Tidy Data & Iteration

Thrisha Rajkumar

2024-12-26

Load Packages

```
library(palmerpenguins)
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
penguins
```

```
## # A tibble: 344 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.1          18.7               181        3750
##  2 Adelie  Torgersen           39.5          17.4               186        3800
##  3 Adelie  Torgersen           40.3          18                 195        3250
##  4 Adelie  Torgersen           NA            NA                  NA          NA
##  5 Adelie  Torgersen           36.7          19.3               193        3450
##  6 Adelie  Torgersen           39.3          20.6               190        3650
##  7 Adelie  Torgersen           38.9          17.8               181        3625
##  8 Adelie  Torgersen           39.2          19.6               195        4675
##  9 Adelie  Torgersen           34.1          18.1               193        3475
## 10 Adelie  Torgersen           42            20.2               190        4250
## # i 334 more rows
## # i 2 more variables: sex <fct>, year <int>
```

the example will be based on the package tidyr

1. Reshaping data (Narrow data & wide data)

Create summary data of the penguins dataset (group by species):

```
penguins_summary <-  penguins %>%
  group_by(species) %>%
  summarise(bill=round(mean(bill_length_mm, na.rm=TRUE),digits=1), flipper=round(mean(flipper_length_mm
print(penguins_summary)
```

```
## # A tibble: 3 x 4
##   species    bill flipper weight
##   <fct>     <dbl>   <dbl>  <dbl>
## 1 Adelie     38.8    190   3701.
## 2 Chinstrap  48.8    196.  3733.
## 3 Gentoo     47.5    217.  5076
```

Create a data frame in a narrow format

```
penguins_summary_narrow <- penguins_summary %>%
  pivot_longer(c(bill, flipper, weight), names_to = 'property', values_to = 'value')
print(penguins_summary_narrow)
```

```
## # A tibble: 9 x 3
##   species   property  value
##   <fct>     <chr>     <dbl>
## 1 Adelie    bill       38.8
## 2 Adelie    flipper   190
## 3 Adelie    weight   3701.
## 4 Chinstrap bill       48.8
## 5 Chinstrap flipper   196.
## 6 Chinstrap weight   3733.
## 7 Gentoo    bill       47.5
## 8 Gentoo    flipper   217.
## 9 Gentoo    weight   5076
```

Suppose that we are given the above narrow data. We want to reshape the narrow data to wide data, using the function pivot_wider:

```
penguins_summary_wide <- penguins_summary_narrow %>%
  pivot_wider(names_from = property, values_from = value)
print(penguins_summary_wide)
```

```
## # A tibble: 3 x 4
##   species    bill flipper weight
##   <fct>     <dbl>   <dbl>  <dbl>
## 1 Adelie     38.8    190   3701.
## 2 Chinstrap  48.8    196.  3733.
## 3 Gentoo     47.5    217.  5076
```

Similarly, to get the narrow data from the wide data, we can use the pivot_longer

```
penguins_summary_wide %>%
  pivot_longer(c(bill, flipper, weight), names_to='property', values_to='value')
```

```
## # A tibble: 9 x 3
##   species   property  value
##   <fct>     <chr>     <dbl>
## 1 Adelie    bill       38.8
## 2 Adelie    flipper   190
## 3 Adelie    weight   3701.
## 4 Chinstrap bill       48.8
## 5 Chinstrap flipper   196.
## 6 Chinstrap weight   3733.
## 7 Gentoo    bill       47.5
## 8 Gentoo    flipper   217.
## 9 Gentoo    weight   5076
```

Note that we have used "c(bill, flipper, weight)" to specify the list of columns that we want to 'fold' into the property column. This is equivalent to excluding the species column from the list. So we can also write

```
penguins_summary_wide %>%
  pivot_longer(cols = !species, names_to='property', values_to='value')
```

```
## # A tibble: 9 x 3
##   species   property  value
##   <fct>     <chr>     <dbl>
## 1 Adelie    bill       38.8
## 2 Adelie    flipper   190
## 3 Adelie    weight   3701.
## 4 Chinstrap bill       48.8
## 5 Chinstrap flipper   196.
## 6 Chinstrap weight   3733.
## 7 Gentoo    bill       47.5
## 8 Gentoo    flipper   217.
## 9 Gentoo    weight   5076
```

The result is the same as before.

2. Uniting and separating data

Using penguins data again

```
print(penguins_summary)
```

```
## # A tibble: 3 x 4
##   species    bill flipper weight
##   <fct>     <dbl>   <dbl>  <dbl>
## 1 Adelie     38.8    190   3701.
## 2 Chinstrap  48.8    196.  3733.
## 3 Gentoo     47.5    217.  5076
```

First, we will combine the flipper and weight columns into a single column called flipper_over_weight, by putting their values together with a separator character "/" between them. This is done via the function unite.

```r
uni_df <- penguins_summary %>%
  unite(flipper_over_weight, flipper, weight, sep = "/")
print(uni_df)
```

```
## # A tibble: 3 x 3
##   species    bill flipper_over_weight
##   <fct>     <dbl> <chr>
## 1 Adelie     38.8 190/3700.7
## 2 Chinstrap  48.8 195.8/3733.1
## 3 Gentoo     47.5 217.2/5076
```

Second, we can reverse this process, by splitting a column into two, i.e., separating the numbers with a separator character "/" between them. This is done via the function "separate".

```r
sep_df <- uni_df %>%
  separate(flipper_over_weight, into=c("flipper", "weight"), sep="/")
print(sep_df)
```

```
## # A tibble: 3 x 4
##   species    bill flipper weight
##   <fct>     <dbl> <chr>   <chr>
## 1 Adelie     38.8 190     3700.7
## 2 Chinstrap  48.8 195.8   3733.1
## 3 Gentoo     47.5 217.2   5076
```

The obtained data frame sep_df is different from penguins_summary. The data type of the columns flipper and weight are characters, while that of the penguins_summary are numeric. This is because, by default, the separate function preserves the type of column (flipper_over_weight is a character column).

```r
mode(sep_df$weight)
mode(penguins_summary$weight)
```

```
## [1] "character"
## [1] "numeric"
```

separate function to convert the column types to numeric by using "convert = TRUE", for example:

```r
sep_df_double <- uni_df %>%
  separate(flipper_over_weight, into=c("flipper", "weight"), sep="/", convert = TRUE)
print(sep_df_double)
```

```
## # A tibble: 3 x 4
##   species    bill flipper weight
##   <fct>     <dbl>   <dbl>  <dbl>
## 1 Adelie     38.8    190   3701.
## 2 Chinstrap  48.8    196.  3733.
## 3 Gentoo     47.5    217.  5076
```

3. Nesting and unnesting

Suppose that we are given a data frame called musicians, which contains information about the band of each musician and the instrument that they play.

```
musicians <- full_join(band_members, band_instruments)
```

```
## Joining with `by = join_by(name)`
```

```
print(musicians)
```

```
## # A tibble: 4 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
## 4 Keith <NA>    guitar
```

We want to convert each group in the data frame into a data frame. This can be done by nest(). The nest function outputs a data frame, each row of which contains a tibble (a special type of data frame) corresponding to an individual group. So the number of rows is equal to the number of groups in the original data frame. Nest is often used together with the group_by function.

```
musicians_nest <- musicians %>%
  group_by(name) %>%
  nest()
print(musicians_nest)
```

```
## # A tibble: 4 x 2
## # Groups:   name [4]
##   name  data
##   <chr> <list>
## 1 Mick  <tibble [1 x 2]>
## 2 John  <tibble [1 x 2]>
## 3 Paul  <tibble [1 x 2]>
## 4 Keith <tibble [1 x 2]>
```

```
print(filter(musicians_nest,name=='Mick')$data) # the data of Mick is a data frame
```

```
## [[1]]
## # A tibble: 1 x 2
##   band   plays
##   <chr>  <chr>
## 1 Stones <NA>
```

By default, the tibbles associated with the group are contained in a column called “data”. The type of this column is list, hence it is called a list-column.

We can undo the nest operations, i.e., flattening a data frame in a nested form into regular columns, by using the unnest function:

```r
musicians_nest %>%
  unnest(cols = data)
```

```
## # A tibble: 4 x 3
## # Groups:   name [4]
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
## 4 Keith <NA>    guitar
```

Note that here we specify the column to be unnested. In this example, this is the 'data' column we created through the nest function.

4. Iteration based on the map function

The map function in R can be used to implement iterations. Particularly, the map function transforms its input by applying a function to each element of a list or atomic vector and returning an object of the same length as the input. Let's understand this using an example:

```r
is_div_2_3 <- function(x){
  if(x%%2 ==0 | x%%3 ==0){
    return (TRUE)
  } else {
    return(FALSE)
  }
}
v <- c(1,2,3,5,6)
map(v, is_div_2_3)
```

```
## [[1]]
## [1] FALSE
##
## [[2]]
## [1] TRUE
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] FALSE
##
## [[5]]
## [1] TRUE
```

Note that the function map returns a list. There are several variants of the map function that return a vector of a specific type, such as map_lgl() returns booleans, map_int() return integers, map_dbl() returns double and map_chr() return strings. E.g., type ?map_int to see more.

```
map_int(v, is_div_2_3)
```

```
## [1] 0 1 1 0 1
```

5. Example: Finding variables of maximal correlation

Now let's consider an example. Suppose that we want to create a function that

1) Takes as input a data frame and a variable name (column name)
2) Computes the correlation with all other numeric variables
3) Returns the name of the variable with maximal absolute correlation, and the corresponding correlation.

Recall that the correlation between vector x and y is defined as (Pearson formula):

$$\frac{\sum_1^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2}\sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

In R, the correlation can be computed using the function cor (type ?cor for more details)

We will start with a script in a specific case where the dataset penguins is used and suppose the variable name is 'body_mass_g'.

```
col_name <- 'body_mass_g'
df = penguins

v_col <- select(df, all_of(col_name)) # extract column based on col_name
df_num <- select_if(df, is.numeric) %>% select(-all_of(col_name)) # select all numeric columns excludin

cor_func <- function(x){ cor(x, v_col, use='complete.obs') } # a function that computes cor between v_c
correlations <- unlist(map(df_num, cor_func)) # compute correlations with all other numeric columns (wi
print('the computed correlations are:'); print(correlations)

max_abs_cor_var <- names( which( abs(correlations)==max(abs(correlations))  ) ) # extract the name of t
cor_val <- as.double(correlations[max_abs_cor_var])
print('\ncolumn with maximal correlation:' ); print(max_abs_cor_var)
```

```
## [1] "the computed correlations are:"
##    bill_length_mm     bill_depth_mm flipper_length_mm             year
##        0.59510982       -0.47191562        0.87120177        0.04220939
## [1] "\ncolumn with maximal correlation:"
## [1] "flipper_length_mm"
```

Here we used map to apply the function cor_func on each element (column) of df_num.

We then convert the above script into a function max_cor_var:

```
max_cor_var <- function(df, col_name){

  v_col <- select(df, all_of(col_name)) # extract column based on col_name
  df_num <- select_if(df, is.numeric) %>% select(-all_of(col_name)) # select all numeric columns exclud

  cor_func <- function(x){ cor(x, v_col, use='complete.obs') } # a function that computes cor between v
  correlations <- unlist(map(df_num, cor_func)) # compute correlations with all other numeric columns (

  max_abs_cor_var <- names( which( abs(correlations)==max(abs(correlations))  ) ) # extract the name of
  cor_val <- as.double(correlations[max_abs_cor_var])

  return (data.frame(var_name=max_abs_cor_var, cor=cor_val)) # return as a data frame
}

max_cor_var(penguins, "body_mass_g")
```

```
##            var_name       cor
## 1 flipper_length_mm 0.8712018
```

We can also perform the above analysis on individual groups of the data frame, with the help of the nest and unnest functions:

```
cor_by_group <- penguins %>%
  group_by(species) %>%
  nest() %>%
  mutate(max_cor=map(data, function(x){max_cor_var(x, 'body_mass_g')}))

print(cor_by_group)
```

```
## # A tibble: 3 x 3
## # Groups:   species [3]
##   species   data              max_cor
##   <fct>     <list>            <list>
## 1 Adelie    <tibble [152 x 7]> <df [1 x 2]>
## 2 Gentoo    <tibble [124 x 7]> <df [1 x 2]>
## 3 Chinstrap <tibble [68 x 7]>  <df [1 x 2]>
```

```
select(cor_by_group, -data) %>%
  unnest(cols=max_cor)
```

```
## # A tibble: 3 x 3
## # Groups:   species [3]
##   species   var_name            cor
##   <fct>     <chr>             <dbl>
## 1 Adelie    bill_depth_mm     0.576
## 2 Gentoo    bill_depth_mm     0.719
## 3 Chinstrap flipper_length_mm 0.642
```

Note that here we use nest() to create a data frame of nested variables, associated with the individual groups. This allows us to perform group-wise operations. In this example, the max_cor_var is applied to each group, represented as a data frame in the data column.

6. Missing data

Missing data is remarkably common in practical Data Science applications.

Consider for example the following data frame called stocks

```
stocks <- tibble(
  year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr    = c(   1,    2,    3,    4,    2,    3,    4),
  return = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)
print(stocks)
```

```
## # A tibble: 7 x 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4  NA
## 5  2016     2   0.92
## 6  2016     3   0.17
## 7  2016     4   2.66
```

We can see that the data stocks has missing values.

Explicit missing data: a NA value appears in the return column, representing a missing value.

Implicit missing data: Data about the first quarter of 2016 is missing, i.e., the whole row does not appear.

To make the implicit missing data explicit, we can insert rows that include NA values. We use the complete function to do this:

```
complete(stocks, year, qtr)
```

```
## # A tibble: 8 x 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4  NA
## 5  2016     1  NA
## 6  2016     2   0.92
## 7  2016     3   0.17
## 8  2016     4   2.66
```

Now, a row is created for the first quarter of 2016, where the return column is filled with NA.

We can find the row with missing values using the function complete.cases, which returns a logical vector indicating which cases are complete.

```
complete.cases(stocks)
```

```
## [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

With the complete case analysis, we can remove the incomplete cases, where there are missing values

```
filter(stocks, complete.cases(stocks))
```

```
## # A tibble: 6 x 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2016     2   0.92
## 5  2016     3   0.17
## 6  2016     4   2.66
```

Now the NA value has been removed from the table.

In some cases, we might want to replace the missing values with some numbers, instead of deleting them.

For example, we can replace them with the mean of the column.

```
replace_by_mean <- function(x){
  mu <- mean(x, na.rm=TRUE) # first compute the mean of x

  impute_f <- function(z){ # imputation on a single element z
    if (is.na(z)){
      return (mu)
    } else {
      return (z)
    }
  }
  return (map_dbl(x, impute_f)) # apply the function to impute across the whole vector x
}

x <- c(1,2,NA,4)
replace_by_mean(x)
```

```
## [1] 1.000000 2.000000 2.333333 4.000000
```

The third element NA is replaced with the mean of the vector.

```
mutate(stocks, return=replace_by_mean(return))
```

```
## # A tibble: 7 x 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4   1.10
## 5  2016     2   0.92
## 6  2016     3   0.17
## 7  2016     4   2.66
```

Here NA on the 4th row has been replaced by the mean value.