How to Calculate the Mean of Multiple Columns in R Using dplyr

Authored by

stats writer

November 21, 2025

Analyzing large datasets often requires calculating summary statistics across specific variables or observations. When working with data frame objects in R, computing the arithmetic mean across multiple columns for each row can be a complex task if not approached efficiently. Fortunately, the dplyr package provides a robust and highly readable framework for tackling such challenges.

While previous versions of dplyr might have relied on functions like `summarise_at()`, modern dplyr idioms favor the combination of `rowwise()` and `c_across()` to perform calculations row by row, granting immense control and clarity over complex operations. This methodology allows data scientists to precisely specify which columns should be included in the calculation, ensuring accurate and rapid results returned within the existing data frame structure.

This guide will detail how to calculate the row-wise mean across selected columns using these powerful dplyr verbs, demonstrating a versatile technique essential for data manipulation and preparation in R. We will explore how to handle missing values and utilize helper functions to select columns dynamically.

The Modern Approach: Leveraging rowwise() and c_across()

To calculate the mean across several columns, we must first instruct dplyr to treat each row independently. This is achieved using the `rowwise()` function. Once the data structure is oriented for row-wise calculation, we use `mutate()` to create a new column, where the calculation itself is performed using `c_across()` combined with the standard R `mean()` function.

The core syntax for calculating a row-wise mean for specific columns within a data frame (here named `df`) is demonstrated below. This technique is highly versatile and is the recommended practice for modern data wrangling in R.

library(dplyr)

```
df %>%
rowwise() %>%
mutate(game_mean = mean(c_across(c('game1', 'game2', 'game3')), na.rm=TRUE))
```

This sequence of operations clearly defines the scope of the calculation. The `rowwise()` call sets the stage, ensuring that subsequent operations like `mutate()` are applied independently to each row. The resulting new column, named `game_mean`, stores the calculated mean for the designated set of variables: `game1`, `game2`, and `game3`. This specific approach ensures precision when dealing with subsets of columns.

Syntax Breakdown: Understanding the Core Functions

Understanding each component of the syntax is crucial for mastering this technique. The dplyr pipeline uses the pipe operator (`%>%`) to chain operations, making the sequence logical and easy to read. The process begins with the input data frame, which is then sequentially modified.

The function `rowwise()` is perhaps the most critical step here. By default, dplyr operations are vectorized and applied column-wise or group-wise. `rowwise()` changes the grouping structure of the data, treating every single row as its own group. This ensures that when `mean()` is applied in the next step, it calculates the average of values within that specific row, rather than attempting to find the average of an entire column.

Inside the `mutate()` call, which is responsible for creating or updating columns, we encounter `c_across()`. This function is designed to work exclusively within `rowwise()` operations. Its purpose is to select a set of columns and return their values for the current row as a vector. This vector is then passed directly to the `mean()` function, allowing the calculation to proceed smoothly across the specified variables. The combination of `rowwise()` and `c_across()` is exceptionally effective for these cross-column aggregations.

Practical Demonstration: Setting Up the Basketball Dataset

To illustrate this method concretely, let us establish a sample data frame representing performance statistics. This dataset contains the points scored by basketball players across different teams over three distinct games. Note that this dataset intentionally includes a missing value (`NA`) in the fifth row under `game2`, which we will address later when discussing robustness.

The creation of the sample data structure is straightforward using the base R `data.frame()` constructor. We define four variables: `team` (categorical identifier) and three quantitative variables (`game1`, `game2`, `game3`) which hold the numerical scores.

#create data frame
df <- data.frame(team=c('A', 'A', 'A', 'B', 'B', 'B', 'C', 'C'),
game1=c(10, 12, 17, 18, 24, 29, 29, 34),
game2=c(8, 10, 14, 15, NA, 19, 18, 29),
game3=c(4, 5, 5, 9, 12, 12, 18, 20))

#view data frame
df

team game1 game2 game3
1 A 10 8 4

2 A 12 10 5
3 A 17 14 5
4 B 18 15 9
5 B 24 NA 12
6 B 29 19 12
7 C 29 18 18
8 C 34 29 20

Our objective is to calculate the average points scored by each player across the three games. This requires calculating the mean horizontally (row-wise) only for the score columns (`game1`, `game2`, and `game3`), ignoring the categorical `team` column. This structure mirrors common scenarios in data analysis where summary statistics are required for specific metric variables.

Executing the Row-Wise Mean Calculation

By applying the established dplyr syntax to our basketball dataset, we successfully generate the desired aggregated column. The following code demonstrates the complete operation, resulting in a new column named `game_mean` appended to the original data frame structure.

library(dplyr)

#calculate mean value in each row for game1, game2 and game3 columns
df %>%
rowwise() %>%
mutate(game_mean = mean(c_across(c('game1', 'game2', 'game3')), na.rm=TRUE))

A tibble: 8 x 5
Rowwise:
team game1 game2 game3 game_mean

1 A 10 8 4 7.33
2 A 12 10 5 9
3 A 17 14 5 12
4 B 18 15 9 14
5 B 24 NA 12 18
6 B 29 19 12 20
7 C 29 18 18 21.7
8 C 34 29 20 27.7

The output clearly shows the new `game_mean` column, which holds the arithmetic mean for the

corresponding row, calculated exclusively from the scores in columns `game1` through `game3`. This result confirms that the `rowwise()` grouping successfully applied the `mean()` function to the vector created by `c_across()` for each observation.

We can verify the correctness of the output by manually calculating the average for a few initial rows. Notice how the calculation works precisely as intended, aggregating the individual game scores for the player associated with that row. This is powerful for assessing individual performance or longitudinal trends.

Mean value of row 1: (10 + 8 + 4) / 3 = **7.33**
Mean value of row 2: (12 + 10 + 5) / 3 = **9**
Mean value of row 3: (17 + 14 + 5) / 3 = **12**

Handling Missing Values with na.rm = TRUE

A critical aspect of robust data analysis is managing missing data, often represented by the `NA` value in R. In our basketball example, row 5 contains an `NA` score for `game2`. If we were to omit the `na.rm = TRUE` argument from the `mean()` function, the result for that row would simply be `NA`, as R cannot calculate the average of a set containing an unknown value.

The parameter `na.rm` stands for "NA Remove." Setting `na.rm = TRUE` instructs the `mean()` function to automatically exclude any missing values from the calculation. For the player in row 5 (scores 24, NA, 12), the calculation proceeds using only the available scores (24 and 12), dividing the sum by the count of non-missing values (2), thus resulting in a mean of 18.

This setting is essential when dealing with real-world data where incomplete observations are common. Always consider the impact of missing values on your descriptive statistics, and utilize `na.rm = TRUE` when calculating row-wise means unless the presence of an `NA` should explicitly propagate a missing result, signaling incomplete data for that observation.

Advanced Column Selection: Utilizing starts_with()

Specifying columns individually, as we did with `c('game1', 'game2', 'game3')`, works well for small, fixed datasets. However, in scenarios involving dozens or hundreds of similarly named variables, manually listing them becomes impractical and prone to errors. dplyr provides powerful selection helpers to overcome this limitation, allowing for dynamic selection based on column names.

One of the most useful selection helpers is `starts_with()`. When nested inside `c_across()`, this function automatically identifies and selects all columns whose names begin with a specified character string. In our example, since all the score variables start with 'game', we can use

`starts_with('game')` to select them without listing them explicitly.

library(dplyr)

#calculate mean value in each row for columns that start with 'game'
df %>%
rowwise() %>%
mutate(game_mean = mean(c_across(c(starts_with('game'))), na.rm=TRUE))

A tibble: 8 x 5
Rowwise:
team game1 game2 game3 game_mean

1 A 10 8 4 7.33
2 A 12 10 5 9
3 A 17 14 5 12
4 B 18 15 9 14
5 B 24 NA 12 18
6 B 29 19 12 20
7 C 29 18 18 21.7
8 C 34 29 20 27.7

As demonstrated by the identical output, using `starts_with()` achieves the exact same computational result as manually listing the columns. The benefit here is future-proofing: if we add `game4` or `game5` to the data frame later, this single line of code will automatically include them in the row-wise mean calculation, drastically reducing maintenance overhead for the analysis script.

Expanding Column Selection Capabilities

Beyond `starts_with()`, dplyr provides a suite of other selection helpers that are highly valuable when integrating with `c_across()`. These functions allow analysts to select columns based on various criteria, not just the beginning of the name.

Key selection helpers include:

`ends_with('pattern')`: Selects columns whose names end with the specified pattern. Useful if you have columns like `score_1`, `score_2`, and `penalty_score`, and only want the primary scores.
`contains('pattern')`: Selects columns whose names contain the specified substring anywhere. This is ideal for broad matching, for instance, selecting all columns that include the word 'score'.
`matches('regex')`: Allows selection using regular expressions, offering the highest level of flexibility for complex naming conventions.

`where(is.numeric)`: Selects columns based on their data type. This is incredibly powerful if you want to calculate the row mean across all numerical columns in your data frame, regardless of their specific names.

By using these functions, data manipulation scripts become highly adaptable. If the underlying data structure changes slightly (e.g., column order changes, new non-metric columns are added), the dplyr code remains stable and functional, ensuring reliable data processing workflows.

Contextualizing the Approach: Modern vs. Legacy Methods

While the combination of `rowwise()` and `c_across()` represents the contemporary best practice in dplyr for row-wise calculations, it is helpful to understand the context of previous methods.

Historically, row-wise operations in R often relied on base functions like `rowMeans()` or `apply(df, 1, mean)`. While functional, these methods integrate less seamlessly into the pipe-based workflow of dplyr and can be less readable, especially when selecting complex subsets of columns or needing to retain the tibble structure.

Within dplyr itself, functions like `summarise_at()`, `mutate_all()`, and `mutate_if()` were commonly used for column-wise operations and could sometimes be creatively adapted for row-wise tasks. However, these "scoped" variants (ending in `_at`, `_all`, `_if`) have been largely superseded by the new selection syntax using `across()` (for column summaries) and `c_across()` (for row calculations). The modern approach is preferred because it offers a more unified and consistent syntax across all dplyr operations.

Summary of dplyr's Flexibility in Data Aggregation

Calculating the mean across multiple specific columns is a frequent requirement in data analysis. By mastering the sequence of `rowwise()`, `mutate()`, and `c_across()`, R users gain an exceptionally clear and efficient way to perform these row-wise aggregations within the powerful dplyr framework.

This method ensures that calculations are performed accurately, even in the presence of missing values, thanks to the necessary inclusion of `na.rm = TRUE`. Furthermore, the ability to integrate advanced selection helpers like `starts_with()` allows scripts to remain concise, maintainable, and highly robust against evolving dataset structures.

Ultimately, employing these modern dplyr techniques transforms what could be a cumbersome task into a clean, single pipeline operation, significantly enhancing data preparation efficiency and reproducibility.