13 How to do rowwise calculations

What will this tutorial cover?

In this tutorial you will learn how to compute on a data frame per row. You will be able to calculate summary statistics one row at a time with `rowwise` and some other vectorized functions that are faster than `rowwise`.

Who do I have to thank?

I have to thank Jeffrey Girard for his excellent tutorial on row-wise means in dplyr. He provided some excellent examples and ideas that I built this tutorial on. Also, I want to thank the guys on the tidyverse team who wrote a really helpful vignette on row-wise operations in dplyr.

Some of the simplest calculations in R can be confusing at times. Take, for example, this: You want to calculate the mean for each row of your data frame. Suppose this data frame represents the test scores of four students at three measurement times: first, second, and third.

```r
(dframe <- tibble(
  name = c("Jakob", "Eliud", "Jamal", "Emily"),
  first = c(1, 4, 7, 10),
  second = c(2, 5, 8,11),
  third = c(3, 6, 9, 12)
))
```

```
# A tibble: 4 x 4
  name  first second third
  <chr> <dbl>  <dbl> <dbl>
1 Jakob     1      2     3
2 Eliud     4      5     6
3 Jamal     7      8     9
4 Emily    10     11    12
```

Calculating the mean for each person from these three measurement points should be straightforward:

```r
dframe %>%
  mutate(
    mean_performance = mean(c(first, second, third))
  )
```

```
# A tibble: 4 x 5
  name  first second third mean_performance
  <chr> <dbl>  <dbl> <dbl>            <dbl>
1 Jakob     1      2     3              6.5
2 Eliud     4      5     6              6.5
3 Jamal     7      8     9              6.5
4 Emily    10     11    12              6.5
```

Apparently not. `6.5` is certainly not the average value of one of these rows. What's going on here? It turns out that `mean` calculates the average of all 9 values, not three for each row. Similar to this:

```r
mean(c(c(1, 4, 7, 10),
       c(2, 5, 8,11),
       c(3, 6, 9, 12)))
```

```
[1] 6.5
```

We would say that `mean` is not a vectorized function because it does not perform its calculations vector by vector. Instead, it throws each vector into a box and calculates the overall average of all its values.

Sooner or later, most of us will stumble upon this problem. And this applies not only to `mean`, but also to `sum`, `min`, `max`, and `median`:

```r
dframe %>%
  mutate(
    mean_performance   = mean(c(first, second, third)),
    sum_performance    = sum(c(first, second, third)),
    min_performance    = min(c(first, second, third)),
    max_performance    = max(c(first, second, third)),
    median_performance = median(c(first, second, third))
  )
```

```
# A tibble: 4 x 9
  name  first second third mean_performance sum_perfor~1 min_p~2 max_p~3 media~4
  <chr> <dbl>  <dbl> <dbl>            <dbl>        <dbl>   <dbl>   <dbl>   <dbl>
1 Jakob     1      2     3              6.5           78       1      12     6.5
2 Eliud     4      5     6              6.5           78       1      12     6.5
3 Jamal     7      8     9              6.5           78       1      12     6.5
4 Emily    10     11    12              6.5           78       1      12     6.5
# ... with abbreviated variable names 1: sum_performance, 2: min_performance,
#   3: max_performance, 4: median_performance
```

So far, so good. What confuses many is the fact that some row-wise calculations yield unexpected results, while others do not. These three use cases, for example, work flawlessly.

```
dframe %>%
  mutate(
    combined = paste(first, second, third),
    sum      = first + second + third,
    mean     = (first + second + third) / 3
  )
```

```
# A tibble: 4 x 7
  name  first second third combined   sum  mean
  <chr> <dbl>  <dbl> <dbl> <chr>    <dbl> <dbl>
1 Jakob     1      2     3 1 2 3        6     2
2 Eliud     4      5     6 4 5 6       15     5
3 Jamal     7      8     9 7 8 9       24     8
4 Emily    10     11    12 10 11 12    33    11
```

So the problem occurs with functions that are not vectorized (e.g. `mean`) and with base R functions that compute summary statistics. There are several solutions to this problem. The first is `rowwise`:

13.1 Introducing `rowwise`

`rowwise` was introduced in dplyr in 2020 with version 1.1.0. Essentially, the function ensures that operations are performed row by row. It works similar to `group_by`. It does not change how the data frame looks, but how calculations are performed with the data frame. Let's apply the function to our toy data frame and see what results we get:

```r
dframe %>%
  rowwise()
```

```
# A tibble: 4 x 4
# Rowwise:
  name  first second third
  <chr> <dbl>  <dbl> <dbl>
1 Jakob     1      2     3
2 Eliud     4      5     6
3 Jamal     7      8     9
4 Emily    10     11    12
```

As you can see, nothing changes. The output just tells you that further calculations will be performed row by row.

Now, if we calculate the mean of the individual's performance, we get the correct results:

```r
dframe %>%
  rowwise() %>%
  mutate(
    mean_performance = mean(c(first, second, third))
  )
```

```
# A tibble: 4 x 5
# Rowwise:
  name  first second third mean_performance
  <chr> <dbl>  <dbl> <dbl>            <dbl>
1 Jakob     1      2     3                2
2 Eliud     4      5     6                5
3 Jamal     7      8     9                8
4 Emily    10     11    12               11
```

To illustrate that `rowwise` is just a special case of `group_by`, we use the `group_by` function to obtain the same results:

```r
dframe %>%
  group_by(name) %>%
  mutate(
    mean = mean(c(first, second, third))
  )
```

```
# A tibble: 4 x 5
# Groups:   name [4]
  name  first second third  mean
  <chr> <dbl>  <dbl> <dbl> <dbl>
1 Jakob     1      2     3     2
2 Eliud     4      5     6     5
3 Jamal     7      8     9     8
4 Emily    10     11    12    11
```

The same logic applies to all other base R functions that compute summary statistics:

```
dframe %>%
  rowwise() %>%
  mutate(
    mean_performance   = mean(c(first, second, third)),
    sum_performance    = sum(c(first, second, third)),
    min_performance    = min(c(first, second, third)),
    max_performance    = max(c(first, second, third)),
    median_performance = median(c(first, second, third))
  )
```

```
# A tibble: 4 x 9
# Rowwise: 
  name  first second third mean_performance sum_perfor~1 min_p~2 max_p~3 media~4
  <chr> <dbl>  <dbl> <dbl>            <dbl>        <dbl>   <dbl>   <dbl>   <dbl>
1 Jakob     1      2     3                2            6       1       3       2
2 Eliud     4      5     6                5           15       4       6       5
3 Jamal     7      8     9                8           24       7       9       8
4 Emily    10     11    12               11           33      10      12      11
# ... with abbreviated variable names 1: sum_performance, 2: min_performance,
#   3: max_performance, 4: median_performance
```

13.2 How to use tidyselect functions with rowwise

It turns out that it is not possible to add a tidyselect function within `mean` or any of the other functions:

```
dframe %>%
  rowwise() %>%
  mutate(
    mean_performance = mean(where(is.numeric))
  )
```

```
Error in `mutate()`:
! Problem while computing `mean_performance = mean(where(is.numeric))`.
  The error occurred in row 1.
Caused by error in `where()`:
! could not find function "where"
Run `rlang::last_error()` to see where the error occurred.
```

In these cases `c_across` comes to your rescue. It was developed especially for `rowwise` and can be considered a wrapper around `c()`. Let's try it first without a tidyselect function:

```
dframe %>%
  rowwise() %>%
  mutate(
    mean_performance = mean(c_across(c(first, second, third)))
  )
```

```
# A tibble: 4 x 5
# Rowwise:
  name  first second third mean_performance
  <chr> <dbl>  <dbl> <dbl>            <dbl>
1 Jakob     1      2     3                2
2 Eliud     4      5     6                5
3 Jamal     7      8     9                8
4 Emily    10     11    12               11
```

The expected results. However, instead of `c()` you can use any tidyselect function of your choice:

```
dframe %>%
  rowwise() %>%
  mutate(
    mean_performance = mean(c_across(where(is.numeric)))
  )
```

```
# A tibble: 4 x 5
# Rowwise:
  name  first second third mean_performance
  <chr> <dbl>  <dbl> <dbl>            <dbl>
1 Jakob     1      2     3                2
2 Eliud     4      5     6                5
3 Jamal     7      8     9                8
4 Emily    10     11    12               11
```

This is especially important if you need to perform calculations on many columns. For example, the `billboard` data frame:

```r
billboard %>%
  rowwise() %>%
  transmute(
    artist,
    track,
    sum = sum(c_across(contains("wk")), na.rm = TRUE)
  )
```

```
# A tibble: 317 x 3
# Rowwise:
   artist         track                     sum
   <chr>          <chr>                   <dbl>
 1 2 Pac          Baby Don't Cry (Keep...   598
 2 2Ge+her        The Hardest Part Of ...   270
 3 3 Doors Down   Kryptonite               1403
 4 3 Doors Down   Loser                    1342
 5 504 Boyz       Wobble Wobble            1012
 6 98^0           Give Me Just One Nig...   753
 7 A*Teens        Dancing Queen             485
 8 Aaliyah        I Don't Wanna            1041
 9 Aaliyah        Try Again                 533
10 Adams, Yolanda Open My Heart            1355
# ... with 307 more rows
```

13.3 Don't forget to ungroup()

Not using `rowwise` can sometimes lead to problems, but using `rowwise` can also lead to problems. The most common mistake is forgetting to `ungroup` the data frame. I told you that `rowwise` does nothing but group the data so that the calculations are performed row by row. Similar to `group_by`, `rowwise` can also be ungrouped with `ungroup()`. If you don't do that, problems like this can occur:

```r
dframe %>%
  rowwise() %>%
  mutate(
    mean = mean(c(first, second, third))
  ) %>%
  summarise(
    mean_across_students = mean(mean)
  )
```

```
# A tibble: 4 x 1
  mean_across_students
                 <dbl>
1                    2
2                    5
3                    8
4                   11
```

Instead of calculating the mean for each person, the mean values are repeated for each student. The reason for this is that `summarise` still performs its computations row by row. To correct this logical error, we must `ungroup` the data frame:

```r
dframe %>%
  group_by(name) %>%
  mutate(
    mean = mean(c(first, second, third))
  ) %>%
  ungroup() %>%
  summarise(
    mean_across_students = mean(mean)
  )
```

```
# A tibble: 1 x 1
  mean_across_students
                 <dbl>
1                  6.5
```

So remember that `rowwise` should never join a party without `ungroup`.

13.4 Calculating proportions with `rowwise`

A nice use case of `rowwise` is to convert your values into proportions (I found the example in the official vignette). To calculate proportions, you must first calculate the sum of all values. Since the data is in a wide format, this can be done with `rowwise`:

```r
(sums_per_row <- dframe %>%
  rowwise() %>%
  mutate(sum_per_row = sum(first, second, third)) %>%
  ungroup())
```

```
# A tibble: 4 x 5
  name  first second third sum_per_row
  <chr> <dbl>  <dbl> <dbl>       <dbl>
1 Jakob     1      2     3           6
2 Eliud     4      5     6          15
3 Jamal     7      8     9          24
4 Emily    10     11    12          33
```

With the column `sum_per_row`, we can use `across` and convert all numeric columns to proportions:

```r
sums_per_row %>%
  transmute(
    name,
    across(
      .cols = where(is.numeric),
      .fns  = ~ . / sum_per_row
    )
  )
```

```
# A tibble: 4 x 5
  name  first second third sum_per_row
  <chr> <dbl>  <dbl> <dbl>       <dbl>
1 Jakob 0.167  0.333 0.5             1
2 Eliud 0.267  0.333 0.4             1
3 Jamal 0.292  0.333 0.375           1
4 Emily 0.303  0.333 0.364           1
```

13.5 If you care about performance, choose alternativ approaches

For small data sets `rowwise` is sufficient. However, if your data frame is very large, `rowwise` will show performance problems compared to alternative approaches. It is simply not very fast. Let's have a look:

To measure the performance, we will use the `bench` package:

```r
library(bench)
```

Let's see how long it takes to calculate the sums of our data frame with four rows and three columns:

```r
bench::mark(
  dframe %>%
    rowwise() %>%
    mutate(
      mean_performance = min(c(first, second, third))
    )
)$total_time
```

```
[1] 470ms
```

Your results will vary. On my computer it took about 450 milliseconds. What I haven't told you yet is that you can use other functions to calculate summary statistics on a row-by-row basis without using `rowwise`. `pmin` is one of those functions. Let's see how fast the code executes with `pmin`:

```r
bench::mark(
  dframe %>%
    mutate(
      mean_performance = pmin(first, second, third)
    )
)$total_time
```

```
[1] 470ms
```

It is slightly faster than with `rowwise`. However, the performance advantage comes when the data frame is very large. For example, the diamond dataset has 53,940 rows (not super large, of course, but an approximation). Let's compare the efficiency of the two approaches using this data frame:

```
bench::mark(
  diamonds %>%
    rowwise() %>%
    mutate(
      mean_performance = min(c(x, y, z))
    ) %>%
    ungroup(),
  diamonds %>%
    mutate(
      mean_performance = pmin(x, y, z)
    )
)$total_time
```

```
Warning: Some expressions had a GC in every iteration; so filtering is disabled.
```

```
[1] 533ms 500ms
```

A difference of more than 50 milliseconds on my computer. Next, we simply duplicate the diamond data set 10 times and compare the results again:

```
duplicated_diamonds <- bind_rows(
  diamonds, diamonds, diamonds, diamonds, diamonds,
  diamonds, diamonds, diamonds, diamonds, diamonds
)

bench::mark(
  duplicated_diamonds %>%
    rowwise() %>%
    mutate(
      mean_performance = min(c(x, y, z))
    ) %>%
    ungroup(),
  duplicated_diamonds %>%
    mutate(
      mean_performance = pmin(x, y, z)
    )
)$total_time
```

```
Warning: Some expressions had a GC in every iteration; so filtering is disabled.
```

```
[1]   1.6s  502.6ms
```

For half a million rows, `pmap` is three times faster than working with `rowwise`. So if you are working with really large data frames, keep in mind that `rowwise` is not the most efficient method. Consider the alternatives. Let's take a closer look at these alternatives at the end of this tutorial.

13.6 A short deep-dive into `pmax` and `pmin`

You have already seen `pmin`. Compared to `min`, `pmin` works because it is a vectorized function. It calculates the minimum value for one or more vectors. In our case, these vectors are rows. If you have `pmin`, you must also have `pmax`. With `pmax` you can calculate the maximum value for all columns on a row basis:

```
dframe %>%
  mutate(
    max = pmax(first, second, third),
    min = pmin(first, second, third)
  )
```

```
# A tibble: 4 x 6
  name  first second third   max   min
  <chr> <dbl>  <dbl> <dbl> <dbl> <dbl>
1 Jakob     1      2     3     3     1
2 Eliud     4      5     6     6     4
3 Jamal     7      8     9     9     7
4 Emily    10     11    12    12    10
```

Unfortunately there is no function `pmean`, `pmedian` or `psum`. However, we can calculate row-based sums with `rowSums`.

13.7 A deep-dive into `rowSums` and `rowMeans`

`rowSums` does what it says. It calculates the sums for rows. The function works with a matrix or a data frame. Providing a vector of values will not work, so we have to use this matrix trick to make it work:

```r
dframe %>%
  mutate(
    sum = rowSums(matrix(c(first, second, third), ncol = 3))
  )
```

```
# A tibble: 4 x 5
  name  first second third   sum
  <chr> <dbl>  <dbl> <dbl> <dbl>
1 Jakob     1      2     3     6
2 Eliud     4      5     6    15
3 Jamal     7      8     9    24
4 Emily    10     11    12    33
```

Similarly, we can use a subset of the data frame and put it into `rowSums`:

```r
rowSums(dframe %>% select(-name))
```

```
[1]  6 15 24 33
```

A trick to not use a matrix is to use `select` with `mutate` and the piped data frame `.`:

```r
dframe %>%
  mutate(
    sum1 = rowSums(select(., first, second, third)),
    sum2 = rowSums(across(first:third)),
    sum3 = rowSums(select(., matches("first|second|third"))),
  )
```

```
# A tibble: 4 x 7
  name  first second third  sum1  sum2  sum3
  <chr> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>
1 Jakob     1      2     3     6     6     6
2 Eliud     4      5     6    15    15    15
3 Jamal     7      8     9    24    24    24
4 Emily    10     11    12    33    33    33
```

Since `rowSums` can take a data frame, we can simply use a subset of our data frame as in `select(., first, second, third)` or `select(., matches("first|second|third"))`.

Similarly, we can calculate the mean of each row with `rowMeans`.

```r
dframe %>%
  mutate(
    sum1 = rowMeans(matrix(c(first, second, third), ncol = 3)),
    sum2 = rowMeans(across(first:third)),
    sum3 = rowMeans(select(., first, second, third)),
    sum4 = rowMeans(select(., matches("first|second|third")))
  )
```

```
# A tibble: 4 x 8
  name  first second third  sum1  sum2  sum3  sum4
  <chr> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Jakob     1      2     3     2     2     2     2
2 Eliud     4      5     6     5     5     5     5
3 Jamal     7      8     9     8     8     8     8
4 Emily    10     11    12    11    11    11    11
```

Summary

Here's what you can take away from this tutorial.

- Be careful to calculate summary statistics row-wise using base R functions. Since these functions are not vectorized, they will calculate the summary statistics with all values from your data frame.
- `rowwise` ensures that computations are done one row at a time.
- Never forget to `ungroup rowwise`. If you don't your calculations might be wrong.
- If you care about performance, use `pmin`, `pmap`, `rowSums` or `rowMeans` instead of `rowwise`.