor"
##
## $test
## [1] "factor"
```

```r
# Another example
age <- as.data.frame(my_data[, 2])
age
```

```
##   my_data[, 2]
## 1          100
## 2           50
```

```r
lapply(age, mean)
```

```
## $`my_data[, 2]`
## [1] 75
```

sapply()

sapply() uses lapply behind the scenes and then *simplifies* the result transforming the list in vector, hence the *s* fro simplify.

```r
a <- c("John", 100, 65, 55, 45, 1)
b <- c("Mary", 50, 45, 68, 45, 2)
my_data <- data.frame(rbind(a, b)) # without data.frame, my_data would be a matrix and coerce all integ
cnames <- c("patient", "age", "weight", "bp", "rating", "test")
colnames(my_data) <- cnames
my_data$age <- c(100, 50)
my_data
```

```
##   patient age weight bp rating test
## a    John 100     65 55     45    1
## b    Mary  50     45 68     45    2
```

```r
sapply(my_data, class) # sapply resturns vector
```

```
##   patient       age    weight        bp    rating      test
##  "factor" "numeric"  "factor"  "factor"  "factor"  "factor"
```

```r
# Another example

age <- as.data.frame(my_data[, 2])
age
```

```
##   my_data[, 2]
## 1          100
## 2           50
```

```r
sapply(age, mean)
```

```
## my_data[, 2]
##           75
```

Extract percentage. See *Binary data makes it easy to compute percentage* below

```
a <- c(1, 0, 1)
b <- c(1, 1, 0)
c <- c(1, 1, 1)
d <- c(1, 0, 0)
e <- c(1, 1, 0)
my_data <- data.frame(rbind(a, b, c, d, e)) # without data.frame, my_data would be a matrix and coerce
cnames <- c("Red", "Blue", "Yellow")
colnames(my_data) <- cnames
my_data
```

```
##   Red Blue Yellow
## a   1    0      1
## b   1    1      0
## c   1    1      1
## d   1    0      0
## e   1    1      0
```

```
sapply(my_data, mean)
```

```
##    Red   Blue Yellow
##    1.0    0.6    0.4
```

```
# 1 is 100%, 0.6 is 60%, 0.4 is 40%. % of times that each color is used (value = 1)
```

sapply output

> In general, if the result is a list where every element is of length one, then sapply() returns a vector. If the result is a list where every element is a vector of the same length (> 1), sapply() returns a matrix. If sapply() can't figure things out, then it just returns a list, no different from what lapply() would give you. *Source: swirl() package.*

Binary data makes it easy to compute percentage

> Perhaps it's more informative to find the proportion of flags (out of 194) containing each color. Since $>$ each column is just a bunch of 1s and 0s, the arithmetic mean of each column will give us the proportion $>$ of 1s. (If it's not clear why, think of a simpler situation where you have three 1s and two 0s – $(1 + 1 > + 1 + 0 + 0)/5 = 3/5 = 0.6$). *Source: swirl() package*

range()

Returns the min and max values of a numeric vector (first argument)

```
a <- c(0, 1, 0)
b <- c(30, 1, 0)
c <- c(1, 0, 15)
d <- c(1, 5, 0)
e <- c(1, 10, 0)
```

```r
my_data <- data.frame(rbind(a, b, c, d, e)) # without data.frame, my_data would be a matrix and coerce
cnames <- c("Star", "Square", "Circle")
colnames(my_data) <- cnames
my_data
```

```
##   Star Square Circle
## a    0      1      0
## b   30      1      0
## c    1      0     15
## d    1      5      0
## e    1     10      0
```

```r
sapply(my_data, range)
```

```
##      Star Square Circle
## [1,]    0      0      0
## [2,]   30     10     15
```

```r
# returns the min (row 1) and max (row 2) values for each col.
```