Filter rows

Prof. Dr. Nicolas Meseth

Summary

This chapter introduces the following new concepts and functions:

- `filter()`
- Arithmetic operators such as `==`, `!=`, `>´,`<,>=, `and`<=‘
- Logical operators such as `&`, `|`, and `xor()`
- The `between()` function
- `slice()` and its variants

The `filter` command

Besides selecting the columns we need, we need tools to restrict the rows in a data frame. For this, the `{dplyr}` package offers the `filter` command.

The `filter` command takes one or more expressions, which must evaluate to `TRUE` or `FALSE`. These types of expressions are called *boolean expressions*, named after George Boole, who invented the Boolean algebra. Every expression passed to the `filter` command is evaluated for every row in the data frame. Only if the expression returns `TRUE` for a row, this row is included in the resulting data frame.

To form expressions, we can use a number of operators and functions. This chapter introduces the basic ways to express filter conditions on our data.

Equals operator

The simplest way to filter data is to compare column to a given value. This way, we can get all orders from female customers:

```r
orders %>%
  filter(customer_gender == "f")
```

```
#> filter: removed 1,613 rows (56%), 1,261 rows remaining
```

As you can see, the equals operator in R consists of two equal signs in a row (`==`). This is important, as using only one equals sign results in an error. A single equals sign is reserved for assignments, such as when we create a new column with `mutate`.

In the example above, the `customer_gender` column is of the data type `chr`, which means it contains alphanumeric symbols. For such columns, when comparing values, we must enclose the literal values with quotations marks. This is because the data type `chr` can contain spaces. If we didn't use quotation marks, R wouldn't know where the string of alphanumeric character starts and ends.

The equals comparison `==` is useful mostly for discrete data types. Un R, these include strings (or `chr`), whole numbers (`integer`), dates, and factors. Data types such as decimal numbers (`double`) or datetime can in principle compared to a specific value using the comparison operator `==`, but given their continuous nature, it usually doesn't make too much sense. Arithmetic operators, such as less than or greater than, are much more useful in these cases.

Arithmetic operators

The following filter removes all rows where the total price is below 50 euros:

```
orders %>%
  filter(total_price < 50)

#> filter: removed 633 rows (22%), 2,241 rows remaining
```

We can combine filter conditions by listing them comma-separated:

```
orders %>%
  filter(total_price < 50, customer_gender == "f")

#> filter: removed 1,868 rows (65%), 1,006 rows remaining
```

This is equivalent to having two subsequent `filter` statements in a pipeline:

```
orders %>%
  filter(total_price < 50) %>%
  filter(customer_gender == "f")

#> filter: removed 633 rows (22%), 2,241 rows remaining
#> filter: removed 1,235 rows (55%), 1,006 rows remaining
```

Logical combinations of filter expressions

As shown above, When we list two filter expressions separated by comma, they are connected with the logical operator *and*:

```
# Customer who are female and university staff at the same time
orders %>%
  filter(customer_gender == "f", customer_is_hsos == TRUE)

#> filter: removed 2,651 rows (92%), 223 rows remaining
```

We can do that explicitly by using the official *and* operator, which is denoted by the symbol `&`.

```
# Same as above, with explicit AND symbol
orders %>%
  filter(customer_gender == "f" & customer_is_hsos == TRUE)

#> filter: removed 2,651 rows (92%), 223 rows remaining
```

Or by having two subsequent `filter` command in our pipeline:

```
# Same as above, but with two filter commands in a row
orders %>%
  filter(customer_gender == "f") %>%
  filter(customer_is_hsos == TRUE)

#> filter: removed 1,613 rows (56%), 1,261 rows remaining
#> filter: removed 1,038 rows (82%), 223 rows remaining
```

An advantage of two `filter` commands is that the `{tidylog}` package prints the effect for each of the two filter expressions separately. So if we are interested in that, this is a good option.

Another way to logically combine filter expressions is the *OR* operator, which is symbolized by the `|` character:

```
# Customers who are either female or university staff (or both)
orders %>%
  filter(customer_gender == "f" | customer_is_hsos == TRUE)

#> filter: removed 1,352 rows (47%), 1,522 rows remaining
```

The *OR* operator is fundamentally different to the *AND* operator. In contrast to the example with the *AND*, a row in the *OR* example must only meet one of the two conditions to be kept in the result. It can meet both, but only one is required. Only if both evaluate to `FALSE`, the row is removed.

The `between` function

If we want to keep records whose value for numerical column is within a give range, we can achieve this with the logical *AND*:

```
orders %>%
  filter(total_price >= 10 & total_price <= 20) %>%
  select(total_price)

#> filter: removed 2,392 rows (83%), 482 rows remaining
#> select: dropped 67 variables (order_id, name, order_number, app_id, created_at, …)
#> # A tibble: 482 x 1
#>    total_price
#>          <dbl>
#>  1        10
#>  2        12
#>  3        15.0
#>  4        14.9
#>  ...
```

For filtering on ranges, the `between()` function is an alternative:

```
# This is equivalent and a bit more efficient than a combination of >= and <=
orders %>%
  filter(between(total_price, 10, 20))

#> filter: removed 2,392 rows (83%), 482 rows remaining
```

Filtering based on a record's index

```
# Keep only the first row
orders %>%
  slice(1)
```

```r
# Keep the first 10 rows
orders %>%
  slice(1:10)
```