Usefulness of R for Psychology

Fábio A. Silva

2021-10-15

Contents

Chapter 1

Introduction

What you'll learn here:

- Extract data from single and multiple output files (in different formats).
- Mess around with the data, changing the dataframe to your preference.
- Extract target measures (accuracy, response times, sensitivity, criterion, etc.)
- Visualize the data as it comes in.
- Create multiple parameters at the beginning that allow you to change minor details while running the script.
- Extract data for mixed models.
- Produce simple analysis (for a more detailed vision on analysis, see P2).
- ...

Chapter 2

OPENING FILES

Before opening the data
First, you need to tell R where the files are located. He is lazy that way, and will not look everywhere on your computer for them. So, tell him by using the command `setwd()`.

Reading the data
So, by now you should have a general idea on how to import data files to R. Here we just remind you how to read some of the most common formats, atlthough R can read just about anything (just google read "file type" in R on google). Be sure to install the `xlsx` and `haven` packages to open Excell and SPSS files, respectively. Additionally, there are multiple ways to read the same file. Some will be built in R itself, others will require external packages. This is important to know because some functions, although working, may be outdated or just give you some sort of weird error. Maybe you can overcome this by using a different package.

If you want to import/read an Excel file, just use:
```
read.xlsx(file = 'example.xlsx', sheetName = 'page_1', header =
TRUE)
```
If a text file:
```
read.delim(file = 'example.txt', header = TRUE, sep = ',', dec =
'.')
```
CSV:
```
read.csv(file = 'example.csv', header = TRUE, sep = ',', dec =
'.')
```
SAV (SPSS):
```
read_sav(file = 'example.sav')
```

Managing your imported data
To have your data in your environment, so that you can mess with it, you should assign your `read` command to a variable (object). Lets say you do the

following `df <- read.delim(file = 'example.txt', header = TRUE, sep = ',', dec = '.')`. Now, your df object is the dataframe containing that imported data set.

Possible problems
You may encounter several problems. Perhaps one of the most common problems is the something like "the number of columns is superior to the data". This probably has to due with R separating columns where it shouldn't or making more columns than it should. You can fix this perhaps by making sure the `sep` command is specifying the exact separator of the file. It helps to open the file with excel for instance, and double check the separator. Also be sure that you have the decimal symbol is different from the new columns symbol. For instance, sometimes R reads the .csv file (which means comma separated file) and you have commas as decimals. This creates way to many columns that mismatch the number of headers present.
There will surely be more problems, but you can find a way around them by using google.

Checking the data
After you've opened the data, you should take a peak at it. There's several ways of doing this such as `head(df)` or some others I'm not recalling at the moment. Lets see bellow.

2.1 Opening multiple files

Lets say you have a folder, lets assume is named "data", and in there you have all your data files (files from each participant). Ideally, to analyze and view the data as a whole, you would want to import all of the data files, and then merge them into a big data file, containing all the participants (identified accordingly). + Here's a snippet of code that would allow you to do that. Beware though, any file that matches the criteria (in this case ".csv" files) will be gathered from the folder (your working directory).

Firstly, lets gather the names of the files in our directory that we wanna import. P.s. Ignore the warning, it just tells you that the directory is only changed for this chunk and then it goes back to normal.

```
# Setting our working directory for this chunk (in this case our data folder)
setwd('data')

# Listing all txt files in folder
list_of_files <- list()

# Searches for ".csv" files in the folder
list_all_data <- dir(pattern='.csv')
```

```r
# For each ".csv" file, append this file to a list that will contain all ".csv" file names.
for (file in list_all_data){
  list_of_files <- c(list_of_files, file)
  cat('\nFile processed:', file)
}
```

```
##
## File processed: Participant_1.csv
## File processed: Participant_10.csv
## File processed: Participant_2.csv
## File processed: Participant_3.csv
## File processed: Participant_4.csv
## File processed: Participant_5.csv
## File processed: Participant_6.csv
## File processed: Participant_7.csv
## File processed: Participant_8.csv
## File processed: Participant_9.csv
```

```r
# Organizing the files
list_of_files <- as.character(list_of_files)  # To character
#list_of_files <- mixedsort(sort(list_of_files))  # Sorting them by name
list_of_files <- as.list(list_of_files)  # Transforming it to list

# How many files were processed?
cat('\nTotal number of files processed:', length(unlist(list_of_files)))
```

```
##
## Total number of files processed: 10
```

Then we need to create a function that iterates over all of this file names. It will need to import each file (by their name) and append them to a general dataframe.

```r
# Setting the working directory to the data folder for this chunk as well
setwd('data')

# Create our function with "file_name" as input.
import_data <- function(file_name){
  df <- read.csv(file_name, header=TRUE, encoding = 'utf-8', sep=',')  # Import one file and name
  df_part <<- df  # Make this file available outside the function
}

# We create an empty dataframe where all individual dataframes will be merged
```

```
df_all <- data.frame()

# Create a index for number of name in list
ind <- 1  # Begins with file name 1.

# Run the function
for (x in 1:length(list_of_files)){
  file_name <- unlist(list_of_files[ind])  # Select one name from our list
  import_data(file_name) # Run our function
  ind <- ind + 1  # Move index of number along.
  df_all <- rbind(df_all, df_part)  # Adds the individual dataframe to our "big" dataf
}

# Seeing our dataframe (you can see it all by clicking on it or using "View(df_all)").
head(df_all)
```

```
##   Participant_ID   Condition       RT
## 1              1 Condition_1 1.478651
## 2              1 Condition_1 1.495121
## 3              1 Condition_1 1.506271
## 4              1 Condition_1 1.561987
## 5              1 Condition_1 1.508967
## 6              1 Condition_1 1.512732
```

2.2 Exporting files

Aside from important, sometimes we also want to export the files we created/modified in R. We can do this with several commands, but perhaps the simpler ones are:
`write.table(x = df, file = 'namewewant.txt', sep = ',', dec = '.')` - We are export the `df` dataframe, to a not existing file with a name "namewewant.txt", that is separated by commas and has "." for decimal points. We can also export to an existing data file, and ask for `append = TRUE`.

`write.csv(x = df, file = 'namewewant.txt')` - Same thing as above, but instead creates a ".csv" file.

As an example, lets export the dataframe we created in the chunks above.

```
write.csv(x = df_all, file = 'some_data.csv')
```

Chapter 3

MANAGING THE DATA

Now, in R you can manage your dataframe as you please. You can do anything. And I truly mean anything. Anything you can do in Excel and then some.

3.1 Classes, Types and Structures

The data you imported can be in a wide range of classes (types). There are 3 basic types of classes, built-in (different functions can use more), you need to be aware (altough there are more). These are:

- *character*: strings (words), such as `"hello"` or `"hi123"`.
- *numeric*: any type of number, such as `2`or `30.4`.
- *logical*: The true or false values. `TRUE` or `FALSE`.

You can ask R about what type of object it is by using the `class(object)` command or the `typeof(object)`

R also has different data structures. The ones worth talking about here are:

- *atomic vector*: Basically every R data structure. A vector could be a character or an numeric object, for instance.
- *lists*: a list of objects. Can be created by using `list()`. You can retrieve the value by using `[[]]`.
- *matrix*: are like tables.
- *data frame*: fancy matrices (more common)
- *factor*: more of a type of class than anything. This object is a factor with levels.

- *tibble*: a special data frame from tidyverse

https://swcarpentry.github.io/r-novice-inflammation/13-supp-data-structures/

3.2 Columns

Lets start by some simply manipulations. Lets say you want to change column names. Ideally, I would avoid spaces in the headers (and overall actually) but you do as you please.

```
df <- iris  # mtcars is a built-in dataset. Just imagine I'm reading from a file
# Option 1
colnames(df) <- c('Colname 1', 'Colname 2', 'Colname 3', 'Colname 4', 'Colname 5')

# Option 2
names(df) <- c('Colname 1', 'Colname 2', 'Colname 3', 'Colname 4', 'Colname 5')

# Or just change a specific column name
colnames(df)[2] <- 'Colname 2 - New'

# Finale result
head(df)
```

```
##   Colname 1 Colname 2 - New Colname 3 Colname 4 Colname 5
## 1       5.1             3.5       1.4       0.2    setosa
## 2       4.9             3.0       1.4       0.2    setosa
## 3       4.7             3.2       1.3       0.2    setosa
## 4       4.6             3.1       1.5       0.2    setosa
## 5       5.0             3.6       1.4       0.2    setosa
## 6       5.4             3.9       1.7       0.4    setosa
```

We can also change the order of the columns.

```
df <- df[,c(3,2,1,4,5)]
```

We can sort by a specific (or multiple columns).

```
df <- iris  # Just restoring the dataframe to be less confusing


df <- df[order(df[, 1]), ]  # Orders by first column
df <- df[order(-df[, 1]), ]  # Orders by first column descending

df <- df[order(-df[, 1], df[, 3]), ]  # Orders by first columns descending and then by
```

We can create new columns.

```
new_data <- rep('New Data', nrow(df))  # Creating new irrelevant data

df$NewColumn <- new_data  # Added this data (data must have same length as dataframe!)

head(df)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species NewColumn
## 132          7.9         3.8          6.4         2.0 virginica  New Data
## 136          7.7         3.0          6.1         2.3 virginica  New Data
## 118          7.7         3.8          6.7         2.2 virginica  New Data
## 123          7.7         2.8          6.7         2.0 virginica  New Data
## 119          7.7         2.6          6.9         2.3 virginica  New Data
## 106          7.6         3.0          6.6         2.1 virginica  New Data
```

How to remove columns.

```
df$Petal.Length <- NULL
# or
df <- within(df, rm(Sepal.Length))
```

We can create and transform the columns.

```
df <- iris

df$Sepal_Area <- df$Sepal.Length * df$Sepal.Width  # Creating new variable with is the multiplica

head(df)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal_Area
## 1          5.1         3.5          1.4         0.2  setosa      17.85
## 2          4.9         3.0          1.4         0.2  setosa      14.70
## 3          4.7         3.2          1.3         0.2  setosa      15.04
## 4          4.6         3.1          1.5         0.2  setosa      14.26
## 5          5.0         3.6          1.4         0.2  setosa      18.00
## 6          5.4         3.9          1.7         0.4  setosa      21.06
```

```
df$Sepal_Area <- round(df$Sepal_Area, 1)  # Transforming existing variable, making it just 1 dec

head(df)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal_Area
```

```
## 1          5.1         3.5          1.4         0.2  setosa       17.8
## 2          4.9         3.0          1.4         0.2  setosa       14.7
## 3          4.7         3.2          1.3         0.2  setosa       15.0
## 4          4.6         3.1          1.5         0.2  setosa       14.3
## 5          5.0         3.6          1.4         0.2  setosa       18.0
## 6          5.4         3.9          1.7         0.4  setosa       21.1
```

3.3 Rows

Altering specific rows is a bit trickier. Nevertheless, this is usually less relevant, since we usually just want to change or apply a condition to an entire column. Having said this, here's some relevant commands.

Say you want to alter rows that meet a condition.

```
df$Sepal.Length[df$Sepal.Length <= 5] <- 0  # Any row in the Sepal.Length column less

df$Sepal.Length[df$Sepal.Length == 7.9] <- 8  # Changing rows with 7.9 to 8.
```

How to create a new entry (i.e., row).

```
row <- data.frame(5.6, 3.2, 1.9, 0.1, 'new_species', 10000)  # Create a new row (all c
colnames(row) <- colnames(df)
df <- rbind(df, row)

tail(df)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width     Species Sepal_Area
## 146          6.7         3.0          5.2         2.3   virginica       20.1
## 147          6.3         2.5          5.0         1.9   virginica       15.8
## 148          6.5         3.0          5.2         2.0   virginica       19.5
## 149          6.2         3.4          5.4         2.3   virginica       21.1
## 150          5.9         3.0          5.1         1.8   virginica       17.7
## 151          5.6         3.2          1.9         0.1 new_species    10000.0
```

How to delete a row.

```
df <- df[-c(151, 152),]
```

3.4 Tidyverse

Tidyverse allows us to do lots of things in a useful and simple way. Lets explore some of its several commands. More than ever, tidyverse is useful here. Lets load it first.

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.4     v dplyr   1.0.7
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   2.0.1     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Now, lets say we want to **filter** the dataframe.

We can **arrange** the dataframe as we wish. We can sort by just 1 column or more. In the latter case the second, third and so on variables will break the ties. Missing values are sorted to the end.

Another useful trick is to **select** columns. With this command we can select the columns we want, or do not want.

To create new columns, we can also use **mutate**. This a versatile command that allows you to do several things. Here are a bunch of examples:

We can also create summaries of the data with the command **summarise()**.

```
head(df)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal_Area
## 1          5.1         3.5          1.4         0.2  setosa       17.8
## 2          0.0         3.0          1.4         0.2  setosa       14.7
## 3          0.0         3.2          1.3         0.2  setosa       15.0
## 4          0.0         3.1          1.5         0.2  setosa       14.3
## 5          0.0         3.6          1.4         0.2  setosa       18.0
## 6          5.4         3.9          1.7         0.4  setosa       21.1
```

```
# Summarising mean Sepatl.length by species
df %>%
  group_by(Species) %>%  # Grouping by this variable
  summarise(Mean_By_Species = mean(Sepal.Length))
```

```
## # A tibble: 3 x 2
##   Species    Mean_By_Species
##   <fct>                <dbl>
```

```
## 1 setosa               2.34
## 2 versicolor           5.64
## 3 virginica            6.49
```

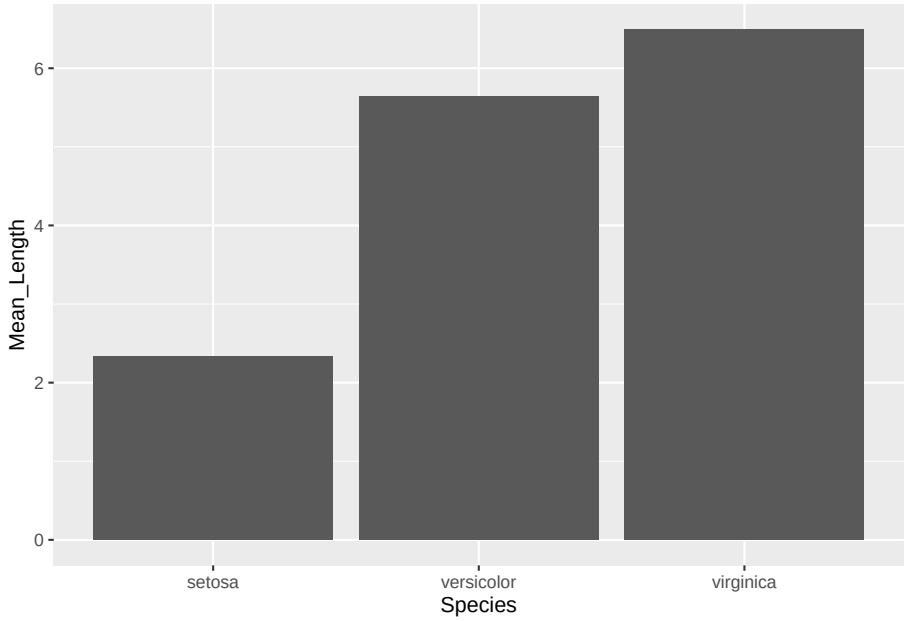
You can group by more than one factor and ask for other summaries, such as median, sd, and other basic operations. For instance:

```
df %>%
  group_by(Species) %>%
  summarise(count = n())  # Gives you the number of entries in each group
```

```
## # A tibble: 3 x 2
##   Species    count
##   <fct>      <int>
## 1 setosa        50
## 2 versicolor    50
## 3 virginica     50
```

You can then build operations on top of your summaries (like mutations or plots)

```
df %>%
  group_by(Species) %>%
  summarise(Mean_Length = mean(Sepal.Length)) %>%
  ggplot(aes(Species, Mean_Length)) +
  geom_col()
```



3.5 Missing Values

We have several ways of dealing with missing values `NA` (which mean "Not Available" by the way). We can remove then, or omit them, depending on the situation. The important thing to note is that you ask R for certain commands. For instance, if you ask a mean of a column that contains `NA` the result will be `NA`. You can either a) specify `na.rm = TRUE` on the command (if the specific command allows you to do so), or just remove the `NA` values prior to running the command.

```
df$Sepal.Length <- as.numeric(df$Sepal.Length)
# Asking a mean with NA values
df %>%
  summarise(Mean = mean(Sepal.Length))
```

```
##   Mean
## 1   NA
```

```
# Removing NAs when asking the mean
df %>%
  summarise(Mean = mean(Sepal.Length, na.rm=TRUE))
```

```
##       Mean
## 1 4.822667
```

```
# Removing NAs then asking for the mean
df %>%
  filter(!is.na(Sepal.Length)) %>%
  summarise(Mean = mean(Sepal.Length))
```

```
##       Mean
## 1 4.822667
```

3.6 Counts

Already mentioned above. Gives you the number of entries.

```
df %>%
  group_by(Species) %>%
  summarise(count = n())
```

```
## # A tibble: 4 x 2
##   Species     count
##   <fct>       <int>
## 1 setosa         50
## 2 versicolor     50
## 3 virginica      50
## 4 new_species     1
```

3.7 Ungrouping

Lastly, you can use `ungroup()` in a pipe to ungroup whay you've did.

See tidyverse book page 107 for Exploratory Data Analysis