R-intro - Session 4

João Gonçalves

31 de Julho de 2018

Contents

Objectives of session 4

- A brief tour to the `tidyverse`
- Data manipulation with the `dplyr` package
  - Main `dplyr` verbs
  - The pipe `%>%` operator
- Making pretty graphics with `ggplot2` - principles and main plot types
  - Histograms
  - Boxplots
  - Bar-plots
  - Scatter-plots
  - Trend lines (linear, loess, . . .)
  - Making plots by groups
  - Facets

The *dplyr* package

What is dplyr?

- `dplyr` is a powerful R-package used to transform and summarize tabular data (typically a dataframe)
- It is built to be fast, highly expressive, and open-minded about how your data is stored

Why is it useful?

- The `dplyr` package is one of the most useful packages in the `tidyverse` (which is a collection of R packages designed for data science) to manipulate data in R, offering a more expressive, human-readable and friendly alternative to many base R functions
- The package contains a set of functions (or "verbs") that perform common data manipulation operations
- Examples of these operations are filtering rows, selecting specific columns (or column ranges), re-ordering rows, adding new columns and summarizing data

Important dplyr verbs to remember

`dplyr` aim is to provide a function (or "verb") for each type of data manipulation thus helping you to translate your thoughts into code. Here are some of the most important ones:

dplyr verbs	Description
`select()`	Select columns
`filter()`	Filter rows by certain conditions
`arrange()`	Re-order or arrange rows
`mutate()`	Create new columns
`group_by()`	Allows for group aggregation operations
`summarise()`	Summarise values by groups

To start using the `dplyr` package first you have to install and then load it. For that purpose, use the following lines of R code:

```
install.packages("dplyr", dependencies = TRUE)
```

After installing the package, let´s load it!

```
library(dplyr)
```

Now you are reading for some fancy data manipulation! ;-)

Selecting columns using `select()`

Put simply, `select` allows to subset or extract columns from a dataframe in a way similar to the indexation operator (`[]`). The advantage here is that using an explicit verb for that operation makes the code more clear and readable.

```
# An example with the iris dataframe for selecting two columns
select(iris, Petal.Length, Petal.Width)
```

It is also possible to select ranges of columns using a sequence of integer positions:

```
# Selecting the two first columns
select(iris, 1:2)
```

It's even possible to select ranges of columns using the name of the first and of the last:

```
# Selecting the clumns from Sepal.Width to Petal.Width
select(iris, Sepal.Width:Petal.Width)
```

To select all the columns except a specific column, use the - (subtraction) operator (also known as negative indexing).

```
# Selecting all columns except Species
select(iris, -Species)
```

Some additional options to select columns based on a specific criteria:

Select criterion	Description
`starts_with()`	Select columns that starts with a character string
`ends_with()`	Select columns that end with a character string
`contains()`	Select columns that contain a character string
`matches()`	Select columns that match a regular expression
`one_of()`	Select columns names that are from a group of names

```
# Selecting the columns that with with the string "petal"
select(iris, starts_with("petal"))

# Selecting the columns that end with the string "length"
select(iris, ends_with("length"))
```

QUICK EXERCISE 1

Using the `select` verb from the `dplyr` package and the `airquality` dataset, answer to the following questions:

a) Select the first three columns

b) Select all columns except `Ozone` and `Temp`

c) Select columns ranging from `Solar.R` to `Day`

Selecting rows with `filter()`

The `filter` verb allows you to select a subset of rows in a data frame.

Like all single verbs, the first argument is a dataframe. The second and subsequent arguments refer to variables within that dataframe which are used to write test conditions used for row selection.

Places where the condition evaluates to `TRUE` are selected.

One of the coolest things about `filter` is that it allows to combine multiple test conditions.

You can use the Boolean operators (e.g. & (AND), | (OR), ! (NOT), >, <, >=, <=, !=, %in%) to create the logical tests.

Let's see some examples:

```
# Select iris specimens with sepal length higher than 7 (cm)
filter(iris, Sepal.Length > 7)

# This is equivalent to:
iris[iris$Sepal.Length > 7, ]
```

```r
# Combining multiple conditions (using the comma is equivalent to using & (AND) operator)
filter(iris, Sepal.Length > 7, Sepal.Width < 3.5)
```

QUICK EXERCISE 2

Using the `filter` verb from the `dplyr` package and the `iris` dataset, answer to the following questions:

a) Select iris specimens with a Petal length equal or higher than 5

b) Select iris specimens with a Petal length equal or higher than 3 *and* a Petal width smaller than 1.2 *or* a Sepal width smaller than 2.8

c) Select only the specimens from the *versicolor* species

The pipe operator: %>%

This operator allows you to pipe the output from one function to the input of another function thus making possible to combine multiple `dplyr` functions for data manipulation.

Instead of nesting functions (reading from the inside to the outside), the idea of of piping is to read the functions from left to right.

Overall, this makes the code more readable and easier to understand.

Here's an example:

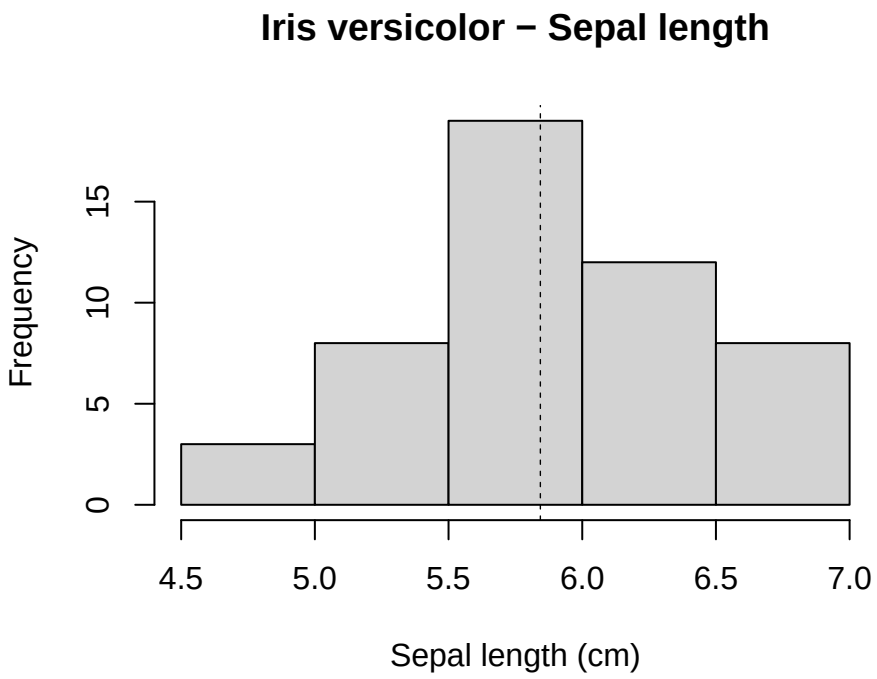
```r
# Applying a filter to select only the virginica species
iris %>% filter(Species == "virginica")
```

So you can see that the left side of the `%>%` pipe operator (the iris dataset) is used as the input for the function in the right side.

This way of manipulating is really powerful if used in a sequence of operations, like in this example used to make a histogram of sepal length for the *versicolor* species:

```r
# Note: pull is used to extract a single variable as a vector (not a table column)

iris %>%
  filter(Species == "versicolor") %>%
  pull(Sepal.Length) %>%
  hist(main="Iris versicolor - Sepal length", col="light grey", xlab="Sepal length (cm)") %>%
  abline(v = mean(iris %>% pull(Sepal.Length)), lwd=0.7, lty=2)
```



Pretty cool right? ;-) Piping really helps to make the code more readable, otherwise you would need to create temporary variables and use nested functions to do such plot.

To make things easier, in RStudio the `Ctrl` + `Shift` + `M` command can be used to write a pipe operator automatically.

Sorting rows with `arrange()`

The `arrange` verb allows reordering rows.

It takes a data frame, and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
# Sort by ascending order (default) of sepal length
arrange(iris, Sepal.Length)
iris %>% arrange(Sepal.Length) # OR using the pipe

# Sort by ascending order of sepal and then by petal length
arrange(iris, Sepal.Length, Petal.Length)
iris %>% arrange(Sepal.Length, Petal.Length)

# Sort by descending order of petal width
arrange(iris, desc(Petal.Width))
iris %>% arrange(desc(Petal.Width))
```

QUICK EXERCISE 3

Using the `arrange` verb from the `dplyr` package and the `iris` dataset, answer to the following question:

a) Using the pipe operator sort by descending order of petal length and then by ascending order of sepal length

Adding new columns with `mutate()`

In data manipulation it is often useful to add new columns that are functions of existing columns. This is the job of the `mutate` verb. Let's see some examples

```
# Calculate tree diameter times the height and divide by two:
mutate(trees, GirthHeight = (Girth * Height) / 2)
```

QUICK EXERCISE 4

Using the `mutate` verb from the `dplyr` package and the `trees` dataset (containing girth (inches), height (feet) and volume (vol timber in cubic ft) for black cherry trees), answer to the following question:

a) Create two new columns named `Girth_cm` and `Height_mt` that respectively convert the tree diameter and the height to cm and meters (hint: 1 meter = 3.2808399 feet, and 1 inch = 2.54cm);

b) Using the pipe operator make a sequence of processes for:

(i) add a new column named `hvr` as the ratio between the height and the volume,
(ii) select trees that have a `hrv` in the interval [3, 5[and
(iii) count how many elements correspond to that condition.

Creating summaries of the data frame using `summarise()` and `group_by()`

The `summarise` function is very useful to create summary statistics for a given column or variable in a dataframe such as finding the mean or the standard-deviation. This is done by applying a function over selected columns. Let's see an example (with pipes too):

```
iris %>% summarise(sepalLength_avg = mean(Sepal.Length, na.rm=TRUE), # Average
                   sepalLength_std = sd(Sepal.Length, na.rm=TRUE),   # Standard-deviation
                   sepalLength_min = min(Sepal.Length, na.rm=TRUE),  # Minimum
                   sepalLength_max = max(Sepal.Length, na.rm=TRUE),  # Maximum
                   count = n())                                      # Number of observations
```

```
##   sepalLength_avg sepalLength_std sepalLength_min sepalLength_max count
## 1        5.843333       0.8280661             4.3             7.9   150
```

Of course there are many other statistics that can used with `summarise`, for example: `quantile()`, `median()`, `n()` (which returns the length of a vector), `first()` (returns the first value), `last()` (returns the last value) and `n_distinct()` (which counts the number of distinct values). Custom functions are also supported.

Although `summarise` is very useful for performing overall summaries it is even more handy when combined with `group_by`to make summaries for different groups in the data.

The `group_by` uses the *"split-apply-combine"* concept to make statistical summaries by groups of observations. Using this we are able to "split" a data frame by some variable (e.g. a species, an experimental group, different regions), apply a function to each individual data frame (by different levels/groups) and then combine the output.

Let's check out some examples using (again) the iris dataset to compare different species:

```
iris %>%
  group_by(Species) %>%
  summarise(sepalLength_avg = mean(Sepal.Length, na.rm=TRUE), # Average
```

```
                    sepalLength_std = sd(Sepal.Length, na.rm=TRUE),    # Standard-deviation
                    sepalLength_min = min(Sepal.Length, na.rm=TRUE),   # Minimum
                    sepalLength_max = max(Sepal.Length, na.rm=TRUE),   # Maximum
                    count = n())                                       # Number of observations
```

```
## # A tibble: 3 x 6
##   Species  sepalLength_avg sepalLength_std sepalLength_min sepalLength_max
##   <fct>              <dbl>           <dbl>           <dbl>           <dbl>
## 1 setosa              5.01           0.352             4.3             5.8
## 2 versico~            5.94           0.516             4.9             7
## 3 virgini~            6.59           0.636             4.9             7.9
## # ... with 1 more variable: count <int>
```

Notice how the `group_by` was used to calculate the same summary but this time by different groups of observations made by each one of the iris species.

The `group_by` verb also enables to use more than one grouping variable (separated by a comma). Here's an example with the `npk` dataset (a classical N, P, K factorial plant growth experiment):

```
npk %>%
  group_by(block, N) %>%
  summarise(avg_yield = mean(yield),
            std_yield = sd(yield),
            count = n())
```

```
## # A tibble: 12 x 5
## # Groups:   block [?]
##    block N     avg_yield std_yield count
##    <fct> <fct>     <dbl>     <dbl> <int>
##  1 1     0          48.2     1.91      2
##  2 1     1          59.9     4.10      2
##  3 2     0          55.8     0.354     2
##  4 2     1          59.2     0.919     2
##  5 3     0          58.9     5.52      2
##  6 3     1          62.6     9.69      2
##  7 4     0          44.8     0.919     2
##  8 4     1          55.4     9.33      2
##  9 5     0          50.2     1.91      2
## 10 5     1          50.9     1.56      2
## 11 6     0          54.6     1.98      2
## 12 6     1          58.1     1.27      2
```

QUICK EXERCISE 5

a) Using the `group_by` and the `summarise` verbs with the `PlantGrowth` dataset, calculate the minimum and the maximum, the quartiles 25%, 50% and 75% of the `weight` along with the number of observations by experimental group (variable `group`).