Introduction to R with Tidyverse

Session 3: Data preparation and manipulation

Sophie Lee

Contents

Chapter 5: Data wrangling and summarising

5.1 Combining two datasets

We may need to combine data from different files within R to perform an analysis. For example, in our case we have the core spending power for each year between 2015 and 2020. If our analysis required comparing this spending over the time period, we would need to combine these files together.

Before the data can be combined, it must be loaded into R. We will begin combining data from 2015 and 2016, then extend this to the entire period.

```
# Return a list of files to copy from the working directory
list.files(path = "data")
```

```
## [1] "CSP_2015.csv"        "CSP_2016.csv"        "CSP_2017.csv"
## [4] "CSP_2018.csv"        "CSP_2019.csv"        "CSP_2020.csv"
## [7] "CSP_long_201520.csv"
```

```
# Load the 2015 data and attach as an object
CSP_2015 <- read_csv("data/CSP_2015.csv")

# Load the 2016 data and attach as an object
CSP_2016 <- read_csv("data/CSP_2016.csv")
```

Next, we will combine these datasets by joining them using key variable(s) which are shared between them. In this case, each local authority has a unique identifier code (`ons_code`) and naming variable (`authority`), they also should have the same `region` listed across both datasets.

In Tidyverse, there is a family of 'joining' functions that combine two datasets at a time. The choice of function depends on which observations we wish to keep where the joining variables do not match between data. In this example, we expect all local authority values to be the same across years, so will use the `full_join` function.

For more information about different joining options, check the helpfile via `?full_join`.

```
# Create a new object by joining the two datasets
csp_201516 <- full_join(CSP_2015, CSP_2016,
                        # List the key joining variables (in speech marks)
                        by = c("ons_code", "authority", "region"))
```

5.2 Joining multiple datasets

R's joining functions can only be applied to two datasets at a time. To combine all 6 core spending power datasets from 2015 to 2020 in this way would require a lot of repetitive coding (which we want to avoid where necessary).

An alternative approach would be to automate this process by using **functional programming**, implemented using tidyverse's `purrr` package.

The first step of this process requires loading all csv files into R by repeatedly applying `read_csv`. This requires a list of file names from the working directory. The function `list.files` introduced earlier contains an optional argument, `pattern` which can be used to return files and folders that match a naming pattern. In this case, all csv files begin "CSP_20", so to return this list of names from the *data* folder, we use the function:

```
csp_201520 <- list.files(path = "data", pattern = "CSP_20")
```

Next, we apply `read_csv` to each element of the list of file names. The function `map` allows us to do this and return a list of tibbles. As the data lies in a folder in the working directory, we must add this file path to the file names:

```
# Return a list of files in the data folder containing CSP_20
csp_201520 <- list.files(path = "data", pattern = "CSP_20") %>%
  # Add "data/" to each of these file names
  paste0("data/", .) %>%
  # Apply read_csv to every element of the list (of file names)
  map(read_csv)
```

Finally, we require a function that apply `full_join` iteratively to the list of tibbles and reduce it to a single tibble containing core spending powers for all years. The function that does this is `reduce`:

```r
# Return a list of files in the data folder containing CSP_20
csp_201520 <- list.files(path = "data", pattern = "CSP_20") %>%
  # Add "data/" to each of these file names
  paste0("data/", .) %>%
  # Apply read_csv to every element of the list (of file names)
  map(read_csv) %>%
  # Reduce the list of tibbles to a single object by iteratively joining
  reduce(full_join, by = c("ons_code", "authority", "region"))
```

5.3 Transforming data

The dataset containing core spending power in England between 2015 and 2020 is currently in what is known as **wide format**. This means there is a variable per measure per year, making the object very wide.

Some analyses and visualisations, particularly those used for temporal data, require a time variable in the dataset (for example, year). This requires the data to be in a different format, known as **long format**. Long format is where each row contains an observation per year (making the data much longer and narrower).

To convert data between wide and long formats, we can use the tidyverse functions `pivot_longer` and `pivot_wider`.

The first argument required by `pivot_longer` is the data we wish to transform. This is followed by the columns we wish to pivot (in this case, all variable other than the local authority codes, names, and regions). The next steps will depend on the format of data we wish to transform, format of the data we would like to generate, the values we need to include in the long dataset, and where this information will be extracted from.

For worked examples and a detailed explanation of different approaches that can be used to pivot data, access the vignette for these function by entering `vignette("pivot")` into the R console.

In the core spending power example, the new dataset will contain a row per local authority per year. A new `year` variable will be created using the suffix of the original variable names, and the prefix of the original names (e.g. `sfa`) will be retained for the new variable names.

Using a combination of the helpfile (`?pivot_longer`) and vignette, the arguments required to convert this data are `names_to`, to specify the old variable names will be used in the new data, and `names_pattern` to define how the old variable names will be separated.

```
# Create an object csp_long by pivotting csp_201520
csp_long <- pivot_longer(csp_201520,
                         # Pivot columns sfa_2015 up to and including rsdg_2020
                         cols = sfa_2015:rsdg_2020,
                         # Separate the old variable names in two,
                         # keep the prefix as it was, and put the suffix
                         # into a new variable, year
                         names_to = c(".value", "year"),
                         # The name prefix and suffix were separated by an _,
                         # the prefix can take different lengths, the suffix
                         # is always the final 4 characters
                         names_pattern = "(.*)_(....)")

# Check the new, long dataset's structure
str(csp_long)
```

```
## tibble [2,376 x 10] (S3: tbl_df/tbl/data.frame)
##  $ ons_code   : chr [1:2376] "E07000223" "E07000223" "E07000223" "E07000223" ...
##  $ authority  : chr [1:2376] "Adur" "Adur" "Adur" "Adur" ...
##  $ region     : chr [1:2376] "SE" "SE" "SE" "SE" ...
##  $ year       : chr [1:2376] "2015" "2016" "2017" "2018" ...
##  $ sfa        : num [1:2376] 3.02 2.39 1.92 1.7 1.74 ...
##  $ under_index: num [1:2376] 0.0234 0.0234 0.0248 0.039 0.0567 ...
##  $ ct_total   : num [1:2376] 5.47 5.68 5.85 6.08 6.35 ...
##  $ nhb        : num [1:2376] 0.652 0.767 0.553 0.202 0.126 ...
##  $ nhb_return : num [1:2376] 0.00523 0.00374 0.00397 0 0 ...
##  $ rsdg       : num [1:2376] 0 0 0 0 0 ...
```

Notice that the new `year` variable is recognised as a character, not a numeric variable as we would like. This is because these values were taken from variable names, which R treats as characters. To fix this, we can use the `mutate` function to convert the new variable into a numeric variable.

We may also wish to calculate the total core spending power for each local authority per year to compare this over time:

```
# Create a new object based on the long data
csp_long2 <- mutate(csp_long,
                    # Convert year to a numeric variable
                    year = as.numeric(year),
                    # Create a new total spend variable
                    total_spend = sfa + under_index + ct_total + nhb +
                      nhb_return + rsdg)
```

After manipulating and transforming the data into the format we need for analysis and visualisation, we can save this object to reload later. Tibbles and data frame objects can be saved as CSV files using the `write_csv` function. Remember to save the data with a different name than the raw data to avoid overwriting these files.

```
write_csv(csp_long2, file = "data/CSP_long_201520.csv")
```

5.4 Summary tables

Summary tables can be created using the `summarise` function. This returns tables in a tibble format, meaning they can easily be customised and exported as CSV files (using the `write_csv` function).

The `summarise` function is set up similarly to the `mutate` function: summaries are listed and given variable names, separated by a comma. The difference between these functions is that `summarise` collapses the tibble into a single summary row, and the new variables must be created using a summary function.

Common examples of summary functions include:

- `mean`
- `median`
- `range` (gives the minimum and maximum values)
- `min`

- `max`
- `IQR` (interquartile range, gives the range of the middle 50% of the sample)
- `sd` (standard deviation, a measure of the spread when data are normally distributed)
- `sum`
- `n` (counts the number of rows the summary is calculated from)

For a full list of compatible summary functions, view the helpfile `?summarise`.

If we wanted to summarise the total core spending power between 2015 and 2020 across all local authorities, we can apply `summarise` to the long format data from the previous section:

```
summarise(csp_long2,
          # Return sum of the total_spend variable
          total_spend_all = sum(total_spend),
          # Return the mean total spend
          mean_total_spend = mean(total_spend),
          # Return the median total spend
          median_total_spend = median(total_spend),
          # Return the 10th percentile (the value that 10% of the sample lies below)
          quantile10_total_spend = quantile(total_spend, 0.1),
          # Count the number of rows that have been summarised
          total_obs = n())
```

```
## # A tibble: 1 x 5
##   total_spend_all mean_total_spend median_total_spend quantile10_total_spend
##             <dbl>            <dbl>              <dbl>                  <dbl>
## 1         263484.             111.               17.6                   8.34
## # i 1 more variable: total_obs <int>
```

The `summarise` function can be used to produce grouped summaries. This is done by first grouping the data with the `group_by` function. For example, if we wished to produce a summary table with a row per local authority, summarising the total spending between 2015 and 2020, we would use the following:

```r
csp_long2 %>%
  # Group by the local authority's unique identifiers
  group_by(ons_code, authority) %>%
  # Total spend 2015 - 2020
  summarise(total_spend_all = sum(total_spend),
            # Mean spend 2015 - 2020
            mean_total_spend = mean(total_spend),
            # Median spend 2015 - 2020
            median_total_spend = median(total_spend),
            # 10th percentile of total spend
            quantile10_total_spend = quantile(total_spend, 0.1),
            # Number of rows summarised over
            total_obs = n()) %>%
  # Remove grouping structure
  ungroup()
```

```
## # A tibble: 396 x 7
##    ons_code  authority       total_spend_all mean_total_spend median_total_spend
##    <chr>     <chr>                     <dbl>            <dbl>              <dbl>
##  1 -         Greater London~          12416.           2069.              2022.
##  2 E06000001 Hartlepool                 485.             80.8               80.3
##  3 E06000002 Middlesbrough              711.            119.               118.
##  4 E06000003 Redcar And Cle~            660.            110.               109.
##  5 E06000004 Stockton-on-Te~            832.            139.               138.
##  6 E06000005 Darlington                 474.             79.0               78.5
##  7 E06000006 Halton                     598.             99.7               99.0
##  8 E06000007 Warrington                 806.            134.               133.
##  9 E06000008 Blackburn with~            692.            115.               115.
## 10 E06000009 Blackpool                  750.            125.               124.
## # i 386 more rows
## # i 2 more variables: quantile10_total_spend <dbl>, total_obs <int>
```

WARNING: Whenever using `group_by`, make sure to `ungroup` the data before proceeding. The grouping structure can be large and slow analysis down, or may interact with other functions to produce unexpected analyses.

Exercise 4

1. Create a data frame with the minimum, maximum and median total spend per year for each region.
2. Produce a frequency table containing the number and percentage of local authorities in each region.
3. Convert the data object `csp_long2` back into wide format, with one row per local authority and one variable per total spend per year (**HINT:** start by selecting only the variables you need from the long data frame). Use the help file `?pivot_wider` and `vignette("pivot")` for more hints.
4. Using your new wide data frame, calculate the difference in total spending for each local authority between 2015 and 2020. How many local authorities have had an overall reduction in spending since 2015?