Problem Set 2: STAT243

Cristobal Pais - cpaismz@berkeley.edu

September 14, 2017

Problem 1

a) Comparing file sizes

Before starting the analysis of each file and its size, important is to note the fact that depending on the operating system (Windows or UNIX in this case) there are some differences in terms of the size of the files. In Windows the size of a new line character is 2 bytes while in UNIX it is only 1 byte. The same situation occurs with the end of the file (EOF) character. Therefore, executing the R-code (exactly the same code in both operating systems) provided in the following questions in Windows OS will modify the results obtained (size will be larger in Windows files). Thus, we perform our analysis for both operating systems, generating the files from R in UNIX -via R for UNIX using the UBUNTU installation in Windows 10- and from R in Windows, covering both operating systems cases for completeness.

i) In the case of the first file (tmp1.csv), we are including 1 million characters inside a .csv file, where (by default) will be separated by a new line character. Hence, the size of the file will be equal to 1e6 bytes (characters) plus the size of the corresponding new lines and *end of the file* characters as follows:

Once we created the files (in */ps2/windows* when using the Windows version of R) we can check the size of each file and analyze it:

- UNIX: we have 1e6 characters + 1e6 new lines characters, thus, we have 1e6+1e6 = 2e6 bytes in total since each character has a size of 1 byte by definition

```
# Check file size and type by calling bash from windows: UNIX format
system('bash -c "cd ~/ps2/unix
        ls -l tmp1.csv
        file tmp1.csv"', intern = TRUE)

## [1] "-rw-rw-rw- 1 cpaismz cpaismz 2000000 sep 10 06:20 tmp1.csv"
## [2] "tmp1.csv: ASCII text"
```

- Windows: we have 1e6 characters + 1e6 new lines characters. However, in Windows OS a new line character has a size of 2 bytes each and thus, we have 1e6+2e6 = 3e6 bytes in total.

```
# Check file size and type by calling bash from windows: Windows format
system('bash -c "cd ~/ps2/windows
        ls -l tmp1.csv
        file tmp1.csv"', intern = TRUE)

## [1] "-rwxrwxrwx 1 cpaismz cpaismz 3000000 sep 10 06:00 tmp1.csv"
## [2] "tmp1.csv: ASCII text, with CRLF line terminators"
```

Based on the previous outputs, we can clearly see the consistency of the previous analysis.

ii) For the second file (tmp2.csv), we are including 1 million characters inside a .csv file without separations between characters. Hence, the size of the file will be equal to 1e6 bytes (characters) plus the size of the corresponding end of the file character:

Once we created the files (in */ps2/windows* when using the Windows version of R) we can check the size of each file and analyze it:

- UNIX: we have 1e6 characters + 1 end of the file (EOF, 26 in ASCII) character, thus, we have 1e6+1 = 1.000001e6 bytes in total since each character has a size of 1 byte by definition.

```
# Check file size and type by calling bash from windows: UNIX format
system('bash -c "cd ~/ps2/unix
        ls -l tmp2.csv
        file tmp2.csv"', intern = TRUE)

## [1] "-rw-rw-rw- 1 cpaismz cpaismz 1000001 sep 10 06:22 tmp2.csv"
## [2] "tmp2.csv: ASCII text, with very long lines, with no line terminators"
```

- Windows: we have 1e6 characters + 1 end of the file character. However, in Windows an end of the file (EOF) character has a size of 2 bytes each and thus, we have 1e6+2 = 1.000002e6 bytes in total.

```
# Check file size and type by calling bash from windows: Windows format
system('bash -c "cd ~/ps2/windows
        ls -l tmp2.csv
        file tmp2.csv"', intern = TRUE)

## [1] "-rwxrwxrwx 1 cpaismz cpaismz 1000002 sep 10 06:00 tmp2.csv"
## [2] "tmp2.csv: ASCII text, with very long lines, with no line terminators"
```

iii) For the third file (tmp3.csv), we are including 1 million of random numbers (taking from a normal distribution) inside a .csv file. As a general analysis for the rest of the files, it is important to note that since we are creating a random vector of numbers following a standard normal distribution $N(0, 1)$, besides the number of decimals that we are including (random, depends on the number generated), we have to take into account the number of negative numbers (extra character '-' at the beginning) and positive numbers. In addition, we have the new line characters at the end of each line.

Furthermore, we have to take into account the fact that the file format is a .Rda file generated by the save function. This function can take any object in R and represent it in a binary format that can be read by a computer but it is not human readable. R binary files extension .Rda are used to keep R object "AS-IS" with the benefit of a reduction in the size of the file due to its ability to reduce the file size, even more (as we will see with the file tmp7.csv) when some character patterns can be identified.

As the documentation of the function indicates, "to maximize the number of observations in a data package, the data needs to be compressed. Compression is done through an algorithm that creates

a dictionary table of more commonly used patterns. This enables the reduction of the file size as large patterns are reduced to smaller patterns". There are three different types of compression offered by the function: *gzip*, *bzip2*, and *xz*. By default, the *save()* function will perform a *gzip* compression to the .Rda files, a compression format that reduces the size of the given files using the well-known Lempel-Ziv coding (LZ77).

Based on the previous analysis, we are aware of the fact that the file will be non-human readable, compressed using *gzip* and therefore, we will not have a match between the number of characters and the size of the file:

Once we created the files (in */ps2/windows* when using the Windows version of R) we can check the size of each file and analyze it:

- UNIX: we do not know the total number of characters, hence, we can count them using the *wc -m* command and we can then check the size of the file and its format in order to reflect the facts stated by our previous analysis.

```
# Check file size and type by calling bash from windows: UNIX format
system('bash -c "cd ~/ps2/unix
       wc -m tmp3.Rda
       ls -l tmp3.Rda
       file tmp3.Rda"', intern = TRUE)

## [1] "3974998 tmp3.Rda"
## [2] "-rw-rw-rw- 1 cpaismz cpaismz 7678196 sep 10 06:22 tmp3.Rda"
## [3] "tmp3.Rda: gzip compressed data, from Unix"
```

- Windows: as in the UNIX case, we count the number of characters and check the file's size and format in order to check our general analysis.

```
# Check file size and type by calling bash from windows: Windows format
system('bash -c "cd ~/ps2/windows
       wc -m tmp3.Rda
       ls -l tmp3.Rda
       file tmp3.Rda"', intern = TRUE)

## [1] "3977053 tmp3.Rda"
## [2] "-rwxrwxrwx 1 cpaismz cpaismz 7678064 sep 10 06:00 tmp3.Rda"
## [3] "tmp3.Rda: gzip compressed data, from HPFS filesystem (OS/2, NT)"
```

Based on the outputs from these codes, we can easily check that we no longer have a perfect relationship between the number of characters inside the file and the size (they are not equal). In addition, there is still a difference between UNIX and Windows file's size. Important is to note that the *file* command indicates in both cases that we are dealing with a gzip file. Hence, there are no inconsistencies with these values (sizes).

Note: UNIX and Windows files for tmp3.csv were not created using the same random seed and therefore, their sizes do not need to match or be similar.

iv) For the fourth file (tmp4.csv), we are including the same 1 million random numbers but in this case, we are saving it in a .csv file using the *write.table()* function in R. Thus, no compression is performed and we can follow the same analysis from the beginning of this section: we can count the number of characters inside the file, such that we know that one character represents a size of 1 byte in the .csv file (numbers, negative signs and new line characters).

Once we created the files (in */ps2/windows* when using the Windows version of R) we can check the size of each file and analyze it:

- UNIX: we do not know the total number of characters, hence, we can count them using the *wc -m* command and we can then check the size of the file and its format.

```
# Check file size and type by calling bash from windows: UNIX format
system('bash -c "cd ~/ps2/unix
       wc -m tmp4.csv
       ls -l tmp4.csv
       file tmp4.csv"', intern = TRUE)

## [1] "18152709 tmp4.csv"
## [2] "-rw-rw-rw- 1 cpaismz cpaismz 18152709 sep 10 06:22 tmp4.csv"
## [3] "tmp4.csv: ASCII text"
```

- Windows: same as with UNIX, we do not know the total number of characters, hence, we can count them using the *wc -m* command and we can then check the size of the file and its format.

```
# Check file size and type by calling bash from windows: Windows format
system('bash -c "cd ~/ps2/windows
       wc -m tmp4.csv
       ls -l tmp4.csv
       file tmp4.csv"', intern = TRUE)

## [1] "19159795 tmp4.csv"
## [2] "-rwxrwxrwx 1 cpaismz cpaismz 19159795 sep 10 06:00 tmp4.csv"
## [3] "tmp4.csv: ASCII text, with CRLF line terminators"
```

Based on the previous outputs, we can clearly check that the size of the files is equal to the number of characters inside them. Hence, our analysis is consistent with the fact that no compression has been made to any of these files. **Note**: UNIX and Windows files for tmp4.csv were not created using the same random seed and therefore, their sizes do not need to match or be similar.

v) For the fifth file (tmp5.csv), we are including the same 1 million random numbers but in this case, we are rounding them to (up to: for example 0 remains as 0 and 0.X are rounded down to 0.) two decimals before saving them in to .csv file using the *write.table()* function in R. Again, no compression is performed and we can follow the same analysis from the previous section: we can count the number of characters inside the file, such that we know that one character represents a size of 1 byte in the .csv file (numbers, negative sign, and new line characters).

In addition, we can easily check that the new size of the file is smaller than in tmp4.csv:

Once we created the files (in */ps2/windows* when using the Windows version of R) we can check the size of each file and analyze it:

- UNIX: we do not know the total number of characters, hence, we can count them using the *wc -m* command and we can then check the size of the file and its format.

```
# Check file size and type by calling bash from windows: UNIX format
system('bash -c "cd ~/ps2/unix
       wc -m tmp5.csv
       ls -l tmp5.csv
       file tmp5.csv"', intern = TRUE)

## [1] "5377265 tmp5.csv"
## [2] "-rw-rw-rw- 1 cpaismz cpaismz 5377265 sep 10 06:22 tmp5.csv"
## [3] "tmp5.csv: ASCII text"
```

- Windows: we do not know the total number of characters, hence, we can count them using the *wc -m* command and we can then check the size of the file and its format.

```
# Check file size and type by calling bash from windows: Windows format
system('bash -c "cd ~/ps2/windows
       wc -m tmp5.csv
       ls -l tmp5.csv
       file tmp5.csv"', intern = TRUE)

## [1] "6378819 tmp5.csv"
## [2] "-rwxrwxrwx 1 cpaismz cpaismz 6378819 sep 10 06:00 tmp5.csv"
## [3] "tmp5.csv: ASCII text, with CRLF line terminators"
```

Based on the previous outputs, we can clearly check that the size of the files is equal to the number of characters inside them. Hence, our analysis is consistent with the fact that no compression has been made to any of these files. Note: UNIX and Windows files for tmp5.csv were not created using the same random seed and therefore, their sizes do not need to match or be similar.

b) Compression and repetitions

As we already indicated in the previous section, the *save()* function performs a *gzip* compression by default, creating a binary file *.rda* that follows that compression method. Therefore, we end up with a non-human readable file with a smaller size due to the compression process. In particular, the *gzip* compression method uses the Lempel-Ziv coding (LZ77), an algorithm that compresses the file by replacing repeated occurrences inside the data file using references to a single copy of that particular data in the uncompressed data stream. As we can find in [1], "*rather than use an abbreviation table, LZ77 cleverly leaves the first occurrence of a repeated wor d in its original position in the text, and when the repetition occurs, the repeated word is replaced by: go backwards XX characters and copy the YY letters at that point*".

This overall process can be understood inside the "data deduplication" concept, a technique for eliminating duplicate copies of repeating data. In [2], this process is defined as a process where "*unique chunks of data, or byte patterns, are identified and stored during a process of analysis. As the analysis continues, other chunks are compared to the stored copy and whenever a match occurs, the redundant chunk is replaced with a small reference that points to the stored chunk. Given that the same byte pattern may occur dozens, hundreds, or even thousands of times (the match frequency is dependent on the chunk size), the amount of data that must be stored or transferred can be greatly reduced*".

We will analyze the different sizes of the file tmp6.Rda and tmp7.Rda using the previous definitions and information.

```
# Setting the working directory (UNIX)
setwd("~/ps2/unix")

chars <- sample(letters, 1e6, replace = TRUE)
chars <- paste(chars, collapse = '')
save(chars, file = 'tmp6.Rda')
```

Once we created the files (in */ps2/windows* when using the Windows version of R) we can check the size of each file and analyze it:

```
# Check file size and type by calling bash from windows: UNIX format
system('bash -c "cd ~/ps2/unix
       wc -m tmp6.Rda
       ls -l tmp6.Rda
       file tmp6.Rda"', intern = TRUE)

## [1] "344641 tmp6.Rda"
## [2] "-rw-rw-rw- 1 cpaismz cpaismz 635323 sep 12 04:21 tmp6.Rda"
## [3] "tmp6.Rda: gzip compressed data, from Unix"
```

- UNIX: we know the total number of characters, however, since the generated file is a compressed binary one -and as we already mentioned in the previous section- we cannot use the number of characters as the size of the file.

```
# Check file size and type by calling bash from windows: Windows format
system('bash -c "cd ~/ps2/windows
       wc -m tmp6.Rda
       ls -l tmp6.Rda
       file tmp6.Rda"', intern = TRUE)

## [1] "344457 tmp6.Rda"
## [2] "-rwxrwxrwx 1 cpaismz cpaismz 635217 sep 14 01:53 tmp6.Rda"
## [3] "tmp6.Rda: gzip compressed data, from HPFS filesystem (OS/2, NT)"
```

- Windows: same as for UNIX.

In order to check and complete our analysis, we will generate four new files using different options for the save command:

1. The first one will use the ascii flag, generating a non-compressed file
2. The second one will be an ascii file that is compressed using *gzip* format
3. The third one will consist of a non-ascii file without compression.
4. The last one will be a non-ascii file with *xz* compression

```
# Generate the extra files for the analysis (different save options)
save(chars, file = 'tmp6_V2.Rda', ascii = TRUE)
save(chars, file = 'tmp6_V3.Rda', ascii = TRUE, compress = "gzip")
save(chars, file = 'tmp6_V4.Rda', ascii = FALSE, compress = FALSE)
save(chars, file = 'tmp6_V5.Rda', ascii = FALSE, compress = "xz")
save(chars, file = 'tmp6_V6.Rda', ascii = FALSE, compress = "gzip")
```

- UNIX and Windows: Now we can compare all files sizes and types. Clearly, we expect smaller file sizes for those where a compression is performed (Original, V3, V5 and V6) while for the others (V2 and V4), we would be able to obtain their sizes (larger sizes) by simply count the number of characters inside them (no compression has been made).

```
# Check files sizes and types by calling bash from windows: UNIX format
system('bash -c "cd ~/ps2/unix
       echo \\"Files sizes comparison:\\"
       ls -l tmp6.Rda tmp6_V2.Rda tmp6_V3.Rda tmp6_V4.Rda tmp6_V5.Rda tmp6_V6.Rda
       echo \\"Files types comparison:\\"
       file tmp6.Rda tmp6_V2.Rda tmp6_V3.Rda tmp6_V4.Rda tmp6_V5.Rda tmp6_V6.Rda"',
       intern = TRUE)

##  [1] "Files sizes comparison:"
##  [2] "-rw-rw-rw- 1 cpaismz cpaismz  635323 sep 12 04:21 tmp6.Rda"
##  [3] "-rw-rw-rw- 1 cpaismz cpaismz 1000070 sep 12 04:21 tmp6_V2.Rda"
##  [4] "-rw-rw-rw- 1 cpaismz cpaismz  635340 sep 12 04:21 tmp6_V3.Rda"
##  [5] "-rw-rw-rw- 1 cpaismz cpaismz 1000060 sep 12 04:21 tmp6_V4.Rda"
##  [6] "-rw-rw-rw- 1 cpaismz cpaismz  606296 sep 12 04:22 tmp6_V5.Rda"
##  [7] "-rw-rw-rw- 1 cpaismz cpaismz  635323 sep 12 04:22 tmp6_V6.Rda"
##  [8] "Files types comparison:"
##  [9] "tmp6.Rda:    gzip compressed data, from Unix"
## [10] "tmp6_V2.Rda: ASCII text, with very long lines"
## [11] "tmp6_V3.Rda: gzip compressed data, from Unix"
## [12] "tmp6_V4.Rda: data"
## [13] "tmp6_V5.Rda: XZ compressed data"
## [14] "tmp6_V6.Rda: gzip compressed data, from Unix"
```

As we expected based on the *save()* function arguments, the original file and the V6 file have exactly the same size (and they are the same type) because we are simply given the default options (explicitly) for generating the sixth version of the file. On the other hand, we can see that V2 and V4 reached the largest sizes due to the lack of compression. Interesting is to note that the *xz* compression obtains the best performance in terms of reducing ratio (smallest file size). Note also that V2 and V4 differ in their format: while V2 is treated as a very long ASCII text, V4 is identified as a data file, therefore, some (small) differences in terms of sizes are detected.

Now, we can generate the file with repeated characters (tmp7.Rda) in order to compare its size with tmp6.Rda and check our initial analysis regarding the performance of the compression algorithms were repeated patterns can be found inside the data:

```
# Setting the working directory (UNIX)
setwd("~/ps2/unix")
```

```
# Create the file with repeated characters
chars <- rep('a', 1e6)
chars <- paste(chars, collapse = '')
save(chars, file = 'tmp7.Rda')
```

Once we created the files (in */ps2/windows* when using the Windows version of R) we can check the size of each file and analyze it:

```
# Check file size and type by calling bash from windows: UNIX format
system('bash -c "cd ~/ps2/unix
       wc -m tmp7.Rda
       ls -l tmp7.Rda
       file tmp7.Rda"', intern = TRUE)

## [1] "1020 tmp7.Rda"
## [2] "-rw-rw-rw- 1 cpaismz cpaismz 1056 sep 12 04:11 tmp7.Rda"
## [3] "tmp7.Rda: gzip compressed data, from Unix"
```

- UNIX: as before, we know the total number of characters, however, since the generated file is a compressed binary one -and as we already mentioned in the previous section- we cannot use the number of characters as the size of the file.

```
# Check file size and type by calling bash from windows: Windows format
system('bash -c "cd ~/ps2/windows
       wc -m tmp7.Rda
       ls -l tmp7.Rda
       file tmp7.Rda"', intern = TRUE)

## [1] "1020 tmp7.Rda"
## [2] "-rwxrwxrwx 1 cpaismz cpaismz 1056 sep 14 01:53 tmp7.Rda"
## [3] "tmp7.Rda: gzip compressed data, from HPFS filesystem (OS/2, NT)"
```

- Windows: same as for UNIX.

Again, we generate the extra files using different arguments inside the *save()* function for completeness:

```
save(chars, file = 'tmp7_V2.Rda', ascii = TRUE)
save(chars, file = 'tmp7_V3.Rda', ascii = TRUE, compress = "gzip")
save(chars, file = 'tmp7_V4.Rda', ascii = FALSE, compress = FALSE)
save(chars, file = 'tmp7_V5.Rda', ascii = FALSE, compress = "xz")
save(chars, file = 'tmp7_V6.Rda', ascii = FALSE, compress = "gzip")
```

- UNIX and Windows: Now we can compare all files sizes and types. Clearly, we expect smaller file sizes for those where a compression is performed (Original, V3, V5 and V6) while for the others (V2 and V4), we would be able to obtain their sizes (larger sizes) by simply count the number of characters inside them (no compression has been made).

```
# Check files sizes and types by calling bash from windows: UNIX format
system('bash -c "cd ~/ps2/unix
       echo \\"Files sizes comparison:\\"
       ls -l tmp7.Rda tmp7_V2.Rda tmp7_V3.Rda tmp7_V4.Rda tmp7_V5.Rda tmp7_V6.Rda
       echo \\"Files types comparison:\\"
       file tmp7.Rda tmp7_V2.Rda tmp7_V3.Rda tmp7_V4.Rda tmp7_V5.Rda tmp7_V6.Rda"',
       intern = TRUE)

##  [1] "Files sizes comparison:"
##  [2] "-rw-rw-rw- 1 cpaismz cpaismz    1056 sep 12 04:11 tmp7.Rda"
##  [3] "-rw-rw-rw- 1 cpaismz cpaismz 1000070 sep 12 04:11 tmp7_V2.Rda"
##  [4] "-rw-rw-rw- 1 cpaismz cpaismz    1057 sep 12 04:11 tmp7_V3.Rda"
##  [5] "-rw-rw-rw- 1 cpaismz cpaismz 1000060 sep 12 04:11 tmp7_V4.Rda"
##  [6] "-rw-rw-rw- 1 cpaismz cpaismz     320 sep 12 04:11 tmp7_V5.Rda"
##  [7] "-rw-rw-rw- 1 cpaismz cpaismz    1056 sep 12 04:42 tmp7_V6.Rda"
##  [8] "Files types comparison:"
##  [9] "tmp7.Rda:    gzip compressed data, from Unix"
## [10] "tmp7_V2.Rda: ASCII text, with very long lines"
## [11] "tmp7_V3.Rda: gzip compressed data, from Unix"
## [12] "tmp7_V4.Rda: data"
## [13] "tmp7_V5.Rda: XZ compressed data"
## [14] "tmp7_V6.Rda: gzip compressed data, from Unix"
```

As expected, the sizes' pattern is the same as with the previous file, however, file sizes are clearly smaller than the ones obtained with the random vector inside tmp6.Rda, although the number of characters remains the same. The explanation behind this pattern lies on the previous results and the general analysis stated at the beginning of this section, the file tmp7.Rda presents a reduced size in comparison to tmp6.csv due to the fact that it contains only one character that is repeated 1 million times. Hence, the compression algorithm can take advantage of it using the fact that repetitions can be referenced in order to reduce the size of the file, obtaining very good compression ratios. Since we know the way that *gzip* works, we can easily understand that the difference in sizes radicates in this deduplication process performed by the compression.

Finally, note that as we expected, the size of tmp6_V2.Rda and tmp4_V4.Rda files are equal to the ones obtained in tmp7_V2.Rda and tmp7_V4.Rda respectively. The explanation is simple: in both cases no compression is performed and thus the size of the files is simply the total number of characters inside them and there is no difference between recording a random vector of characters or one that consists of a unique (repeated) character.

Problem 2

The general strategy to solve this problem consists of gathering all the necessary information from the *.html* files that can be found in the relevant Google Schoolar urls. At the beginning, our code obtains the *.html* file that defines the website reached right after entering the name of the researcher in http://scholar.google.com. Based on the structure of this file and the pattern of the queries, we can obtain the user ID (if the name is a valid and registered author's name) and the corresponding link to its personal citation website. Using this information, we can obtain the HTML file associated with the author's personal citation website. All these operations are performed by our first function called *CitesScholar()* such that the input is the name of the researcher and the output is a list object containing both the user ID and the HTML file from the personal citation website.

Once we have a parsed HTML object from the author's personal website, we can easily analyze it and check/understand its structure. Thus, we are able to perform a series of queries using the XML library functions and some post-processing with the help of the stringr library. By the end of the process, we obtain five vectors with all the relevant information per article: Title, Authors, Journal, Year, and number of Citations. Note that some authors may have some articles where not all the information is available (blank fields) or papers that appear multiple times with different number of citations, in those cases, we leave them 'as-is' in our table, being consistent with the information provided by the Google Scholar website.

Since Google Scholar website could try to block our connections after repeating the procedure, we develop an extra function called *DownloadfromScholar()* that, instead of creating a HTML file on the fly using *readLines()* function, downloads the relevant *.html* files to the current working directory and then parses them. Thanks to this implementation, we are able to reuse previous queries and create their correspondent DataFrames. Thus, we can use this function as an alternative when our original implementation is being blocked.

All the previous operations, starting from the parsed HTML file (from the personal citation website) and ending with a DataFrame with all the relevant information are performed by our function *CreateDataFrame_subset()* when dealing with only the first 20 publications and by our function *CreateDataFrame()* for the case when we include all the author's publications inside our DataFrame. A slight modification to the original link and a loop must be added in order to be able to perform a series of queries that allow us to get all the publications.

Finally, some tests are performed to our functions using the package testthat as required in section c).

Thus, we have the following steps:

i) Download/extract the *.html* file associated with the provided author's name.

ii) Obtain the user ID and personal website link from it.

iii) Download/extract the HTML file from the author's profile, parse it.

iv) Process it and create the relevant columns.

v) Return a DataFrame containing the relevant information.

a) Google Scholar citation function: ID and html

The function's input consists of the author's name, only allowing names that contain a first and a last name separated by a blank space, following the example provided for Geoffrey Hinton. Its outputs are

the user ID and the parsed HTML associated with the author's personal citation website.

The implementation is as follows:

1. We declare our function *CitesScholar()* that requires the name of the author as main input. In order to get access to all the functions that are used inside of it, the main libraries are invoked and warnings are suppresed for visualization purposes. In addition, two "valid input" checks are performed: (1) determining if the input is a character or not. If not, then an error message is printed to the main console and the function is no longer executed, and (2) checking if the given name follows the specific pattern we want (using a regular expression) from the user where a first name must be followed by an empty space and then by the last name of the author. If not, an error message with a usage example is given to the user via the console terminal.

```
# Function declaration
CitesScholar <- function(name) {
  # Loading libraries
  library(XML)
  library(RCurl)
  library(stringr)

  # Turning off the warnings for visualization purposes
  options(warn = -1)

  # Check if the input is a string
  if (!is.character(name)){
    stop("The input must be a name (e.g Geoffrey Hinton)")
  }

  # Check if we have only letters and a first and last name
  if(grepl("^[[:alpha:]]+[[:space:]]+[[:alpha:]]+$", name) == FALSE){
    stop("Only two words (no digits) are allowed as the author's name.
         The name string must have a space between the first and last names,
         please try again (e.g Geoffrey Hinton)")
  }
```

2. After passing the checks, we know that the user has given a valid input, thus, we extract the first and last names from the original variable name based on the blank space between them and the initial url is generated based on the structure followed by the Google Scholar website. This url is the one that a normal user will get after writing the name of the author in the search field of the Google Scholar main web (http://scholar.google.com)

```
  # Separate the name in first and last names
  firstname <- strsplit(name, " ")[[1]][[1]]
  lastname <- strsplit(name, " ")[[1]][[2]]

  # Generate the URL with the profile in google scholar
  urlInit <- paste("https://scholar.google.com/scholar?hl=es&q=", firstname,
                   "+", lastname, "&btnG=&lr=", sep = "")
```

3. We extract the HTML file information from the generated url using the classic *readLines()* function, obtaining a rawHTML file. Then we parse this file thanks to the function included in the XML library called *htmlParse()*. Once we have the parsed HTML object, we can obtain the link associated with the author's personal website thanks to our knowledge of the *.html* structure. Hence, we perform a query that returns the link (attribute) associated with a node that contains it.

```
# Get the Google Scholar ID and the link to the personal citations website
rawHTML <- readLines(urlInit)

# Parse the raw HTML code
parsedHTML <- htmlParse(rawHTML)

# Find the link to the author's personal website
link <- xpathSApply(parsedHTML, "//h4[@class = 'gs_rt2']//a",
                    xmlGetAttr, "href")
```

4. We include an additional check that validates the existence of a link. If the query returns an empty value/string, it means that the author does not have a personal citation website and therefore, it does not exist inside the Google Scholar Database as a registered researcher. In that case, an error message is given to the user.

```
# Check if there are valid links inside the file: if not, stop
if (length(link) <= 0) {
  stop("New error: The author does not exist in the Google Scholar Database,
       please try again with a different name")
}
```

5. Once the link to the personal website is obtained, it can be processed using regular expressions in order to extract the user ID from it as a string. Again, an extra safety check is included, testing the case that no user ID is found indicating that there is no personal website available for the given author name.

```
# Extract the user
UserID <- str_match(link, ".*user=([\\d\\w]+)&")[[2]]

# If no User ID, then author does not exists in Google Scholar: stop
if (is.na(UserID) == TRUE){
  stop("The author does not exist in the Google Scholar Database,
       please try again with a different name")
}
```

6. Finally, the url associated with the author's personal citation website is generated based on the structure that we previously identified from the Google Scholar website. A new HTML extraction is performed via the *readLines()*. After parsing the resulting object, a list containing both the user ID and parsed HTML code is created in order to be able to return multiple objects via the return statament. This is a peculiarity of R: the return statement does not allow multiple arguments, hence, we can create a list that contains all the elements that we want to return from the function.

```
  # Get the citations html
  urlcites <- paste("https://scholar.google.com", link, sep = "")
  htmlcites <- readLines(urlcites)
  parhtmlcites <- htmlParse(htmlcites)

  # Return the user ID and parsed html from the personal citations web
  returnlist <- list(UserID, parhtmlcites)
  return(returnlist)
}
```

Therefore, the function returns the user ID and the HTML object (parsed) that contains all the information of the author's personal website. From this, we will be able to extract the title, journal, authors, year of publication, and number of citations of each of his/her published articles.

a.2) Extra function

As we indicated above, we developed a second function for performing the previous operations in order to have an alternative method in the case that Google Scholar blocks our queries. In this case, instead of extracting the HTML file information using the *readLines()* function, we are downloading the relevant *.html* files and then we are processing them via the *download.file()* function (similar to the wget function in bash).

Our alternative function is as follows:

1. All the code (except for the name of the function) is identical up to the second chunk of code stated above. Then, for the third chunk of code, instead of using the *readLines()* function, we download the content of the html file into a *.html* file with the name of the author. After downloading it, we simply parse its content into a new variable.

```
  # Generate the initial URL to download the .html file
  urlInit <- paste("https://scholar.google.com/scholar?q=", firstname,
                   "+", lastname, "&hl=en&as_sdt=0,5", sep = "")
  filename <- paste(firstname, lastname, ".html", sep = "")

  # Download the file and parse the html file
  download.file(urlInit, filename)
  parsedHTML <- htmlParse(readLines(filename))
```

2. Then, the other modification occurs at the end of the function (chunk number 6 in the previous function) where we again download the HTML file from the author's personal citation website instead of extracting it using the *readLines()* function. We parse it and we create the list for returning both the user ID and the parsed HTML object.

```
  # Get the citations html: generate the url and download it
  urlcites <- paste("https://scholar.google.com", link, sep = "")
  filenamecites <- paste(UserID, ".html", sep = "")
  download.file(urlcites, filenamecites)

  # Parse the new html
```

```
  parhtmlcites <- htmlParse(readLines(filenamecites))

  # Return the user ID and parsed html from the personal citations web
  returnlist <- list(UserID, parhtmlcites)
  return(returnlist)
}
```

With this slight modifications, we are able to perform queries when our previous function is being blocked by the Google Scholar server. We can easily use the *CitesScholar()* (or the *Downloadfrom-Scholar()*) function as follows:

```
# Obtain the User ID and HTML object of the author
ResultsList <- CitesScholar("Geoffrey Hinton")

# Separate the UserID and Authorhtml (for simplicity)
UserID <- ResultsList[[1]]
Authorhtml <- ResultsList[[2]]

# Display them (summary of the html file for visualization purposes)
# User ID
UserID

# HTML summary
summary(Authorhtml)
```

b) First 20 citations as DataFrame

The following function *CreateDataFrame_subset()* takes as input the parsed HTML object obtained as the second output from the previous *CitesScholar()* function. Based on the structure of this file, a series of queries are performed in order to obtain the Title, Authors, Journals, Years and number of Citations associated with each paper/article published and indexed by the Google Scholar engine for the corresponding author. After all queries are done, some final checks regarding the length and type of vectors are performed in order to avoid errors related to missing values or wrong type entries (like strings inside the number of citations or year). Finally, a DataFrame is generated and returned to the user including the five main fields for the first 20 articles (ordered by the number of citations) presented in the author's personal citation website.

The details are as follow:

1. We declare the function including the parsed HTML object as the main output and all the necessary libraries are invoked. Warnings are omitted for visualization purposes as before.

```
# Function definition
CreateDataFrame_subset <- function(html){
  # Libraries
  library(XML)
  library(RCurl)
  library(stringr)

  # Turning off the warnings for visualization purposes
  options(warn = -1)
```

2. A first check related to the input type is performed in order to validate its format. If the type is not the adequate for the function (parsed HTML) then an error message is output to the user and the function stops. Otherwise, the first query is performed, generating the Titles vector containing the first 20 publications of the author (ordered by the number of citations, from higher to lower). A second check is performed: if the query is empty, it means that the current HTML object does not contain citations from the Google Scholar website, indicating that the author's citation page provided does not have any article or that it is not registered in Google Scholar. In this case, the function stops and an error message is prompted to the user.

```
  # Check if the file is a parsed HTML
  if (typeof(html) != "externalptr"){
    stop("Not a valid html file from author's citation website")
  }

  # Articles titles
  Titles <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//tr[@class ='gsc_a_tr']
                        //td[@class = 'gsc_a_t']//a[@class = 'gsc_a_at']",
                        xmlValue, trim = TRUE, encoding = "UTF-8")

  # Check if there are valid titles: if not, not a valid file
  if (length(Titles) <= 0){
    stop("No Titles: There is no author's citation page inside Google Scholar
         associated with the html file being processed, please
```

```
        try again with a different file")
}
```

3. If all preliminary checks are passed, a query is performed in order to get the Journal and the Authors of each entry. Note that we are explicitly encoding the result of the query using UTF-8 by default, allowing the function to deal with non-common characters like European accents/characters and Slavic letters. In addition, we can notice that the query is based on the knowledge of the HTML tree structure. Once the query is performed, the journal and authors information is split using our knowledge of the resulting string from the query (Journals are located on even entries while Authors are located on odd entries). The year of publication is deleted from the journal information.

```
# Journals and authors array
JournalAuthors <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//
                              tr[@class ='gsc_a_tr']//td[@class = 'gsc_a_t']
                              //div[@class = 'gs_gray']", xmlValue,
                              trim = TRUE, encoding = "UTF-8")

# Getting the Journals (even numbers of previous vector)
Journals <- JournalAuthors[seq(2, length(JournalAuthors), 2)]

# We delete the year of publication after the Journal
Journals <- substring(Journals, 0, nchar(Journals) - 6)

# Getting the authors (odd numbers)
Authors <- JournalAuthors[seq(1, length(JournalAuthors), 2)]
```

4. More queries are performed in order to obtain the Year of publication and the number of Citations associated with each entry/paper. In order to deal with non-numerical elements (like *, empty spaces, etc.) inside the Citations vector, we transform it into a numeric vector, obtaining NA values in those conflicting entries. Then, we simply remove all the NA values from inside. Note that since Google Scholar orders the articles by number of citations, this procedure will not produce any inconsistency since in the case that some entries are null (0 citations), we will fill the Citations vector with 0s in the next step.

```
# Years of publication
Year <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//tr[@class ='gsc_a_tr']
                    //td[@class = 'gsc_a_y']//span[@class = 'gsc_a_h']",
                    xmlValue, trim = TRUE, encoding = "UTF-8")

# Number of citations (including repeated articles)
Citations <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']
                         //tr[@class ='gsc_a_tr']//td[@class = 'gsc_a_c']
                         //a", xmlValue, trim = TRUE, encoding = "UTF-8")

# Delete potential non numerical elements by transforming and filtering the array
Citations <- as.numeric(Citations)
remove <- c(NA)

# Re-Transform it to a character array for the DF
Citations <- as.character(Citations[!Citations %in% remove])
```

5. If there are some articles without Citations, we fill those entries with 0s in order to match the length of the vector with the other fields for creating a consistent DataFrame.

```
  # Dimension checks: fill with 0 empty citations
  if (length(Citations) != length(Titles)){
    Citations <- c(Citations, integer(length(Titles) - length(Citations)))
  }
```

6. Finally, we declare the name of each column of the DataFrame and we generate it from the data gathered by the previous queries.

```
  # Create a DataFrame
  x_name <- "Title"
  y_name <- "Authors"
  z_name <- "Journals"
  w_name <- "Year"
  v_name <- "Citations"

  # Create the DF using the reshape2 melt function for simplicity
  require(reshape2)
  ScholarDF <- melt(data.frame(Titles, Authors, Journals, Year, Citations))

  # Define the columns' names
  colnames(ScholarDF) <- c(x_name, y_name, z_name, w_name, v_name)

  # Return the DataFrame
  return(ScholarDF)
}
```

Therefore, the function returns the DataFrame with the first 20 publications of the author under study based on the parsed HTML code obtained from the previous function *CitesScholar()* (or *DownloadfromScholar()*).

We can easily use one of these functions and then we can run our *CreateDataFrame_subset()* function:

```
# Call the function and create a DataFrame from the author's website
ResultsList <- CitesScholar("Geoffrey Hinton")

# Separate the UserID and Authorhtml (for simplicity)
UserID <- ResultsList[[1]]
Authorhtml <- ResultsList[[2]]

# Create the data frame
SubsetScholarDF <- CreateDataFrame_subset(Authorhtml)

# Display the head of the DataFrame
head(SubsetScholarDF)

# Printing a summary using the apply function
apply(SubsetScholarDF, 1:5, function(x) substr(x,start = 0,stop = 10))
```

c) Testing our code

i) We have already included checks inside all our functions, hence, we develop some tests using the *testthat* library in the next section.

ii) In order to perform some tests using the *testthat* library in our two functions *CitesScholar()* and *CreateDataFrame_subset()*, we generate two *.R* files containing them and two associated test files: *CitesScholar.R*, *CreateDataFrame_subset()*, and *test_CitesScholar.R*, *test_CreateDataFrame_subset()* respectively.

CitesScholar Tests

For this function we perform three main types of tests: (1) Test the function output using a known researcher name as input, (2) Test the function using a non-registered Google Scholar author and (3) Check different input possibilities and how our function is handling them.

1. We check if the output obtained is a list if its length is valid (2) including the user ID + HTML object, and we check the type of each component inside the list. We test it with a well-known researcher name such as Albert Einstein.

```
# Loading the function file to be tested
source("CitesScholar.R")

# Testing the CitesScholar function (input = HTML, output = list)
test_that("Types and lengths are consistent when using a real author's name", {

  # As an example, we are using the citation website of the well known Albert Einstein
  ResultsList <- CitesScholar("Albert Einstein")

  # Check if the final output is a list
  expect_that( ResultsList, is_a("list") )

  # Check its length
  expect_that( length(ResultsList), equals(2) )

  # Check the types of both outputs
  expect_that( ResultsList[[1]], is_a("character") )
  expect_that( ResultsList[[2]], is_a("HTMLInternalDocument") )
})
```

2. Here we test how the function handles the situation where the given name is valid but it is not registered in Google Scholar.

```
  # Testing the CitesScholar function (input = HTML, output = list)
test_that("Output when a non-existent author name is provided", {
  # A non-registered user name is provided
  ResultsList <- CitesScholar("Cristobal Pais")
})
```

3. Finally, we perform a series of tests for checking how our function handles different non-valid input situations.

```
# Testing the CitesScholar function for different inputs
test_that("Only first name", {
  # One word is provided
  ResultsList <- CitesScholar("Albert")
})

test_that("More than two names", {
  # More than 2 words are provided
  ResultsList <- CitesScholar("Albert Einstein Canalejo")
})

test_that("Words and numbers", {
  # First and last name with a number
  ResultsList <- CitesScholar("Albert Einstein2")
})

test_that("Words and numbers (mixed)", {
  # Numbers in between characters
  ResultsList <- CitesScholar("Albert E1nstein")
})

test_that("No space between names", {
  # First and last names are not separated
  ResultsList <- CitesScholar("AlbertEinstein")
})

test_that("Uppercase name", {
  # Name is written in uppercase
  ResultsList <- CitesScholar("ALBERT EINSTEIN")
})

test_that("Lowercase name", {
  # Name is written in lowercase
  ResultsList <- CitesScholar("albert einstein")
})

test_that("Erroneous input is provided (numbers)", {
  # Numeric input
  ResultsList <- CitesScholar(1123)
})

test_that("Erroneous input is provided (digit strings)", {
  # String with digits
  FinalDF2 <- CitesScholar("1222")
})
```

In order to run these tests, we can use the following simple code:

```
# Turning off the warnings for visualization purposes
options(warn = -1)
```

```
# Loading the library
suppressMessages(library(testthat))

# Set working directory
setwd("C:/Users/chile/ps2/RTests/")

# Invoking the test
test_file("test_CitesScholar.R")
```

Based on the output we can see that we get a series of warnings regarding the end of the html files we are using for testing. This is due to the fact that these files do not end with a \ n or a \ r \ n characters and thus, the system outputs the corresponding warning. Then, we can easily check that all our tests for checking the input valid formats (when no valid ones were provided) ended as expected: stopping the function and displaying an error message to the user regarding the structure of a valid input argument. On the other hand, tests for checking the output types and their format when a valid input name is given are passed without any errors, as expected.

CreateDataFrame_subset Tests

For this function we also perform three main types of tests: (1) Test the function output using a known researcher name as input, (2) Test the function using a non-existent/non-valid Google Scholar author and (3) Check different input possibilities and how our function is handling them.

Important is to note that in this case since the function uses a HTML object as main input, we are using some previously downloaded *html* files using our *DownloadfromScholar()* function presented in section a). This allows us to perform valid and useful tests.

1. We check if the output obtained is a Dataframe, if the length of its columns is valid and if the number of citations does not contain NA values after transforming it to a numeric object. Again, we are using Albert Einstein as our testing researcher with the difference that in this case, we are loading its HTML file, previously downloaded using *DownloadfromScholar()* (see how we are running the tests at the end of this section).

```
# Loading the function file to be tested
source("CreateDataFrame_subset.R")

# Testing the CreateDataFrame_subset function (input = HTML, output = Dataframe)
test_that("Types and lengths are consistent when using a known html file", {
  # Save the output in the FinalDF variable, reading a html file from Google Scholar
  # As an example, we are using the citation website of Albert Einstein
  FinalDF <- CreateDataFrame_subset(htmlParse(readLines("~/ps2/download/
                                                        qc6CJjYAAAAJ.html")))

  # Check if the final output is a DataFrame
  expect_that( FinalDF, is_a("data.frame") )

  # Check if all columns have the same length
```

```
  expect_that( length(FinalDF$Title), equals(length(FinalDF$Authors)) )
  expect_that( length(FinalDF$Title), equals(length(FinalDF$Year)) )
  expect_that( length(FinalDF$Title), equals(length(FinalDF$Citations)) )
  expect_that( length(FinalDF$Title), equals(length(FinalDF$Journals)) )

  # Check if the Citations column has only numbers
  expect_that( as.numeric(FinalDF$Citations), is_a("numeric") )

})
```

2. Here we test how the function handles the situation where the given HTML is not valid (does not contain the information expected by the function).

```
# Testing the CreateDataFrame_subset function for different inputs
test_that("Output when a non-existent user html file is provided", {
  # Save the output in the FinalDF variable, after reading a .html
  # file from the Google Scholar website, but from a non-existing user
  FinalDF <- CreateDataFrame_subset(htmlParse(readLines("~/ps2/download/
                                                        nonexistent.html")))
})

# Another html file (not from Google Scholar)
test_that("A html file/code from any page", {
  # HTML downloaded from google.com
  FinalDF <- CreateDataFrame_subset(htmlParse(readLines("~/ps2/download
                                                        /index.html")))
})
```

3. Finally, we perform a series of tests for checking how our function handles different non-valid input situations.

```
# Numbers as inputs
test_that("Erroneous input is provided (numbers)", {
  # Testing with a numeric input
  FinalDF2 <- CreateDataFrame_subset(1222)
})

# String with numbers
test_that("Erroneous input is provided (digit strings)", {
  # String composed by digits
  FinalDF2 <- CreateDataFrame_subset("1222")
})

# Any random string
test_that("Erroneous input is provided (character strings)", {
  # Characters string
  FinalDF2 <- CreateDataFrame_subset("hello")
})
```

In order to run these tests, we can use the following simple code.

```
# Loading the library
suppressMessages(library(testthat))

# Set working directory
setwd("C:/Users/chile/ps2/RTests/")

# Downloading the files (using wget for simplicity): valid, non-existent and non-valid
system('bash -c "wget -q -O qc6CJjYAAAAJ.html "https://scholar.google.com/"\
"citations?user=qc6CJjYAAAAJ&hl=es&oi=ao""')
system('bash -c "wget -q -O nonexistent.html "https://scholar.google.com/scholar?"\
"q=Cristobal+Pais&btnG=&hl=es&as_sdt=0%2C5""')
system('bash -c "wget -q www.google.com"')

# Invoking the test
test_file("test_CreateDataFrame_subset.R")
```

Based on the output we can easily check that we again get some warnings regarding the end of the HTML files and the NAs values introduced by our function when we are cleaning the Citations array. None of these warnings are relevant for our implementation. Then, we can clearly see that all the tests developed when a valid input is given to the function (Albert Einstein html file in this example) are passed without any problem and then, we get a series of errors when non-valid input arguments are passed to our function: non-existing author query, numbers, strings containing numbers, a random string, and non-valid html files.

d) Extra Credit: All citations as DataFrame

Based on the code developed for section b), we just need to add some simple modifications in order to download all the author's articles. In this case, we will perform a series of queries based on our knowledge of how Google Scholar implements the "show more" logic when hitting the button at the bottom of the author's personal website, that will allow us to extract different HTML files structures, each one associated with (at most) one hundred publications. In order to obtain this information, we simply used the developer tools available in Google Chrome and we check the structure of the query that is triggered when we press the button.

The structure of the query consists of a starting and size values: the start value indicates the number of the article from which the query will start displaying minus 1 (e.g start = 0 implies displaying from the first article, start = 100 then 101, and so on) and the size value indicates the maximum number of articles that can be displayed at the same time (same window) with an internal limit of 100 entries. Hence, we will perform a series of queries of size 100, while changing the start value at every iteration until we obtain an empty query, indicating that no more articles are available for the author.

The function is as follows:

1. As in the original function *CreateDataFrame_subset()* we declare the function, load the libraries and perform the first check of the input, validating its type.

```
# Declaring the dunction
CreateDataFrame <- function(html){
  # Libraries
  library(XML)
  library(RCurl)
  library(stringr)

  # Turning off the warnings for visualization purposes
  options(warn = -1)

  # Check if the file is a parsed HTML
  if (typeof(html) != "externalptr"){
    stop("Not a valid html file from author's citation website
         please check the format of the input file")
  }
```

2. Then, the first difference consists of the fact that we are extracting (again) the author's personal citation website from the HTML object because we will need it for generating the iterative queries. Note that we could have simply used the user ID and then, based on the known structure of the link, generated it. However, since the function in part b) can only use the HTML object as input (as stated in the question), we are also satisfying that requirement in this case.

   A second check is performed: if there are no links, it means that it is not a valid HTML object for Google Scholar and the function stops.

```
  # Get the links inside the html
  links <- getHTMLLinks(html, baseURL = urlInit, relative = FALSE)
```

```
# Check if there are valid links in the file
if (length(links) <= 0){
  stop("Not a valid html file from author's citation website
       the file does not contain any valid link")
}

# Generate the link containing the query
url4queries <- paste("http://scholar.google.com", links[8], sep = '')
```

3. We generate five global vectors for our function associated with each of the columns of the desired DataFrame. On the other hand, we initialize the "from and size" values for the first query (0+1 to 100 papers).

```
# Global vectors for recording all the queries
GTitles <- c()
GJournals <- c()
GAuthors <- c()
GYears <- c()
GCitations <- c()

# Initial values: from (start) and size
from <- 0
size <- 100
```

4. A while loop starts with TRUE as the main condition, allowing us to use a *break* statement inside it. The url containing the query is generated using its known structure and we extract the content of the HTML file on that website. This HTML will contain up to 100 references, depending on the number of valid entries associated with the author's Google Scholar account.

```
# Loop for getting all the articles based on the query structure
while (TRUE){
  query <- paste("&cstart=",from,"&pagesize=",size,sep = '')
  urlq <- paste(url4queries,query,sep = '')
  html <- readLines(urlq)
  html <- htmlParse(html)
```

5. Same queries as in the original case are performed in order to extract all the relevant information from the HTML object. Important is to note that the *break* condition consists of returning an empty query (no new Titles), indicating that the author does not have more articles published and linked to the Google Scholar engine.

```
  # Articles titles
  Titles <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//
                        tr[@class ='gsc_a_tr']//td[@class = 'gsc_a_t']
                        //a[@class = 'gsc_a_at']", xmlValue,
                        trim = TRUE, encoding = "UTF-8")
```

```
# Break condition: no titles/papers inside the HTML object
if (length(Titles) == 0 ) {
  break
}

# Journals and authors array
JournalAuthors <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//
                              tr[@class ='gsc_a_tr']//td[@class = 'gsc_a_t']
                              //div[@class = 'gs_gray']", xmlValue,
                              trim = TRUE, encoding = "UTF-8")

# Getting the Journals (even numbers of previos vector)
Journals <- JournalAuthors[seq(2, length(JournalAuthors), 2)]

# We delete the year of publication after the Journal
Journals <- substring(Journals, 0, nchar(Journals)-6)

# Getting the authors (odd numbers)
Authors <- JournalAuthors[seq(1, length(JournalAuthors), 2)]

# Years of publication
Year <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//tr[@class ='gsc_a_tr']
                    //td[@class = 'gsc_a_y']//span[@class = 'gsc_a_h']",
                    xmlValue, trim = TRUE, encoding = "UTF-8")

# Number of citations (No repetitions)
Citations <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//
                         tr[@class ='gsc_a_tr']//td[@class = 'gsc_a_c']//a",
                         xmlValue, trim = TRUE, encoding = "UTF-8")

# Delete potential non numerical elements by transforming and filtering the array
Citations <- as.numeric(Citations)
remove <- c(NA)

# Re-Transform it to a character array for the DF
Citations <- as.character(Citations[!Citations %in% remove])
```

6. We update the content of the global vectors. Note that we are appending the vectors obtained from the new queries in order to preserve all values per iteration (not replacing them). We end the loop by updating the from value by 100 in order to perform a new query including the next (up to) 100 entries. Note that we can add a command like Sys.sleep(2) in between of the iterations in order to not perform a series of queries without some delay between them (good webscraping practices, as suggested in class).

```
# Record in global vectors
GTitles <- c(GTitles, Titles)
GAuthors <- c(GAuthors, Authors)
GJournals <- c(GJournals, Journals)
GYears <- c(GYears, Year)
```

```
    GCitations <- c(GCitations, Citations)

    # Update the counter
    from <- from + 100
  }
```

7. As in the original function, we perform a length check regarding the number of citations vector. We fill it with as many zeros as needed in order to match the length of the other vectors. This can be done without any inconsistency since the articles are ordered by number of citations starting with the highest one.

```
  # Dimension checks: fill with 0 empty citations
  if (length(GCitations) != length(GTitles)){
    GCitations <- c(GCitations, integer(length(GTitles) - length(GCitations)))
  }
```

8. The DataFrame information and labels are defined. The complete DataFrame (with all citations) is returned to the user.

```
  # Create a DataFrame
  x_name <- "Title"
  y_name <- "Authors"
  z_name <- "Journals"
  w_name <- "Year"
  v_name <- "Citations"

  require(reshape2)
  ScholarDF <- melt(data.frame(GTitles, GAuthors, GJournals, GYears, GCitations))
  colnames(ScholarDF) <- c(x_name, y_name, z_name, w_name, v_name)

  # Return the DataFrame
  return(ScholarDF)
}
```

Thus, we extended our original function such that the new one uses the same HTML object as input but returns all the results for a researcher instead of only the first 20. We can easily run the function using the following code:

```
# Call the function and create a DataFrame from the author's website
ResultsList <- CitesScholar("Geoffrey Hinton")

# Separate the UserID and Authorhtml (for simplicity)
UserID <- ResultsList[[1]]
Authorhtml <- ResultsList[[2]]

# Create the full data frame
FullScholarDF <- CreateDataFrame(Authorhtml)
```

```r
# Display the length of the DataFrame (checking the length of any column)
length(FullScholarDF$Title)

# Display the head of the DataFrame
head(FullScholarDF)

# Printing a summary using the apply function
apply(FullScholarDF, 1:5, function(x) substr(x,start = 0,stop = 10))
```

Bibliography

1. http://people.cs.ksu.edu/ schmidt/115/ch7.html, "Chapter 7: Data Compression", visited 09/14/2017
2. https://en.wikipedia.org/wiki/Data_deduplication, "Data deduplication", visited 09/14/2017

Appendix: Full Codes

Problem 2

a) Google Scholar citation function: ID and html

```
# Declaring the function
CitesScholar <- function(name) {
  # Turning off the warnings for visualization purposes
  options(warn = -1)

  # Loading libraries
  suppressMessages(library(XML))
  suppressMessages(library(RCurl))
  suppressMessages(library(stringr))

  # Check if the input is a string
  if (!is.character(name)){
    stop("The input must be a name (e.g Geoffrey Hinton)")
  }

  # Check if we have only letters and a first and last name
  if(grepl("^[[:alpha:]]+[[:space:]]+[[:alpha:]]+$", name) == FALSE){
    stop("Only two words (no digits) are allowed as the author's name,
         please try again (e.g Geoffrey Hinton)")
  }

  # Check if the name string has a first and last name
  if (length(strsplit(name, " ")[[1]]) < 2){
    stop("The name string must have a space between the first and last names,
         please try again (e.g Geoffrey Hinton)")
  }

  # Separate the name in first and last names
  firstname <- strsplit(name, " ")[[1]][[1]]
  lastname <- strsplit(name, " ")[[1]][[2]]

  # Generate the URL with the profile in google scholar
  urlInit <- paste("https://scholar.google.com/scholar?hl=es&q=", firstname,
                   "+", lastname, "&btnG=&lr=", sep = "")

  # Get the Google Scholar ID and the link to the personal citations website
  rawHTML <- readLines(urlInit)

  # Parse the raw HTML code
```

```
  parsedHTML <- htmlParse(rawHTML)

  # Find the link to the author's personal website
  link <- xpathSApply(parsedHTML, "//h4[@class = 'gs_rt2']//a",
                      xmlGetAttr, "href")

  # Check if there are valid links inside the file: if not, stop
  if (length(link) <= 0) {
    stop("New error: The author does not exist in the Google Scholar Database,
         please try again with a different name")
  }

  # Extract the user
  UserID <- str_match(link, ".*user=([\\d\\w]+)&")[[2]]

  # If no User ID, then author does not exists in Google Scholar: stop
  if (is.na(UserID) == TRUE){
    stop("The author does not exist in the Google Scholar Database,
         please try again with a different name")
  }

  # Get the citations html
  urlcites <- paste("https://scholar.google.com", link, sep = "")
  htmlcites <- readLines(urlcites)
  parhtmlcites <- htmlParse(htmlcites)

  # Return the user ID and parsed html from the personal citations web
  returnlist <- list(UserID, parhtmlcites)
  return(returnlist)
}
```

Run the code

```
# Obtain the User ID and HTML object of the author
ResultsList <- CitesScholar("Geoffrey Hinton")

# Separate the UserID and Authorhtml (for simplicity)
UserID <- ResultsList[[1]]
Authorhtml <- ResultsList[[2]]

# Display them (summary of the html file for visualization purposes)
# User ID
UserID

# HTML summary
summary(Authorhtml)
```

a.2) Extra function

```
# Declaring the function
DownloadfromScholar <- function(name) {
  # Setting working directory
  setwd("~/ps2/")

  # Loading libraries
  library(XML)
  library(stringr)

  # Turning off the warnings for visualization purposes
  options(warn = -1)

  # Check if the input is a string
  if (!is.character(name)){
    stop("The input must be a name (e.g Geoffrey Hinton)")
  }

  # Check if we have only letters and a first and last name
  if(grepl("^[[:alpha:]]+[[:space:]]+[[:alpha:]]+$", name) == FALSE){
    stop("Only two words (no digits) are allowed as the author's name,
         please try again (e.g Geoffrey Hinton)")
  }

  # Check if the name string has a first and last name
  if (length(strsplit(name, " ")[[1]]) < 2){
    stop("The name string must have a space between the first and last names,
         please try again (e.g Geoffrey Hinton)")
  }

  # Separate the name in first and last names
  firstname <- strsplit(name, " ")[[1]][[1]]
  lastname <- strsplit(name, " ")[[1]][[2]]

  # Generate the initial URL to download the .html file
  urlInit <- paste("https://scholar.google.com/scholar?q=", firstname,
                   "+", lastname, "&hl=en&as_sdt=0,5", sep = "")
  filename <- paste(firstname, lastname, ".html", sep = "")

  # Download the file and parse the html file
  download.file(urlInit, filename)
  parsedHTML <- htmlParse(readLines(filename))

  # Find the link to the author's personal website
  link <- xpathSApply(parsedHTML, "//h4[@class = 'gs_rt2']//a",
                      xmlGetAttr, "href")

  # Check if the link exists (is valid)
  if (length(link) <= 0) {
    stop("New error: The author does not exist in the Google Scholar Database,
```

```
        please try again with a different name")
  }

  # Extract the user
  UserID <- str_match(link, ".*user=([\\d\\w]+)&")[[2]]

  # If no User ID, then author does not exists in Google Scholar: stop
  if (is.na(UserID) == TRUE){
    stop("The author does not exist in the Google Scholar Database,
        please try again with a different name")
  }

  # Get the citations html: generate the url and download it
  urlcites <- paste("https://scholar.google.com", link, sep = "")
  filenamecites <- paste(UserID, ".html", sep = "")
  download.file(urlcites, filenamecites)

  # Parse the new html
  parhtmlcites <- htmlParse(readLines(filenamecites))

  # Return parsed html file
  return(parhtmlcites)
}
```

Run the code

```
# Call the function and create a DataFrame from the author's website
ResultsList <- CitesScholar("Geoffrey Hinton")

# Separate the UserID and Authorhtml (for simplicity)
UserID <- ResultsList[[1]]
Authorhtml <- ResultsList[[2]]

# Create the data frame
SubsetScholarDF <- CreateDataFrame_subset(Authorhtml)

# Display the head of the DataFrame
head(SubsetScholarDF)

# Printing a summary using the apply function
apply(SubsetScholarDF, 1:5, function(x) substr(x,start = 0,stop = 10))
```

b) First 20 citations as DataFrame

```
# Declaring the function
CreateDataFrame_subset <- function(html){
  # Turning off the warnings for visualization purposes
  options(warn = -1)

  # Libraries
  suppressMessages(library(XML))
  suppressMessages(library(RCurl))
  suppressMessages(library(stringr))

  # Check if the file is a parsed HTML
  if (typeof(html) != "externalptr"){
    stop("Not a valid html file from author's citation website")
  }

  # Articles titles
  Titles <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//tr[@class ='gsc_a_tr']
                        //td[@class = 'gsc_a_t']//a[@class = 'gsc_a_at']",
                        xmlValue, trim = TRUE, encoding = "UTF-8")

  # Check if there are valid titles: if not, not a valid file
  if (length(Titles) <= 0){
    stop("No Titles: There is no author's citation page inside Google Scholar
         associated with the html file being processed, please
         try again with a different file")
  }

  # Journals and authors array
  JournalAuthors <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//
                                tr[@class ='gsc_a_tr']//td[@class = 'gsc_a_t']
                                //div[@class = 'gs_gray']", xmlValue,
                                trim = TRUE, encoding = "UTF-8")

  # Getting the Journals (even numbers of previos vector)
  Journals <- JournalAuthors[seq(2, length(JournalAuthors), 2)]

  # We delete the year of publication after the Journal
  Journals <- substring(Journals, 0, nchar(Journals) - 6)

  # Getting the authors (odd numbers)
  Authors <- JournalAuthors[seq(1, length(JournalAuthors), 2)]

  # Years of publication
  Year <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//tr[@class ='gsc_a_tr']
                      //td[@class = 'gsc_a_y']//span[@class = 'gsc_a_h']",
                      xmlValue, trim = TRUE, encoding = "UTF-8")

  # Number of citations (including repeated articles)
  Citations <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']
```

```
                            //tr[@class ='gsc_a_tr']//td[@class = 'gsc_a_c']
                            //a", xmlValue, trim = TRUE, encoding = "UTF-8")

  # Delete potential non numerical elements by transforming and filtering the array
  Citations <- as.numeric(Citations)
  remove <- c(NA)

  # Re-Transform it to a character array for the DF
  Citations <- as.character(Citations[!Citations %in% remove])

  # Dimension checks: fill with 0 empty citations
  if (length(Citations) != length(Titles)){
    Citations <- c(Citations, integer(length(Titles) - length(Citations)))
  }
  # Create a DataFrame
  x_name <- "Title"
  y_name <- "Authors"
  z_name <- "Journals"
  w_name <- "Year"
  v_name <- "Citations"

  # Create the DF using the reshape2 melt function for simplicity
  suppressMessages(require(reshape2))
  ScholarDF <- suppressMessages(melt(data.frame(Titles, Authors, Journals, Year,


  # Define the columns' names
  suppressMessages(colnames(ScholarDF) <- c(x_name, y_name, z_name, w_name, v_name))

  # Return the DataFrame
  return(ScholarDF)
}
```

Run the code

```
# Call the function and create a DataFrame from the author's website
ResultsList <- CitesScholar("Geoffrey Hinton")

# Separate the UserID and Authorhtml (for simplicity)
UserID <- ResultsList[[1]]
Authorhtml <- ResultsList[[2]]

# Create the data frame
SubsetScholarDF <- CreateDataFrame_subset(Authorhtml)

# Display the head of the DataFrame
head(SubsetScholarDF)

# Printing a summary using the apply function
apply(SubsetScholarDF, 1:5, function(x) substr(x,start = 0,stop = 10))
```

c) Testing our code

c.1) CitesScholar Tests

```
source("CitesScholar.R")

# Testing the CitesScholar function for different inputs
test_that("Only first name", {
  # One word is provided
  ResultsList <- CitesScholar("Albert")
})

test_that("More than two names", {
  # More than 2 words are provided
  ResultsList <- CitesScholar("Albert Einstein Canalejo")
})

test_that("Words and numbers", {
  # First and last name with a number
  ResultsList <- CitesScholar("Albert Einstein2")
})

test_that("Words and numbers (mixed)", {
  # Numbers in between characters
  ResultsList <- CitesScholar("Albert E1nstein")
})

test_that("No space between names", {
  # First and last names are not separated
  ResultsList <- CitesScholar("AlbertEinstein")
})

test_that("Uppercase name", {
  # Name is written in uppercase
  ResultsList <- CitesScholar("ALBERT EINSTEIN")
})

test_that("Lowercase name", {
  # Name is written in lowercase
  ResultsList <- CitesScholar("albert einstein")
})

test_that("Erroneous input is provided (numbers)", {
  # Numeric input
  ResultsList <- CitesScholar(1123)
})

test_that("Erroneous input is provided (digit strings)", {
  # String with digits
  FinalDF2 <- CitesScholar("1222")
})
```

```
test_that("Erroneous input is provided (digit strings)", {
  # String with digits
  FinalDF2 <- CitesScholar("1222")
})

test_that("Erroneous input is provided (anything)", {
  # Html sintaxis
  FinalDF <- CitesScholar("<html><body>")
})
```

Run the code

```
# Loading the library
suppressMessages(library(testthat))

# Set working directory
setwd("C:/Users/chile/ps2/RTests/")

# Invoking the test
test_file("test_CitesScholar.R")
```

c.2) CreateDataFrame_subset Tests

```
source("CreateDataFrame_subset.R")

# Testing the CreateDataFrame_subset function (input = HTML, output = Dataframe)
test_that("Types and lengths are consistent when using a known html file", {
  # Save the output in the FinalDF variable, reading a html file from Google Scholar
  # As an example, we are using the citation website of Albert Einstein
  FinalDF <- CreateDataFrame_subset(htmlParse(readLines("qc6CJjYAAAAJ.html")))

  # Check if the final output is a DataFrame
  expect_that( FinalDF, is_a("data.frame") )

  # Check if all columns have the same length
  expect_that( length(FinalDF$Title), equals(length(FinalDF$Authors)) )
  expect_that( length(FinalDF$Title), equals(length(FinalDF$Year)) )
  expect_that( length(FinalDF$Title), equals(length(FinalDF$Citations)) )
  expect_that( length(FinalDF$Title), equals(length(FinalDF$Journals)) )

  # Check if the Citations column has only numbers
  expect_that( as.numeric(FinalDF$Citations), is_a("numeric") )
})

# Testing the CreateDataFrame_subset function for different inputs
test_that("Output when a non-existent user html file is provided", {
  # Save the output in the FinalDF variable, after reading a html
  # file from Google Scholar website, but from a non-existing user
  FinalDF2 <- CreateDataFrame_subset(htmlParse(readLines("nonexistent.html")))
})

# Another html file (not from Google Scholar)
test_that("A html file/code from any page", {
  # Html taken from google.com
  FinalDF <- CreateDataFrame_subset(htmlParse(readLines("index.html")))
})

# Numbers as inputs
test_that("Erroneous input is provided (numbers)", {
  # Testing with a numeric input
  FinalDF2 <- CreateDataFrame_subset(1222)
})

# String with numbers
test_that("Erroneous input is provided (digit strings)", {
  # String composed by digits
  FinalDF2 <- CreateDataFrame_subset("1222")
})

# Any random string
test_that("Erroneous input is provided (character strings)", {
  # Characters string
```

```
  FinalDF2 <- CreateDataFrame_subset("hello")
})

# Some html sintax (check if function is confused)
test_that("Erroneous input is provided (html sintax)", {
  # Html sintaxis
  FinalDF <- CreateDataFrame_subset("<html><body>")
})
```

Run the code

```
# Loading the library
suppressMessages(library(testthat))

# Set working directory
setwd("C:/Users/chile/ps2/RTests/")

# Downloading the files (using wget for simplicity): valid, non-existent and non-valid
system('bash -c "wget -q -O qc6CJjYAAAAJ.html https://scholar.google.com/\
       citations?user=qc6CJjYAAAAJ&hl=es&oi=ao"')
system('bash -c "wget -q -O nonexistent.html https://scholar.google.com/\
       \scholar?q=Cristobal+Pais&btnG=&hl=es&as_sdt=0%2C5"')
system('bash -c "wget -q www.google.com"')

# Invoking the test
test_file("test_CreateDataFrame_subset.R")
```

d) Extra Credit: All citations as DataFrame

```
# Declaring the function
CreateDataFrame <- function(html){
  # Turning off the warnings for visualization purposes
  options(warn = -1)

  # Libraries
  suppressMessages(library(XML))
  suppressMessages(library(RCurl))
  suppressMessages(library(stringr))

  # Check if the file is a parsed HTML
  if (typeof(html) != "externalptr"){
    stop("Not a valid html file from author's citation website
         please check the format of the input file")
  }
  # Get the links inside the html
  links <- getHTMLLinks(html, baseURL = urlInit, relative = FALSE)

  # Check if there are valid links in the file
  if (length(links) <= 0){
    stop("Not a valid html file from author's citation website
         the file does not contain any valid link")
  }

  # Generate the link containing the query
  url4queries <- paste("http://scholar.google.com", links[8], sep = '')

  # Global vectors for recording all the queries
  GTitles <- c()
  GJournals <- c()
  GAuthors <- c()
  GYears <- c()
  GCitations <- c()

  # Initial values: from (start) and size
  from <- 0
  size <- 100

  # Loop for getting all the articles based on the query structure
  while (TRUE){
    query <- paste("&cstart=",from,"&pagesize=",size,sep = '')
    urlq <- paste(url4queries,query,sep = '')
    html <- readLines(urlq)
    html <- htmlParse(html)
    # Articles titles
    Titles <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//
                          tr[@class ='gsc_a_tr']//td[@class = 'gsc_a_t']
                          //a[@class = 'gsc_a_at']", xmlValue,
                          trim = TRUE, encoding = "UTF-8")
```

```
  # Break condition: no titles/papers inside the HTML object
  if (length(Titles) == 0 ) {
    break
  }

  # Journals and authors array
  JournalAuthors <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//
                                tr[@class ='gsc_a_tr']//td[@class = 'gsc_a_t']
                                //div[@class = 'gs_gray']", xmlValue,
                                trim = TRUE, encoding = "UTF-8")

  # Getting the Journals (even numbers of previos vector)
  Journals <- JournalAuthors[seq(2, length(JournalAuthors), 2)]

  # We delete the year of publication after the Journal
  Journals <- substring(Journals, 0, nchar(Journals)-6)

  # Getting the authors (odd numbers)
  Authors <- JournalAuthors[seq(1, length(JournalAuthors), 2)]

  # Years of publication
  Year <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//tr[@class ='gsc_a_tr']
                      //td[@class = 'gsc_a_y']//span[@class = 'gsc_a_h']",
                      xmlValue, trim = TRUE, encoding = "UTF-8")

  # Number of citations (No repetitions)
  Citations <- xpathSApply(html, "//tbody[@id = 'gsc_a_b']//
                           tr[@class ='gsc_a_tr']//td[@class = 'gsc_a_c']//a",
                           xmlValue, trim = TRUE, encoding = "UTF-8")

  # Delete potential non numerical elements by transforming and filtering the array
  Citations <- as.numeric(Citations)
  remove <- c(NA)

  # Re-Transform it to a character array for the DF
  Citations <- as.character(Citations[!Citations %in% remove])

  # Record in global vectors
  GTitles <- c(GTitles, Titles)
  GAuthors <- c(GAuthors, Authors)
  GJournals <- c(GJournals, Journals)
  GYears <- c(GYears, Year)
  GCitations <- c(GCitations, Citations)

  # Update the counter
  from <- from + 100
}

# Dimension checks: fill with 0 empty citations
if (length(GCitations) != length(GTitles)){
```

```
    GCitations <- c(GCitations, integer(length(GTitles) - length(GCitations)))
  }

  # Create a DataFrame
  x_name <- "Title"
  y_name <- "Authors"
  z_name <- "Journals"
  w_name <- "Year"
  v_name <- "Citations"

  require(reshape2)
  ScholarDF <- melt(data.frame(GTitles, GAuthors, GJournals, GYears, GCitations))
  colnames(ScholarDF) <- c(x_name, y_name, z_name, w_name, v_name)

  # Return the DataFrame
  return(ScholarDF)
}
```

Run the code

```
# Call the function and create a DataFrame from the author's website
ResultsList <- CitesScholar("Geoffrey Hinton")

# Separate the UserID and Authorhtml (for simplicity)
UserID <- ResultsList[[1]]
Authorhtml <- ResultsList[[2]]

# Create the full data frame
FullScholarDF <- CreateDataFrame(Authorhtml)

# Display the length of the DataFrame (checking the length of any column)
length(FullScholarDF$Title)

# Display the head of the DataFrame
head(FullScholarDF)

# Printing a summary using the apply function
apply(FullScholarDF, 1:5, function(x) substr(x,start = 0,stop = 10))
```