Data visualisation with R - part 4: data processing with dplyr

Hannah Meyer

January 2020

Contents

Introduction

In the previous tutorials, you have learned how to visualise your data, from simple scatter plots with default settings to compound figures with elaborate color schemes and labels. For all of these, we used the data from (Smith et al. 2004) (made available at the following http://www.antigenic-cartography.org/). I mentioned briefly in the previous exercises, that I had formated the data for us to work with. What I expressed there in a half-sentence, usually contains a lot of work, often more than the actual analysis: cleaning your data, bringing it into the right format, checking it for sanity. In the following sections, we will learn how to use `dplyr` to reformat data into a 'tidy' format that we can use for visualisation and analysis.

For a more detailed description and additional examples, refer to chapter 5 in Hadley Wickham book 'R for Data Science' (Wickham and Grolemund 2017). I would generally highly recommend this book, as it introduces concepts we have discussed in this course and beyond - its online version is available for free here!

In addition, for an overview and help on `dplyr` functions take a look at the `dplyr` cheat sheet accessible by choosing *Help > Cheatsheets > Data Transformation with* ***`dplyr`*** in the RStudio tool bar.

Set-up

First, we are going to set up our analysis script,

```
knitr::opts_chunk$set(echo = TRUE,
                      comment = "#>",
                      collapse = TRUE,
                      fig.width = 6,
                      fig.align = "center",
                      fig.pos = 'h',
                      out.width = "70%")
```

load the required libraries

```
library("tidyverse")
```

and read our dataset into R again:

```
coord <- read_csv("data/2004_Science_Smith_data.csv")
#> Parsed with column specification:
#> cols(
#>   name = col_character(),
#>   year = col_double(),
#>   cluster = col_character(),
#>   type = col_character(),
#>   x.coordinate = col_double(),
#>   y.coordinate = col_double(),
#>   location = col_character(),
#>   lat = col_double(),
#>   lng = col_double()
#> )
```

dplyr: a grammar of data manipulation

The `dplyr` package is a core member of the tidyverse. Its main functionality relies on six functions that let us solve the majority of data reformating. These functions, often described as the 'verbs for the language of data manipulation', are:

- `arrange()`: to change the order of observations;
- `select()`: to select a subset of variables;
- `mutate()`: to add new variables that are functions of existing variables;
- `filter()`: to subset observations based on their values;
- `summarise()`: to summarise observations to a single row;
- `group_by()`: to change the unit of analysis from the complete dataset to individual groups.

All verbs work in a similar fashion:

- their first argument is a `tibble` or `data.frame`;
- the following arguments describe the action to apply to that `tibble`/`data.frame` by specifying the variable names
- the result is a new `tibble`/`data.frame`, dependent on the initial input

Note: If you want to save the results of a `dplyr` function, you have to use the assignment operator `<-`, as `dplyr` functions never modify their input data.

Change the order of observations with `arrange()`

Using `arrange()`, we can change the order of rows. As input, `arrange` takes a `tibble` and a set of variable names - at least one is required. It will re-arrange the observations of the `tibble` based on variable selected for re-ordering. If you provided more than one variable to order the tibble by, each additional variable will be used to break ties in the values of preceding variables.

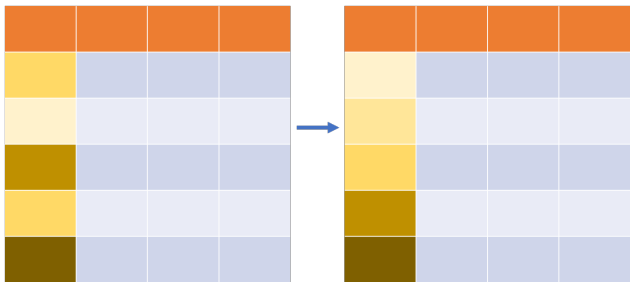


Figure 1: arrange()

Note: These visualisations for `dplyr` verb function are inspired by the `dplyr` cheat sheet.

Here, we are going to order `coord` by location and year:

```
arrange(coord, location, year)
#> # A tibble: 322 x 9
#>    name   year cluster type  x.coordinate y.coordinate location   lat
#>    <chr> <dbl> <chr>   <chr>        <dbl>        <dbl> <chr>    <dbl>
#>  1 AK/4~  1993 BE92    AG           6.21         -3.79  AKITA     39.7
#>  2 AM/1~  1977 VI75    AG           0.895         9.41  AMSTERD~  52.4
#>  3 AM/1~  1977 VI75    SR           0.211         7.10  AMSTERD~  52.4
#>  4 AM/4~  1992 BE89    AG          -2.00         -0.924 AMSTERD~  52.4
#>  5 AT/3~  1988 SI87    AG          -0.765         2.75  ATLANTA   33.8
#>  6 AT/2~  1989 SI87    AG          -1.61          1.03  ATLANTA   33.8
#>  7 AU/1~  1997 SY97    AG          -3.28        -10.1   AUCKLAND -36.8
#>  8 AU/1~  1997 SY97    SR          -2.02         -9.26  AUCKLAND -36.8
#>  9 AL/4~  1982 BK79    AG          -4.07          6.18  AUSTRAL~ -33.9
#> 10 BA/1~  1979 BK79    AG          -5.95          5.44  BANGKOK   13.8
#> # ... with 312 more rows, and 1 more variable: lng <dbl>
```

As a default, `arrange` orders the variables in ascending order. To re-order in descending order use `desc()`:

```
arrange(coord, desc(location), year)
#> # A tibble: 322 x 9
#>    name   year cluster type  x.coordinate y.coordinate location   lat   lng
#>    <chr> <dbl> <chr>   <chr>        <dbl>        <dbl> <chr>    <dbl> <dbl>
#>  1 YA/5~  1993 BE92    AG           6.59         -3.91  YAMAGA    33.0  131.
#>  2 YA/6~  1993 BE92    AG           7.74         -2.70  YAMAGA    33.0  131.
#>  3 YA/6~  1993 BE92    AG           6.93         -3.59  YAMAGA    33.0  131.
#>  4 WU/3~  1995 WU95    AG           0.0395       -6.20  WUHAN     30.6  114.
#>  5 WE/4~  1985 BK79    AG          -2.93          4.27  WELLING~ -41.3  175.
#>  6 WE/4~  1985 BK79    SR          -1.72          3.62  WELLING~ -41.3  175.
#>  7 WE/5~  1989 SI87    AG           0.891         0.733 WELLING~ -41.3  175.
#>  8 WE/3~  1990 BE89    AG          -2.06         -1.45  WELLING~ -41.3  175.
#>  9 WE/5~  1993 BE92    AG           5.22         -4.56  WELLING~ -41.3  175.
#> 10 WK/1~  1989 SI87    AG           0.382         1.53  WAIKATO  -37.6  175.
#> # ... with 312 more rows
```

Exercises

1. How does the ouput change if you sorted by year first, then location?
2. Sort `coord` to find when the last virus was isolated.
3. Find the largest and smallest x.coordinate in the dataset.

Select a subset of variables with `select()`

`select()` let's us pick a subset of variables. In the most simple case, we specify the `tibble` from which we want to `select()` variables, followed by the names of the variables that we want to pick.

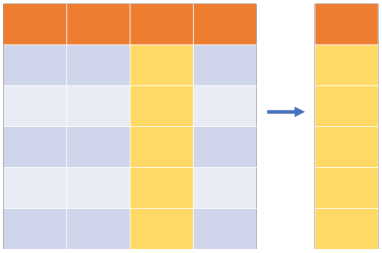


Figure 2: select()

For instance, we could select only name and location:

```
select(coord, name, location)
#> # A tibble: 322 x 2
#>    name         location
#>    <chr>        <chr>
#>  1 BI/15793/68 BILTHOVEN
#>  2 BI/16190/68 BILTHOVEN
#>  3 BI/16398/68 BILTHOVEN
#>  4 BI/808/69   BILTHOVEN
#>  5 BI/908/69   BILTHOVEN
#>  6 BI/17938/69 BILTHOVEN
#>  7 BI/93/70    BILTHOVEN
#>  8 BI/2668/70  BILTHOVEN
#>  9 BI/6449/71  BILTHOVEN
#> 10 BI/21438/71 BILTHOVEN
#> # ... with 312 more rows
```

To exclude a variable, use `select()` with the variable name preceded by a minus `-`:

```
select(coord, -type)
#> # A tibble: 322 x 8
#>    name         year cluster x.coordinate y.coordinate location   lat   lng
#>    <chr>       <dbl> <chr>           <dbl>        <dbl> <chr>    <dbl> <dbl>
#>  1 BI/15793/68  1968 HK68             4.05         15.0 BILTHOV~  52.1  5.02
#>  2 BI/16190/68  1968 HK68             4.10         14.8 BILTHOV~  52.1  5.02
#>  3 BI/16398/68  1968 HK68             4.36         13.9 BILTHOV~  52.1  5.02
#>  4 BI/808/69    1969 HK68             3.87         14.3 BILTHOV~  52.1  5.02
#>  5 BI/908/69    1969 HK68             4.87         14.1 BILTHOV~  52.1  5.02
#>  6 BI/17938/69  1969 HK68             4.40         14.9 BILTHOV~  52.1  5.02
#>  7 BI/93/70     1970 HK68             5.06         14.5 BILTHOV~  52.1  5.02
#>  8 BI/2668/70   1970 HK68             4.82         15.5 BILTHOV~  52.1  5.02
#>  9 BI/6449/71   1971 HK68             3.87         15.9 BILTHOV~  52.1  5.02
```

```
#> 10 BI/21438/71  1971 HK68            4.27         14.1 BILTHOV~  52.1  5.02
#> # ... with 312 more rows
```

In addition to that, there a number of functions that can be used in conjunction with `select()` that let us select multiple variables with common elements in their name. These include:

- `starts_with("year")`: matches names that begin with "year".
- `ends_with("coordinate")`: matches names that end with "coordinate".
- `contains("coord")`: matches names that contain "coord".
- `num_range("X", 1:3)`: matches X1, X2 and X3.
- `everything()`: matches everything that has not specifically been named before (see Exercises to mutate for example).

Exercises

1. What are two other ways of only selecting the `x.coordinate` and `y.coordinate` columns?
2. Can you use `select` to move the `location` variable from second column to last?

Add new variables with `mutate()`

To add new variables to a `tibble`, we use `mutate`. Again, `mutate` first expects the name of the `tibble` to which we want to add a variable, followed by the name of the new variable, an equal sign `=` and the data to add. The data has to have the same number of observations as our input `tibble` and can be created by using a transformation of an existing variable.

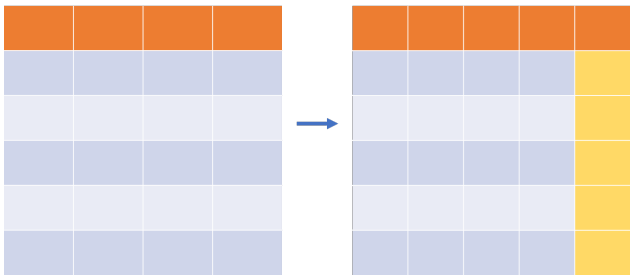


Figure 3: mutate()

Here we create a new variable, that contains the Euclidean distance of each antigen from the origin at (0,0):

```
mutate(coord, distance=sqrt(x.coordinate^2 + y.coordinate^2))
#> # A tibble: 322 x 10
#>    name   year cluster type  x.coordinate y.coordinate location   lat   lng
#>    <chr> <dbl> <chr>   <chr>        <dbl>        <dbl> <chr>    <dbl> <dbl>
#>  1 BI/1~  1968 HK68    AG            4.05         15.0 BILTHOV~  52.1  5.02
#>  2 BI/1~  1968 HK68    AG            4.10         14.8 BILTHOV~  52.1  5.02
#>  3 BI/1~  1968 HK68    AG            4.36         13.9 BILTHOV~  52.1  5.02
#>  4 BI/8~  1969 HK68    AG            3.87         14.3 BILTHOV~  52.1  5.02
#>  5 BI/9~  1969 HK68    AG            4.87         14.1 BILTHOV~  52.1  5.02
#>  6 BI/1~  1969 HK68    AG            4.40         14.9 BILTHOV~  52.1  5.02
#>  7 BI/9~  1970 HK68    AG            5.06         14.5 BILTHOV~  52.1  5.02
#>  8 BI/2~  1970 HK68    AG            4.82         15.5 BILTHOV~  52.1  5.02
#>  9 BI/6~  1971 HK68    AG            3.87         15.9 BILTHOV~  52.1  5.02
#> 10 BI/2~  1971 HK68    AG            4.27         14.1 BILTHOV~  52.1  5.02
#> # ... with 312 more rows, and 1 more variable: distance <dbl>
```

Note: `mutate()` adds the column at the end of the `tibble`; if you want it at a different position use `select` to reorder the variables afterwards.

Exercises

1. Add a new column that contains the time difference between this year, 2020, and the year the virus was isolated.
2. Save the result of this `mutate` in a new object.
3. Move the new column between the `year` and `cluster` columns.

Select of subset of observations with `filter()`

`select` creates a subset of the input data by selecting variables. `filter()` allows us to subset our input data by observations. As a first argument it takes the name of the `tibble`; this is followed by expressions that filter observations based on their value in the specified variables. Filtering uses the standard set of comparison operators available in R.

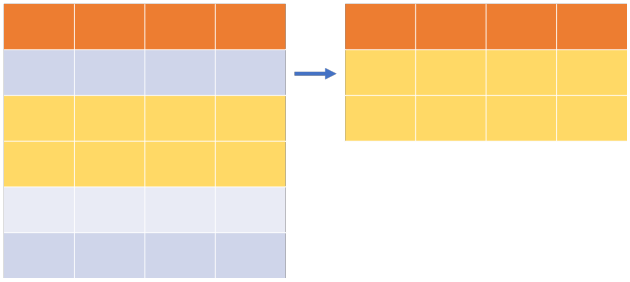


Figure 4: filter()

Comparisons

Comparison operators in R:

- `==`: equal to
- `!=`: not equal to
- `>`: greater than
- `>=`: greater or equal than
- `<`: less than
- `<=`: less or equal than

Note: When testing for equality, make sure to use `==` and not a simple `=`!

For instance, the following code let's us select all observations of viruses in circulation before 1988:

```
filter(coord, year < 1988)
#> # A tibble: 88 x 9
#>    name   year cluster type  x.coordinate y.coordinate location   lat   lng
#>    <chr> <dbl> <chr>   <chr>        <dbl>        <dbl> <chr>    <dbl> <dbl>
#>  1 BI/1~  1968 HK68    AG            4.05         15.0 BILTHOV~  52.1  5.02
#>  2 BI/1~  1968 HK68    AG            4.10         14.8 BILTHOV~  52.1  5.02
#>  3 BI/1~  1968 HK68    AG            4.36         13.9 BILTHOV~  52.1  5.02
#>  4 BI/8~  1969 HK68    AG            3.87         14.3 BILTHOV~  52.1  5.02
#>  5 BI/9~  1969 HK68    AG            4.87         14.1 BILTHOV~  52.1  5.02
#>  6 BI/1~  1969 HK68    AG            4.40         14.9 BILTHOV~  52.1  5.02
```

```
#>  7 BI/9~  1970 HK68    AG            5.06         14.5 BILTHOV~  52.1  5.02
#>  8 BI/2~  1970 HK68    AG            4.82         15.5 BILTHOV~  52.1  5.02
#>  9 BI/6~  1971 HK68    AG            3.87         15.9 BILTHOV~  52.1  5.02
#> 10 BI/2~  1971 HK68    AG            4.27         14.1 BILTHOV~  52.1  5.02
#> # ... with 78 more rows
```

In addition to these comparison operators, we can also use boolean operators to filter the input data. The simplest boolean operator is intrinsic to the `filter` function: multiple arguments to `filter()` are combined with *and*, i.e. every expression has to be true for an observation to be kept in the output. For instance, all observations of viruses in circulation before 1988 in BILTHOVEN.

```
filter(coord, year < 1988, location == "BILTHOVEN")
#> # A tibble: 40 x 9
#>    name   year cluster type  x.coordinate y.coordinate location   lat   lng
#>    <chr> <dbl> <chr>   <chr>        <dbl>        <dbl> <chr>    <dbl> <dbl>
#>  1 BI/1~  1968 HK68    AG            4.05         15.0 BILTHOV~  52.1  5.02
#>  2 BI/1~  1968 HK68    AG            4.10         14.8 BILTHOV~  52.1  5.02
#>  3 BI/1~  1968 HK68    AG            4.36         13.9 BILTHOV~  52.1  5.02
#>  4 BI/8~  1969 HK68    AG            3.87         14.3 BILTHOV~  52.1  5.02
#>  5 BI/9~  1969 HK68    AG            4.87         14.1 BILTHOV~  52.1  5.02
#>  6 BI/1~  1969 HK68    AG            4.40         14.9 BILTHOV~  52.1  5.02
#>  7 BI/9~  1970 HK68    AG            5.06         14.5 BILTHOV~  52.1  5.02
#>  8 BI/2~  1970 HK68    AG            4.82         15.5 BILTHOV~  52.1  5.02
#>  9 BI/6~  1971 HK68    AG            3.87         15.9 BILTHOV~  52.1  5.02
#> 10 BI/2~  1971 HK68    AG            4.27         14.1 BILTHOV~  52.1  5.02
#> # ... with 30 more rows
```

Any more complicated combinations, like in circulation before 1988 but not in BILTHOVEN and not in MADRID, can be constructed using the logical operators *and* `&`, *or* `|` and *not* `!`. The graphic below shows a complete overview of selection logical subsets of observations for two variables "red" and "blue":

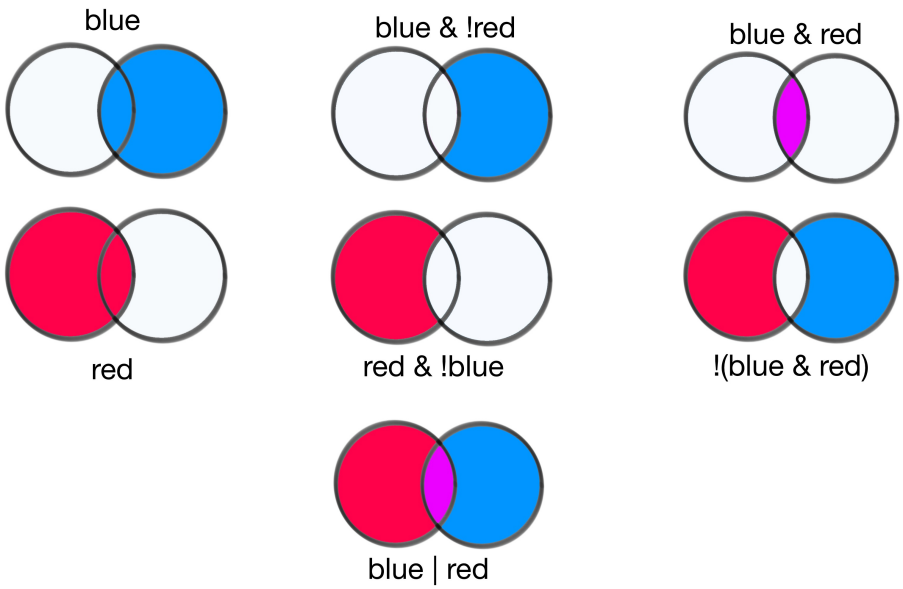


Figure 5: Boolean Operators

Using these logical operators, we can filter for viruses in circulation before 1988 but only in BILTHOVEN or in MADRID:

```
filter(coord, year < 1988, location == "BILTHOVEN" | location == "MADRID")
#> # A tibble: 40 x 9
#>    name   year cluster type  x.coordinate y.coordinate location   lat   lng
#>    <chr> <dbl> <chr>   <chr>        <dbl>        <dbl> <chr>    <dbl> <dbl>
#>  1 BI/1~  1968 HK68    AG            4.05         15.0 BILTHOV~  52.1  5.02
#>  2 BI/1~  1968 HK68    AG            4.10         14.8 BILTHOV~  52.1  5.02
#>  3 BI/1~  1968 HK68    AG            4.36         13.9 BILTHOV~  52.1  5.02
#>  4 BI/8~  1969 HK68    AG            3.87         14.3 BILTHOV~  52.1  5.02
#>  5 BI/9~  1969 HK68    AG            4.87         14.1 BILTHOV~  52.1  5.02
#>  6 BI/1~  1969 HK68    AG            4.40         14.9 BILTHOV~  52.1  5.02
#>  7 BI/9~  1970 HK68    AG            5.06         14.5 BILTHOV~  52.1  5.02
#>  8 BI/2~  1970 HK68    AG            4.82         15.5 BILTHOV~  52.1  5.02
#>  9 BI/6~  1971 HK68    AG            3.87         15.9 BILTHOV~  52.1  5.02
#> 10 BI/2~  1971 HK68    AG            4.27         14.1 BILTHOV~  52.1  5.02
#> # ... with 30 more rows
```

Alternatively, we can solve the above using the `x %in% y` syntax:

```
filter(coord, year < 1988, location %in% c("BILTHOVEN", "MADRID"))
#> # A tibble: 40 x 9
#>    name   year cluster type  x.coordinate y.coordinate location   lat   lng
#>    <chr> <dbl> <chr>   <chr>        <dbl>        <dbl> <chr>    <dbl> <dbl>
#>  1 BI/1~  1968 HK68    AG            4.05         15.0 BILTHOV~  52.1  5.02
#>  2 BI/1~  1968 HK68    AG            4.10         14.8 BILTHOV~  52.1  5.02
#>  3 BI/1~  1968 HK68    AG            4.36         13.9 BILTHOV~  52.1  5.02
#>  4 BI/8~  1969 HK68    AG            3.87         14.3 BILTHOV~  52.1  5.02
#>  5 BI/9~  1969 HK68    AG            4.87         14.1 BILTHOV~  52.1  5.02
#>  6 BI/1~  1969 HK68    AG            4.40         14.9 BILTHOV~  52.1  5.02
#>  7 BI/9~  1970 HK68    AG            5.06         14.5 BILTHOV~  52.1  5.02
#>  8 BI/2~  1970 HK68    AG            4.82         15.5 BILTHOV~  52.1  5.02
#>  9 BI/6~  1971 HK68    AG            3.87         15.9 BILTHOV~  52.1  5.02
#> 10 BI/2~  1971 HK68    AG            4.27         14.1 BILTHOV~  52.1  5.02
#> # ... with 30 more rows
```

where we select every row where `x`, in this case location, is contained in `y`, here a vector of names.

Note: vectors are R data objects and are constructed by enclosing the variables`x`, `y`, `z` to be put in the vector in `c()`: `c(x,y,z)`

Exercises

1. Find viruses whose isolation is specified as NETHERLANDS.
2. Filter for rows that are of type antigen (AG).
3. Find viruses in circulation after 1997 and are assigned to either cluster SY97 or cluster WU95.

Group analyses and summarise observations with `group_by` and `summarise()`

`group_by` and `summarise()` often go hand in hand: using `group_by`, we can first group observation based on values in the specified variable and then apply a summary statistic on this group. This might sound a bit complicated, so let's have a look at an example. We are fist going to group `coord` by cluster. To pass this grouped `tibble` onto the `summarise` function, we have to save the grouped `tibble` into a new object (for now, we will see further down how we can do this more elegantly). We pass this new object to `summarise`, to find the start and end year of circulation for viruses per cluster.

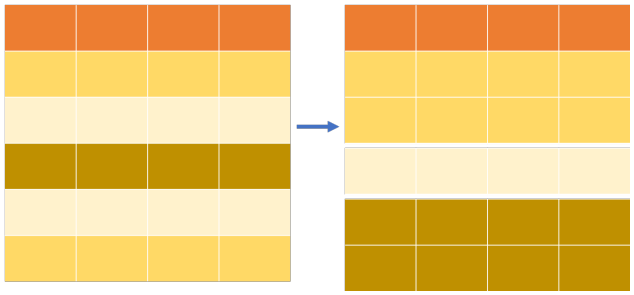


Figure 6: (ref:groupby-caption)

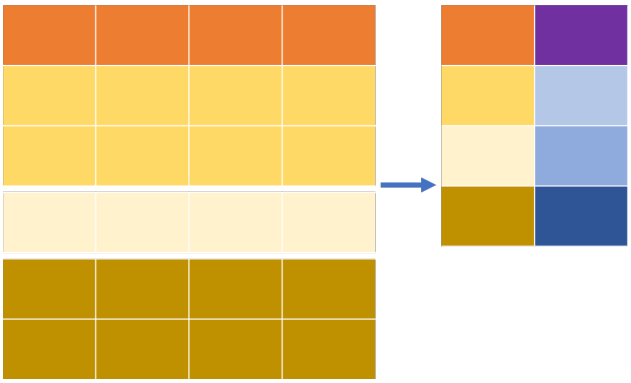


Figure 7: summarise()

(ref:groupby-caption) group_by().

To find the start and end year, we can use the R functions `min` and `max`, that find the minimum and maximum entry of a `tibble` column or vector, respectively:

```
coord_grouped <- group_by(coord, cluster)
summarise(coord_grouped,
          start=min(year),
          end=max(year))
#> # A tibble: 11 x 3
#>    cluster start   end
#>    <chr>   <dbl> <dbl>
#>  1 BE89     1989  1993
#>  2 BE92     1992  1996
#>  3 BK79     1979  1988
#>  4 EN72     1972  1975
#>  5 FU02     2002  2003
#>  6 HK68     1968  1972
#>  7 SI87     1987  1991
#>  8 SY97     1997  2003
#>  9 TX77     1976  1977
#> 10 VI75     1975  1977
#> 11 WU95     1993  1998
```

Functions used with `summarise` have to return a single value. Other useful functions are for instance:

- `mean`: returns mean value;
- `median`: returns median values;

- `sum`: return the sum of input values;
- `n()`: returns total count (this is the only one that does not take an argument);
- `n_distinct`: returns the unique count;

Exercises

1. How do you know if a `tibble` is grouped or not? How do you ungroup a grouped `tibble` (Hint: use the help function of `group_by`).
2. What happens if you use the same `summarise` command on the original, ungrouped `tibble`?
3. Find the mean x and y coordinate of each cluster.
4. Find the total and distinct number of locations per cluster.

Creating a workflow with pipes

We have seen above how useful it is to first use `group_by` and then `summarise` on the grouped `tibble`. However, we had to first save the grouped `tibble` in a new object, that we passed on to summarise. This object solely served the purpose of an intermediate step in our workflow. To avoid having to create intermediate objects, we can use the pipe function `%>%`. To do the same grouping and summarising as above, we can now simply write:

```
coord %>%
    group_by(cluster) %>%
    summarise(start=min(year), end=max(year))
#> # A tibble: 11 x 3
#>    cluster start   end
#>    <chr>   <dbl> <dbl>
#>  1 BE89     1989  1993
#>  2 BE92     1992  1996
#>  3 BK79     1979  1988
#>  4 EN72     1972  1975
#>  5 FU02     2002  2003
#>  6 HK68     1968  1972
#>  7 SI87     1987  1991
#>  8 SY97     1997  2003
#>  9 TX77     1976  1977
#> 10 VI75     1975  1977
#> 11 WU95     1993  1998
```

where we pipe `coord` into the grouping function, and then pipe the output of that into the summarise function.

This is really powerful and elegant, when we have to apply many processing steps to our data and want to avoid having to create lot's of intermediate objects.

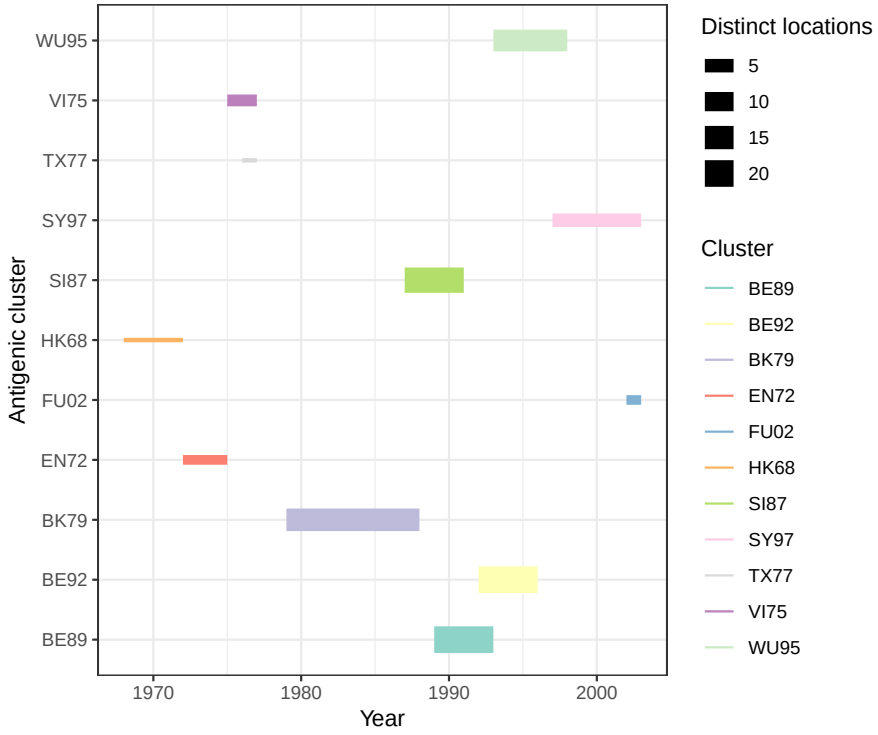
For instance, in this chunk, we apply grouping by cluster, then summarise start, end and number of locations and then sort in ascending order by start year. We save the entire process in a new object by using the `<-` operator.

```
circulation_summary <- coord %>%
    group_by(cluster) %>%
    summarise(start=min(year),
              end=max(year),
              location=n_distinct(location)) %>%
    arrange(start)
```

We can then visualise the cluster transition over time in a segment plot. We additionally map the number of distinct locations to the size aesthetic, to visualise the its spread.

```
circulation_summary <- coord %>%
    group_by(cluster) %>%
    summarise(start=min(year),
              end=max(year),
              location=n_distinct(location)) %>%
    arrange(start)

p <- ggplot(circulation_summary)
p + geom_segment(aes(x=start, xend=end, y=cluster, yend=cluster,
                     size=location, color=cluster)) +
    scale_color_brewer(type="qual", palette = "Set3") +
    labs(x="Year",
         y="Antigenic cluster",
         size="Distinct locations",
         color="Cluster") +
    theme_bw()
```



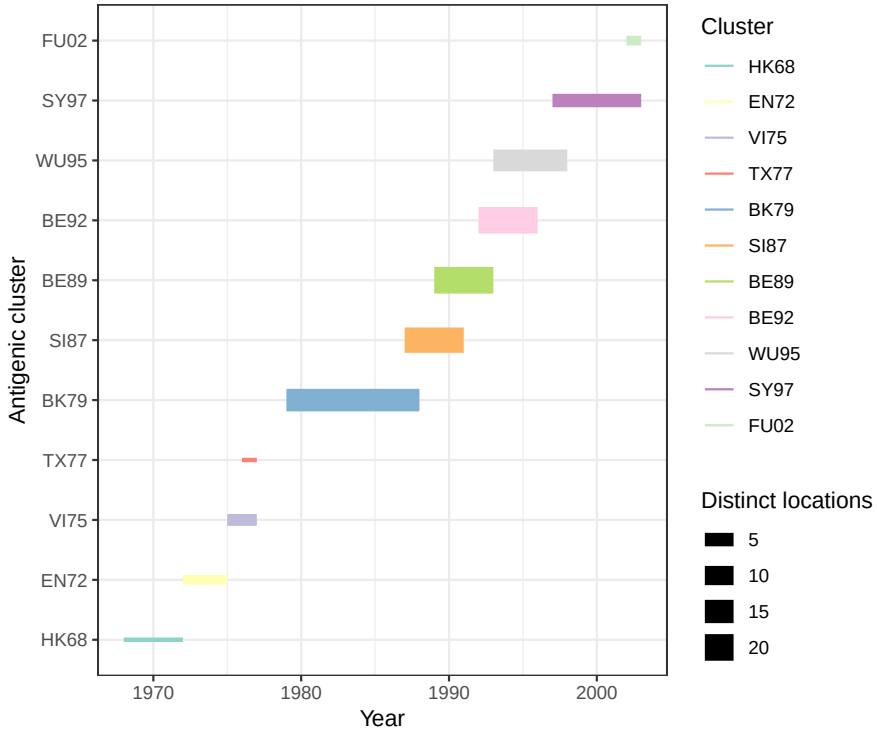
This plot does not yield the result we expected. Despite us having arranged the `tibble` by start year, this is not the order it was plotted. This is something very confusing to beginners and even more experienced R users stumble over this from time to time. When using a character column (here cluster names) as aes, `ggplot` treats it as a factor column (we mentioned factors as a representation of categorical variables with fixed possible values in part 2 of this course). By converting characters to factors, the values are internally ordered alphabetically, which in this case disrupts the ordering we desired.

To prevent this, we can explictly convert the cluster column into a factor and enforce the original order by the `fct_inorder` function. We can chain this processing onto our previous workflow using a pipe and the `mutate` function.

The segment plot with the ordered factor as y-labels looks as we expect:

```
circulation_summary <- coord %>%
    group_by(cluster) %>%
    summarise(start=min(year),
              end=max(year),
              location=n_distinct(location)) %>%
    arrange(start) %>%
    mutate(cluster=fct_inorder(cluster))

p <- ggplot(circulation_summary)
p + geom_segment(aes(x=start, xend=end, y=cluster, yend=cluster,
                     size=location, color=cluster)) +
    scale_color_brewer(type="qual", palette = "Set3") +
    labs(x="Year",
         y="Antigenic cluster",
         size="Distinct locations",
         color="Cluster") +
    theme_bw()
```



Exercises

1. Create a piped workflow that finds the number of distinct clusters and the first virus isolation for each location.

References

Smith, Derek J., Alan S. Lapedes, Jan C. de Jong, Theo M. Bestebroer, Guus F. Rimmelzwaan, Albert D. M. E. Osterhaus, and Ron A. M. Fouchier. 2004. "Mapping the Antigenic and Genetic Evolution

of Influenza Virus." *Science* 305 (5682). American Association for the Advancement of Science: 371–76. https://doi.org/10.1126/science.1097211.

Wickham, Hadley, and Garrett Grolemund. 2017. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. 1st ed. O'Reilly Media, Inc. https://r4ds.had.co.nz/.