SimOb\_Rinjects

## 1. Install packages and load libraries

# Check if the 'pacman' package is installed, if not install it:  
if (!requireNamespace("pacman", quietly = TRUE)) install.packages("pacman")  
  
# Load the required libraries into the current R session:  
pacman::p\_load(rio,   
 here,   
 tidyverse,   
 skimr,  
 plyr,  
 janitor,  
 lubridate,  
 gtsummary,   
 flextable,  
 officer,  
 epikit,   
 apyramid,   
 scales)  
  
# Load EpiStats from Github:  
# pacman::p\_load\_gh("Epiconcept-Paris/EpiStats")  
# pacman::p\_load\_gh("Epiconcept-Paris/EpiStats@release\_1.4-1\_2020-04-21")

## 2. Import your data

# Import the raw data set:  
linelist <- rio::import(here::here("data", "Copenhagen\_raw.csv"))

## 3. Explore and clean your data

head(linelist)  
dim(linelist)  
str(linelist)  
skimr::skim(linelist)  
names(linelist)

Let’s explore some variables.

#### Age

Through visual exploration of the age histogram we see that there is at least one very high value, likely implausible.

You can then create a cross-tabulation of variables age and group to have a better idea of how your data looks like.

# Have a look at the histogram  
hist(linelist$age)

# Create cross-tab with the group variable:  
janitor::tabyl(dat = linelist, age, group)

With this table, we can more easily identify ages that are likely to be typographic errors. Specifically:

* There is one teacher aged 16 (likely digit reversal - should be 61)
* There is one student aged 8 (likely missing a digit - should be 18)
* There is one student aged 180 (likely has an extra digit - should be 18)

Assuming you have contacted the school to make sure your suspicions about the actual ages are correct, we can now correct them, using case\_when(). We create logical conditions to identify the incorrect ages, combining the values for age with the group they belong to:

# Update incorrect ages to the correct values with case\_when:  
linelist <- linelist %>%   
 mutate(age =   
 case\_when(  
 # Where the respondent is 16 and a teacher, change their age to 61:  
 age == 16 & group == "teacher" ~ 61,   
 # where the respondent is 8 or 180 and a student, change their age to 18:  
 age %in% c(8, 180) & group == "student" ~ 18,   
 # Keep remaining values as is:   
 TRUE ~ as.numeric(age)  
 )  
 )

#### Dose response

Now let’s create a summary table of the dose response columns.

# Create summary table for dose response columns:   
drtable <- linelist %>%   
 # Select all the columns with column names that end in upper case 'D':   
 select(ends\_with("D", ignore.case = FALSE)) %>%   
 # Create the summary table, excluding missing values:   
 gtsummary::tbl\_summary(missing = "no") %>%   
 # Convert to flextable:   
 gtsummary::as\_flex\_table()   
  
 # Print the summary table:   
drtable

### Modify variables format

We want to modify the “variable format” or “column type” of many variables, so that we can do future calculations with them.

Variable types to modify

| Variable name | Original | Desired | Hint |
| --- | --- | --- | --- |
| sex | character | factor |  |
| group | integer | factor |  |
| class | integer | factor |  |
| All the clinical symptom variables | integer | logical |  |
| All the food variables | integer | logical |  |
| All the food variables representing the amount of specific foods eaten (those finishing with a capital “D”) | integer | factor |  |
| dayonset | character | date |  |
| starthour | integer |  |  |

#### Sex, group and class

Let’s start transforming one-by-one the first three variables in the table: sex, group, and class. We will also provide labels to the variable group with plyr::revalue, as 0 and 1 can be difficult to interpret when these mean something other than “no” and “yes”, respectively.

linelist <- linelist %>%   
 dplyr::mutate(  
 sex = as.factor(sex),  
 group = as.factor(group),  
 group = plyr::revalue(  
 group, c("0" = "teacher", "1" = "student")),  
 class = as.factor(class))

#### Symptoms and Food variables

For these variables, we are going to show you a couple of different ways to carry out the same variable type transformation in a *set* of variables, so you don’t need to do one variable at a time. We are showing you these ways so you see alternative ways to do the same thing.

1. For the variables that are clinical symptoms, we will list them one by one and show you the use of mutate(across( )).

linelist <- linelist %>%   
 dplyr::mutate(  
 # clinical symptoms  
 across(c(diarrhoea, bloody, vomiting,  
 abdo, nausea, fever,headache, jointpain),   
 ~as.logical(.)  
 )  
 )

1. For the variables that are food items and food doses, we will show you how to first create a vector of names, following by how to the use of mutate(across(all\_of( ))) on this vector.

# Create a vector with all the food variables representing the amount of specific foods items eaten (those finishing with a capital "D")  
food\_dose <- linelist %>%   
 select(ends\_with("D", ignore.case = FALSE)) %>%   
 names()  
food\_dose <- c("tunaD", "shrimpsD", "greenD", "vealD",   
 "pastaD", "rocketD", "sauceD", "breadD",  
 "champagneD", "beerD", "redwineD", "whitewineD")  
  
# Create a vecor with all food items. To do this, we can remove the capital D of food\_dose by using gsub() to replace the pattern "D" by nothing ("").  
food\_items <- gsub("D", "", food\_dose)  
# Add variable "meal"  
food\_items <- c(food\_items, "meal")   
  
  
linelist <- linelist %>%   
 dplyr::mutate(  
 # food dose variables  
 across(all\_of(food\_dose), ~as.factor(.)),  
 # food items variables  
 across(all\_of(food\_items), ~as.logical(.)))

***Note***: The tilde (~) bellow is used to apply the transformation as.logical(.) to each selected column, which in our case is either all columns included in food\_items and food\_dose.

#### Date and time variables

You can use lubridate::dmy() to mutate the dayonset variable into a date variable. Note that we are using the function dmy() because dates are formatted as day, then month (abbreviated character string), then year (i.e. “12nov2006”).

# Have a look at how the data is stored  
head(linelist$dayonset)  
class(linelist$dayonset)  
  
# Update linelist:  
linelist <- linelist %>%   
 # Change column to date class:  
 dplyr::mutate(dayonset = lubridate::dmy(dayonset))  
  
# Check class of updated column:  
class(linelist$dayonset)

Having a variable that defines “time” in an outbreak investigation can be very useful when creating a case definition. An hour of the day, without a date associated with it doesn’t help you much, thus, you should merge together day and time of onset of symptoms into a single variable. Moreover, you will be using this combined variable later on to create your epicurve. We can combine these two variables by using the lubridate::ymd\_h() function.

Before we proceed, it would be wise to check if any respondents have a value for dayonset but not starthour, or vice versa. The lubridate date-time conversion functions do not have an explicit argument for dealing with missing values, but the truncated = … argument can help prevent spurious date-times being derived from a date-time combination where one value is missing.

We can check if we have any missing values by cross-tabulating starthour with dayonset:

# Cross-tabulate dayonset with starthour:  
janitor::tabyl(dat = linelist,   
 starthour,   
 dayonset)

This shows us that there are two respondents who had an onset date, but are missing onset time (starthour). Since starthour is represented by 1 - 2 digits, we can specify that we want lubridate to also parse date-time combinations that are truncated by up to two digits:

linelist <- linelist %>%   
 # Combine dayonset and starthour in a new date time variable:  
 mutate(onset\_datetime = lubridate::ymd\_h(paste(dayonset, starthour),   
 # Deal with missing starthour:  
 truncated = 2))

Note that we needed to paste dayonset and starthour together before we could convert the variable to a date-time object. This is because the function expects a single character string, containing both the date and the time, as input.

The argument truncated = 2 will result in dates with missing starthour still being converted to date-time, with the missing time being set to 00:00 (midnight). Whether you want to deal with missing starthour in this way or prefer to code these date-times as NA will depend on how you want them to be represented in your analysis.

Now we can check that everything in the new combined date-time variable has parsed correctly:

head(linelist$dayonset)  
head(linelist$starthour)  
head(linelist$onset\_datetime)

# 4. Export clean data

Save the cleaned data set before proceeding with using your case definition to identify cases in your dataset.

rio::export(x = linelist,   
 file = here::here("data", "Copenhagen\_clean.xlsx"))