Lecture 4 - Data Cleaning, Loops and `apply` Functions

Sajid Bashir, PhD

15 February, 2020

Contents

Agenda

- Tip of the Day
- Data Cleaning
- Data Wrangling
- Loops in R
- The `apply()` Family
- Specifying Scope

Tip of the Day

`search()`

- Gives list of attached packages in R search path

```
# The default packages in search path
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

- Adding `BOD` (Biochemical Oxygen Demand) in searach path

```
attach(BOD)
search()
```

```
##  [1] ".GlobalEnv"        "BOD"               "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

- Remove `BOD` from searach path

```
detach(BOD)
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

- List the full path of the packages:

```
searchpaths()
```

```
## [1] ".GlobalEnv"
## [2] "C:/Program Files/R/R-3.6.2/library/stats"
## [3] "C:/Program Files/R/R-3.6.2/library/graphics"
## [4] "C:/Program Files/R/R-3.6.2/library/grDevices"
## [5] "C:/Program Files/R/R-3.6.2/library/utils"
## [6] "C:/Program Files/R/R-3.6.2/library/datasets"
## [7] "C:/Program Files/R/R-3.6.2/library/methods"
## [8] "Autoloads"
## [9] "C:/PROGRA~1/R/R-36~1.2/library/base"
```

Data Cleaning

A Common Challenge

One of the most common problems when importing manually-entered data is inconsistent data types within columns. For a simple example, let's look at `TVhours` column in a messy version of the student survey data.

```
messy.data <- read.csv("messy_data.csv", header=TRUE)
```

What's going on? Let us look at the summary:

```
str(messy.data)
```

```
## 'data.frame':    31 obs. of  7 variables:
##  $ X              : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Program        : Factor w/ 3 levels "CS","DS","SW": 2 2 2 3 2 2 2 3 2 3 ...
##  $ PriorExp       : Factor w/ 3 levels "Extensive experience",..: 3 2 3 3 2 2 3 3 3 2 ...
##  $ Rexperience    : Factor w/ 4 levels "Basic competence",..: 3 3 1 1 3 3 1 1 1 3 ...
##  $ OperatingSystem: Factor w/ 2 levels "Mac OS X","Windows": 1 2 1 1 2 2 1 2 2 2 ...
##  $ TVhours        : Factor w/ 18 levels "0","1","10","14",..: 8 6 7 1 9 12 1 11 1 1 ...
##  $ Editor         : Factor w/ 4 levels "Excel","LaTeX",..: 3 3 3 2 3 3 3 3 3 3 ...
```

Several entries have *non-numeric values and contain strings*. As a result, `TVhours` is being imported as `factor`.

Fixing Options

1. **A Simple Approach:** Let us try to cast it back to `numeric`?

   ```
   tv.hours.messy <- messy.data$TVhours
   tv.hours.messy
   ```

   ```
   ##  [1] 2              15 incl movies 16             0              2 hours
   ##  [6] 5ish           0              4              0              0
   ## [11] 14             7-Jun          10             15             4
   ## [16] 4              10             7              33 (Netflix)   8
   ## [21] 8              falkjklj       0              0              0
   ## [26] none           6              1              0              0
   ## [31] 4
   ## 18 Levels: 0 1 10 14 15 15 incl movies 16 2 2 hours 33 (Netflix) 4 5ish 6 ... none
   ```

   ```
   as.numeric(tv.hours.messy)
   ```

   ```
   ##  [1]  8  6  7  1  9 12  1 11  1  1  4 15  3  5 11 11  3 14 10 16 16 17  1  1  1
   ## [26] 18 13  2  1  1 11
   ```

Well, that didn't work!

This has converted all the values into the **rank** of *integer-coded levels* of the *factor* and is not what we wanted. The correct numeric values also get corrupted.

2. **A Better Approach:** Protect the already correct numeric values. Consider the following simple example

   ```
   num.vec <- c(3.1, 2.5)
   as.factor(num.vec)
   ```

   ```
   ## [1] 3.1 2.5
   ```

```
## Levels: 2.5 3.1
```

```
as.numeric(as.factor(num.vec)) # replaces numerics with rank of factor values
```

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

```
as.numeric(as.character(as.factor(num.vec))) # retains numeric values
```

```
## [1] 3.1 2.5
```

Result

If we take a number that's being coded as a factor and *first* turn it into a `character` string, *then* converting the string to a `numeric` gets the number back.

Applying the approach to the corrupted `TVhours` column.

```
as.character(tv.hours.messy)
```

```
##  [1] "2"              "15 incl movies" "16"             "0"
##  [5] "2 hours"        "5ish"           "0"              "4"
##  [9] "0"              "0"              "14"             "7-Jun"
## [13] "10"             "15"             "4"              "4"
## [17] "10"             "7"              "33 (Netflix)"   "8"
## [21] "8"              "falkjklj"       "0"              "0"
## [25] "0"              "none"           "6"              "1"
## [29] "0"              "0"              "4"
```

```
as.numeric(as.character(tv.hours.messy))
```

```
## Warning: NAs introduced by coercion
```

```
##  [1]  2 NA 16  0 NA NA  0  4  0  0 14 NA 10 15  4  4 10  7 NA  8  8 NA  0  0  0
## [26] NA  6  1  0  0  4
```

```
typeof(as.numeric(as.character(tv.hours.messy)))  # Success!! (Almost...)
```

```
## Warning in typeof(as.numeric(as.character(tv.hours.messy))): NAs introduced by
## coercion
```

```
## [1] "double"
```

We can observe a small improvement.

- Correct numerica values are retained.
- All the corrupted cells now appear as `NA`, which is R's missing indicator.

We can do a even better by

- Cleaning up the vector once we get it to `character` form.
- Deleting non-numeric (or .) characters.

  Use `gsub()` to replace everything except `digits` and `.` with a blank `" "`

```
tv.hours.strings <- as.character(tv.hours.messy)
tv.hours.strings
```

```
##  [1] "2"              "15 incl movies" "16"             "0"
##  [5] "2 hours"        "5ish"           "0"              "4"
##  [9] "0"              "0"              "14"             "7-Jun"
## [13] "10"             "15"             "4"              "4"
## [17] "10"             "7"              "33 (Netflix)"   "8"
## [21] "8"              "falkjklj"       "0"              "0"
```

```
## [25] "0"              "none"           "6"              "1"
## [29] "0"              "0"              "4"
```

```
gsub("[^0-9.]", "", tv.hours.strings)
```

```
##  [1] "2"  "15" "16" "0"  "2"  "5"  "0"  "4"  "0"  "0"  "14" "7"  "10" "15" "4"
## [16] "4"  "10" "7"  "33" "8"  "8"  ""   "0"  "0"  "0"  ""   "6"  "1"  "0"  "0"
## [31] "4"
```

The Final Product

```
tv.hours.messy
```

```
##  [1] 2              15 incl movies 16             0              2 hours
##  [6] 5ish           0              4              0              0
## [11] 14             7-Jun          10             15             4
## [16] 4              10             7              33 (Netflix)   8
## [21] 8              falkjklj       0              0              0
## [26] none           6              1              0              0
## [31] 4
## 18 Levels: 0 1 10 14 15 15 incl movies 16 2 2 hours 33 (Netflix) 4 5ish 6 ... none
```

```
tv.hours.clean <- as.numeric(gsub("[^0-9.]", "", tv.hours.strings))
tv.hours.clean
```

```
##  [1]  2 15 16  0  2  5  0  4  0  0 14  7 10 15  4  4 10  7 33  8  8 NA  0  0  0
## [26] NA  6  1  0  0  4
```

As a last step, we should go through and figure out if any of the `NA` values should really be `0`. This step is not shown here and you can exercise using the `gsub()` function.

3. **A Different Approach:** We can also handle this problem by setting `stringsAsFactors = FALSE` when importing our data.

```
messy.data <- read.csv("messy_data.csv", header=TRUE, stringsAsFactors=FALSE)
str(messy.data)
```

```
## 'data.frame':    31 obs. of  7 variables:
##  $ X              : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Program        : chr  "DS" "DS" "DS" "SW" ...
##  $ PriorExp       : chr  "Some experience" "Never programmed before" "Some experience" "Some exp
##  $ Rexperience    : chr  "Never used" "Never used" "Basic competence" "Basic competence" ...
##  $ OperatingSystem: chr  "Mac OS X" "Windows" "Mac OS X" "Mac OS X" ...
##  $ TVhours        : chr  "2" "15 incl movies" "16" "0" ...
##  $ Editor         : chr  "Microsoft Word" "Microsoft Word" "Microsoft Word" "LaTeX" ...
```

Now everything is a `character` instead of a `factor`

One-line Cleanup

Let's clean up the `TVhours` column and cast it to numeric all in one command.

```
survey <- transform(messy.data,
                    TVhours = as.numeric(gsub("[^0-9.]", "", TVhours)))
str(survey)
```

```
## 'data.frame':    31 obs. of  7 variables:
##  $ X              : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Program        : chr  "DS" "DS" "DS" "SW" ...
##  $ PriorExp       : chr  "Some experience" "Never programmed before" "Some experience" "Some exp
##  $ Rexperience    : chr  "Never used" "Never used" "Basic competence" "Basic competence" ...
```

```
##  $ OperatingSystem: chr  "Mac OS X" "Windows" "Mac OS X" "Mac OS X" ...
##  $ TVhours        : num  2 15 16 0 2 5 0 4 0 0 ...
##  $ Editor         : chr  "Microsoft Word" "Microsoft Word" "Microsoft Word" "LaTeX" ...
```

What about all those other character variables?

```
table(survey[["Program"]])
```

```
##
## CS DS SW
##  4 19  8
```

```
table(as.factor(survey[["Program"]]))
```

```
##
## CS DS SW
##  4 19  8
```

Having factors coded as characters may be OK for many parts of our analysis.

But to be safe, let's fix things:

```
# Figure out which columns are coded as characters
chr.indexes <- sapply(survey, FUN = is.character)
chr.indexes
```

```
##               X         Program        PriorExp     Rexperience OperatingSystem
##           FALSE            TRUE            TRUE            TRUE            TRUE
##         TVhours          Editor
##           FALSE            TRUE
```

```
# Re-code all of the character columns to factors
survey[chr.indexes] <- lapply(survey[chr.indexes], FUN = as.factor)
```

Here's the outcome

```
str(survey)
```

```
## 'data.frame':    31 obs. of  7 variables:
##  $ X              : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Program        : Factor w/ 3 levels "CS","DS","SW": 2 2 2 3 2 2 2 3 2 3 ...
##  $ PriorExp       : Factor w/ 3 levels "Extensive experience",..: 3 2 3 3 2 2 3 3 3 2 ...
##  $ Rexperience    : Factor w/ 4 levels "Basic competence",..: 3 3 1 1 3 3 1 1 1 3 ...
##  $ OperatingSystem: Factor w/ 2 levels "Mac OS X","Windows": 1 2 1 1 2 2 1 2 2 2 ...
##  $ TVhours        : num  2 15 16 0 2 5 0 4 0 0 ...
##  $ Editor         : Factor w/ 4 levels "Excel","LaTeX",..: 3 3 3 2 3 3 3 3 3 3 ...
```

Success!

Data Wrangling

Refers to the process of cleaning, restructuring and enriching the raw data into a more usable format. It helps to quicken the process of decision making, and get better insights in less time.

Steps in Data Wrangling

1. **Discovering:** Before implementing the cleaning process develop a deeper understanding the data. We identify criteria to demarcate and divide the data accordingly.

2. **Structuring:** Raw data is in a haphazard manner, generally there is no structure to it. Based on the criteria identified in the first step, data is separated/ structured for for better analysis.
3. **Cleaning:** All datasets have some outliers which can skew the results of the analysis. Data is cleaned for better results and high-quality analysis. Change null values, standardize formatting etc.
4. **Enriching:** Take stock of what is in the data and strategise whether you will have to augment it using some additional data in order to make it better. Brainstorm about whether we can derive new data from existing clean data set.
5. **Validation:** Refers to some repetitive programming steps which are used to verify the consistency, quality and the security of the data. E.g. ascertain whether the data set fields are accurate, whether attributes are normally distributed.
6. **Publish:** The prepared wrangled data is published so that it can be used further down the line. If needed, document the steps which were taken or logic used to wrangle the said data.

We will be dealing with Data Wrangling more in next sections

Another Challenge

Another common problem is that When data is entered manually, misspellings and case changes are very common. E.g., a column showing life support mechanism may look like,

```
life.support <- as.factor(c("dialysis", "Ventilation", "Dialysis", "dialysis", "none", "None", "nnone",
summary(life.support)
```

```
##    dialysis    Dialysis    dyalysis       nnone        none        None
##           3           1           1           1           2           1
## ventilation Ventilation
##           1           1
```

```
summary(life.support)
```

```
##    dialysis    Dialysis    dyalysis       nnone        none        None
##           3           1           1           1           2           1
## ventilation Ventilation
##           1           1
```

This factor has 8 levels even though it should have 3 (dialysis, ventilation, none).

We can fix many of the typos by running `spellcheck` (if available) before importing the data, or by changing the values on a case-by-case basis later. There's a faster way to fix just the capitalization issue (Homework 2).

Loops in R

loops are ways of iterating over data

- **For Loops**

  A **for loop** executes a chunk of code for every value of an **index variable** in an **index set**

  The basic syntax takes the form

```
for(index.variable in index.set) {
  code to be repeated at every value of index.variable
}
```

The index set is often a vector of integers, but can be more general

Example 01

```
index.set <- list(name="Ahmad", weight=185, is.male=TRUE) # a list
for(i in index.set) {
  print(c(i, typeof(i)))
}
```

```
## [1] "Ahmad"     "character"
## [1] "185"    "double"
## [1] "TRUE"    "logical"
```

Example 02

```
for(i in 1:4) {
  print(i)
}
```

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

```
phrase <- "Good Night, "
for(word in c("and", "Good", "Luck")) {
  phrase <- paste(phrase, word)
  print(phrase)
}
```

```
## [1] "Good Night,  and"
## [1] "Good Night,  and Good"
## [1] "Good Night,  and Good Luck"
```

  - `paste()` concatenates vectors after converting to charatec.

Example 03: Calculate Sum of Each Column

```
fake.data <- matrix(rnorm(500), ncol=5) # create fake 100 x 5 data set
head(fake.data,2) # print first two rows
```

```
##            [,1]       [,2]       [,3]      [,4]       [,5]
## [1,] -0.3669806 -0.3343998 -0.4384081 0.5317731 -0.1469892
## [2,] -0.2543469  1.1324551 -0.4862656 1.5847693  0.8614055
```

```
col.sums <- numeric(ncol(fake.data)) # returns a numeric vector with `ncol` elements
for(i in 1:nrow(fake.data)) {
  col.sums <- col.sums + fake.data[i,] # add ith observation to the sum
}
col.sums
```

```
## [1] -1.323855 17.270144  4.852998 -5.677463  0.611398
```

```
colSums(fake.data) # A better approach (see also colMeans())
```

```
## [1] -1.323855 17.270144  4.852998 -5.677463  0.611398
```

  - `rnorm(n, mean, sd)` generates vector of numbers having normal distribution with mean = `mean` and standard deviation = `sd`.

- **While Loops**

repeat a chunk of code while the specified condition remains true

```
day <- 1
num.days <- 365
while(day <= num.days) {
  day <- day + 1
}
```

We might not be using `while` loops quite frequently. Just be aware that they exist, and that they may become useful to you at some point in your analytics career.

The apply() Family

loops are ways of iterating over data and `apply()` functions are good *alternatives* to loops.

These are all efficient ways of applying a function to `MARGIN` of an array or elements of a list. Where `MARGIN` is a vector giving the subscripts or dimension over which the function will be applied over.

Various `apply()` Functions

Command	Description
`apply(X, MARGIN, FUN)`	Obtain a `vector/array/list` by applying `FUN` along the specified `MARGIN` of an array or matrix `X`
`lapply(X, FUN)`	Obtain a `list` by applying `FUN` to the elements of a list `X`
`sapply(X, FUN)`	Simplified version of `lapply`. Returns a `vector/array` instead of `list`.
`tapply(X, INDEX, FUN)`	Obtain a `table` by applying `FUN` to each combination of the factors given in `INDEX`

These functions are (good!) alternatives to loops. They are typically *more efficient* than loops (often run considerably faster on large data sets). Takes practice to get used to, but make analysis easier to debug and less prone to error when used effectively. You can always type `example(function)` to get code examples (E.g., `example(apply)`)

Example 1: apply()

```
colMeans(fake.data)
```

```
## [1] -0.01323855  0.17270144  0.04852998 -0.05677463  0.00611398
```

```
apply(fake.data, MARGIN=2, FUN=mean) # MARGIN = 1 for rows, 2 for columns
```

```
## [1] -0.01323855  0.17270144  0.04852998 -0.05677463  0.00611398
```

```
# Function that calculates proportion of vector indexes that are > 0
propPositive <- function(x) mean(x > 0)
apply(fake.data, MARGIN=2, FUN=propPositive)
```

```
## [1] 0.47 0.56 0.54 0.53 0.53
```

Example 2: lapply(), sapply()

```
lapply(survey, is.factor) # Returns a list
```

```
## $X
```

```
## [1] FALSE
##
## $Program
## [1] TRUE
##
## $PriorExp
## [1] TRUE
##
## $Rexperience
## [1] TRUE
##
## $OperatingSystem
## [1] TRUE
##
## $TVhours
## [1] FALSE
##
## $Editor
## [1] TRUE
```

```
sapply(survey, FUN = is.factor) # Returns a vector with named elements
```

```
##               X         Program        PriorExp     Rexperience OperatingSystem
##           FALSE            TRUE            TRUE            TRUE            TRUE
##         TVhours          Editor
##           FALSE            TRUE
```

Example 3: apply(), lapply(), sapply()

```
apply(cars, 2, FUN=mean) # Data frames are arrays
```

```
## speed  dist
## 15.40 42.98
```

```
lapply(cars, FUN=mean) # Data frames are also lists
```

```
## $speed
## [1] 15.4
##
## $dist
## [1] 42.98
```

```
sapply(cars, FUN=mean) # sapply() is just simplified lapply()
```

```
## speed  dist
## 15.40 42.98
```

Example 4: tapply()

Think of `tapply()` as a generalized form of the `table()` function

```
library(MASS)
```

```
##
## Attaching package: 'MASS'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     survey
```

```r
# Get a count table, data broken down by Origin and DriveTrain
table(Cars93$Origin, Cars93$DriveTrain)
```

```
##
##           4WD Front Rear
##   USA       5    34    9
##   non-USA   5    33    7
```

```r
# Calculate average MPG.City, broken down by Origin and Drivetrain
tapply(Cars93$MPG.city, INDEX = Cars93[c("Origin", "DriveTrain")], FUN=mean)
```

```
##          DriveTrain
## Origin     4WD    Front     Rear
##   USA     17.6 22.14706 18.33333
##   non-USA 23.4 24.93939 19.14286
```

Example 5: tapply()

Let's get the average horsepower by car `Origin` and `Type`

```r
tapply(Cars93[["Horsepower"]], INDEX = Cars93[c("Origin", "Type")], FUN=mean)
```

```
##          Type
## Origin     Compact    Large  Midsize    Small   Sporty    Van
##   USA     117.4286 179.4545 153.5000 89.42857 166.5000 158.40
##   non-USA 141.5556       NA 189.4167 91.78571 151.6667 138.25
```

What's that `NA` doing there?

```r
any(Cars93$Origin == "non-USA" & Cars93$Type == "Large")
```

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

None of the non-USA manufacturers produced Large cars!

Example 6: using tapply() to mimic table()

Here's how one can use `tapply()` to produce the same output as the `table()` function

```r
library(MASS)
# Get a count table, data broken down by Origin and DriveTrain
table(Cars93$Origin, Cars93$DriveTrain)
```

```
##
##           4WD Front Rear
##   USA       5    34    9
##   non-USA   5    33    7
```

```r
# This one may take a moment to figure out...
tapply(rep(1, nrow(Cars93)), INDEX = Cars93[c("Origin", "DriveTrain")], FUN=sum)
```

```
##          DriveTrain
## Origin    4WD Front Rear
##   USA       5    34    9
##   non-USA   5    33    7
```

Specifying Scope

Thus far we've repeatedly typed out the data frame name when referencing its columns. This is because the data variables don't exist in our working environment.

Using `with(data, expr)`:

> Lets us specify that the code in `expr` should be evaluated in an environment that contains the elements of `data` as variables

```
library(MASS)
with(Cars93, table(Origin, Type))
```

```
##          Type
## Origin    Compact Large Midsize Small Sporty Van
##   USA           7    11      10     7      8   5
##   non-USA       9     0      12    14      6   4
```

Example:

```
any(Cars93$Origin == "non-USA" & Cars93$Type == "Large")
```

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

```
with(Cars93, any(Origin == "non-USA" & Type == "Large")) # Same effect!
```

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

```
with(Cars93, tapply(Horsepower, INDEX = list(Origin, Type), FUN=mean))
```

```
##          Compact    Large  Midsize    Small   Sporty    Van
## USA     117.4286 179.4545 153.5000 89.42857 166.5000 158.40
## non-USA 141.5556       NA 189.4167 91.78571 151.6667 138.25
```

Using `with()` makes code simpler, easier to read, and easier to debug.

Reminders

- Homework 2 will be uploaded on shared folder during next week.
- Lab 3 (uploaded on shared folder) - **due on Wednesday, 21 Februray 2020**
- If you have questions, feel free to post on the course group