dplyr 1.0.0: working within rows

Photo by Oleksandr Hrebelnyk

2020/04/10

dplyr, dplyr-1-0-0

Hadley Wickham

Today, I wanted to talk a little bit about the renewed `rowwise()` function that makes it easy to perform operations "row-by-row". I'll show how you can use `rowwise()` to compute summaries "by row", talk about how `rowwise()` is a natural pairing with list-columns, and show a couple of use cases that I think are particularly elegant. You can learn more about all of these topics in `vignette("rowwise")`.

Update: as of June 1, dplyr 1.0.0 is now available on CRAN! Read all about it or install it now with `install.packages("dplyr")`.

Basic operation

Contents

verbs operate on the data. Let's see how this works with a simple example. Here I have some imaginary test results for students in a class:

```
library(dplyr, warn.conflicts = FALSE)

df <- tibble(
  student_id = 1:4,
  test1 = 10:13,
  test2 = 20:23,
  test3 = 30:33,
  test4 = 40:43
)
df
#> # A tibble: 4 x 5
#>   student_id test1 test2 test3 test4
#>        <int> <int> <int> <int> <int>
#> 1          1    10    20    30    40
#> 2          2    11    21    31    41
#> 3          3    12    22    32    42
#> 4          4    13    23    33    43
```

I'd like to be able to compute the mean of the test scores for each student, but `mutate()` and `mean()` don't do what I want:

```
df %>% mutate(avg = mean(c(test1, test2, test3,
test4)))
#> # A tibble: 4 x 6
#>   student_id test1 test2 test3 test4   avg
#>        <int> <int> <int> <int> <int> <dbl>
#> 1          1    10    20    30    40  26.5
#> 2          2    11    21    31    41  26.5
#> 3          3    12    22    32    42  26.5
#> 4          4    13    23    33    43  26.5
```

The problem is that I'm getting a mean over the whole data frame, not for each student. I can resolve this problem of getting a mean for each student by creating a "row-wise" data frame with `rowwise()`:

`rowwise()` doesn't need any additional arguments unless you have variables that identify the rows, like `student_id` here. Much like grouping variables, identifier variables will be automatically preserved when you `summarise()` the data.

```
rf
#> # A tibble: 4 x 5
#> # Rowwise:  student_id
#>   student_id test1 test2 test3 test4
#>        <int> <int> <int> <int> <int>
#> 1          1    10    20    30    40
#> 2          2    11    21    31    41
#> 3          3    12    22    32    42
#> 4          4    13    23    33    43
```

`rf` *looks* very similar to `df`, but *behaves* very differently:

```
rf %>% mutate(avg = mean(c(test1, test2, test3,
test4)))
#> # A tibble: 4 x 6
#> # Rowwise:  student_id
#>   student_id test1 test2 test3 test4   avg
#>        <int> <int> <int> <int> <int> <dbl>
#> 1          1    10    20    30    40    25
#> 2          2    11    21    31    41    26
#> 3          3    12    22    32    42    27
#> 4          4    13    23    33    43    28
```

An additional advantage of `rowwise()` is that it's paired with `c_across()`, which works like `c()` but uses the same tidyselect syntax as `across()`. That makes it easy to operate on multiple variables:

```
rf %>% mutate(avg =
mean(c_across(starts_with("test"))))
#> # A tibble: 4 x 6
#> # Rowwise:  student_id
#>   student_id test1 test2 test3 test4   avg
#>        <int> <int> <int> <int> <int> <dbl>
#> 1          1    10    20    30    40    25
#> 2          2    11    21    31    41    26
```

Other ways of achieving the same result

Some summary functions have alternative ways of computing row-wise summaries that take advantage of built-in vectorisation. For example, if you wanted to compute the sum, you could use `+`:

```
df %>% mutate(total = test1 + test2 + test3 + test4)
#> # A tibble: 4 x 6
#>   student_id test1 test2 test3 test4 total
#>        <int> <int> <int> <int> <int> <int>
#> 1          1    10    20    30    40   100
#> 2          2    11    21    31    41   104
#> 3          3    12    22    32    42   108
#> 4          4    13    23    33    43   112
```

And you *could* use the same basic idea to compute the mean:

```
df %>% mutate(avg = (test1 + test2 + test3 + test4)
/ 4)
#> # A tibble: 4 x 6
#>   student_id test1 test2 test3 test4   avg
#>        <int> <int> <int> <int> <int> <dbl>
#> 1          1    10    20    30    40    25
#> 2          2    11    21    31    41    26
#> 3          3    12    22    32    42    27
#> 4          4    13    23    33    43    28
```

Another family of summary functions have "parallel" extensions where you can provide multiple variables in the arguments:

```
df %>% mutate(
  min = pmin(test1, test2, test3, test4),
  max = pmax(test1, test2, test3, test4),
  string = paste(test1, test2, test3, test4, sep =
"-")
)
#> # A tibble: 4 x 8
#>   student_id test1 test2 test3 test4   min   max
```

```
#> 1          1    10    20    30    40    10    40
10-20-30-40
#> 2          2    11    21    31    41    11    41
11-21-31-41
#> 3          3    12    22    32    42    12    42
12-22-32-42
#> 4          4    13    23    33    43    13    43
13-23-33-43
```

Where these functions exist, they'll usually be faster than `rowwise()`. The advantage of `rowwise()` is that it works with any function, not just those that are already vectorised.

List-columns

`rowwise()` is useful for computing simple summaries, but its real power comes when you use it with list-columns. Because lists can contain anything, you can use list-columns to keep related objects together, regardless of what type of thing they are. List-columns give you a convenient storage mechanism and `rowwise()` gives you a convenient computation mechanism.

Let's make those ideas concrete by creating a data frame with a list-column. A little later, we'll come back to how you might actually get a list-column in a more realistic situation. The following data frame uses list columns to store things that would otherwise be challenging:

- `x` contains vectors of different lengths.
- `y` contains vectors of different types
- `z` contains functions, which can't usually live in a data frame.

```
df <- tibble(
  x = list(1, 2:3, 4:6),
  y = list(TRUE, 1, "a"),
```

```
#> # A tibble: 3 x 3
#>   x         y         z
#>   <list>    <list>    <list>
#> 1 <dbl [1]> <lgl [1]> <fn>
#> 2 <int [2]> <dbl [1]> <fn>
#> 3 <int [3]> <chr [1]> <fn>
```

When you have list-columns in a row-wise data frame, you can easily compute with each element of the list:

```
df %>%
  rowwise() %>%
  summarise(
    x_length = length(x),
    y_type = typeof(y),
    z_call = z(1:5)
  )
#> `summarise()` ungrouping output (override with
`.groups` argument)
#> # A tibble: 3 x 3
#>   x_length y_type    z_call
#>      <int> <chr>      <dbl>
#> 1        1 logical    15
#> 2        2 double      3
#> 3        3 character   1.58
```

This makes a row-wise `mutate()` or `summarise()` a general vectorisation tool, in the same way as the apply family in base R or the map family in purrr do. It's now much simpler to solve a number of problems where we previously recommended learning about `map()`, `map2()`, `pmap()` and friends.

Use cases

To finish up, I wanted to show off a couple of use cases where I think `rowwise()` provides a really elegant solution: simulations and modelling.

Simulation

The basic idea of using `rowwise()` to perform simulation is to store all your simulation paramters in a data frame:

```
df <- tribble(
  ~id, ~ n, ~ min, ~ max,
    1,   3,     0,     1,
    2,   2,    10,   100,
    3,   2,   100,  1000,
)
```

Then you can either generate a list-column containing the simulated values with `mutate()`:

```
df %>%
  rowwise(id) %>%
  mutate(data = list(runif(n, min, max)))
#> # A tibble: 3 x 5
#> # Rowwise:  id
#>      id     n   min   max data
#>   <dbl> <dbl> <dbl> <dbl> <list>
#> 1     1     3     0     1 <dbl [3]>
#> 2     2     2    10   100 <dbl [2]>
#> 3     3     2   100  1000 <dbl [2]>
```

Or take advantage of `summarise()`'s new capabilities and return one element per row:

```
df %>%
  rowwise(id) %>%
  summarise(x = runif(n, min, max))
#> `summarise()` regrouping output by 'id' (override
with `.groups` argument)
#> # A tibble: 7 x 2
#> # Groups:   id [3]
#>      id       x
#>   <dbl>   <dbl>
#> 1     1   0.579
```

```
#> 5     2  50.8
#> 6     3 451.
#> 7     3 985.
```

Note that `id` is preserved in the output here because we defined it as an identifier variable in the call to `rowwise()`.

`vignette("rowwise")` expands on this idea to show how you can generate parameter grids and vary the random distribution used in each row.

Group-wise models

The new `nest_by()` function works similarly to `group_by()` but instead of storing the grouping data as metadata, visibly changes the structure. Now we have three rows (one for each group), and we have a list-col, `data`, that stores the data for that group. Also note that the output is a `rowwise()` object; this is important because it's going to make working with that list of data frames much easier.

```
by_cyl <- mtcars %>% nest_by(cyl)
by_cyl
#> # A tibble: 3 x 2
#> # Rowwise:  cyl
#>     cyl                data
#>   <dbl> <list<tbl_df[,10]>>
#> 1     4           [11 × 10]
#> 2     6            [7 × 10]
#> 3     8           [14 × 10]
```

Now we can use `mutate()` to fit a model to each data frame:

```
by_cyl <- by_cyl %>% mutate(model = list(lm(mpg ~
wt, data = data)))
by_cyl
#> # A tibble: 3 x 3
```

```
#> 1     4           [11 × 10] <lm>
#> 2     6            [7 × 10] <lm>
#> 3     8           [14 × 10] <lm>
```

(Note that we need to wrap the output of `lm()` into a list; if you forget this, the error message will remind you.)

And then extract model summaries or coefficients with `summarise()` and broom functions:

```
by_cyl %>% summarise(broom::glance(model))
#> `summarise()` regrouping output by 'cyl'
(override with `.groups` argument)
#> # A tibble: 3 x 12
#> # Groups:   cyl [3]
#>     cyl r.squared adj.r.squared sigma statistic
p.value    df logLik   AIC   BIC
#>   <dbl>     <dbl>         <dbl> <dbl>     <dbl>
<dbl> <int>  <dbl> <dbl> <dbl>
#> 1     4     0.509         0.454  3.33      9.32
0.0137     2 -27.7   61.5  62.7
#> 2     6     0.465         0.357  1.17      4.34
0.0918     2  -9.83  25.7  25.5
#> 3     8     0.423         0.375  2.02      8.80
0.0118     2 -28.7   63.3  65.2
#> # … with 2 more variables: deviance <dbl>,
df.residual <int>

by_cyl %>% summarise(broom::tidy(model))
#> `summarise()` regrouping output by 'cyl'
(override with `.groups` argument)
#> # A tibble: 6 x 6
#> # Groups:   cyl [3]
#>     cyl term        estimate std.error statistic
p.value
#>   <dbl> <chr>          <dbl>     <dbl>     <dbl>
<dbl>
#> 1     4 (Intercept)    39.6      4.35       9.10
0.00000777
#> 2     4 wt             -5.65     1.85      -3.05
0.0137
#> 3     6 (Intercept)    28.4      4.18       6.79
```

```
#> 5     8 (Intercept)    23.9      3.01       7.94
0.00000405
#> 6     8 wt             -2.19     0.739     -2.97
0.0118
```