Learning to Summarize Multiple Columns with dplyr in R

Authored by

Mohammed looti

October 29, 2025

In the realm of data analysis, the ability to efficiently summarize large datasets is not merely a convenience--it is a fundamental requirement. Whether the goal is to uncover initial patterns during exploratory analysis, prepare clean features for machine learning models, or generate concise, aggregated reports, condensing information into meaningful statistics is paramount. When dealing with tabular data within the R environment, the **dplyr** package, an integral part of the Tidyverse, stands out as the definitive tool, offering a consistent and highly readable syntax for complex data manipulation tasks.

This comprehensive guide focuses on one of **dplyr's** most powerful and efficiency-boosting features: the capacity to summarize numerous columns in a single, streamlined command. This capability is essential for any analyst working with wide data frames containing dozens or even hundreds of variables. We will navigate the various strategies to accomplish this, ranging from applying functions to all columns to targeting specific selections based on column names or data types, all through the mastery of the versatile **across()** function, introduced in **dplyr** 1.0.0.

By the conclusion of this article, you will possess a robust understanding of how to implement different programmatic summarization strategies using **dplyr**, enabling you to extract valuable insights from your data with significantly greater speed and precision. We will utilize practical, executable R code examples to illustrate each method, ensuring clarity and providing a foundation for immediately applying these concepts to your own analytical workflows.

The Foundation: `dplyr`, Grouping, and `across()`

dplyr is often described as a grammar for data manipulation because it provides a small, coherent set of verbs designed to solve the vast majority of data wrangling challenges. It is a cornerstone of the Tidyverse, an ecosystem that champions clarity, consistency, and reproducibility in data organization and analysis. Core functions that are crucial for summarization include `filter()`, `select()`, and, most importantly for our task, `group_by()` and `summarise()`.

The process of aggregation typically relies heavily on the **pipe operator** (`%>%`), which allows you to chain multiple operations sequentially, transforming complex data steps into a readable, left-to-right flow. When summarizing data, we first use **group_by()** to partition the data frame based on categorical variables, defining the levels at which the subsequent calculations should occur. Following this, the **summarise()** function (or `summarize()`) collapses the grouped data, calculating aggregate statistics such as the **mean**, count, or **standard deviation** for each group.

The true efficiency gain in modern **dplyr** comes from the **across()** helper function. This function serves as a powerful abstraction layer, providing a flexible mechanism to apply the identical transformation or summary function to numerous columns simultaneously. By eliminating the necessity for manually writing out identical code blocks for every column, **across()** vastly improves coding efficiency, minimizes redundancy, and reduces the potential for manual errors in large-scale

data processing workflows.

A crucial consideration in any statistical summarization is the handling of missing values, typically represented by `NA` in R. If not handled correctly, the presence of even a single `NA` can render the entire statistical calculation invalid, often resulting in an `NA` output for the whole column summary. To counteract this, most summary functions within R accept the `na.rm` argument, short for "NA remove." By consistently setting `na.rm = TRUE` within our summary functions, we instruct **dplyr** to safely exclude missing values before performing the aggregation, ensuring that a valid numeric result is returned whenever possible.

Method 1: Summarizing All Relevant Columns

The first method is ideal for initial exploratory data analysis (EDA) or when you require a rapid, comprehensive statistical snapshot of your entire dataset. It allows you to apply a summary function across virtually all non-grouping variables within your data frame without the tedious task of listing column names individually. This blanket approach saves substantial development time, particularly when working with datasets that frequently change or contain a large number of quantitative metrics.

To execute this strategy, we utilize **across()** in combination with the **everything()** selection helper. The **everything()** helper is highly convenient because it dynamically selects every column in the data frame that has not already been used or specified (e.g., as the grouping variable in `group_by()`). This makes it the go-to choice for applying a single function uniformly across all remaining variables.

The following code demonstrates how to calculate the average (**mean**) of all columns, aggregated separately for each unique value found in `group_var`:

#summarise mean of all columns
df %>%
group_by(group_var) %>%
summarise(across(everything(), mean, na.rm=TRUE))

In this pipeline, `group_by(group_var)` establishes the necessary groups. Subsequently, `summarise(across(everything(), mean, na.rm=TRUE))` instructs **across()** to apply the `mean` function to every column selected by **everything()**, ensuring that missing values are gracefully handled. While powerful, remember that this method will attempt to run the function on all columns, meaning non-numeric variables might result in `NA` or be automatically dropped, depending on the specific summary function used.

Method 2: Summarizing a Targeted Subset of Columns

In contrast to the broad approach of Method 1, analytical requirements frequently demand a more focused summarization, concentrating only on a select few variables critical to the current question. This method grants precise control, allowing you to explicitly name and aggregate only those columns that are truly relevant, which is essential for maintaining clarity and avoiding computational overhead on unnecessary data points in large data frames.

To implement targeted selection, you simply pass a vector of column names directly into the **across()** function, typically constructed using the **c()** function. This vector clearly dictates the subset of columns that will be subjected to the summarization function. This explicit declaration ensures that your summary outputs are perfectly tailored to your analytical needs, streamlining the resulting tibble and focusing attention on key metrics.

For instance, if your analysis mandates calculating the **mean** solely for `col1` and `col2`, the required code structure remains straightforward and highly readable:

```
#summarise mean of col1 and col2 only
df %>%
group_by(group_var) %>%
summarise(across(c(col1, col2), mean, na.rm=TRUE))
```

Within this command, the argument `c(col1, col2)` precisely limits the scope of **across()**. The pipeline remains consistent: the data is grouped by `group_var`, and the `mean` is computed exclusively for the listed columns, while preserving the necessary missing value removal logic. This method provides surgical precision, ensuring computational resources are focused and the resulting summary is maximally informative.

Method 3: Summarizing Numeric Columns with Multiple Statistics

Real-world datasets are heterogeneous, comprising a blend of data types, including character strings, factors, and logical values alongside numeric data. Attempting to calculate quantitative measures like the **mean** or **standard deviation** on non-numeric columns will inevitably lead to errors or yield statistically meaningless results. To ensure the robustness and reliability of your analysis, it is frequently necessary to automatically filter and summarize only those columns that contain numeric data.

dplyr offers a sophisticated and elegant solution using the **where()** selection helper in combination with a predicate function. A predicate function is any function that accepts a column vector and returns a single Boolean (`TRUE` or `FALSE`) value, indicating whether the column meets a specified

criterion. For identifying numeric data, we employ the **is.numeric()** function within **where()**. This powerful combination dynamically selects all appropriate variables, ensuring that summary statistics are only applied where they are mathematically valid.

Furthermore, this method allows for the calculation of multiple summary statistics simultaneously for the selected numeric columns. By passing a **list()** of functions--such as `mean` and `sd`--to **across()**, you can generate comprehensive statistical profiles in a single, highly efficient operation, providing metrics like both central tendency and dispersion.

The following code snippet demonstrates how to compute both the **mean** and **standard deviation** for all numeric columns, aggregated by a grouping variable:

```
#summarise mean and standard deviation of all numeric columns
df %>%
group_by(group_var) %>%
summarise(across(where(is.numeric), list(mean=mean, sd=sd), na.rm=TRUE))
```

The expression `where(is.numeric)` intelligently filters the column selection. The argument `list(mean=mean, sd=sd)` then specifies the two functions to run. Critically, the names given in the **list()** (`mean`, `sd`) are appended to the original column names in the resulting output, yielding descriptive columns like `variable_mean` and `variable_sd`. This technique is invaluable for automating detailed statistical reporting across varied datasets.

Practical Application: Illustrative Examples

To fully internalize the power and mechanics of these summarization techniques, we will now transition to concrete, practical examples utilizing a small, representative data frame. These demonstrations will clearly show how each method discussed is implemented in a real-world context and what the structure of the resulting aggregated output looks like. We will use a fictional dataset detailing team sports statistics.

First, we must construct the sample data frame that will serve as the input for all subsequent examples:

```
#create data frame
df <- data.frame(team=c('A', 'A', 'A', 'B', 'B', 'B'),
points=c(99, 90, 86, 88, 95, 90),
assists=c(33, 28, 31, 39, 34, 25),
rebounds=c(NA, 28, 24, 24, 28, 19))

#view data frame
```

df

team points assists rebounds
1 A 99 33 NA
2 A 90 28 28
3 A 86 31 24
4 B 88 39 24
5 B 95 34 28
6 B 90 25 19

This data frame, `df`, contains a mix of team identifiers (`team`) and numeric performance metrics (`points`, `assists`, `rebounds`). Importantly, the intentional inclusion of an `NA` value in the `rebounds` column for Team A will allow us to verify the effectiveness of the `na.rm = TRUE` argument in correctly calculating group-wise summaries despite missing data.

Example 1: Summarize All Columns

We begin by implementing Method 1 on our sample data. The objective is to calculate the average (**mean**) for every non-grouping column, segregated by `team`. This output provides an immediate, broad overview of the central tendency for each team across all recorded statistical metrics.

library(dplyr)

#summarise mean of all columns, grouped by team
df %>%
group_by(team) %>%
summarise(across(everything(), mean, na.rm=TRUE))

A tibble: 2 x 4
team points assists rebounds

1 A 91.7 30.7 26
2 B 91 32.7 23.7

The output is a grouped **tibble**, the enhanced **dplyr** equivalent of a data frame. It presents the group (`team`) along with the average `points`, `assists`, and `rebounds`. Crucially, notice that the mean for `rebounds` in Team A (26) was calculated correctly from the two available values (28 and 24), successfully ignoring the `NA` thanks to the `na.rm = TRUE` argument. This showcases the power and convenience of using **everything()** for blanket summaries.

Example 2: Summarize Specific Columns

Next, we execute Method 2, employing a targeted focus. Suppose our interest lies strictly in comparing the average `points` and `rebounds`, potentially viewing `assists` as a secondary metric for this analysis. We must explicitly select only the two primary columns for summarization.

library(dplyr)

#summarise mean of points and rebounds, grouped by team
df %>%
group_by(team) %>%
summarise(across(c(points, rebounds), mean, na.rm=TRUE))

A tibble: 2 x 3
team points rebounds

1 A 91.7 26
2 B 91 23.7

As anticipated, the resulting **tibble** is streamlined, containing only the grouping column (`team`) and the two requested summary columns (`points` and `rebounds`). The data for `assists` has been intentionally excluded. This selective approach is invaluable when minimizing complexity and ensuring that the output focuses precisely on the variables most pertinent to the current stage of the data analysis.

Example 3: Summarize Numeric Columns with Multiple Statistics

Finally, we demonstrate Method 3, which offers the most detailed statistical profile while maintaining robustness against varying data types. Here, we calculate both the **mean** and the **standard deviation** for all numeric columns, aggregated by team. This provides a comprehensive view of team performance, capturing both average values and the degree of performance variation.

library(dplyr)

#summarise mean and standard deviation of all numeric columns
df %>%
group_by(team) %>%
summarise(across(where(is.numeric), list(mean=mean, sd=sd), na.rm=TRUE))

A tibble: 2 x 7
team points_mean points_sd assists_mean assists_sd rebounds_mean rebounds_sd

1 A 91.7 6.66 30.7 2.52 26 2.83
2 B 91 3.61 32.7 7.09 23.7 4.51

The resulting **tibble** is significantly expanded, now featuring distinct columns for both summary statistics for each numeric variable (e.g., `points_mean` and `points_sd`). This output elegantly illustrates the efficiency of combining the dynamic selection capabilities of `where(is.numeric)` with the ability to pass a **list()** of functions, allowing for sophisticated, automated statistical summaries that are both detailed and error-resistant.

Conclusion and Further Exploration

Mastering the methodology for summarizing multiple columns within R using **dplyr** and the **across()** function is a key milestone in enhancing your data analysis capabilities. We have successfully demonstrated three highly effective, distinct methods: summarizing all variables using **everything()**, targeting specific columns using vectors, and intelligently aggregating only numeric columns using **where()**. Each technique is designed to address different analytical needs while maximizing code efficiency.

The structural elegance and operational efficiency provided by **dplyr**, especially when coupled with the adaptability of **across()**, empower analysts to produce cleaner, more scalable, and highly maintainable data manipulation scripts. By leveraging dynamic selection features and the ability to apply multiple functions simultaneously via a **list()**, complex summarization tasks that once required extensive manual coding can now be achieved with remarkable simplicity and speed.

We strongly encourage practitioners to apply these methods to their own datasets, experimenting with different grouping variables and summary functions to fully grasp their potential. Keep in mind that the **across()** function is exceptionally versatile; consulting its official documentation will reveal even more advanced patterns and use cases, further streamlining your **dplyr** and R programming journey.

Additional Resources for `dplyr` Mastery

For those committed to deepening their technical expertise in **dplyr** and the broader Tidyverse framework, consulting the official documentation remains the most authoritative source. The documentation for **across()**, in particular, offers detailed insights into its advanced syntax for column selection and function application, including ways to handle non-standard evaluation.

The following resources provide excellent supplementary material and tutorials on other core data manipulation functions within **dplyr**, helping you build a comprehensive toolkit for effective data analysis in R:

Official **dplyr** Website and Reference Manual
Introduction to the Tidyverse
Tutorials on Data Filtering and Selecting (`filter()` and `select()`)
Guides on Creating and Modifying Columns (`mutate()`)

These guides are crucial for mastering all stages of the data analysis workflow, from initial data cleaning and transformation to the sophisticated summarization techniques covered here.