Data manipulation and visualization with R

2.3 Data manipulation with dplyr

2.3 Data manipulation with dplyr

Introduction

We already learned the basics of R programming, how to import data and how to select specific information from data containers. In this lesson we will explore the package “dplyr” which eases the manipulation of data. Under “data manipulation” we understand any action that subsets and filters data, adds more data to existing data structures or summarises data.

We will learn how to install the package and how its basic grammar works. After that we will explore six of the most fundamental functions provided by the package:

- filter( ) reduces the datasets to rows fulfilling specific conditions
- select( ) reduces the dataset to columns fulfilling specific conditions
- mutate( ) introduces new columns to a dataset
- summarise( ) creates a dataset providing statistical summaries
- group_by( ) organizes the data into categories
- arrange( ) reorganizes how data is sorted

Introduction

Installation guide and online resources

First, we need to install the dplyr package. This needs to be done only once.

```
install.package("dplyr")
```

Next, we add the package to the library. This must be done each session.

```
library("dplyr")
```

dplyr is also part of the package “tidyverse”, which includes “ggplot2” as well. Installing the tidyverse package therefore installs both packages we discuss in this course automatically.

A comprehensive documentation of dplyr can be found here: https://cran.r-project.org/web/packages/dplyr/dplyr.pdf

Furthermore, the “cheat sheet” provides an excellent overview about the most important functions included within this package: https://github.com/rstudio/cheatsheets/blob/master/data-transformation.pdf

Introduction

A new “grammar”

dplyr adds a new type of “grammar” to R that is used to combine several individual function calls into one call. The “pipe” %>% links functions together and automatically passes the input data to the next step.

The code on the right shows the same call of three functions; the first example uses “Vanilla R” without packages while the second one uses dplyr.

In the first example, function1( ) uses “data” to create a “result” dataset which is then uses as input for the next two steps. Each time, “result” has to be saved and loaded. With big datasets this could slow down the computation.

The second example uses the pipe %>% to link all three functions together. Note that the “result” only has to be saved once. This reduces computation time and also cleans up the code a little.

```
# applying three functions to the same dataset
result <- function1(data, arguments)
result <- function2(result, arguments)
result <- function3(result, arguments)


# with dplyr functions can be linked via the pipe: %>%
result <- function1(data, arguments) %>%
    function2(arguments)             %>%
    function3(arguments)
```

Introduction

A new “grammar”

The following example shows the potential of simplifying code with dplyr. The task at hand is to summarise the cars in the mtcars dataset so we can easily see the average horsepower for each category of cylinder numbers.

In the first example we have to create two empty variables “cyl” and “meanHP” that will later be filled. We loop through each unique number of cylinders “i” and add the average horsepower per category “i” to “meanHP”. We also add each “i” to “cyl”. After the loop we combine both vectors into a dataframe and sort them by number of cylinders. This approach is almost as incomprehensible without proper annotations as the code itself.

The second example is simpler: We pass “mtcars” to group it by the “cyl” column and calculate the “meanHP” for each group.

```
# using basic functions
cyl <- NULL
meanHP <- NULL
for(i in unique(mtcars$cyl)){
   meanHP <- append(meanHP, mean(mtcars$hp[which(mtcars$cyl==i)]))
   cyl <- append(cyl, i)
}
cbind(cyl,meanHP)[order(cyl),]

# using dplyr
mtcars %>%
   group_by(cyl) %>%
   summarise(meanHP = mean(hp))
```

```
# result:
     cyl    meanHP
[1,]   4  82.63636
[2,]   6 122.28571
[3,]   8 209.21429
```

2.3 Data manipulation with dplyr | Hannes Seller, Jonathan Young (2020)

dplyr functions

filter( )

The filter( ) function allows us to reduce the data to rows that fulfill specific conditions.

In the first example we only apply the nrow( ) function to the dataset “mtcars”, which is available in RStudio. This shows us that a) the original dataset has 32 observations (rows) and b) that dplyr and Vanilla functions can be combined by the pipe.

The second and third examples reduce the dataset to only those cars with 8 cylinders. While both examples return the same result, the first one created with filter( ) looks more compact and easier to understand. A selection with “data[which(data$column),]” uses a lot of different brackets as well as a comma, which quickly leads to typos while programming.

The fourth and fifth examples use inequalities. We can use “<”, “<=”, “==”, “>=” and “>” to pick certain intervals. The logical AND (“&”) means that two adjacent conditions must be fulfilled, while the logical OR (vertical stoke: “|”) only needs at least one condition fulfilled. The last example also shows how results can be stored in variables.

```
mtcars %>%
  nrow() # no filter, 32 observations

mtcars %>%
  filter(cyl==8) %>%
  nrow() # 14 observations

# this is equivalent to:
nrow(mtcars[which(mtcars$cyl==8),])

mtcars %>%
  filter(cyl<8) %>%
  nrow() # 18 observations

# store result in new variable (14 observations)
new_data <- mtcars %>%
  filter(cyl==4 & mpg > 20)
```

dplyr functions

select( )

The select( ) function allows us to reduce the data to columns that fulfill specific conditions. We can also rename columns at the same time.

In the first example we reduce the mtcars dataset to only the column “cyl”. The result remains a dataframe even though it has only one column. The call “mtcars$cyl”, however, would only return a vector of the column’s values. Selecting “-cyl” will exclude this column from the dataset but keeps the rest.

Several columns can be selected at once by adding more column names to the select( ) function call. This does not require a c( ) to wrap them together as in most Vanilla functions. When several adjacent columns should be selected, the call can be specified as a range “columnX:columnY”. Both approaches are combinable, as in the fifth example.

Example six shows how selected columns can be renamed. In the last example, the function “everything( )” adds all non-selected columns to the back of the dataset. This can be a handy way to put selected columns to the front of the dataset.

```
mtcars %>%  select(cyl) # select only column “cyl”

mtcars %>% select(-cyl) # s. all BUT column “cyl”

# select only columns “cyl” and “gear” in that order
mtcars %>% select(cyl, gear)

# select all columns from “mpg” to “wt”
mtcars %>% select(mpg:wt)

# select all columns from “mpg” to “wt”, then “carb”
mtcars %>% select(mpg:wt, carb)

# select and rename columns “cyl” and “wt”
mtcars %>% select(cylinders = cyl, weight=wt)

# select “wt”, then the remaining columns
mtcars %>% select(weight=wt, everything())
```

dplyr functions

mutate( )

The mutate( ) function allows us to add new columns to the dataset.

In the first example we add the column ”wtMT” which converts the “wt” column’s values in “1,000 lbs” to metric tonnes (rounded to three decimal digits).

The second example uses inequalities and calculations based on other columns. At first the code calculates the mean value of the “wt” column and then compares the “wt” to its average. This returns a logical where “TRUE” indicates that the car is heavier than the average and “FALSE” if not.

In the third example we multiply two existing columns together. This makes little sense in this scenario, but is often required for physical measurements (e.g. weight divided by volume)

The last examples show the combination of several mutate( ) functions. The functions can be piped together or integrated into the same mutate( ) function.

```
# mutate once
mtcars %>%
  mutate(wtMT = round(wt*0.453,3))

mtcars %>%
  mutate(wtAboveAVG = wt > mean(wt))

mtcars %>%
  mutate(gearCarb = gear * carb)

# mutate several times
mtcars %>%
  mutate(wtMT = round(wt*0.453,3)) %>%
  mutate(wtAboveAVG = wt > mean(wt))

mtcars %>%
  mutate(wtMT = round(wt*0.453,3),
         wtAboveAVG = wt > mean(wt))
```

dplyr functions

summarise( ) & group_by( )

The summarise( ) function creates a new dataset with statistics about the original dataset. This can be combined with the group_by( ) function that subsets the dataset into groups.

The first example creates a summary of mtcars where the arithmetic average (mean), the central value (median) and the standard deviation of horsepower is displayed. The function can also be called by its American spelling: summarize( ).

Summaries can be combined with the group_by( ) function to receive more details about the result. In this case, we grouped the cars by “cyl”, i.e. all cars with 4, 6 or 8 cylinders will be sorted into their own respective group. summarise( ) will then give a summary of each group individually. The function n( ) counts how many observations are present in each group.

The group_by( ) function can also group the dataset by several columns and conditions, e.g. cars with more than 6 cylinders and fewer.

```
mtcars %>%
  summarise(hpMean = mean(hp),
            hpMedian = median(hp),
            hpSD = sd(hp))
   hpMean    hpMedian    hpSD
1 146.6875    123         68.56287

mtcars %>%
  group_by(cyl) %>%
  summarise(count = n(),
            hpMean = mean(hp),
            hpMedian = median(hp),
            hpSD = sd(hp))

   cyl  count  hpMean hpMedian  hpSD
1   4     11     82.6      91     20.9
2   6      7    122.      110     24.3
3   8     14    209.      192.    51.0
```

dplyr functions

arrange( )

The arrange( ) function allows us to change the order in which observations are sorted.

The first example reorganizes the dataset to show the cars with the fewest cylinders first (ascending order). As “cyl” can be regarded as a categorical value is this case (there are only three categories), there will be “ties”. In these cases, the observation coming earlier in the dataset will be given the priority.

The second example solves this issue by introducing a second sorting condition. If two cars have the same “cyl” value, their “hp” value will decide which one is given priority. The function can take each column into consideration, but that might decrease the performance if the dataset is big.

In the third example, we use “desc( )” to sort the cars by “wt” in descending order, i.e. the heaviest cars will appear first. The fourth example combines ascending order (by default) and descending order (by specification).

```
# sort by cylinders, ascending order
mtcars %>%
  arrange(cyl)


# sort by cylinders, then by hp, ascending order
mtcars %>%
  arrange(cyl, hp)


# sort by cylinders, descending order
mtcars %>%
  arrange(desc(wt))


# sort by cylinders, descending order, then by hp
mtcars %>%
  arrange(desc(wt), hp)
```

dplyr functions

Bring it together! (1/2)

Now we want to apply what we learned to do some statistics on a few csv files.

This course provides eight csv files containing information about the eight generations of *Pokémon**. Our goal is to find out which is the strongest Pokémon in each generation.

To load the csv files we have set a path, or our working directory, to the location of the files. This is done here with “setwd(“...”)”. The path must use “/” instead of “\” to be valid in R. Make sure your directory only contains the eight specified files and nothing else.

We create an empty dataframe “pkmn” and a counter variable “gen” for the generations. The function “dir( )” lists all files in the working directory. Please make sure it does not contain other data. Looping through all elements “i” (each i represents a file name), we store each file’s content into a temporary variable “temp”. With “mutate( )” we add a column for the generation’s number. The “temp” variable is connected row-wise with “rbind( )”, and the “gen” variable is increased, so the next loop starts with “gen=2”. In the end, the dataframe “pkmn” should include 1,017 observations.

*Data collected from: https://bulbapedia.bulbagarden.net/wiki/List_of_Pok%C3%A9mon_by_base_stats_(Generation_VIII-present)

```
# set wd to location of csv files
setwd("...")


pkmn <- data.frame() # empty df
gen <- 1 # generation counter, starting with 1

# loop through all files in wd
for(i in dir()){
# read csv files, one at a time
  temp <- read.csv(i, sep=",", header=TRUE) %>%
    # add generation column
    mutate(generation = gen)
  # add generation to “pkmn” df
  pkmn <- rbind(pkmn, temp)
  # increase “gen” for next iteration
  gen = gen + 1
}
```

dplyr functions

Bring it together! (2/2)

Once “pkmn” is ready, we use dplyr functions to find the strongest Pokémon from each generation. The strength will be measured as total sum of all six attributes.

Firstly, we create a “total” column that sums up all attribute values per Pokémon. We group by “generation”. To only get the highest value per generation, we filter out the observation that coincides with the max( ) value of its group. Finally, we arrange the remaining observations in descending order and select only relevant columns.

The result should look like this. “Eternamax Eternatus” has the highest total value of all Pokémons. There are a few ties because some Pokémons’ values add up to the same totals.

number	pokemon	generation	total
1 890E	Eternamax Eternatus	8	1125
2 150MX	Mega Mewtwo X	1	780
3 150MY	Mega Mewtwo Y	1	780
4 384M	Mega Rayquaza	3	780
5 800U	Ultra Necrozma	7	754
6 493	Arceus	4	720
7 718C	Zygarde (Complete)	6	708
8 248M	Mega Tyranitar	2	700
9 646B	Black Kyurem	5	700
10 646W	White Kyurem	5	700

```
pkmn %>%
# add column “total”
mutate(total = hp + attack + defense + spAttack +
spDefense + speed)  %>%

# group by generation
group_by(generation) %>%

# filter the observation with max value per group
filter(total == max(total)) %>%

# arrange observations in descending order
arrange(desc(total)) %>%

# select relevant columns
select(number, pokemon, generation, total)
```

2.3 Data manipulation with dplyr

What have we learned?

In this chapter we learned how to install and use a package in R. The “dplyr” package provides us with functions that exceed functionality and efficiency of standard (“Vanilla”) functions.

The functions filter( ) and select( ) can be used to subset a dataset to specific rows or columns respectively. While filter( ) uses conditional statements, the latter function simply names columns that should be either included or excluded. Both functions are great to reduce a dataset to the most essential information. With mutate( ) we can add columns to a dataset. arrange( ) sorts observations by ascending or descending values of specified columns.

summarise( ) and group_by( ) are useful functions to calculate statistics of a dataset. By default, a statistic like the arithmetic mean or the standard deviation will take all observations into consideration to create one resulting value. If the dataset is grouped by a certain criterion (e.g. age groups, biological sex, profession), the statistics are more faceted.

The dplyr package has many more functions. For the scope of this course, these six functions suffice to provide an introduction to the package’s usefulness.

In the next lessons we will learn how to visualize data.