README - Getting and Cleaning Data Course Project

Klismam Pereira

19/02/2020

Project and Data Overview

This project's objective was to "tidy" the original data tables using knowledge acquired during the Getting and Cleaning Data MOOC, by the Johns Hopkins University, available on the Coursera platform (https://www.coursera.org/learn/data-cleaning/home/welcome).

The data used in this project was generated in a study entitled "Human Activity Recognition on Smartphones using a Multiclass Hardware-Friendly Support Vector Machine" (IWAAL 2012). The study involved the measurement and process of 3-axial linear accelerations and 3-axial angular velocities obtained through the accelerometer and gyroscope of a smartphone. The device was attached to the volunteers' waist, which were asked to perform 6 activities while being recorded - walking, walking upstairs, walking downstairs, sitting, standing and laying. For more information on the study: http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones.

Data Processing

This document aims to explain the process of the script **run_analysis.R**, used to tidy the data. To reproduce the process on your computer change the archives' paths accordingly.

Loading R Packages

The script begins loading the necessary R packages:

```
library(dplyr)

library(tidyr)
```

Uniting *test* and *training* sets

The *test* and *training* sets, *X_test.txt* and *X_train.txt*, separated in the original study, were read and united by rows in the **total_set** table. Both sets had the same number of variables and several observations each.

```
test_set <- read.table("./UCI HAR Dataset/test/X_test.txt")

training_set <- read.table("./UCI HAR Dataset/train/X_train.txt")

total_set <- bind_rows(test_set, training_set)
```

Processing activity labels

For both *test* and *training* sets there are tables that contain the activities performed for each observation on it's rows - *y_test.txt* and *y_train.txt*. These have a number from 1 to 6 in each row that represent the activities performed by the volunteers.

```
table(read.table("./UCI HAR Dataset/test/y_test.txt", stringsAsFactors = FALSE))
```

```
##
##   1   2   3   4   5   6
## 496 471 420 491 532 537
```

Also, with the data available there is a table named *activity_labels.txt* which contains keys from 1 to 6 and the names of the activities.

```
read.table("./UCI HAR Dataset/activity_labels.txt", stringsAsFactors = FALSE)
```

```
##   V1                 V2
## 1  1            WALKING
## 2  2   WALKING_UPSTAIRS
## 3  3 WALKING_DOWNSTAIRS
## 4  4            SITTING
## 5  5           STANDING
## 6  6             LAYING
```

These activity tables, *y_test* and *y_train*, were read and united much like the *test* and *training* sets, then joined with the *activity_labels* table using the keys through the function `inner-join` from the dplyr package. The `select(2)` code was used to subset a vector with the names of the activities. Finally, the characters in the vector were all set to lower case and the result of this process was saved on the **activities** table.

```
test_activity <- read.table("./UCI HAR Dataset/test/y_test.txt", stringsAsFactors = FALSE)

training_activity <- read.table("./UCI HAR Dataset/train/y_train.txt", stringsAsFactors = FALSE)

activity_labels <- read.table("./UCI HAR Dataset/activity_labels.txt", stringsAsFactors = FALSE)

activities <- bind_rows(test_activity, training_activity)
activities <- inner_join(activities, activity_labels) %>% select(2)
activities <- tolower(activities[[1]])
```

The index `[[1]]` was used on `activities <- tolower(activities[[1]])` so that the resulting vector is not of class data.frame. This is important as other functions used later on do not allow the use of objects of this class in it's arguments.

Processing subjects

Both *test* and *training* sets also have subjects tables, *subject_test.txt* and *subject_train.txt*, which contain rows with numbers ranged from 1 to 30, representing the volunteers that performed the activities. 30% of the volunteers were selected to generate the test data and 70% to generate the training data.

```
table(read.table("./UCI HAR Dataset/test/subject_test.txt", stringsAsFactors = FALSE))
```

```
##
##   2   4   9  10  12  13  18  20  24
## 302 317 288 294 320 327 364 354 381
```

```
table(read.table("./UCI HAR Dataset/train/subject_train.txt", stringsAsFactors = FALSE))
```

```
##
##   1   3   5   6   7   8  11  14  15  16  17  19  21  22  23  25  26  27  28  29
## 347 341 302 325 308 281 316 323 328 366 368 360 408 321 372 409 392 376 382 344
##  30
## 383
```

These were united like the other tables. There's one catch, though, the variables on each table must have the same name for the function `bind_rows` to work, and this wasn't the case, that's why they're each named *"subject"*. The result was stored in the **subjects** vector.

```
test_subjects <- read.table("./UCI HAR Dataset/test/subject_test.txt", stringsAsFactors = FALSE)

training_subjects <- read.table("./UCI HAR Dataset/train/subject_train.txt", stringsAsFactors = FALSE)

names(test_subjects) <- "subject"
names(training_subjects) <- "subject"

subjects <- bind_rows(test_subjects, training_subjects)[[1]]
```

The index was used on `subjects <- bind_rows(test_subjects, training_subjects)[[1]]` for the same reason as in the previous section.

Obtaining variables' names and mean/standard deviation indexes

The labels of the 561 variables of the original study can be found in an archive named *features.txt*.

```
str(read.table("./UCI HAR Dataset/features.txt", stringsAsFactors = FALSE))
```

```
## 'data.frame':    561 obs. of  2 variables:
##  $ V1: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ V2: chr  "tBodyAcc-mean()-X" "tBodyAcc-mean()-Y" "tBodyAcc-mean()-Z" "tBodyAcc-std()-X" ...
```

The next step is to read and extract only the measurements on the mean and standard deviation for each measurement, as required by the project guidelines. The index `[[2]]` is utilized in the following `read.table` call so that only the column related to the variable names is saved in **features** as a character vector. Next, the function `grepl` searches for the characters that contain "mean()" or "std()" in features, returning a logical vector which is then stored as **mean_std_index**. These strings were selected based on the description given by the *features_info.txt* archive that accompany the original data. The argument `fixed = TRUE` guarantees that the exact characters are matched.

```
features <- read.table("./UCI HAR Dataset/features.txt", stringsAsFactors = FALSE)[[2]]

mean_std_index <- grepl("mean()", features, fixed = TRUE) |
        grepl("std()", features, fixed = TRUE)
```

Processing variables' names

As the variables' names contain information on the domain (time or frequency), type of signal (body acceleration, gravity acceleration, etc.), type of measure (mean or standard deviation), axis (x, y, z), it makes sense to separate these into exclusive variables.
Later, *regular expressions* are used to extract these pieces of information from the labels. Basically, it's code that indicates where each piece is located in the character string, provided there is a pattern or separator. The website https://regexr.com/ was very useful to create and test patterns, considering the result expected. That said, some processing is needed to create these patterns, and that's why the next chunk of code focuses only on altering some of the labels in the **features** vector, mainly through the `gsub` function, which substitutes characters that match the first argument by the second argument.

```
features <- tolower(features)
features <- gsub("^f", "frequency-", features)
features <- gsub("^t", "time-", features)
features <- gsub("\\()","", features)
features <- gsub("BodyBody","Body", features, ignore.case = TRUE)
features <- gsub("BodyAcc","body_acceleration", features, ignore.case = TRUE)
features <- gsub("GravityAcc","gravity_acceleration", features, ignore.case = TRUE)
features <- gsub("BodyGyro","body_gyroscope", features, ignore.case = TRUE)
features <- gsub("Mag-mean","_magnitude-mean-not_applicable", features, ignore.case = TRUE)
features <- gsub("Mag-std","_magnitude-std-not_applicable", features, ignore.case = TRUE)
features <- gsub("Jerk","_jerk", features, ignore.case = TRUE)
```

- The `tolower` function, as seen before, sets all the characters on the vector to lower case
- The domain related characters "t" and "f" are substituted by "frequency-" and "time-", these "-" will aid on creating a useful pattern
- All "()" are removed
- There is a labeling error in the original data, some labels contain the "BodyBody" string instead of just "Body", these are also fixed with the code above
- "BodyAcc", "GravityAcc" and "BodyGyro" were given human readable names
- Some of the labels indicated that they're referred to the magnitude of the signals. These were sightly different from the other labels and were also addressed in these codes
  - "Mag-mean" and "Mag-std" were substituted by "_magnitude-mean-not_applicable" and "_magnitude-std-not_applicable": the underscore"_" at the beginning and the "-not_applicable" and the end of the strings are also going to be used later to help identify where these pieces of information are on the string. Specifically, "-not_applicable" is used to fill the need of information on the *axis*, which magnitude signals do not have
- "Jerk" was substituted by "_jerk". The underscore"_" helps to create a useful pattern

The modified labels are stored in the **features** vector.

Applying variables' labels, adding activities and subjects columns

Now, the modified labels on the **features** vector substitute the names of the <**total_set**> table.

```
names(total_set) <- features
```

Then, the mean and standard deviation measures are selected from the complete table by using the **mean__std__index**, and the `mutate` function from the dplyr package adds the **activities** and **subjects** vectors as columns. The result remains stored as **total__set**.

```
total_set <- total_set[,mean_std_index] %>%
        mutate(activity = activities, subject = subjects)
```

Transforming labels in observations using tidyr

The last chunk of code creates the tidy data, it uses functions from the dplyr and tidyr package.

```
tidy_data_1 <-
        pivot_longer(total_set, -c("activity", "subject"),
                     names_to = c("domain", "signal", "measure", "axis"),
                     names_pattern = "(frequency|time)-([bg][or]\\w+)-(std|mean)-([xyz]|n\\w+)",
                     values_to = "values") %>%
        select(subject, activity, domain, signal, axis, measure, values) %>%
        arrange(subject, activity, signal, desc(domain), measure, axis)
```

In order, the code:

- Calls the function `pivot_longer` from the tidyr package, which transforms variables into observations, given a pattern or a separator. The arguments are:
  - `total_set`: the table that is being modified
  - `-c("activity", "subject")`: selects all the variables names, except "activity" and "subject", which are good as they are
  - `names_to = c("domain", "signal", "measure", "axis")`: defines to which new column each piece of information is going to
  - `names_pattern = "(frequency|time)-([bg][or]\\w+)-(std|mean)-([xyz]|n\\w+)"`: this pattern, which use *regular expressions*, is able to find, in the labels, the information targeted between "-", like so:
    1. `(frequency|time)`: it will try to match "frequency" or "time"
    2. `([bg][or]\\w+)`: then, it'll look for words beginning with "b" or "g", followed by "o" or "r", followed by a number of letters, like in "body_acceleration" and "gravity_acceleration"
    3. `(std|mean)`: next, it searches for "std" or "mean"
    4. `([xyz]|n\\w+)`: finally, it matches any "x", "y" and "z" or words that begin with "n" and are followed by any other number of letters, like in "not_applicable"
  - `values_to = "values"`: the values from the previous columns are going to be observations on the "values" column
- `select` reorder the columns on the data frame
- `arrange` sorts the data, based on the order of columns given
- The tidy data is stored as **tidy__data__1**

```
str(read.table("./tidy_data_1.txt"))
```

```
## 'data.frame':    679735 obs. of  7 variables:
##  $ V1: Factor w/ 31 levels "1","10","11",..: 31 1 1 1 1 1 1 1 1 1 ...
```

```
##  $ V2: Factor w/ 7 levels "activity","laying",..: 1 2 2 2 2 2 2 2 2 2 ...
##  $ V3: Factor w/ 3 levels "domain","frequency",..: 1 3 3 3 3 3 3 3 3 3 ...
##  $ V4: Factor w/ 11 levels "body_acceleration",..: 11 1 1 1 1 1 1 1 1 1 ...
##  $ V5: Factor w/ 5 levels "axis","not_applicable",..: 1 3 3 3 3 3 3 3 3 3 ...
##  $ V6: Factor w/ 3 levels "mean","measure",..: 2 1 1 1 1 1 1 1 1 1 ...
##  $ V7: Factor w/ 649664 levels "-0.00010067201",..: 649664 632834 622030 620949 622637 620927 621877
```

The guidelines also require that a new tidy table is created from the previous. This second table needs to contain the average of each variable for each activity and each subject, and is obtained through the next code chunk.

```
tidy_data_2 <- tidy_data_1 %>%
        group_by(subject, activity, domain, signal, axis, measure) %>%
        summarise(average = mean(values)) %>%
        arrange(subject, activity, signal, desc(domain))
```

In order, the code:

- Calls the function `group_by` from the dplyr package, which groups the columns based on their observations, creating factor variables. All of the new variables are given, except for the "values" column
- Summarizes the table with `summarise` by calculating mean of the "values" column, based on the groups defined previously. The results are stored on the new "average" column
- The `arrange` function sorts the data, based on the order of columns given
- The second tidy data is stored as **tidy_data_2**

```
head(read.table("./tidy_data_2.txt"))
```

```
##        V1       V2     V3               V4   V5      V6              V7
## 1 subject activity domain           signal axis measure         average
## 2       1   laying   time body_acceleration    x    mean    0.22159824394
## 3       1   laying   time body_acceleration    x     std    -0.9280564692
## 4       1   laying   time body_acceleration    y    mean -0.0405139534294
## 5       1   laying   time body_acceleration    y     std   -0.83682740562
## 6       1   laying   time body_acceleration    z    mean   -0.11320355358
```

Writing tidy data

Finally, the tidy data sets obtained, **tidy_data_1** and **tidy_data_2** are written as .txt archives.

```
write.table(tidy_data_1, file = "./tidy_data_1.txt", row.name = FALSE)

write.table(tidy_data_2, file = "./tidy_data_2.txt", row.name = FALSE)
```

Disclaimer: the **measure** variable has two possible observations, mean and std, which are variables per se. Indeed, these could be pivoted to the variables and the **values** could be reorganized as it's observations. Nevertheless, since there are many observations in the **values** column that would be addressed to the mean or std based on the other variables (ex. many "std" observations for the combination "subject = 1, activity = walking, domain = time, signal = body_acceleration, axis = x"), all the observations would be coerced into a vector. Then, for every combination of subject, activity, domain, signal and axis, there would be a vector of values on the "mean" and "std" columns.
