R Workshop: Getting Started in R

Dr. Ashutosh Singh

Winter 2024

Contents

1 What will I learn?

Welcome to this introductory tutorial for Marketing Analytics! Throughout the Marketing Analytics module, we will apply analytic tools to strategic marketing questions using R. The current tutorial on 'Getting Started in R' aims to prepare you for these upcoming applications, as there are a few things you should do and know before you can start using R for marketing analytics.

By the end of this tutorial, you will:

- have installed R and RStudio
- be able to create and save your own script files
- know how to install and load R packages
- be able to import data to R

2 FAQ

What is R?

R is a free, open-source software and programming language for statistical computing. R has become one of the dominant software packages for data analysis and is widely used by leading organizations worldwide.

What is RStudio?

Rstudio is a graphical front-end to R and provides a user-friendly environment for using R. We will use RStudio throughout the module.

Why do I have to learn R? Why can we not use Excel or SPSS?

A key objective of the Marketing Analytics Module is to provide you with analytic skills to help you make strategic marketing decisions.

Marketers increasingly make use of larger and more varied (e.g. text data) data sources, as well as more advanced algorithms (e.g. machine learning) to deal with these data. These data and algorithms can not be handled well by tools like Excel and SPSS.

Moreover, leading companies increasingly make data-driven strategic decisions, and they indicate that they are increasingly working with programming languages like R to do so. Throughout this module, you will build analytical skills that are highly valued by top employers.

What if I do not have any background in statistics or programming?

There is absolutely no need to worry about this! This module does not assume any background in programming or R. This module is designed for students without any background in statistics and programming, and you will develop all required skills within this module.

What can I expect to learn?

It is important to note that we will be taking a practical approach to using R. The aim is to teach R in an accessible way that allows you to solve marketing problems, not to convert you into a statistician or a programming expert. The focus will lay on learning how to apply R code in order to answer frequently asked marketing questions.

You may find using R difficult and time-consuming initially, but after carefully working through the R applications, you will become increasingly proficient in using R and develop new analytical skills along the way. And we're, of course, happy to help you in case you experience any difficulties along the way.

R is an open-source language and has a big community where you can find answers to almost every question. Learn to search in R community.

Additionally, you will find some statistical contents that are hard to understand. It is common, especially for students without any statistical background. We do not expect you to understand all of the statistics

content within one single module. You will need to spend more time in self-leaning if you want to grasp the mathematics and statistics behind the models fully.

3 Getting Started

3.1 Download and install R and Rstudio

You will first have to download R and then Rstudio.

To download R, please follow the steps given below:

1. Go to "https://cran.r-project.org/mirrors.html" and click the appropriate link, e.g. "https://cran.ma.imperial.ac.uk/", or " https://mirrors.tuna.tsinghua.edu.cn/CRAN/"
2. Select the appropriate operating system for your device (e.g. "Download R for Windows").
3. Download the latest version of R. Note that on Windows, you will first have to click on "base" after selecting your operating system and then click on "Download R 4.3.2 for Windows". On Mac, click on "R-4.3.2.pkg" under "Latest Release".
4. Open the downloaded file and follow the instructions to complete the installation.

If you already have R installed on your device, I would still recommend downloading R again if you do not have the latest version installed. This is because some of the functions we will be using may not work on older versions of R. R will, by default, use the latest version installed on your device, but you will still have the option of using earlier versions if you wish to do so.

To download RStudio, please follow the below steps:

1. Go to https://rstudio.com/products/rstudio/download/
2. We will use the "Rstudio Desktop" Open Source License. Click on "Download" for this version.
3. Under "All Installers and Tarballs", select the appropriate operating system for your device and download the appropriate file.
4. Open the downloaded file and follow the instructions to complete the installation.

3.2 Initial steps

3.2.1 Opening RStudio

From your start menu, you should now be able to find "Rstudio".

You should then see the window splitting into different panels:

- Top left panel: *source*, where you will write/view R scripts. Some outputs (such as if you view a dataset using View()) will appear as a tab here. If you do not see a window with a script file (top left window), you can open one by clicking on **File**, **New file**, **R script**.
- Bottom left panel: *Console/Terminal*. This is actually where you see the execution of commands. This is the same display you would see if you were using R at the command line without RStudio. You can work interactively (i.e. enter R commands here), but for the most part, we will run a script (or lines in a script) in the source pane and watch their execution and output here. The "Terminal" tab gives you access to the BASH terminal (the Linux operating system, unrelated to R).
- Top right panel: *Environment/History*, where RStudio will show you what datasets and objects (variables) you have created and which are defined in memory. You can also see some properties of objects/datasets, such as their type and dimensions. The "History" tab contains a history of the R commands you have executed R.
- Bottom right panel: *Files/Plots/Packages/Help*. This multipurpose panel will show you the contents of directories on your computer. You can also use the "Files" tab to navigate and set the working directory. The "Plots" tab will show the output of any plots generated. In "Packages", you will see

what packages are actively loaded, or you can attach installed packages. "Help" will display help files for R functions and packages.

3.2.2 Setting your working directory

What is a working directory? The working directory will store your files (e.g. your code and data). Before continuing to work with RStudio, you should set up your working directory.

How do I set my working directory? I would recommend creating a dedicated folder for all R materials on your computer, for instance, called 'Marketing Analytics 2024' (or any other name you would like to use).

Next, in RStudio, you can click on *Session* in the menu bar at the top of your screen. Then you go to *Set Working Directory*, *Choose Directory*, and finally, choose the folder 'Marketing Analytics 2024' that you created on your computer.

Note that you will have to set your working directory every time you start Rstudio! To make this easier, you can set your working directory within your script file. You can copy the line of code that was generated just now in your Console (starting with "setwd") and paste this into your script file. Make sure not to include the ">" sign before "setwd" in the script file.

3.2.3 Saving your script

The code that you enter directly into the console (bottom left window in R) will not be saved by R, so it is best to work in script files. The code that you enter into a script file can be saved as a reproducible record of your analysis.

To save your current script, go to **File**, and click on **Save**, and you can save your script on your device (e.g. in your folder "Marketing Analytics 2024").

3.2.4 Entering and running codes from your script

Let's start with a simple exercise and use R as a basic calculator.

Imagine company ABC expects total yearly revenue of £3,750,000 for the current year and plans to spend 15% of its revenues on marketing for the upcoming year. You want to know how much spending that will amount to next year. We can let R do the calculation for us. To do so, you have to enter the code below into your script. The " * " sign indicates multiplication in R.

To run the code, select this line of code. Then, click on the 'Run' button in the top right corner of the script window. Alternatively, as a shortcut, you can also press Ctrl+R or Ctrl+Enter on Windows and Cmd+Enter on a Mac. As an alternative to selecting lines of code, you can also place your cursor at the end of a line of code you wish to run and then click on 'Run' or use the shortcut.

```
3750000*0.15
```

```
## [1] 562500
```

In your console, the result appears. As you can see, company ABC can expect a total marketing budget of 562,500 pounds for the upcoming year.

3.2.5 Annotating your script

As you will gradually be working with more extensive files of code, it is pretty useful to annotate your script. This can help you identify and segregate distinct components in your code. You can do so by using "#" at the beginning of a line of code. For instance:

```
# Getting started in R
    # Exercise 1
      3750000*0.15
```

```
## [1] 562500
```

3.3 Installing and loading packages

What are packages, and why do I need them?

R automatically loads several functions to do basic operations, but packages provide extra functionality. They typically consist of many functions that can handle specific tasks. For example, a package could provide functions to run a machine-learning algorithm, as we will do later in this course.

How do I install packages?

To install a package, you can type *install.packages("package name goes here")* in your script. To install the package, run this line of code.

Let's try installing the popular *tidyverse* package, one of the packages we will use in this module. Install the package by running the following line of code:

install.packages("tidyverse") # this line installs the tidyverse package

After installing a package, you will need to load it to actually work in R.

library(tidyverse) # this line loads the tidyverse package

- Note that you only need to install packages once on your computer, but you just need to load them every time you open R.
- After installing and loading the *tidyverse* package, you will be able to use the functions that are included in the tidyverse package.
- Installing a package will typically produce a lot of output in the console, which is not important in itself.

How do I know if I have successfully installed a package?

You can check whether you have successfully installed a package by loading the package. If you try to load a package that has not been successfully installed, you will get an error in your Console. For example, suppose you try to load a package called 'marketing':

library(marketing) # load the (non-existent) package 'marketing'

Error in library(marketing): there is no package called 'marketing'

R is telling us that there is no package called 'marketing', because it is not downloaded (in fact, it does not exist). If you get this error while installing the tidyverse package, this means the installation was unsuccessful.

What should I do if the installation of a package is unsuccessful?

- Check carefully to make sure you have not made any spelling mistakes, use quotation marks (" ") where appropriate, and have not used any capital letters as R is case sensitive.
- Alternatively, you can try to install the package manually by using the Packages tab on the bottom right panel of your screen. Click on *Install*, insert the name of the package under *Packages*, and click *Install*. After installing packages manually, you still need to load the package like before.
- For the remainder of this reading, successful installation of the tidyverse package is not needed, so for now, you can continue without having the package installed.

4 Basic objects

4.1 Vectors

The simplest R object is a vector, a one-dimensional collection of data points of a similar kind (such as numbers or text).

```
x <- c(2, 4, 6, 8) # create a vector x using combine function c()
```

Vectors commonly comprise numeric data, logical values, or character strings.

```
xNum <- c(1, 3.14159, 5, 7)
xLog <- c(TRUE, FALSE, TRUE, TRUE)
xChar <- c("foo", "bar", "boo", "far")
xMix <- c(1, TRUE, 3, "Hello, world!")
xNum
```

```
## [1] 1.00000 3.14159 5.00000 7.00000
```

These four objects, *xNum, xLog, xChar,* and *xMix*, have different data *types*.

Vectors may be appended to one another with c():

```
x2 <- c(x, x)
x2
```

```
## [1] 2 4 6 8 2 4 6 8
```

Indexing denotes particular elements of a data structure. Vectors are indexed with square brackets,[ ]. For instance, the second element of xNum is:

```
xNum[2]
```

```
## [1] 3.14159
```

At its core, R is a mathematical language that understands vectors, matrices, and other structures, as well as common mathematical functions and constants. R automatically applies operators across entire vectors:

```
x2 + 1
```

```
## [1] 3 5 7 9 3 5 7 9
```

```
x2 * pi
```

```
## [1]  6.283185 12.566371 18.849556 25.132741  6.283185 12.566371 18.849556
## [8] 25.132741
```

```
(x+cos(0.5)) * x2
```

```
## [1]  5.755165 19.510330 41.265495 71.020660  5.755165 19.510330 41.265495
## [8] 71.020660
```

When working with vectors, R recycles the elements to match a longer set. In the last command, x2 has eight elements, while x has only four. R will line them up and multiply x[1] * x2[1], x[2] * x2[2], and so forth. When it comes to x2[5], there is no matching element in x, so it goes back to x[1] and starts again.

4.2 More on Vectors and Indexing

Let's look at vectors and indexing in detail.

Integer sequences are commonly defined with the : operator.

```
xSeq <- 1:10 # use 1:10 instead of typing 1,2,3,4 ...10.
```

Sequences are useful for indexing and you can use sequences inside[ ]:

```
xNum
```

```
## [1] 1.00000 3.14159 5.00000 7.00000
```

```
xNum[2:4]
```

```
## [1] 3.14159 5.00000 7.00000
```

```
myStart <- 2
xNum[myStart:sqrt(myStart+7)]
```

```
## [1] 3.14159 5.00000
```

Exclude items by using negative indices:

```
xSeq
```

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

```
xSeq[-5:-7]
```

```
## [1]  1  2  3  4  8  9 10
```

In all of the R output, we've seen "[1]" at the start of the row. That indicates the vector position index of the first item printed on each row of the output. Try these:

```
1:300
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
## [109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
## [127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
## [145] 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162
## [163] 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
## [181] 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198
## [199] 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216
## [217] 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234
## [235] 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252
## [253] 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270
## [271] 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288
## [289] 289 290 291 292 293 294 295 296 297 298 299 300
```

```
1001:1300
```

```
##   [1] 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015
##  [16] 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030
##  [31] 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045
##  [46] 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060
##  [61] 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075
##  [76] 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090
##  [91] 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103 1104 1105
## [106] 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120
## [121] 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135
## [136] 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150
## [151] 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165
## [166] 1166 1167 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180
## [181] 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195
## [196] 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210
## [211] 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225
## [226] 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240
## [241] 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1254 1255
## [256] 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270
```

```
## [271] 1271 1272 1273 1274 1275 1276 1277 1278 1279 1280 1281 1282 1283 1284 1285
## [286] 1286 1287 1288 1289 1290 1291 1292 1293 1294 1295 1296 1297 1298 1299 1300
```

The result of an R vector operation is itself a vector. Try this:

```
xNum[2:4]
```

```
## [1] 3.14159 5.00000 7.00000
```

```
xSub <- xNum[2:4]
xSub
```

```
## [1] 3.14159 5.00000 7.00000
```

The new object xSub is created by selecting the elements of xNum. This may seem obvious, yet it has profound implications because it means that the results of most operations in R are fully formed, inspectable objects that can be passed on to other functions. Instead of just output, you get an object you can reuse, query, manipulate, update, save, or share.

Indexing also works with a vector of logical variables (TRUE/FALSE) that indicate which elements you want to select:

```
xNum
```

```
## [1] 1.00000 3.14159 5.00000 7.00000
```

```
xNum[c(FALSE, TRUE, TRUE, TRUE)]
```

```
## [1] 3.14159 5.00000 7.00000
```

This allows you to use logical expressions—which are evaluated as a vector of logical values—to select subsets of data based on specific criteria. Here is an example:

```
xNum[xNum > 3]
```

```
## [1] 3.14159 5.00000 7.00000
```

4.3 Missing and interesting values

In statistics, missing values are important, and in a statistics environment, R understands them and includes a special constant for a missing value: NA. This is not a character object ("NA") but a constant in its own right. It is useful in several contexts. For instance, you might create a data object that will be filled in with values later:

```
my.test.scores <- c(91, NA, NA)
```

Any math performed on a value of NA becomes NA:

```
mean(my.test.scores)
```

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

```
max(my.test.scores)
```

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

Many commands include an argument that instructs them to ignore missing values: *na.rm=TRUE*:

```
mean(my.test.scores, na.rm=TRUE)
```

```
## [1] 91
```

```
max(my.test.scores, na.rm=TRUE)
```

```
## [1] 91
```

A second approach is to remove NA values explicitly before calculating them or assigning them elsewhere. This may be done most easily with the command *na.omit()*:

```
mean(na.omit(my.test.scores))
```

```
## [1] 91
```

A third and more cumbersome alternative is to test for NA using the *is.na()* function and then index data for the values that are not NA by adding the ! ("not") operator

```
is.na(my.test.scores)
```

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

```
my.test.scores[!is.na(my.test.scores)]
```

```
## [1] 91
```

4.4 Lists

Lists are collections of objects of any type. They are useful on their own and are especially important to understand how R stores data sets.

Let's look at two of the objects we defined above, inspecting their structures with the *str()* command:

```
str(xNum)
```

```
##  num [1:4] 1 3.14 5 7
```

```
str(xChar)
```

```
##  chr [1:4] "foo" "bar" "boo" "far"
```

We see that these vectors are of type "numeric" and "character", respectively. All the elements in a vector must be the same type. We can combine these two vectors into a list using *list()*:

```
xList <- list(xNum, xChar)
xList
```

```
## [[1]]
## [1] 1.00000 3.14159 5.00000 7.00000
##
## [[2]]
## [1] "foo" "bar" "boo" "far"
```

Using str(), we see that objects inside the list retain the types that they had as separate vectors:

```
str(xList)
```

```
## List of 2
##  $ : num [1:4] 1 3.14 5 7
##  $ : chr [1:4] "foo" "bar" "boo" "far"
```

Lists are indexed with double brackets [[ ]] instead of the single brackets that vectors use, and thus xList comprises two objects that are indexed with [[1]] and [[2]]. We might index the objects and find summary information one at a time, such as:

```
summary(xList[[1]])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   2.606   4.071   4.035   5.500   7.000
```

Each element in a list may be assigned a name, which you can access with the *names()* function. You may set the *names()* when a list is created or at a later time. The following two list creation methods give the same result:

```
xList <- list(xNum, xChar) # method 1: create, then name
names(xList) <- c("itemnum", "itemchar")
```

```
xList <- list(itemnum=xNum, itemchar=xChar)
# method 2: create & name at once
names(xList)
```

```
## [1] "itemnum"  "itemchar"
```

List may be indexed using its names rather than a numeric index. You can use *$name* or *[["name"]]* as you prefer:

```
xList[[1]] # method 1: numeric
```

```
## [1] 1.00000 3.14159 5.00000 7.00000
```

```
xList$itemnum # method 2: $name reference
```

```
## [1] 1.00000 3.14159 5.00000 7.00000
```

```
xList[["itemnum"]] # method 3: quoted name
```

```
## [1] 1.00000 3.14159 5.00000 7.00000
```

List names are character strings and may include spaces and various special characters. Putting the names in quotes are useful when names include spaces.

This brings us to the most important object type in R: **data frames**.

5 Data frame

Data frames are the workhorse objects in R, used to hold data sets and to provide data to statistical functions and models. A data frame's general structure will be familiar to any analyst: it is a rectangular object comprised of columns of varying data types (often referred to as "variables") and rows that each has a value (or missing value, NA) in each column ("observations").

You may construct a data frame with the *data.frame()* function, which takes as input a set of vectors of the same length:

```
x.df <- data.frame(xNum, xLog, xChar)
```

In the resulting data frame, we find three named columns that inherit their names from the contributing vectors. Each row is numbered, sequentially starting from 1. Elements of a data frame may be indexed using [ROW, COLUMN] notation:

```
x.df[2,1]
```

```
## [1] 3.14159
```

```
x.df[1,3]
```

```
## [1] "foo"
```

Marketing analysts often work with categorical data such as gender, region, or different treatments in an experiment. It is important to convert such *character strings* to *nominal factors* in the data frame. Converting character strings to factors is a good thing for data that you might use in a statistical model because it tells R to handle it appropriately in the model. In R, such nominal factors are stored internally as a vector of

integers and a separate list of labels naming the categories. The latter are called levels and are accessed with the *levels()* function.

You can make the conversion to factors by adding an option to data.frame() that sets stringsAsFactors=TRUE:

```
x.df1 <- data.frame(xNum, xLog, xChar, stringsAsFactors=TRUE)
str(x.df1)
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ xNum : num  1 3.14 5 7
##  $ xLog : logi  TRUE FALSE TRUE TRUE
##  $ xChar: Factor w/ 4 levels "bar","boo","far",..: 4 1 2 3
```

Indices can be left blank, which selects all of those dimension:

```
x.df[2, ] # all of row 2
```

```
x.df[ ,3] # all of column 3
```

```
## [1] "foo" "bar" "boo" "far"
```

Index data frames by using vectors or ranges for the elements you want. Use negative indices to omit elements:

```
x.df[2:3, ]
```

```
x.df[ ,1:2] # two columns
```

```
x.df[-3, ] # omit the third observation
```

```
x.df[, -2] # omit the second column
```

Indexing a data frame returns an object. The object will have whatever type suits that data: choosing a single element (row + column) yields a singular object (a vector of length one); selecting a column returns a vector; and choosing rows or multiple columns yields a new data frame. We can see this by using the *str()* inspector, which tells you more about the structure of the object:

```
str(x.df[2,1])
```

```
##  num 3.14
```

```
str(x.df[, 2])
```

```
##  logi [1:4] TRUE FALSE TRUE TRUE
```

```
str(x.df[c(1,3), ]) # use c() to get rows 1 and 3 only
```

```
## 'data.frame':    2 obs. of  3 variables:
##  $ xNum : num  1 5
##  $ xLog : logi  TRUE TRUE
##  $ xChar: chr  "foo" "boo"
```

As with lists, data frames may be indexed by using the names of their columns:

```
x.df$xNum
```

```
## [1] 1.00000 3.14159 5.00000 7.00000
```

In short, data frames are the way to work with a data set in R. R users encounter data frames all the time, and learning to work with them is perhaps the single most important set of skills in R.

Let's create a new data set which is more representative of the data in marketing research. We'll clean up our workspace and then create new data:

```
rm(list=ls()) # caution, deletes all objects!
store.num <- factor(c(3, 14, 21, 32, 54)) # store id
store.rev <- c(543, 654, 345, 678, 234) # store revenue, $1000
store.visits <- c(45, 78, 32, 56, 34) # visits, 1000s
store.manager <- c("Annie", "Bert", "Carla", "Dave", "Ella")
(store.df <- data.frame(store.num, store.rev, store.visits,
                        store.manager))
```

In the final command above, by putting parentheses around the whole expression, we tell R to assign the result of *data.frame(store.num, store.rev, . . .)* to *store.df* and then evaluate the resulting object *(store.df)*. This has the same effect as assigning the object and then typing its name again to see its contents. This trick sometimes saves typing.

We can now get different information by selecting that column using the same $ notation that we used with lists:

```
store.df$store.manager
```

```
## [1] "Annie" "Bert"  "Carla" "Dave"  "Ella"
```

```
mean(store.df$store.rev)
```

```
## [1] 490.8
```

```
cor(store.df$store.rev, store.df$store.visits)
```

```
## [1] 0.8291032
```

You can obtain basic statistics for a data frame with summary():

```
summary(store.df)
```

```
##  store.num   store.rev      store.visits store.manager
##  3 :1      Min.   :234.0   Min.   :32    Length:5
##  14:1      1st Qu.:345.0   1st Qu.:34    Class :character
##  21:1      Median :543.0   Median :45    Mode  :character
##  32:1      Mean   :490.8   Mean   :49
##  54:1      3rd Qu.:654.0   3rd Qu.:56
##            Max.   :678.0   Max.   :78
```

6 Saving, loading, and importing data

6.1 Saving and loading data

Let's back up the *store.df* object to disk using *save(OBJECT, FILE)*:

```
save(store.df, file="store-df-backup.RData")
```

We'll delete it from memory and use load(FILE) to restore it:

```
rm(store.df) # caution, only if save() gave no error
load("store-df-backup.RData")
```

The memory image of an entire session can be saved with the command *save.image(FILE)*. If *FILE* is excluded, then it defaults to a file named ".RData".Base R and R Studio both prompt you to save a memory image on closing, but you can also do it yourself by typing:

```
save.image() # saves file ".RData"
save.image("mywork.RData")
```

Workspace images are re-loaded with the general *load()* command, not with a special "image" version; an image is a collection of objects and no different than other files produced by *save()*. Loading an image will silently overwrite current memory objects that have the same names as objects in the image but do not remove other objects.

```
load("mywork.RData")
```

6.2 Import data to R

Before we can analyze data in R, we need to know how to import data into R.

6.2.1 Excel files

Let's try to import a dataset containing sales and marketing data for deospray brands into R. The Excel file "deospray.xls" is included in the 'Seminar 1' folder on Minerva. Please make sure to save the Excel file in your working directory.

To import Excel files into R, you will need the package called *readxl*, so first, you will need to install and load this package.

install.packages("readxl") # this line installs the readxl package

library(readxl) # this line loads the readxl package

Next, you can read the dataset. Make sure the Excel file is saved in your working directory.

```
deospray.data <- read_excel(path = "deospray sales.xls", sheet = "deospray")

# path could be with your work directory path, such as:
# path = "C:/Users/busasi/LUBS5403M/Week1_Introduction/deospray sales.xls"
```

- This line reads the "deospray" sheet from the "deospray sales.xls" file from your working directory into R. The function to do this is *read_excel* from the *readxl* package.

read_excel is a function from the *readxl* package. It takes two arguments: the first one is the filename (in this case "deospray sales.xls", and the second argument is the name of the Excel sheet that you want to read (in this case "deospray").

- *deospray.data* reflects the name you are giving to the file within R. You can choose any name you like.

Now, let's have a first look at the data we have imported into R.

```
str(deospray.data)
```

```
## tibble [620 x 22] (S3: tbl_df/tbl/data.frame)
##  $ chain         : num [1:620] 1 1 1 1 1 1 1 1 1 1 ...
##  $ week          : chr [1:620] "W16046" "W16047" "W16048" "W16049" ...
##  $ sales_brand1  : num [1:620] 24 28.1 25.8 24.6 23 ...
##  $ sales_brand2  : num [1:620] 31.3 39.6 102.2 68.4 37.8 ...
##  $ sales_brand3  : num [1:620] 21.1 24 26.8 22.7 20.6 ...
##  $ sales_brand4  : num [1:620] 48.4 52.1 53 43 53.9 ...
##  $ sales_brand5  : num [1:620] 65 64.6 64.7 64.2 52.7 ...
##  $ price_brand1  : num [1:620] 1.09 1.08 1.08 1.08 1.08 ...
##  $ price_brand2  : num [1:620] 0.688 0.642 0.478 0.569 0.576 ...
##  $ price_brand3  : num [1:620] 1.05 1.05 1.05 1.04 1.06 ...
##  $ price_brand4  : num [1:620] 1.12 1.12 1.12 1.12 1.12 ...
##  $ price_brand5  : num [1:620] 1.32 1.32 1.32 1.32 1.32 ...
##  $ display_brand1: num [1:620] 0.02 0.02 0.19 0.18 0.05 0.05 0.08 0.03 0.05 0.04 ...
```

```
##  $ display_brand2: num [1:620] 0 0 0 0 0 0 0 0 0 0 ...
##  $ display_brand3: num [1:620] 0 0 0 0 0 0 0 0 0 0 ...
##  $ display_brand4: num [1:620] 0.01 0 0 0 0 0 0 0 0 0 ...
##  $ display_brand5: num [1:620] 0 0 0 0 0 0 0 0 0 0 ...
##  $ feature_brand1: num [1:620] 0 0 0 0 0 0 0 0 0 0 ...
##  $ feature_brand2: num [1:620] 0 0 0 0 0 0 0 0 0 0 ...
##  $ feature_brand3: num [1:620] 0 0 0 0 0 0 0 0 0 0 ...
##  $ feature_brand4: num [1:620] 0 0 0 0 0 0 0 0 0 0 ...
##  $ feature_brand5: num [1:620] 0 0 0 0 0 0 0 0 0 0 ...
```

As you can see, R tells us the data contains 620 observations of 22 variables, so all observations and variables from the Excel file have been imported.

You can also have a look at the entire data set by running "View(deospray.Data)", or by clicking on the object "deospray.data" in your Environment.

```
View(deospray.data)
```

6.2.2 CSV files

Delimited files such as CSV files are a common type many marketers analysts use to move data between tools such as R, databases, and Microsoft Excel.

Now let's write a real file and then read it using *read.csv(file=...)*:

```
read.csv("Data_Descriptive.csv")
```

By default, read.csv() prints the CSV contents to the R console formatted as a data frame. To assign the data to an object, use the assignment operator (<-). Let's read the CSV file and assign its data to a new object:

```
store.df <- read.csv("Data_descriptive.csv")
store.df$storeNum <- factor(store.df$storeNum)
```

After reading the CSV file, we recreate storNum as a factor variable.

Great, you have now learned how to set up R to get started with analyzing your first dataset.

7 Create your own functions

We examine the basics of writing reusable functions, a fundamental programming skill. R provides *functions* to let you write a set of commands once and reuse it with new data.

For instance, we can declare a function to compute the standard error of the mean for a vector of observed data (such a function already exits in R, but it is simple to write our own):

FUNCTIONNAME <- function (INPUTS) EXPR

In most cases, EXPR is a set of multiple lines that operate on the inputs. When there are multiple lines, they must be enclosed with braces {}. By default, the return value of the function is the output of the last command in the function declaration.

```
se <- function(x){sd(x) / sqrt(length(x))}
```

The new function *se()* can then be used just like any other built-in function in R:

```
se(store.df$store.visits)
```

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

When writing a function, we recommend four conventions:

- Put *braces* around the body using { and }, even if it is just a one-line function.

- Create *temporary values* to hold results along the way inside the function.
- *Comment* on the function profusely
- Use the keyword *return()* to show the explicit value returned by the function.

Putting those recommendation together, the *se()* function above may be rewritten as follows:

```
se <- function(x){
  # computes standard error of the mean
  tmp.sd <- sd(x)  # standard deviation
  tmp.N <- length(x) # sample size
  tmp.se <- tmp.sd / sqrt(tmp.N)  #std error of the mean
  return(tmp.se)
}
```

8 Clean up

R keeps everything in memory by default, and when you exit (use the command *q()*, for quit), R offers to save the memory workspace to disk to be loaded next time. This is convenient but means that your workspace will become crowded unless you keep it clean.

We recommend a few steps to keep your workspace clean. Use the *ls()* command periodically to see what you have in memory. If you do not recognize an object, use the *rm()* command to remove it.

```
ls()
```

```
## [1] "deospray.data" "se"            "store.df"      "store.manager"
## [5] "store.num"     "store.rev"     "store.visits"
```

```
rm(store.num)    #remove a single object
rm(list=c("store.rev", "store.visits"))  #remove a group of objects
rm(list=ls(pattern = "store"))   #remove a whole set following a pattern
```

It is better to start every session clean instead of saving a workspace. It is a good idea to *keep all important and reproducible code* in a *working script file*. This will make it easy to recreate an analysis and keep a workspace clean and reproducible.

To clean out memory and ensure you are starting from scratch at a given time, remove all objects:

```
rm(list=ls())  # delete all visible objects in memory.
```

Alternatively, you can exit without saving the workspace.

==========

You have reached the end of these introductory notes! You should now be able to set your working directory, install and load packages, import data into R, and understand basic objects and data frames in R.