R dataframes–review of class material and exploration of other topics in dataframes

George Chen

October 30, 2017

```
setwd("C:\\fa17\\stat133\\stat133-hws-fall17\\post01\\report")
```

Keep in mind that I have two backslashes between each directory to prevent escape character errors in R.

Introduction

I am going to talk about data frames in R. In most programming languages data frames would not be formally named. They would be considered as an array of different types of arrays, so nothing too special. However, since R is mostly used as a statistical analysis language, there is a specific data structure: the data frame. The data frame is the most important data structure in R, and I want to improve my knowledege of data frames and teach people about how to use them.

Background

Definitions

Dataframe: Data structure in R that consists of a list of vectors. The data frame forms a sort of table-like data structure, grouping seemingly unrelated elements together. All vectors should have the same row length, even if that means adding missing values to a row. In addition, remember that lists can have elements of different types but vectors are *atomic*, and thus cannot have elements of different types.

Part 1: Data frame initialization(giving it an initial value)

Data frame's can be created using the built-in R function "data.frame". For example if my table were describing monthly attendance at Warrior's games, my data frame might look like this:

```
#create months vector
months <- c("January","February","March","April","May")

#create attendance vector
attendance <- c(100000,120000,100000,69000,99000)

#create df data frame based on months and attendance vectors
df = data.frame(months, attendance)
```

If we tried to create a data frame where months only had four elements, we couldn't succesfully execute the data.frame function

```
##months <- c("January","February","March","April")
##attendance <- c(100000,120000,100000,69000,99000)
##df = data.frame(months, attendance)
```

The error we receive is that "arguments imply differing number of rows: 4,5"

One way to fix this is just to add the month "May" in months to make the number of elements 5. We could also, add NA, a missing value to months.

```
#create months vector w' NA 5th element
months <- c("January","February","March","April",NA)

#create attendance vector
attendance <- c(100000,120000,100000,69000,99000)

#create df dataframe based on months and attendance vector.
df = data.frame(months, attendance)
```

As you can see, it doesn't matter about the content of the vectors. We just care about whether or not they have the same number of elements each.

Make sure to note also that each vector in a dataframe represents a column, and not a row. This is a common misconception for beginning R programmers.

```
#revert df to correct form.
df$months[5] <- "May"
```

```
## Warning in `[<-.factor`(`*tmp*`, 5, value = structure(c(3L, 2L, 4L, 1L, :
## invalid factor level, NA generated
```

```
#displaying the df dataframe
df
```

```
##     months attendance
## 1  January     100000
## 2 February     120000
## 3    March     100000
## 4    April      69000
## 5     <NA>      99000
```

Data frame initialization special cases:

Make sure to consider the case with **empty data frame** To create an empty data frame, declare the data frame with empty vectors that are given a type(i.e. character, logical etc.)

```
#creating an empty data frame with character, integer, and logical components
emptydf <- data.frame(a = character(), b = integer(), c = logical())
##https://stackoverflow.com/questions/10689055/create-an-empty-data-frame
```

Part 2: Viewing a dataframe

R has the following methods to view a dataframe:

- str(df): This function is an alternative to the *summary* function. str will show the framework of the dataframe(i.e. how many rows, how many columns, the column names, column types, and some of the column elements–usually first five.)
- names(df): gives set of names of all vectors in dataframe.
- summary(df): alternative to the *str* function. When applied to a dataframe, summary is applied to each column. On most vectors, the count of each value is displayed. On numeric vectors, summary will display the 5-number summary.
- head(df): displays the first few elements of each column(usually first five). head(df, n = k) will display the first k elements of each column vector.
- tail(df): display the last few elements of each column(usually last five). tail(df, n = k) will display the last k elements of each column vector.

```
str(df)
```

```
## 'data.frame':    5 obs. of  2 variables:
##  $ months    : Factor w/ 4 levels "April","February",..: 3 2 4 1 NA
##  $ attendance: num  100000 120000 100000 69000 99000
```

```
#showing off str func
names(df)
```

```
## [1] "months"     "attendance"
```

```
#showing off names func
summary(df)
```

```
##       months    attendance
##  April   :1   Min.   : 69000
##  February:1   1st Qu.: 99000
##  January :1   Median :100000
##  March   :1   Mean   : 97600
##  NA's    :1   3rd Qu.:100000
##               Max.   :120000
```

```
#showing off summary func
head(df)
```

```
##     months attendance
## 1  January     100000
## 2 February     120000
## 3    March     100000
## 4    April      69000
## 5     <NA>      99000
```

```
#showing off head func
tail(df)
```

```
##     months attendance
## 1  January     100000
## 2 February     120000
## 3    March     100000
## 4    April      69000
## 5     <NA>      99000
```

```
#showing off tail func
```

```
head(df, 3)
```

```
##     months attendance
## 1  January     100000
## 2 February     120000
## 3    March     100000
```

```
#showing off head(df, k) func
tail(df, 3)
```

```
##   months attendance
## 3  March     100000
## 4  April      69000
## 5   <NA>      99000
```

```
#showing off tail(df, k) func
```

Part 3: Accessing Values from a dataframe

Accessing values from a dataframe works similarly to accessing values from a vector. This follows the principles of subsetting, which I won't cover in detail.

To select certain columns from a dataframe use the following syntax:

```
#create vars a and b as indexing variables
a <- 1
b <- 2
#accessing cols a:b from df
df[a:b]
```

```
##     months attendance
## 1  January     100000
## 2 February     120000
## 3    March     100000
## 4    April      69000
## 5     <NA>      99000
```

where a, and b are the start and end indices of access.

This can also be accomplished by selecting all rows, for the given columns:

```
#create vars a and b as indexing variables

a <- 1
b <- 2
#accessing cols a:b from df
df[,a:b]
```

```
##     months attendance
## 1  January     100000
## 2 February     120000
## 3    March     100000
## 4    April      69000
## 5     <NA>      99000
```

To select both rows and columns:

```
#create vars row1, row2, a, and b as index variables
row1 <- 1
row2 <- 2
a <- 1
b <- 2

#accessing row1:row2 rows, and a:b cols from df
df[row1:row2,a:b]
```

```
##     months attendance
## 1  January     100000
## 2 February     120000
```

Remember you can do the same as above with logical indexing, but that is another topic altogether.

You can also access specific columns from data frames. For example in my "df" dataframe, I had a column named months. To access that column vector, I can just do: "df"$"months". In general, "df name"$"column name" will give you the column vector.

```
#accessing month col from df
df$months
```

```
## [1] January  February March    April    <NA>
## Levels: April February January March
```

Part 4: Modifying values in a dataframe.

To modify values within a column vector, access the column vector and change the value using normal R assignment operations.

```
#printing out df's attendance column vector
df$attendance
```

```
## [1] 100000 120000 100000  69000  99000
```

```
#setting the value of df attendance column vector element 1
df$attendance[1] <- 690

#printint out df's attendance column vector again
df$attendance
```

```
## [1]    690 120000 100000  69000  99000
```

I changed the attendance of the 1st month January to 690 by reassigning its value. This change is reflected in the data frame.

To change a column vector in the data frame, do the same

```
df$attendance <- rep(1,5)
#This changes df$attendance to 1 for every month

df$attendance <- attendance <- c(100000,120000,100000,69000,99000)
#I immediately change attendance back
```

Part 5: Adding columns to R data frames

To add a column to an R dataframe, just do “df name”$“proposed column name” <- column vector value For example, If I want to add a new column of best opponent(from power rankings) during each month, I can do the following:

```
df$biggest_opponent <- c("Cavaliers", "Rockets", "Pelicans", "Cavaliers", "Cavaliers")
#adding biggest_opponent column vector to df
```

Now the df dataframe will have a new column.

Part 6: Aggregation with data frames

The aggregate function is a function that allows you to do operations on certain parts of a data frame, and not others. aggregate in its most basic form takes in 3 parameters. aggregate(“df column name”, by = list(“display_name” = “dataframe$actual_name”, FUN = “function name”))

aggregate will compute the function give by “function name” for every single element in the column vector with name “df column name”. It will then display a table with column header display_name for the df column vector “dataframe$actual_name” and a column header x for “df column name”.

Let’s say I wanted to make the dataframe ‘df’ include the next year’s data as well. I can expand the months, attendnace, and biggest_opponent vectors to account for this. Now each vector should have 10 elements.

```
#creating a temp_df data frame
temp_df <- data.frame(
 months <- c("January","February","March","April","May", "January", "February", "March", "April", "May"),
attendance <- c(100000,120000,100000,69000,99000,110000,122000,102000,69900,100000) ,
biggest_opponent <- c("Cavaliers", "Rockets", "Pelicans", "Cavaliers", "Cavaliers", "Lakers", "Celtics", "Lakers",
"Celtics","Lakers")
)

#temp_df has column vectors of the same name as df, except that temp_df has two years worth of data rather than 1.
#Note I am imagining an NBA where the season lasts from January to May
```

I could use the aggregate function to show attendance for each month over the 2 year period.

```
aggregate(temp_df$attendance,by = list(months = temp_df$months), FUN = sum)
```

```
##     months      x
## 1    April 138900
## 2 February 242000
## 3  January 210000
## 4    March 202000
## 5      May 199000
```

```
#FUNction acts on the attendance variable to show attendance in each Month over the 2 year period.
```

You can also aggregate with formula notation.
The basic syntax that you need to know is that: y ~ x means that y is dependent on x.

To do the same aggregation as above, just do:

```
aggregate(attendance ~ months,data = temp_df, FUN = sum)
```

```
##     months attendance
## 1    April     138900
## 2 February     242000
## 3  January     210000
## 4    March     202000
## 5      May     199000
```

```
#do the same thing as above, except with 'attendance' dependent on 'months'.
```

Two more tips: You can create your own function. To create a function with parameter x that shows the minimum value of each common value that shows up, do: “function(x) min(x)” A common value would be a month that shows up twice in the “months” column. Every month in the data frame “df” is a common value since each month shows up twice, once for each year that we have data for.

You can also aggregate by multiple data frame columns. Just add a comma and repeat the same format.

```
#aggregate by multiple data frame columns
#this will cause months and bigget_opponent to display in the ouput as columns instead of just months
aggregate(temp_df$attendance,by = list(months = temp_df$months, biggest_opponent = temp_df$biggest_opponent), FUN
= sum)
```

```
##      months biggest_opponent      x
## 1     April        Cavaliers  69000
## 2   January        Cavaliers 100000
## 3       May        Cavaliers  99000
## 4     April          Celtics  69900
## 5  February          Celtics 122000
## 6   January           Lakers 110000
## 7     March           Lakers 102000
## 8       May           Lakers 100000
## 9     March         Pelicans 100000
## 10 February          Rockets 120000
```

Part 7: Using attach to simplify data frames

If you wanna avoid using the df$“column_name” syntax, then I have the solution for you! If you are just as lazy as me do the following:

```
#attaching df
attach(df)
```

```
## The following objects are masked _by_ .GlobalEnv:
##
##     attendance, biggest_opponent, months
```

This now means that you can access columns within df just by writing their names down.

```
#printing out attendance data frame
attendance
```

```
##  [1] 100000 120000 100000  69000  99000 110000 122000 102000  69900 100000
```

To detach data frames, just do:

```
#detaching dataframe
detach(df)
```

Remember it is good practice to not attach multiple data frames at the same time. You can, but sometimes the two data frames will overlap on column vector names. This can be confusing to you as a user when determing which variable you are accessing when you write down the name of the common column vector.

```
#attaching two data frames
attach(df)
```

```
## The following objects are masked _by_ .GlobalEnv:
##
##     attendance, biggest_opponent, months
```

```
attach(temp_df)
```

Since df and temp_df have the same column_vector names for every column_vector, the attached variables are all temp_df’s since it was attached last. See how it can be confusing?

```
#printing out 'months'
months
```

```
##  [1] "January"  "February" "March"    "April"    "May"      "January"
##  [7] "February" "March"    "April"    "May"
```

```
#printing out 'attendance'
attendance
```

```
##  [1] 100000 120000 100000  69000  99000 110000 122000 102000  69900 100000
```

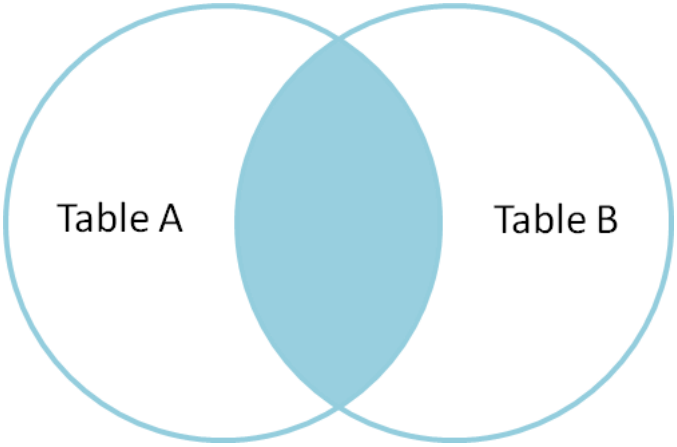
```
#printing out 'biggest_opponent'
biggest_opponent
```

```
##  [1] "Cavaliers" "Rockets"   "Pelicans"  "Cavaliers" "Cavaliers"
##  [6] "Lakers"    "Celtics"   "Lakers"    "Celtics"   "Lakers"
```

```
#detaching both data frames
detach(df)
detach(temp_df)
```

Part 8: Merging data frames

Let me preface this by saying that this will be pretty complicated. Data frame merges are like SQL Joins. In regular joe lingo that means joining tables together. There are *suprise!*, not one, not two but four main ways to join two dataframes in R. The most important one, the natural join(inner join), is the one I will talk about today.



Natural Join

As you can see there are two tables above A and B and the intersection of A and B is shaded in blue. The natural join is just the intersect of A and B. A and B must have a column of the same name and data type in common. In the case that this is true, then the natural join will take all such columns. R will iterate through each common column in a and b. To make the finished merged data frame, R will only keep elements in a given column if it is common to both a and b. Otherwise, it will throw the whole row out.

Let's say we are given two data frames that display weather in California

```
#creating data frame 'a' with column vectors of 'month', and 'temp'
a <- data.frame(
    month = c("Jan", "Feb", "Mar", "Apr"),
    temp = c(95, 96, 99, 100)
)

#creating data frame 'b' with column vectors of 'month', and 'temp' but with different elements inside
#these column vectors
b <- data.frame(
    month = c("Feb", "Mar", "Oct"),
    temp = c(99, 99, 99)
)

#printing 'a', and 'b' out
a
```

```
##   month temp
## 1   Jan   95
## 2   Feb   96
## 3   Mar   99
## 4   Apr  100
```

```
b
```

```
##   month temp
## 1   Feb   99
## 2   Mar   99
## 3   Oct   99
```

Displaying those data frames shows us that a and b have two of the same column names, “month” and “temp”.

Merging those two with an inner join based on the given explanation of inner join will give us a new data frame that looks like this:

```
#merging data frame 'a', and 'b'
merge(a , b)
```

```
##   month temp
## 1   Mar   99
```

Why does the new data frame look like this? Let’s do this in steps.

1. Look for the columns with common names in a and b. That’s just “month”, and “temp”
2. Iterate through each common column in a and b. To make the finished merged data frame, only keep elements in a given column if it is common to both a and b. Otherwise, throw the whole row out.

Note: if we changed the temperature for March in b to 100, then nothing would be returned as the result of merge(a,b)

```
#changing b's temp column vector to have a 2nd element w' value 100
b$temp[2] <- 100

#merging the two data frames
merge(a,b)
```

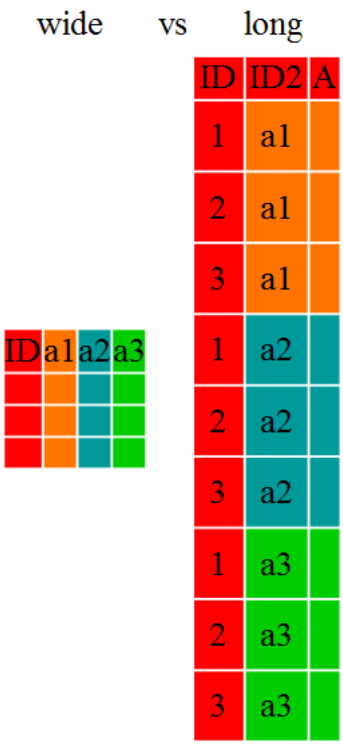
```
## [1] month temp
## <0 rows> (or 0-length row.names)
```

```
#resetting b's temp column vector to have a 2nd element w' value 99
b$temp[1] <- 99
```

Part 9: reshaping data frames

There are two main types of data table formats: long, and wide.

Long format data has repeats of column values, and may be considered “too long” for viewing purposes. Wide data format is considered better looking. Long data format should be used for data processing because most R functions need data to be in that format.



Long and Wide Data formats

As you can see the long data format has multiple repeats of values of both ID, and ID2. Therefore in the wide format, we can succesffuly make ID, a1, a2, and a3 column vectors, making the table wider but better looking.

Here is a r execution of the above example. To make the code easier, I am using the package ‘reshape2’. I really like Hadley Wickham’s packages by the way. He is the best! :

```
#installing reshape2 package
install.packages("reshape2",repos = "http://cran.us.r-project.org")
```

```
## Installing package into 'C:/Users/George chen/Documents/R/win-library/3.4'
## (as 'lib' is unspecified)
```

```
## package 'reshape2' successfully unpacked and MD5 sums checked
##
## The downloaded binary packages are in
##  C:\Users\George chen\AppData\Local\Temp\Rtmpy2iVQC\downloaded_packages
```

```
#load the package 'reshape2'
library(reshape2)
```

```
## Warning: package 'reshape2' was built under R version 3.4.2
```

```
#create long format dataframe 'long_dat'
long_dat <- data.frame(
    ID <- c(1, 2, 3, 1, 2, 3, 1, 2, 3),
    ID2 <- c("a1", "a2", "a3", "a1", "a2", "a3", "a1", "a2", "a3"),
    A <- rep("", 9)
)
#'str' function on 'long_dat'
str(long_dat)
```

```
## 'data.frame':    9 obs. of  3 variables:
##  $ ID....c.1..2..3..1..2..3..1..2..3.                            : num  1 2 3 1 2 3 1 2 3
##  $ ID2....c..a1....a2....a3....a1....a2....a3....a1....a2....a3..: Factor w/ 3 levels "a1","a2","a3": 1 2 3 1 2
3 1 2 3
##  $ A....rep.....9.                                               : Factor w/ 1 level "": 1 1 1 1 1 1 1 1 1
```

```
#'dcast'ing long_dat into 'wide_dat'
wide_dat <- dcast(long_dat, ID ~ ID2)
```

```
## Using A....rep.....9. as value column: use value.var to override.
```

```
## Aggregation function missing: defaulting to length
```

```
#'str' function on 'wide_dat'
str(wide_dat)
```

```
## 'data.frame':    3 obs. of  4 variables:
##  $ ID: num  1 2 3
##  $ a1: int  3 0 0
##  $ a2: int  0 3 0
##  $ a3: int  0 0 3
```

Note: Don't worry about the warnings. They are not important.

By using the dcast function I was able to convert the data from wide to long format. Hooray! Yipee! dcast takes in two params: the data frame, and the formula. I let ID be dependent on ID2 in function notation, therefore causing ID2 to be disbanded so that a1,a2, and a3 could now be column vectors in the wide_dat data frame. A was ignored because I didn't do anything with it in dcast.

In general to use dcast put in a data frame as a first parameter, and a formula such as I had relating one variable in your data frame and another. At least one should have repeats of values, otherwise there is no point in changing from long to wide storage.

Conclusion

Overall data frames are an easy to start, but hard to master data structure. Declaring a data frame is simple if you already know of the vector and list data structures already. However, there are complex operations such as reshaping, and merging data frames that are very difficult to master. Even the examples I showed, which are about middle-level took 30 minutes to figure out. Data frames have a wide variety of uses beyond the basic ones we learned in the classroom. Now go out there and use them friends!

Resources:

Coding: http://www.r-tutor.com/r-introduction/data-frame https://www.tutorialspoint.com/r/r_data_frames.htm
https://onlinecourses.science.psu.edu/stat484/node/247
https://en.wikibooks.org/wiki/R_Programming/Working_with_data_frames#Creating_a_subset_of_the_data
https://www.datacamp.com/community/tutorials/15-easy-solutions-data-frame-problems-r
https://stackoverflow.com/questions/10689055/create-an-empty-data-frame
https://chemicalstatistician.wordpress.com/2013/08/19/exploratory-data-analysis-useful-r-functions-for-exploring-a-data-frame/
https://www.rdocumentation.org/ http://www.dummies.com/programming/r/how-to-use-the-formula-interface-in-r/
http://www.dummies.com/programming/r/how-to-use-the-merge-function-with-data-sets-in-r/ https://www.w3resource.com/sql/joins/natural-join.php https://stackoverflow.com/questions/34590173/long-and-wide-data-when-to-use-what

Images: https://blog.codinghorror.com/content/images/uploads/2007/10/6a0120a85dcdae970b012877702708970c-pi.png
https://swcarpentry.github.io/r-novice-gapminder/fig/14-tidyr-fig1.png