Practical R for MBM

Chuck Frost

6/23/2020

Practical R for MBM field data manipulation and analysis

The purpose of this ongoing short course is to familiarize MBM personnel with the concepts of tidy data, basic data manipulation, and basic programming techniques as implemented in R. It is expected that participants have a working understanding of R and RStudio. We will not go into depth on other types of programming, specifically statistical programming and computer programming. The topics covered here will foster appropriate treatment of MBM data throughout the data life cycle and lead to cleaner and more useful data in the future.

Topics

1. R, RStudio, packages, help, intro to functions
2. Loading and saving, data types and structures
3. Visualization
4. Working with dates
5. Creating functions, conditional statements, looping
6. Tidy data

1.0.1 What is R?

An open-source programming environment that serves as:

1. A giant calculator
2. An interface for data analysis and visualization
3. An interface for data manipulation and file handling
4. An interface for a simple and efficient programming language (S, that became R)

R isn't the "method." For example, "We used R to estimate population size." yuck

1.0.2 What is RStudio?

An integrated development environment (IDE) for R.

Provides tools, menus, help, and easy access to the best parts of R.

1.1.1 Packages

Base R automatically provides you with many common functions.

```
mean(c(1,2,3))
```

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

```
sum(c(1,2,3))
```

```
## [1] 6
```

```
var(c(1,2,3))
```

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

Other useful functions exist thanks to R users. Collections of related functions get wrapped into packages.

If you're looking for something NOT in base R, you need to install the related package.

The CRAN provides R community tested and approved packages. Check the packages tab in RStudio for some you already have installed but (probably) not loaded. Just click them to load. If you know what package you need, there are 2 easy ways to install it. The first is by clicking the install button in the packages tab. Search the CRAN for the package you want and install it. The second is:

```
install.packages("PackageName")
```

Don't forget to load the package after installation, either by checking the box under packages, or by calling:

```
library(PackageName)
```

Note that in install.packages() you are searching a string in quotes, "PackageName" while in library() you are calling a package object without quotes, PackageName.

1.1.2 Installing from GitHub

If a package isn't uploaded to CRAN, it can still be made available through GitHub. Packages installed from GitHub require an intermediate step:

```
install.packages("devtools")

library(devtools)

install_github("USFWS/AKaerial", ref = "development")

library(AKaerial)
```

This code does 4 things:
1. Install devtools package
2. Load devtools package

3. Call install_github (from devtools), look for the GitHub account "USFWS" and repository "AKaerial" and load the package in the "development" branch
4. Load AKaerial package

Installing and loading AKaerial may take a while since it includes dependencies. Dependencies are packages that are specified within another package that must be included for that package to operate appropriately. Akaerial "depends" on several other packages and will check if you have these installed.

1.1.3 Functions

Once you have a package loaded, you can access functions written and included in that package. Typing the name of a function will spit out the code that makes up the function. This can be messy for a long or complicated function. Typing ?FunctionName will access the help file for a given function.

```
AdjustCounts

?AdjustCounts
```

This can be extremely helpful (if the help file is!) when troubleshooting why your code isn't working as you want it to. Note that the top of the help file also tells you FunctionName {PackageName} in case you can't figure out where your function is coming from. This could be the case if you have dozens of packages loaded or have borrowed someone else's code to help run your analysis. Don't underestimate the power of internet searches to find if something already exists to help you do your thing!

We will talk about functions in more depth later, but for now, the general idea is that functions take arguments, run processes, and (usually) return products saved as R objects.

```
numbers = c(1,2,3)

avg = mean(numbers)

avg
```

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

In this example, numbers is initiated as a vector of 3 integers: 1,2,3. We then run the function mean with numbers as its argument, and save the result as the object avg.

It can be helpful to think of functions as recipes and arguments as ingredients.

```
Mix = function(what.to.mix){
  dough = sum(what.to.mix)
  return(dough)
  }

Bake = function(what.to.bake, time.to.bake){

  cookies = what.to.bake * time.to.bake

  return(cookies)

}
```

```
my.ingredients = c("sugar", "eggs", "butter", "flour")

my.dough = Mix(my.ingredients)

my.cookies = Bake(my.dough, 10)
```

2.1.0 Loading Files

There are many ways to read data into R. We will go over the most common way data is stored and used in R; as a data frame. A data frame is a 2-dimensional array (table) where each column represents a variable (category) and each row represents a unique observation. Let's start with creating your own data frame. This might be useful if you have a small data set.

```
small.data = data.frame(
  name = c("Laura", "Zak", "Chuck"),
  size = c("small", "medium", "large"),
  tenure = c(6, "drifter",9)
    )
```

We can now check the structure of the data frame. This is a great first QA/QC check! If we know that tenure should be numeric, the structure should confirm it.

```
str(small.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ name  : Factor w/ 3 levels "Chuck","Laura",..: 2 3 1
##  $ size  : Factor w/ 3 levels "large","medium",..: 3 2 1
##  $ tenure: Factor w/ 3 levels "6","9","drifter": 1 3 2
```

Notice how all 3 variables are shown as Factor, which is incorrect. Factors, in general are tough to work with. If we try to change the value here (use $ to grab a column and brackets to select a row [ ], or use only brackets to select [row, column]), it throws an error.

```
small.data$tenure[2] = 3
```

```
## Warning in `[<-.factor`(`*tmp*`, 2, value = structure(c(1L, NA, 2L), .Label
## = c("6", : invalid factor level, NA generated
```

```
small.data[2,3] = 3
```

```
## Warning in `[<-.factor`(`*tmp*`, iseq, value = 3): invalid factor level, NA
## generated
```

```
levels(small.data$tenure)
```

```
## [1] "6"       "9"       "drifter"
```

Since R couldn't determine the structure of our tenure column, it defaulted to calling it a Factor, which can take any value. The unique set of values gets set as the levels of that factor. If you try to define a value outside of that set, you will get an error that results in NA. This is (almost) never what you want. So we can fix this in 2 ways:

First, we can use the "as." functions in base R to force R to evaluate something as a different data type. In this case, we can't just let R evaluate the factor as numeric since it will then take a numeric representation of that particular level of a factor in the set of all levels of that factor. What a mess. We have to first convert the factor to a character string using as.character, then to a numeric using as.numeric. This is called wrapping functions, when you pass as an argument to a new function the output of another function without saving the intermediate result. In the outer-most function call to as.numeric, anything that R can't represent as numeric will get NA. This is already messy and hard to trace changes to your data!

```
small.data$tenure=as.numeric(as.character(small.data$tenure))

str(small.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ name  : Factor w/ 3 levels "Chuck","Laura",..: 2 3 1
##  $ size  : Factor w/ 3 levels "large","medium",..: 3 2 1
##  $ tenure: num  6 NA 9
```

Or, since we are scripting our analysis, we can change it in our original code chunk and just re-run. Score 1 for a nice scripted workflow... but imagine doing this through thousands or hundreds of thousands of rows of data.

```
small.data = data.frame(
  name = c("Laura", "Zak", "Chuck"),
  size = c("small", "medium", "large"),
  tenure = c(6,3,9)
    )
```

While we are at it, lets just purge the factors from our data entirely. It is easy to make something a factor later, if we ever see the need. To do this, we just pass the argument stringsAsFactors = FALSE to our data frame function. Note that we have to add a comma after the declaration of our final column tenure.

```
small.data = data.frame(
  name = c("Laura", "Zak", "Chuck"),
  size = c("small", "medium", "large"),
  tenure = c(6,3,9),
  stringsAsFactors = FALSE
    )

str(small.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ name  : chr  "Laura" "Zak" "Chuck"
##  $ size  : chr  "small" "medium" "large"
##  $ tenure: num  6 3 9
```

Now we are clean... maybe. Our structure says our name and size columns are chr (character) and our tenure correctly shows (num) numeric. Now we want add a new column (greatness) using the values in an existing column. We can both create and define the values of the new column in one step.

```
small.data$greatness = small.data$tenure^3

str(small.data)
```

```
## 'data.frame':    3 obs. of  4 variables:
##  $ name     : chr  "Laura" "Zak" "Chuck"
##  $ size     : chr  "small" "medium" "large"
##  $ tenure   : num  6 3 9
##  $ greatness: num  216 27 729
```

```
small.data
```

```
##    name   size tenure greatness
## 1 Laura  small      6       216
## 2   Zak medium      3        27
## 3 Chuck  large      9       729
```

Now we have our 4 columns and the structure even gives us what it can display of the values. Now what if we want to add another row of data? The function rbind (row bind) makes it easy. As long as our new row is in the same order and type as the columns of our data frame, we can simply call:

```
small.data = rbind(small.data, c("Hannah", "small", 1, 1))

str(small.data)
```

```
## 'data.frame':    4 obs. of  4 variables:
##  $ name     : chr  "Laura" "Zak" "Chuck" "Hannah"
##  $ size     : chr  "small" "medium" "large" "small"
##  $ tenure   : chr  "6" "3" "9" "1"
##  $ greatness: chr  "216" "27" "729" "1"
```

```
small.data
```

```
##     name   size tenure greatness
## 1  Laura  small      6       216
## 2    Zak medium      3        27
## 3  Chuck  large      9       729
## 4 Hannah  small      1         1
```

As you can imagine, the same thing can be done for columns using cbind. Let's add another one:

```
small.data = cbind(small.data, interests=c("Bedazzling", "K-Pop", "Couponing", "Oranges"))

small.data
```

```
##     name   size tenure greatness  interests
## 1  Laura  small      6       216 Bedazzling
## 2    Zak medium      3        27      K-Pop
## 3  Chuck  large      9       729  Couponing
## 4 Hannah  small      1         1    Oranges
```

2.2.0 Reading in a data file

If you have a data file that exists (hopefully) as .csv or .txt, base R has a function that will read your data into a data frame for you. We will demo this with a .csv of fake aerial survey data. Remember that R works best (and sometimes only!) with tidy data. Your file should at least be rectangular (no loose columns, no long or dangling rows) and preferably will have column headings, though those can also be defined later. The function read.table will be the most generic and robust method, but there is also read.csv that saves a little time since you are predefining the formatting.

Plain text format is the simplest way to store text. One step up are delimited files. The most common types of delimited files and tab- and comma-delimited. Comma-delimited files are commonly saved as .csv (comma separated values). R does not care how your files are delimited, you just have to know the delimiting character(s) and pass them as arguments in the function call.

```
my.data = read.table("SomeData.txt", header = TRUE, sep = " ")
```

This tells R to find the file SomeData.txt (it will default to your working directory, but you can path it anywhere). It will open it, read the header row into the data frame my.data, then scan the file for spaces (sep = " ") and add entries between spaces to the values in each row. What could possibly go wrong with space-delimiting? Or even comma-delimiting?

To illustrate further concepts below we will be using an artificial data set I modified for this demo. It is located on the GitHub repository for this short course. To read it in, we will use read.csv, which tells R to expect a comma-delimited file. Oh, and we should also get rid of those nasty factors for now.

```
aerial=read.csv(
  "https://raw.githubusercontent.com/cfrost3/MBM_R_Short_Course/master/BLSC_2018_RawObs_Fake.csv",
                header=TRUE,
                stringsAsFactors=FALSE)

str(aerial)
```

```
## 'data.frame':    624 obs. of  16 variables:
##  $ Year      : int  2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 ...
##  $ Month     : chr  "6" "6" "6" "6" ...
##  $ Day       : int  15 15 15 15 15 15 15 15 15 15 ...
##  $ Seat      : chr  "LF" "LF" "LF" "LF" ...
##  $ Observer  : chr  "FAKE" "FAKE" "FAKE" "FAKE" ...
##  $ Strata    : chr  NA NA NA NA ...
##  $ Transect  : chr  "19" "19" "19" "19" ...
##  $ Segment   : logi  NA NA NA NA NA NA ...
##  $ Flight_Dir: chr  "88" "88" "89" "89" ...
##  $ Wind_Dir  : chr  "45" "45" "45" "45" ...
##  $ Wind_Vel  : chr  "24" "24" "24" "24" ...
##  $ Lat       : num  61.1 61.1 61.2 61.2 61.2 ...
##  $ Long      : num  -166 -165 -165 -165 -165 ...
##  $ Species   : chr  "START" "SCAU4" "LTDUF4" "BLSC4" ...
##  $ Num       : chr  "19" "1" "1" "1" ...
##  $ Obs_Type  : chr  "open" "single" "single" "pair" ...
```

Hey! We have real (fake) data! Anything stand out immediately in the structure of the file?

2.3.1 Data Types

R has 4 basic data types you will use (6 in total, actually, including raw and complex):
1. Integer (1, 2, 3, 4)
2. Numeric (2, 4, 2.4, 45.48)
3. Character ("cat", "DOG", "15d", "and so on")
4. Logical (TRUE, FALSE)

2.3.2 Data Structures

Depending on how you combine data types, you can end up with one of several data structures:
1. Vector - a collection of all one data type
2. Matrix - multidimensional collection of vectors
3. Factor - nomical set of unique values and a vector of integer indices
4. List - open format to contain data structure elements
5. Data frame - our most common, discussed above

There are many other data structures in R, including user-defined and package-defined structures. We will mostly stick to the common ones.

2.3.3 Useful Data Structure Functions

Base R comes with many quick ways to assess your data structures. We will go into more depth later, but for now, explore commands such as:

```
a = c(1,2,3,4,10.1)

length(a)  #how long is the vector?
```

```
## [1] 5
```

```
mean(a)  #mean value
```

```
## [1] 4.02
```

```
var(a)  #variance
```

```
## [1] 12.802
```

```
typeof(a)  #what data type?  (double means floating decimal numeric)
```

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

```
is.na(a)  #any NA or missing values?
```

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

```r
summary(a)  #some basic summarizing statistics
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    2.00    3.00    4.02    4.00   10.10
```

```r
max(a)  #maximum value
```

```
## [1] 10.1
```

```r
min(a)  #minimum value
```

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

```r
dim(small.data)  #what are the dimensions of a structure?
```

```
## [1] 4 5
```

```r
class(small.data)  #what class of structure is it?
```

```
## [1] "data.frame"
```

```r
names(small.data)  #what are the names of the objects that make up the structure?
```

```
## [1] "name"      "size"      "tenure"    "greatness" "interests"
```

```r
unique(small.data$size)  #what are all of the unique values in a range?
```

```
## [1] "small"  "medium" "large"
```