

Lab 0: R Tutorial, A Warm Up

First Name: _____ Last Name: _____ NetID: _____

The purpose of this lab is to get you started with R immediately. We will be working in RStudio, a convenient IDE for R. You will get a taste of the basics of the R programming language and then play with a couple of datasets to extract basic information.

You don't need to turn in this lab.

1 Working in the Console Window

1.1 Changing Directories

Run RStudio. In the RStudio console window, type in

```
> getwd()
```

`getwd` is a function that shows the current working directory of R. You can change the working directory by clicking “Session\Set working dir...” from the toolbar and choose the new working directory, or you can call

```
> setwd("Path/to/the/new/working/directory")
```

to do the same thing. To see the contents of your working directory, use `dir()`. After setting up your working directory, **download** `olive-train.dat`, `olive.metadata.txt` and `recalls.xml` from Canvas to your working directory.

1.2 Basic Operations

Before we start playing with data, we will first get a taste of the basics of R. The first thing we would do, like when we learn other programming languages, is to create an object and assign values to it. In the R prompt, type in

```
> a <- 5
```

This assigns value 5 to object `a`. You can also assign values in a way which might be more familiar:

```
> a = 5
```

You may find the first way strange but you're suggested to use it.¹ If you want to see what you have just created, simply type

```
> a
```

and you will get:

```
[1] 5
```

Here “[1]” just indicates the first line of the displayed results. Ignore it. Each object in R has a **class**:

- vector

¹There are subtle differences between the two methods - you might be interested in browsing through <https://stackoverflow.com/questions/1741820/>.

- matrix
- array
- data.frame
- list

There are more classes but these are the ones you need to know currently. You will get familiar with them as the course moves forward. An object of each class can have elements that are of type numeric, character, factor or logical:

- numeric: real-valued/integer valued
- character: character/string
- factor: categorical
- logical: TRUE/FALSE

To check the class of object `a`, simply type

```
> class(a)
```

and you will get:

```
[1] "numeric"
```

Now let's create something more interesting: a vector. There are multiple ways to create a vector, for example:

```
> b <- c(1,2,3)
> c <- (1:5)
> d <- c(1:10)
> e <- rep(0,10)
> f <- c("Monday", "Tuesday")
```

The vectors created can be of type numeric, character, factor or logical, check it:

```
> is.numeric(a)
[1] T
> is.factor(a)
[1] F
> is.character(f)
[1] T
```

To find out the number of elements in a vector, say `d`, call:

```
> length(d)
```

We can also switch the class of an object. For example, we want to change `b` to a vector of categorical data with 3 categories, denoted "1", "2" and "3" respectively:

```
> b <- as.factor(b)
> is.factor(b)
[1] T
```

This is important because sometimes categorical variables are encoded as integers in data sets, e.g. 1=female, 2=male.

To find the different possible values of a factor vector, use the `levels` function:

```
> levels(b)
[1] "1" "2" "3"
```

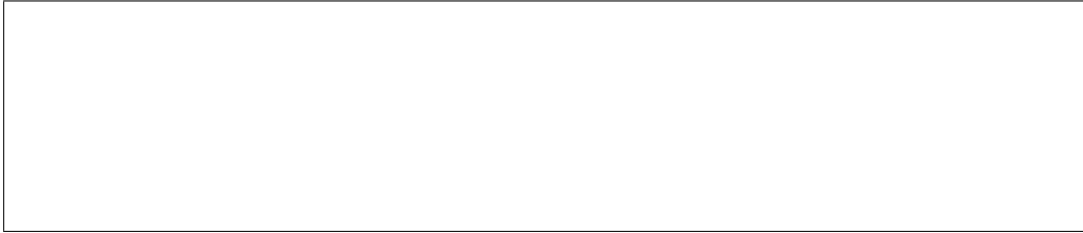
R vector indices start at 1. To obtain the i th element of a vector, e.g. $i=2$, use:

```
> c[2]
[1] 2
```

We can select multiple elements of a vector at once. Type these codes in the console:

```
> d[c(2,3,5)]
> d[c(T,T,F,F,T,T,F,F,T,T)]
> d[d>5]
> f[f=="Monday"] <- "Sunday"
> f[1]
```

What are the results?



1.3 User-Defined Functions

It is easy to write functions in R. The general structure is

```
myfunction <- function(arg1, arg2, ... ){
  statements
  return(object)
}
```

Suppose we want to write a function **Fun1** that takes an object as input and add 2 to all its elements:

```
> Fun1 <- function(a) {
+   return(a + 2)
+ }
```

We can also write functions that call other functions. We now write another function **Fun2** that calls **Fun1** in its body:

```
> Fun2 <- function(a) {
+   print(sum(a))
+   d <- Fun1(a)
+   return(mean(d))
+ }
```

The `sum()` and `mean()` are some built-in functions. To know what they do, we can use the `help` command, type:

```
> help(sum)
> help(mean)
```

The commands will bring out the documentaion for the functions in the help window.

Now call `Fun2`:

```
> b <- Fun2((1:4))
```

What is the result of `b`?

How about adding a loop in the function?

```
> Fun3 <- function(a) {  
+   nData <- length(a)  
+   for (i in (1:nData)) {  
+     j <- nData + 1 - i  
+     print(a[j])  
+   }  
+ }
```

What is the printout from `Fun3(b)` when `b=c(3,5,2,7)`?

1.4 Loading Data

OK. Let's start to play with the data!

Read the training data into R and store the data in an object named `oliveOils` using the `read.table` function. Type:

```
> help(read.table)
```

to see the help document of the function `read.table`. This function has many arguments. For now you need only set the `header` and `sep` arguments correctly(You might need to open the `olive-train.dat` in a text file editor to figure out what these arguments should be!). What are the arguments you use for `header` and `sep`?

The resulting data set `oliveOils` should be a `data.frame` object. Which function will you use to check this? What is the result?

Open the `olive_metadata.txt` file. Based on the information in this file, we will set appropriate column names for `oliveOils`. First check what are the current column names by simply typing:

```
> oliveOils[1:10, ]
```

This command lets you observe the first 10 rows of your imported data set. The first row in the output are the column names. You can also compare the result with the `olive-train.dat` text file directly to check whether the data has been corrupted when reading it into R. Just make sure the data looks the same as the data in the training data text file (if it was read in incorrectly then there might be too few columns, columns with all NA values, etc.).

Now we will set the column names according to `olive.metadata.txt` by calling:

```
> colnames(oliveOils) <- c("region", "area", ...)
```

Our goal is to estimate the region or area of origin for new samples of olive oil (e.g. in a test data set), by learning from the data set we have been given (the training data set). Is this supervised or unsupervised learning?

How many samples are in the training data?(Hint: Use the `dim` command)

Check whether the data has been read in as numeric, character, or factor data. This may be different for different variables (columns) in `oliveOils`. The variables can be selected using the syntax `"oliveOils$region"` (If you have named your first column `"region"`). Think about what function(s) you should use. Region and area clearly are intended to be factor (categorical) variables, so coerce them to factor variables. What function will you use? What function will you use to check if this change has been made successfully? What is the output of your check?

How many different values does the `region` variable take in the data set? What are these values? What function will you use to get this information? What is the result?

Are there any missing values for the region in the data set? Missing values can be encoded in many ways: by a period, dash, or question mark, by a text string such as unknown, or using a number that does not correspond to any of the possible values for the variable, for instance a 9 when the variable should be between 1 and 4. The `levels` function will show you all the values that the variable takes in the data set, except the special NA value which can be used for missing data. Check for NA values in `oliveOils$region` using the `is.na` function. Note that applying `is.na` to `oliveOils$region` will result in a long vector of true/false values. You want to summarize this vector rather than visually inspecting the whole thing. What function should you use?(Hint: try the `sum` function). What is your result?

Final challenge! A more complicated one! We want to write a function which includes a loop to compute the mean of the input vector `a`:

```
> computeMean <- function(a)
```

How will you write this function `computeMean`?

Apply `computeMean` to the 3rd column of `oliveOils`, what is the result you get? Compare it with the result you get by directly calling the `mean` function to that column. Did your function get the correct result?

If you wish to quit the current R session, simply type:

```
> q()
```

or just close RStudio.

1.5 Installing packages

Much of R's power comes from the universe of third-party packages accompanying it. To install a package, we use the `install.packages()` command. For example, if we want to install the `ggplot2` package, we can type the following in the console window:

```
> install.packages("ggplot2")
```

If this is your first time using R, you may be prompted to use a *mirror* from which to download the package. These mirrors form the **C**omprehensive **R** Archive **N**etwork (CRAN), which hosts the latest released versions of R package. To minimize latency, choose the mirror closest to your location. After installing a package, loading it is as simple as typing

```
> library("ggplot2")
```

In fact, `ggplot2` is part of a collection of R packages aimed at maximizing productivity for data science tasks. The name of this collection is called `tidyverse`², and you may install it using

```
> install.packages("tidyverse")
```

You won't need the whole collection for the remainder of this lab. Instead, you may install the following packages:

```
> install.packages(c("tibble", "dplyr"))
```

²<https://www.tidyverse.org/>

1.6 R scripts

We have so far only worked in the console window in RStudio. It serves as a useful tool to try out functions interactively. However, when we are working in larger scale projects, it might be useful to run R commands as a script. There are many ways to create a script file, for example, you can use **Ctrl+Shift+N** or click on the toolbar **File > New File > R Script**. It is easy to run the previous codes in a script file. Instead of typing the commands in the console window, we type the codes line by line into the script.

The script can be run in a couple of ways:

1. Execution by line: to run the script line by line, simply select the line of code and press **Ctrl+Enter** (or click on the **Run** button in the toolbar)
2. Execution by part: to run a particular part of the code, simply select the part and press **Ctrl+Enter** (or click on the **Run** button in the toolbar)

2 Working with data frames

Previously, you worked with a `data.frame` object which contained the olive oil dataset. Data frames are 2-dimensional arrays, with each column containing data of the same type and each row containing a data point with a *unique id*. However, this does **not** mean that data frames cannot contain duplicates! To see for yourself, try the following:

```
> taData <- data.frame(
+   name = c("Tao", "Tonghua", "Duanduan", "Kevin", "Tao", "Tonghua"),
+   role = c("full", "full", "half", "half", "full", "full"))
> taData
      name role
1      Tao full
2 Tonghua full
3 Duanduan half
4      Kevin half
5      Tao full
6 Tonghua full
```

As you see, R automatically generated the first column, containing a unique identifier for each row (notice that this column is **unnamed**). If you suspect that your data might contain duplicates, you can obtain a view containing only distinct data points using `unique()`:

```
> taUnique <- unique(taData)
> taUnique
      name role
1      Tao full
2 Tonghua full
3 Duanduan half
4      Kevin half
```

What if instead we had typed the following?

```
> taData <- data.frame(
+   name = c("Tao", "Tonghua", "Duanduan", "Kevin", "tao", "Tonghua"),
+   role = c("full", "full", "half", "half", "full", "full"))
> taData
```

```

      name role
1      Tao full
2 Tonghua full
3 Duanduan half
4      Kevin half
5      tao full
6 Tonghua full

```

In this case, we can convert all names to lowercase and use `duplicated()` to spot the rows which are duplicates of previous rows:

```

> taData[!duplicated(tolower(taData$name)),]
      name role
1      Tao full
2 Tonghua full
3 Duanduan half
4      Kevin half

```

2.1 The dplyr library

Working with data frames in R can be pain, but luckily there are some great packages to make our lives easier. The `dplyr` package provides a set of handy functions for manipulating data frames, along with the so-called **pipe** operator which we will introduce momentarily. First, let us make sure that we work with a more interesting dataset. The file `recalls.xml` you downloaded from Canvas contains a list of product recalls between 2015 and 2017, provided by the FDA. First, we must load it in memory. The file is stored in XML format, which we can load using R's XML library:

```

> install.packages("XML")
> library("XML")

```

Now, read the contents of the XML file into a data frame.

```

> recalls <- xmlToDataFrame("recalls.xml")

```

Let us try to inspect the data. If you write `print(recalls)` on the R prompt, you will (probably) get a very messy output. This is where the `tibble` library is useful - try the following:

```

> library("tibble")
> library("dplyr")
> recalls <- as_tibble(xmlToDataFrame("recalls.xml"))
> print(recalls)
# A tibble: 1,091 x 8
  DATE   BRAND_NAME PRODUCT_DESCRIP REASON COMPANY COMPANY_RELEASE PHOTOS_LINK
<chr> <chr>      <chr>          <chr> <chr>    <chr>          <chr>
1 Thu, Isomeric "Multiple compo Conce Isomer http://wayback. "\t"
2 Wed, Newport "Newport HT70 a Poten Medtro http://wayback. "\t"
3 Tue, Clancys "Combo snack ba Undec Olde Y http://wayback. "\t"
4 Tue, Seasons "Seasons Choic Liste Aldi i http://wayback. "\t"
5 Mon, Wegmans "Original Kille Undec Wegman http://wayback. "\t"
6 Sun, Hunts "Chili Kit"      Salmo Conagr http://wayback. "\t"
7 Fri, Menu Del "Beans & Ch Liste Sigma http://wayback. "\t"
8 Fri, Resers "Description\tM Undec Reser http://wayback. "\t"
9 Fri, Mylan "EpiPen (epinep Failu Mylan http://wayback. "\t"
10 Thu, NuGo Slim "Crunchy Peanut Undec Lifest http://wayback. "\t"
# with 1,081 more rows, and 1 more variable: text <chr>

```


As you see, a `tibble` provides pretty-printing for your data frame, showing summary information such as the number of data points and features, as well as the datatype of each column. You can *coerce* any data frame to a tibble by using the function `as_tibble()`.

In the lab, it was pointed out that the last column, called `text`, contains only NA values, causing `na.omit()` to drop the entire dataset. To avoid this, it was suggested to remove the last column first by using:

```
> recalls <- select(recalls, -text)
```

or to wait until we `select` the subset of columns shown below until we call `na.omit`.

If you browse over the data, you may notice that the last row has values NA. This is a special value commonly used to indicate missing data. We can remove all rows containing missing values using the following:

```
> recalls <- na.omit(recalls)
```

2.1.1 select() and filter()

Let us try to discard information of no interest to us. We will use `dplyr`'s `select()` function, which selects a given subset of columns. Let us keep everything except the company release link, the photos link, and any additional text. To inspect the names of the columns, we use the function `names()`:

```
> names(recalls)
[1] "DATE"           "BRAND_NAME"      "PRODUCT_DESCRIPTION"
[4] "REASON"         "COMPANY"         "COMPANY_RELEASE_LINK"
[7] "PHOTOS_LINK"    "text"
> recallData <- select(recalls,
+   c("DATE", "BRAND_NAME", "PRODUCT_DESCRIPTION", "REASON", "COMPANY"))
> dim(recallData)
[1] 1091    5
```

Because we are keeping more columns than we are dropping, there is a slightly less tedious way: we need only specify the dropped columns, prepending a “-” ahead of the vector containing their names:

```
> recallData <- select(recalls,
+   -c("COMPANY_RELEASE_LINK", "PHOTOS_LINK", "text"))
```

You might have noticed some entries from a very familiar grocery store in there. To see how many recalls this store had, we want to filter those rows whose `BRAND_NAME` field contains the word “wegmans” (in any combination of lowercase/uppercase letters). The way to accomplish this is to use `grepl()` which returns a logical vector containing `TRUE` if a string matches a given pattern. The `filter()` function then uses this logical vector to keep only the rows indexed by the `TRUE` elements.

```
> wegmansRecalls <- filter(recallData,
+   grepl("wegmans", BRAND_NAME, ignore.case=TRUE))
> wegmansRecalls
# A tibble: 10 x 5
.....
```

The **pipe operator** of `dplyr` allows us to chain several operations on a data frame. It is an example of a *special function* in R, with an unusual syntax best described by an example:

```

> recallReasons <- recallData %>%
+   filter(grepl("wegmans", BRAND_NAME, ignore.case=TRUE)) %>%
+   select(REASON)
> recallReasons
# A tibble: 10 x 1
  REASON
  <chr>
1 Undeclared Peanuts
2 Undeclared peanuts
3 May contain pieces of white plastic
4 Listeria monocytogenes
5 Listeria monocytogenes
6 Listeria monocytogenes
7 Undeclared Cashews and Almonds
8 Listeria monocytogenes
9 Listeria monocytogenes
10 Salmonella

```

Inserting “%>%” applies the function following the operator to the result of the expression preceding it. Mathematically:

$$x \%>\% f(y) = f(x, y)$$

To make yourself familiar with the concept, use the pipe operator to count the number of distinct companies that had products recalled due to salmonella. You might find the function `n.distinct()` useful.

2.1.2 mutate() and summarize()

You might have noticed that the dates in the `recalls.xml` dataset follow a somewhat nonstandard format. Our next objective is to extract the year, month and day of each product recall. To do so, we will use the `mutate()` function, which generates new columns for a data frame. First, install the `parsedate` library:

```
> install.packages("parsedate"); library("parsedate")
```

This library offers a `parse_date()` function which tries to convert an unknown date format to the commonly used ISO 8601 format. Let us create a column called `ISO_DATE`:

```

> isoData <- recallData %>%
+   mutate(ISO_DATE=parse_date(DATE))

```

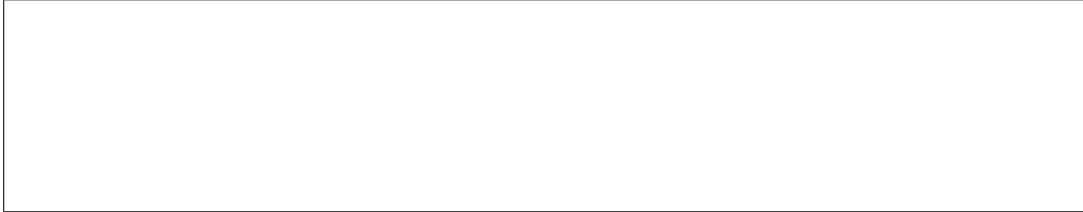
Now, we can parse the `ISO_DATE` column to generate the year, month and day columns. The following example shows how to use `format()` for that conversion:

```

> dt <- as.POSIXct("2022-01-24 12:00:00") # create toy datetime
> year <- as.integer(format(dt, "%Y"))   # 2024
> month <- as.integer(format(dt, "%m"))  # 1
> day <- as.integer(format(dt, "%d"))    # 24

```

Now, your job is to use `mutate()` to create 3 columns called `YEAR`, `MONTH`, `DAY` containing the numerical values, starting from `recallData`, and afterwards drop the columns `DATE`, `ISO_DATE`. Try to do so using as few assignments as possible; remember that you can use the pipe operator to chain multiple function applications. [Store the result into a variable called `dateData`](#).



Finally, the `summarize()` function is useful in conjunction with `group_by()`, which produces so-called *grouped* data frames; a grouped data frame enables the use of **aggregate** functions (such as computing the mean of a quantity) on each group. For example, to compute the average number of recalls per month, given the 3 years of data from `recalls.xml`:

1. we first group the data by month, followed by year
2. we count the number of data points per month of each year, using the built-in function `n()`
3. finally, compute the average of the data point count using the `mean()` function.

```
recallsPerMonth <- dateData %>%
+   group_by(MONTH, YEAR) %>%
+   summarize(n = n()) %>%
+   summarize(avg = mean(n))
# A tibble: 12 x 2
  MONTH    avg
  <int> <dbl>
1     1    36
2     2   41.3
3     3    41
4     4    32
5     5   49.5
6     6   53.5
7     7    34
8     8   29.5
9     9   36.5
10    10   41.5
11    11   35.5
12    12   39.5
```

The above was a very brief and incomplete tour of the `dplyr` library. For a deeper dive into its data manipulation facilities, you are encouraged to go through the official tutorial:

<https://dplyr.tidyverse.org/articles/dplyr.html>

2.2 Some exercises

Using `dplyr`'s data manipulation functions, find:

1. the number of product recalls in 2015 for Wegmans or Whole Foods.

2. the number of product recalls due to undeclared ingredients per month
3. the 5 companies with the most recalls due to undeclared ingredients.