R for Data Science - Final exam (Solutions)

University of Milano - Bicocca, Department of Economics, Management and Statistics

Andrea Gilardi

2023-05-04

Exercise 1 - Exploratory Data Analysis (12pt)

The `covfefe_chat.txt` file (that I sent to each one of you by email) includes a copy of the text messages exchanged in the Covfefe Whatsapp group during the last few months. The file was saved using the "Whatsapp Text Data" format and can be conveniently read into R as follows:

```
library(rwhatsapp)
covfefe_chat <- rwa_read(
  here::here("exam", "covfefe_chat.txt"),
  encoding = "UTF-8"
)
```

Whatsapp text data analysis in R

The vignette of the R package `rwhatsapp` provides a great introduction to the analysis of whatsapp text data. You might want to read it to get inspiration on how to solve the following exercises.

If we check the structure of the dataset

```
library(dplyr, warn.conflicts = FALSE)
glimpse(covfefe_chat)
## Rows: 1,610
## Columns: 6
## $ time       <dttm> 2022-10-25 17:50:39, 2022-10-25 17:51:39, 2022-10-25 17:52~
## $ author     <fct> Tommaso Rigon, Luca Presicce, Roberto Ascari, Tommaso Rigon~
## $ text       <chr> "Hi Valentina! Nice to meet you!", "Welcome Valentina! <U+0001F601>",~
## $ source     <chr> "D:/git/R4DS-PhD-Unimib/exam/covfefe_chat.txt", "D:/git/R4D~
## $ emoji      <list> <NULL>, "<U+0001F601>", "<U+0001F601>", <NULL>, <NULL>, <NULL>, <NULL>,
## $ emoji_name <list> <NULL>, "beaming face with smiling eyes", "beaming face wi~
```

we can see that there are 1610 rows and 6 column with a quite descriptive name.

Using the Covfefe dataset, answer the following questions:

1. Who sent the highest number of messages?

Solution

```r
covfefe_chat |> count(author, sort = TRUE)
## # A tibble: 32 x 2
##   author                n
##   <fct>             <int>
## 1 Andrea Gilardi      233
## 2 Chiara Magnani      192
## 3 Laura D'Angelo      190
## 4 Giorgia Zaccaria    186
## 5 Luca Aiello         181
## 6 Tommaso Rigon       146
## # i 26 more rows
```

Andrea Gilardi (😬) sent the highest number of messages during the considered period.

2. How many messages were exchanged during December 2022? **Tip:** Check the functions exported by `lubridate` package if you need to extract the "month" from the `time` field

Solution

```r
library(lubridate, warn.conflicts = FALSE)
covfefe_chat |> mutate(month = month(time)) |> filter(month == 12) |> nrow()
## [1] 246
```

246 messages were exchanged during December 2022.

3. Who sent the first message of the current year? At which time?

Solution

```r
covfefe_chat |> filter(year(time) >= 2023) |> slice_min(time)
## # A tibble: 2 x 6
##   time                author         text               source emoji  emoji_name
##   <dttm>              <fct>          <chr>              <chr>  <list> <list>
## 1 2023-01-04 15:08:39 Andrea Gilardi "Hi everyone! Fir~ D:/gi~ <chr>  <chr [1]>
## 2 2023-01-04 15:08:39 Andrea Gilardi "<Media omessi>"   D:/gi~ <NULL> <NULL>
```

The first message of the current year was sent by Andrea Gilardi (🙈) on January 4th, 2023 at 15:08:30.

4. On which day did we exchange the highest number of messages? After filtering the corresponding text messages, check their content and try to explain the anomalous behaviour.

Solution

```
covfefe_chat |>
  mutate(day = as_date(time)) |>
  count(day, sort = TRUE) |>
  slice_max(n)
## # A tibble: 1 x 2
##   day            n
##   <date>     <int>
## 1 2023-02-01    60
```

The day in which we exchanged the highest number of messages is February 1st, 2023. If we check the content of those messages:

```
covfefe_chat |> filter(as_date(time) == as.Date("2023-02-01"))
## # A tibble: 60 x 6
##   time                author                text         source emoji  emoji_name
##   <dttm>              <fct>                 <chr>        <chr>  <list> <list>
## 1 2023-02-01 07:14:39 Luca Danese           I'm in!      D:/gi~ <NULL> <NULL>
## 2 2023-02-01 09:24:39 Luca Aiello           I'm in loa~  D:/gi~ <NULL> <NULL>
## 3 2023-02-01 12:29:39 Chiara Magnani        Lunch in 1~  D:/gi~ <NULL> <NULL>
## 4 2023-02-01 12:30:39 Luca Aiello           Yeah         D:/gi~ <NULL> <NULL>
## 5 2023-02-01 12:30:39 Valentina Zangirolami First year~  D:/gi~ <NULL> <NULL>
## 6 2023-02-01 12:30:39 Chiara Magnani        Tooop!       D:/gi~ <NULL> <NULL>
## # i 54 more rows
```

we can see that the participants were discussing about the organisation of an happy hour (to welcome Alessia Caponera) and, unfortunately, there were some problems to decide the final location.

5. How many messages are sent on average per day?

Solution

```
covfefe_chat |> summarise(
  min_date = min(as_date(time)),
  max_date = max(as_date(time)),
  n_days = max_date - min_date,
  avg_num_messages = n() / as.numeric(n_days)
)
## # A tibble: 1 x 4
##   min_date   max_date   n_days   avg_num_messages
##   <date>     <date>     <drtn>              <dbl>
## 1 2022-10-25 2023-03-31 157 days             10.3
```

If we consider the complete set of days in the period under analysis, then we can see that we sent, on average, 10.3 messages per day. On the other hand, if we consider only the days in which we exchanged at least one message

```
covfefe_chat |> summarise(
  nunique_days = length(unique(as_date(time))),
  avg_num_messages = n() / nunique_days
)
## # A tibble: 1 x 2
##   nunique_days avg_num_messages
##          <int>            <dbl>
## 1          111             14.5
```

then we can see that we sent, on average, 14.5 messages per day.

6. Who sent the highest number of messages which included at least one emoji? **Tip:** As we can see from the output of `glimpse()`, the `emoji` column is a `<list>` column. The following code can be used to select only the not-`NULL` values from a list-column named `col` in a dataset named `data`: `data |> filter(!vapply(col, is.null, logical(1)))`.

Solution

```
covfefe_chat |>
  filter(!vapply(emoji, is.null, logical(1))) |>
  count(author, sort = TRUE)
## # A tibble: 23 x 2
##   author                 n
##   <fct>              <int>
## 1 Andrea Gilardi        66
## 2 Giorgia Zaccaria      62
## 3 Chiara Magnani        43
## 4 Claudia Sartirana     42
## 5 Tommaso Rigon         19
## 6 Ludovica De Carolis   14
## # i 17 more rows
```

The author who sent the highest number of messages which included at least one emoji is (one more time 🤦 ...) Andrea Gilardi.

7. **(Difficult)** Determine the most common emoji for each author. In case of ties, you can select any of the equally-used emojies. **Tip:** The `unnest()` function (which is defined in the R package `tidyr`) can be used to "unnest" a list column. See the corresponding help page and the vignette of `rwhatsapp` for more details. In any case, you don't need to "parse" the UTF-8 codes.

Solution

```
library(tidyr)

covfefe_chat |>
  filter(!vapply(emoji, is.null, logical(1))) |>
  select(author, emoji) |>
  unnest(emoji) |>
  group_by(author) |>
  count(emoji) |>
  slice_max(order_by = n, with_ties = FALSE)
## # A tibble: 23 x 3
## # Groups:   author [23]
##   author              emoji                      n
##   <fct>               <chr>                  <int>
## 1 Alessandro Colombi "\U0001f44b\U0001f3fb"     2
## 2 Alessia Caponera   "\U0001f604"               1
## 3 Alice Giampino     "\U0001f601"               2
## 4 Andrea Gilardi     "<U+2615>"                    18
## 5 Caterina Daidone   "\U0001f973"               2
## 6 Chiara Magnani     "\U0001f44d\U0001f3fb"     8
## # i 17 more rows
```

If you want, you can run `View()` at the end of the pipe chain in an interactive Rstudio session to see the parsed emojies.

8. **(More difficult)** Compute and display the total number of messages exchanged in the whatsapp chat after dividing the observations according to the hour of the day AND the day of the week. **Tip:** The function `tidyr::complete` can be used to fill the "implicit" missing values (i.e. those combinations of day and hour where no message was sent). Filling the 0 counts might help for the development of the visualisation.

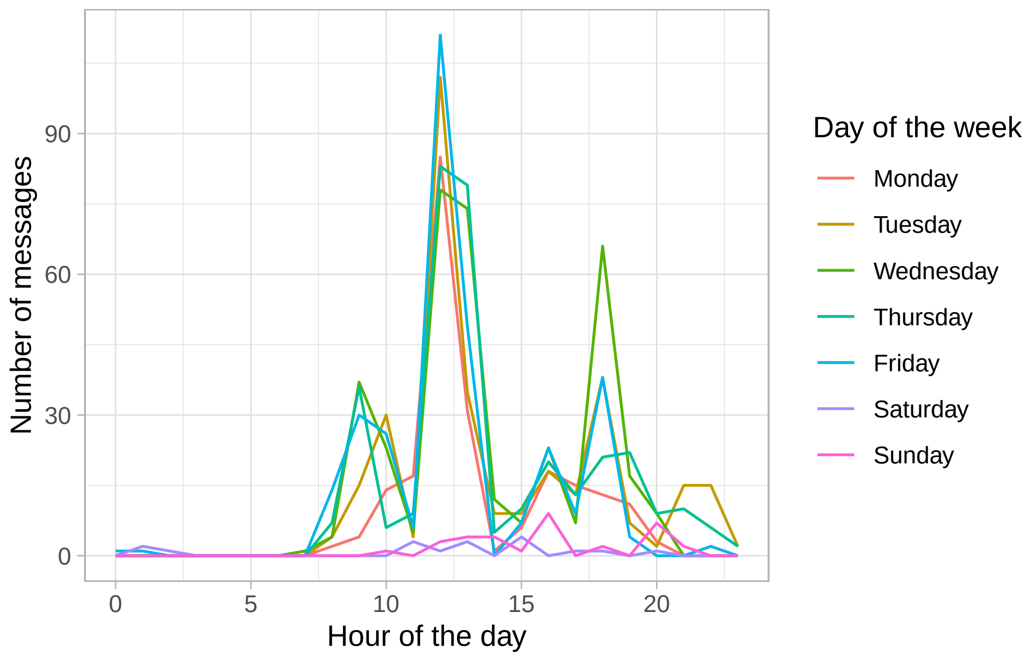
Solution

First, properly format the data.

```
datetime_nmessages <- covfefe_chat |>
  group_by(
    dow = factor(
      weekdays(time),
      levels = c(
        "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
      )
    ),
    hour = factor(hour(time), levels = 0:23)
  ) |>
  summarise(n = n(), .groups = "drop") |>
  complete(dow, hour, fill = list(n = 0L)) |>
  mutate(hour = as.numeric(levels(hour))[hour])
```

The following shows a possible visualisation

```
library(ggplot2)
ggplot(datetime_nmessages, aes(x = hour, y = n, group = dow, col = dow)) +
  geom_line() +
  theme_light() +
  labs(
    x = "Hour of the day",
    y = "Number of messages",
    col = "Day of the week"
  )
```



Exercise 2 - Debugging techniques (8pt)

According to the official R documentation, the `termplot()` function can be used to plot the regression terms included in a linear model against their predictors (i.e. the product of $\hat{\beta}_j$ and x_j). So, for example, if we define a small simulation study such as

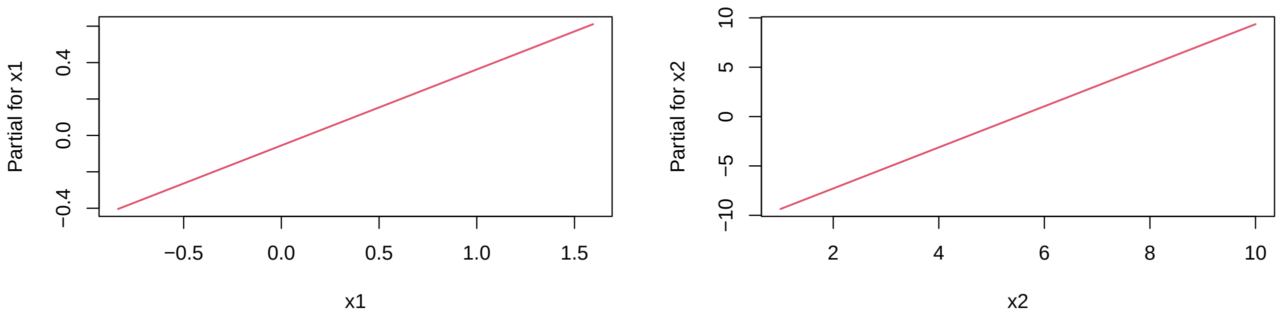
```
set.seed(1)
n <- 10L
x1 <- rnorm(n)
x2 <- seq.int(n)
beta0 <- 0; beta1 <- 1; beta2 <- 2
y <- beta0 + beta1 * x1 + beta2 * x2 + rnorm(n)
```

then we can obtain a least square estimate of β_0, β_1, and β_2 as follows

```r
(mod1 <- lm(y ~ x1 + x2))
##
## Call:
## lm(formula = y ~ x1 + x2)
##
## Coefficients:
## (Intercept)           x1           x2
##     -0.1165       0.4176       2.0804
```

and the following command plots x_j vs $\hat{\beta}_j x_j$, $j = 1, 2$:

```r
termplot(mod1, ask = FALSE, ylim = "free")
```



The `I()` function can be used inside a formula to stop the interpretation of its argument, indicating that it should be treated "as is". See also its help page for more details. Therefore, `mod1` can also be defined as follows:

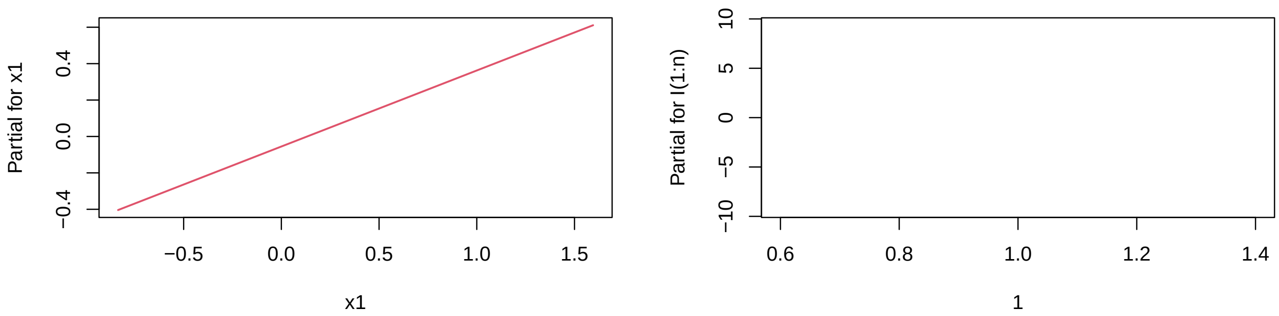
```r
(mod2 <- lm(y ~ x1 + I(1:n)))
##
## Call:
## lm(formula = y ~ x1 + I(1:n))
##
## Coefficients:
## (Intercept)           x1       I(1:n)
##     -0.1165       0.4176       2.0804
```

Nevertheless, the following chunk of code shows that, in this second case, the `termplot` function returns an empty plot and a suspicious warning which is clearly misleading:

```r
termplot(mod2, ask = FALSE, ylim = "free")
## Warning in termplot(mod2, ask = FALSE, ylim = "free"): 'model' appears to
## involve interactions: see the help page
```

Questions:

1. Explain why `termplot()` raises a warning message when we pass `mod2` instead of `mod1` and the steps you took / the techniques you used to tackle this problem.



Solution

The warning message is erroneously raised by the following piece of code

```
strwrap(body(termplot)[[11]])
## [1] "if"
## [2] "any(grepl(\":\", nmt, fixed = TRUE))"
## [3] "warning(\"'model' appears to involve interactions: see the help page\","
## [4] "domain = NA, immediate. = TRUE)"
```

In particular, the object `nmt` inside `grepl()` is defined as

```
body(termplot)[[10]]
## nmt <- colnames(tms)
```

whereas `tms` is defined in

```
body(termplot)[[4]]
## n.tms <- ncol(tms <- as.matrix(if (se) terms$fit else terms))
```

According to the official documentation and the function's definition, `terms` is a dataframe-like object that contains the predicted values for the chosen term(s) and `nmt` are the `colnames` of `tms`, i.e. the names of the covariates included into the regression model (stored as a character vector). Therefore, we could argue that the creator of this function added the `if` clause to raise an informative warning message in case a user passed a model that includes interactions among covariates defined via the `:` operator (like `A:B`), probably because those are not correctly handled by `termplot()`. Unfortunately, the existing approach confuses the term `I(1:n)` as an interaction term and erroneously raises that same warning message.

2. **(Difficult)** Explain why `termplot()` creates an empty plot when displaying the relationship between `x2 := 1:n` and its predicted values.

Solution

The plots generated by `termplot()` are created within a for loop that iterates over the model's terms:

```r
body(termplot)[[34]]
## for (i in 1L:n.tms) {
##     if (identical(ylim, "free")) {
##         ylims <- range(tms[, i], na.rm = TRUE)
##         if (se)
##             ylims <- range(ylims, tms[, i] + 1.05 * 2 * terms$se.fit[,
##                 i], tms[, i] - 1.05 * 2 * terms$se.fit[, i],
##                 na.rm = TRUE)
##         if (partial.resid)
##             ylims <- range(ylims, pres[, i], na.rm = TRUE)
##         if (rug)
##             ylims[1L] <- ylims[1L] - 0.07 * diff(ylims)
##     }
##     if (!in.mf[i])
##         next
##     if (is.fac[i]) {
##         ff <- mf[, nmt[i]]
##         if (!is.null(model$na.action))
##             ff <- naresid(model$na.action, ff)
##         ll <- levels(ff)
##         xlims <- range(seq_along(ll)) + c(-0.5, 0.5)
##         xx <- as.numeric(ff)
##         if (rug) {
##             xlims[1L] <- xlims[1L] - 0.07 * diff(xlims)
##             xlims[2L] <- xlims[2L] + 0.03 * diff(xlims)
##         }
....
```

In our case, `n.tms` is equal to 2 (since the regression model includes two terms), implying that the for loop runs for 2 times (one for each plot). Moreover, considering that we are not dealing with any factor covariate, the plots must be generated by the bottom part of the following call

```r
body(termplot)[[34]][[4]][[4]][[4]]
## {
##     xx <- carrier(cn[[i]], transform.x[i])
##     if (!is.null(use.rows))
##         xx <- xx[use.rows]
##     xlims <- range(xx, na.rm = TRUE)
##     if (rug)
##         xlims[1L] <- xlims[1L] - 0.07 * diff(xlims)
##     oo <- order(xx)
##     plot(xx[oo], tms[oo, i], type = "l", xlab = xlabs[i], ylab = ylabs[i],
##         xlim = xlims, ylim = ylims, main = main[i], col = col.term,
##         lwd = lwd.term, ...)
##     if (se)
##         se.lines(xx[oo], iy = oo, i = i)
## }
```

The result should always be a scatter plot (with dots connected by lines) where the x-axis is given by `xx[oo]` and the y-axis by `tms[oo, i]`. The object `xx` is generated by the `carrier()` function (see the beginning of the previous call), which is also defined inside the body of `termplot()`

```
body(termplot)[[18]]
## carrier <- function(term, transform) {
##     if (length(term) > 1L) {
##         if (transform)
##             tms[, i]
##         else carrier(term[[2L]], transform)
##     }
##     else eval(term, data, enclos = pf)
## }
```

The object `cn` (which is one of the input of `carrier()`) is given by

```
body(termplot)[[12]]
## cn <- str2expression(nmt)
```

where, as we have already seen, `nmt` is a character vector that contains the (col)names of the chosen terms (`"x1"` and `"I(1:n)"` in our example) and, according to the official documentation, the function `str2expression()` converts the vector of (col)names into an expression object, like

```
(cn <- str2expression(c("x1", "I(1:n)")))
## expression(x1, I(1:n))
```

More precisely, the first element of `cn` is a *symbol* or *name* object, which is a class of objects that represent a way to refer to other elements (`x1` in this case) by their name:

```
cn[[1]]
## x1
class(cn[[1]])
## [1] "name"
```

See also `?name` for more details. Looking at the previous tests, everything works fine when we process the first term. On the other hand, the second element of `cn` is an object of type `call`:

```
class(cn[[2]])
## [1] "call"
```

A `call` is a recursive type of objects that "represent the action of calling a function" (Wickham 2019). In this case

```
cn[[2]]
## I(1:n)
class(cn[[2]])
## [1] "call"
```

The first element of each call is the function that gets called

```r
cn[[2]][[1]]
## I
```

and the other elements are the arguments

```r
cn[[2]][[2]]
## 1:n
```

The length of a call is given by one plus the number of arguments provided to the function and, therefore,

```r
length(cn[[2]])
## [1] 2
```

For this reason, when the function `termplot()` runs `carrier(cn[[2]], FALSE)` to generate the second plot, the condition inside the if clause is evaluated as `TRUE` and the function recursively calls `carrier(cn[[2]][[2]], FALSE)` where `cn[[2]][[2]]` is the argument inside `I()` and it is also another `call` object.

```r
cn[[2]][[2]]
## 1:n
class(cn[[2]][[2]])
## [1] "call"
```

In fact, given the nature of the R language, `1:n` can be actually seen as a call object where the function `:` is applied with arguments `1` and `n`:

```r
1:n
##  [1]  1  2  3  4  5  6  7  8  9 10
`:`(1, n)
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Therefore, `carrier(cn[[2]][[2]], FALSE)` recursively calls `carrier(cn[[2]][[2]][[2]], FALSE)` where `cn[[2]][[2]][[2]]` is just the first argument passed to the function `:` i.e. 1:

```r
cn[[2]][[2]][[2]]
## [1] 1
```

Finally, the nested object returned by `cn[[2]][[2]][[2]]` is not a call object

```r
class(cn[[2]][[2]][[2]])
## [1] "numeric"
```

and `xx` is computed as `eval(1, data, encols = pf)` which simply evaluates to 1 since 1 is a constant numeric value. Therefore, all the subsequent steps (e.g. `range()`, `order()`, …) run in an erroneous/pathological way and produce a plot which is just a point centred in $x = 1$.

Exercise 3 - R packages (16pt)

During the last two classes of our course we developed a small R package named `statsAndBooze` that can be used to perform the following task:

```r
library(statsAndBooze)

# Which days are you available to have a beer?
beer_dates_string <- list(
  andrea = c("2023-04-04", "2023-04-05"),
  federico = "2023-04-04"
)

# Convert strings to Date(s)
beer_dates <- parse_dates(beer_dates_string)
decide_happy_hour(beer_dates)
## [1] "2023-04-04"
```

Now you are required to extend the currently-existing functionalities in the following ways!

1. As you may already know, I love beer and, more importantly, I'm a lazy guy… So, I'm (almost) always available to have a beer but I really don't want to manually enter all the single dates into the R script :(Therefore, now you have to extend the `parse_dates` function to allow the specification of time intervals instead of single dates. So, for example, given the following input

```r
beer_dates_string <- list(
  andrea = c("2023-04-01", "2023-04-03 / 2023-04-05"),
  federico = "2023-04-04"
)
```

our package should return something along these lines

```r
parse_dates(beer_dates_string)
# $andrea
# [1] "2023-04-01", "2023-04-03", "2023-04-04", "2023-04-05"

# $federico
# [1] "2023-04-04"
```

Please notice that each element of the list should be a vector of Dates!

> **Date Intervals in R**
>
> The R package `lubridate` implements a class of objects named Interval(s). So, for example, the following code creates and prints a time interval that starts on April 3rd, 2023 and finishes on April 5th, 2023.

```r
library(lubridate, warn.conflicts = FALSE)
(my_interval <- interval("2023-04-03 / 2023-04-05"))
## [1] 2023-04-03 UTC--2023-04-05 UTC
```

Given an `Interval` object you can also access the boundary points:

```r
int_start(my_interval)
## [1] "2023-04-03 UTC"
int_end(my_interval)
## [1] "2023-04-05 UTC"
```

From that point, you may also create a **seq**uence of Days and then…

Solution

The R function that we defined together in class was coded as follows:

```r
parse_dates <- function(x) {
  lapply(x, lubridate::as_date)
}
```

so that

```r
beer_dates_string <- list(
  andrea = c("2023-04-04", "2023-04-05"),
  federico = "2023-04-04"
)
parse_dates(beer_dates_string)
## $andrea
## [1] "2023-04-04" "2023-04-05"
##
## $federico
## [1] "2023-04-04"
```

The following code chunk presents a possible way to extend the previous approach to accommodate the request of this exercise. First we need to load the relevant function (which must be added as an Import to the package)

```r
library(lubridate)
```

then we can defined a utility function

```r
interval_to_date_sequence <- function(x) {
  x <- interval(x)
  if (identical(int_length(x), numeric(0))) {
    return(numeric(0))
  }
  seq(int_start(x), int_end(x), by = "day")
}
```

and finally we can mix everything together expanding the previous version of `parse_dates()`:

```r
parse_dates_v2 <- function(x) {
  lapply(
    X = x,
    FUN = function(x) {
      looks_like_interval <- grepl("/", x, fixed = TRUE)
      out_interval <- interval_to_date_sequence(x[looks_like_interval])
      out_date <- as_date(x[! looks_like_interval])
      c(out_date, out_interval)
    }
  )
}
```

We can see that

```r
beer_dates_string <- list(
  andrea = c("2023-04-01", "2023-04-03 / 2023-04-05"),
  federico = "2023-04-04"
)
parse_dates_v2(beer_dates_string)
## $andrea
## [1] "2023-04-01" "2023-04-03" "2023-04-04" "2023-04-05"
##
## $federico
## [1] "2023-04-04"
```

The (possibly unexported) function `interval_to_date_sequence()` can be defined in a file named `utils.R`.

2. **(Difficult)** Unfortunately, I'm much more lazy than that and I'm also getting old, so I can barely link dates and weekdays… Therefore, you need to further help me extending the `statsAndBooze` package to allow the specification of weekdays instead of numerical dates! For example, suppose that today is April 5th, 2023 (i.e. the last day of classes together) and we know it's Wednesday. Then, if we assume that I will be up for a beer on Thursday and Friday, the `parse_dates()` function should automatically convert those strings into the corresponding Dates:

```r
beer_dates_string <- list(
  andrea = c("thursday", "friday"),
  federico = "2023-04-05"
```

```
)
parse_dates(beer_dates_string)
# $andrea
# [1] "2023-04-06", "2023-04-07"

# $federico
# [1] "2023-04-05"
```

Restriction

If you want, you can assume that the strings indicating the names of the weekdays refer to the subsequent 7 days with respect to the current date when we run the `parse_date()` function.

Solution

Following the same spirit as before, we can improve the existing approach as follows:

```
weekday_to_date <- function(x) {
  weekdays <- c(
    "sunday", "monday", "tuesday", "wednesday", "thursday", "friday", "saturday"
  )
  wday_now <- wday(today())
  wday_x <- match(x, weekdays)
  # match(., .) returns the index position of the match
  days_diff <- (wday_x - wday_now) %% 7L
  # we need to perform operations modulo 7 (e.g. -1 mod 7 = 6 mod 7)
  today() + days_diff
}

parse_dates_v3 <- function(x) {
  lapply(
    X = x,
    FUN = function(x) {
      looks_like_character_weekday <- grepl("[:alpha:]", x)
      looks_like_interval <- grepl("/", x, fixed = TRUE)

      out_weekday <- weekday_to_date(x[looks_like_character_weekday])
      out_interval <- interval_to_date_sequence(x[looks_like_interval])
      out_date <- as_date(x[! (looks_like_interval | looks_like_character_weekday)])

      c(out_date, out_interval, out_weekday)
    }
  )
}
```

such that

```r
beer_dates_string <- list(
  andrea = c("2023-05-04", "friday", "monday"),
  federico = c("2023-04-05", "2023-05-01 / 2023-05-05")
)
parse_dates_v3(beer_dates_string)
## $andrea
## [1] "2023-05-04" "2023-05-05" "2023-05-08"
##
## $federico
## [1] "2023-04-05" "2023-05-01" "2023-05-02" "2023-05-03" "2023-05-04"
## [6] "2023-05-05"
```

The `weekday_to_date()` function could be defined in an ad-hoc `temporal-conversion.R` file or something similar. Moreover, the sequence of `looks_like_*` tests could be defined using alternative approaches (if/else, switch, case_when, …).

3. At the end, you need to showcase the functionalities we developed together and the new ones creating a **beautiful** README file. If you need inspiration, see also here. Please notice that you just need to explain the installation process of your package and present its basic functionalities (i.e. the ones we coded together and the new ones).

References

Wickham, Hadley (2019). *Advanced R*. CRC press. URL: http://adv-r.had.co.nz/.