# Learning dplyr’s across() Function: A Comprehensive Guide with Examples

Authored by

**Mohammed looti**

October 29, 2025

The `across()` function, a core component of the celebrated `dplyr` package in R, represents a significant advancement in data manipulation efficiency. Designed specifically to reduce repetitive code, this powerful tool allows analysts to apply identical transformations or aggregation operations simultaneously across multiple columns within a data frame or tibble. Mastering `across()` is essential for writing clean, scalable, and highly readable analytical scripts in R.

As a key innovation within the broader Tidyverse framework, `across()` elegantly solves the problem of "copy-paste" coding, which frequently occurs when dealing with numerous variables that require the same treatment (e.g., standardizing 20 different feature columns). Instead of manually coding a separate line for each variable, `across()` provides a consolidated and expressive syntax. This functionality not only saves considerable time during the development phase but also minimizes the chances of introducing errors, thereby greatly improving the maintainability of your code.

This comprehensive guide will demonstrate the fundamental versatility of the `across()` function by exploring three distinct and highly practical applications. We will examine how to use `across()` to perform transformations on selected columns, calculate a single summary statistic across a group of variables, and finally, compute multiple statistical summaries simultaneously, illustrating its utility across various stages of the data analysis workflow.

## Setting Up: The Sample Data Frame

To provide a clear and executable demonstration of the `across()` function, we must first establish a common dataset. This sample data frame will serve as the foundation for all subsequent examples, ensuring that the results of each operation are transparent and easy to follow.

Our example dataset, named `df`, is a simplified representation of hypothetical sports team performance statistics. It includes a categorical variable indicating the team's conference affiliation, alongside two crucial numeric metrics: points scored and total rebounds. Familiarity with the structure of this data frame is essential for understanding how `across()` selectively operates on its columns.

Below is the R code used to construct and display the structure of our working data frame. This code initializes the data structure we will be manipulating throughout the tutorial:

```
#create data frame
df <- data.frame(conf=c('East', 'East', 'East', 'West', 'West', 'West'),
points=c(22, 25, 29, 13, 22, 30),
rebounds=c(12, 10, 6, 6, 8, 11))

#view data frame
```

df

conf points rebounds
1 East 22 12
2 East 25 10
3 East 29 6
4 West 13 6
5 West 22 8
6 West 30 11

The resulting `df` data frame clearly contains three variables: `conf` (a categorical variable), and `points` and `rebounds` (both numeric variables). Since the latter two columns share the same data type and represent performance metrics, they are perfect candidates for demonstrating the efficient, parallel application of functions using `across()`.

## Use Case 1: Applying a Single Transformation to Multiple Columns

One of the most fundamental and frequently used applications of the `across()` function is its ability to apply an identical transformation or arbitrary function to a defined set of columns. This scenario is highly relevant whenever data scaling, normalization, or simple arithmetic adjustments are required across several variables simultaneously.

In this initial example, we will demonstrate how to leverage `across()` in combination with the `mutate()` verb. Our goal is to double the values present in both the `points` and `rebounds` columns. The `mutate()` function is utilized here because its purpose is to create new columns or modify existing ones within the scope of the original data frame structure.

The core syntax of `across()` requires two primary arguments: the selection of columns (which can be provided as a character vector, column names, or using selection helpers), and the function to be applied. For straightforward operations like multiplication, defining an anonymous function (e.g., `function(x) x*2`) offers a concise and immediate way to specify the necessary transformation without defining a separate, named function.

**library(dplyr)**

#multiply values in points and rebounds columns by 2
df %>%
mutate(across(c(points, rebounds), function(x) x*2))

conf points rebounds
1 East 44 24

2 East 50 20
3 East 58 12
4 West 26 12
5 West 44 16
6 West 60 22

The output confirms that the values in both target columns--`points` and `rebounds`--have been successfully doubled. This example clearly illustrates how `across()` facilitates column-wise operations with dramatically reduced code volume, leading to scripts that are not only cleaner but significantly easier to review and maintain, especially in datasets containing dozens or hundreds of variables.

## Use Case 2: Calculating One Summary Statistic for Multiple Columns

Beyond simple data transformations, `across()` is exceptionally valuable for exploratory data analysis (EDA), allowing users to compute a single summary statistic across a collection of variables efficiently. This is commonly required when rapidly assessing key metrics such as central tendency or data spread for multiple numeric features.

In this second scenario, we pair `across()` with the `summarise()` verb. Our objective is to calculate the arithmetic mean value for both the `points` and `rebounds` columns across the entire dataset. The `summarise()` function is specifically designed to collapse data, resulting in a concise output that contains the aggregated statistics.

When calculating statistical measures like the mean, it is critical to address potential missing values. By including the argument `na.rm = TRUE` within the `across()` call, we ensure that any `NA` values are automatically excluded from the calculation. This prevents missing data from corrupting the statistical result, which would otherwise propagate to an `NA` output for the entire statistic.

**library(dplyr)**

#calculate mean value of points an rebounds columns
df %>%
summarise(across(c(points, rebounds), mean, na.rm=TRUE))

points rebounds
1 23.5 8.833333

The resulting output provides a clear snapshot of the average `points` and `rebounds` across all teams. This technique is highly efficient for generating key metrics from multiple variables using a

single, streamlined command, significantly speeding up the initial stages of data exploration.

Furthermore, `across()` offers sophisticated flexibility in column selection. Rather than listing columns manually, analysts can utilize helper functions to dynamically select variables based on their properties. A prime example is using the `where()` helper in combination with type-checking functions, such as `where(is.numeric)`. This allows you to automatically compute a summary statistic, like the mean, for every numeric column currently residing in your data frame.

This dynamic selection method is indispensable when handling wide data frames where manual specification is impractical. It ensures that your analytical code adapts robustly, even if the underlying data structure changes slightly (e.g., adding or removing numeric columns).

**library(dplyr)**

#calculate mean value for every numeric column in data frame
df %>%
summarise(across(where(is.numeric), mean, na.rm=TRUE))

points rebounds
1 23.5 8.833333

As confirmed by the identical output, this code successfully identified and processed only the numeric columns. This demonstrates how `across()`, when coupled with selection helpers, significantly enhances the flexibility and generality of your data manipulation scripts.

## Use Case 3: Generating Multiple Statistics Simultaneously

The true power of `across()` becomes apparent when the task requires calculating not just one, but multiple summary statistics for a designated group of columns. This functionality is crucial for generating comprehensive descriptive statistics, providing insights into both the central tendency and the variability of key variables in a single operation.

In this final example, we will combine `across()` and `summarise()` to calculate both the mean and the standard deviation for the `points` and `rebounds` columns. The mechanism for achieving this multi-summary calculation is the `.fns` argument of `across()`, which accepts a named list of functions.

By defining the list as `list(mean=mean, sd=sd)`, we explicitly instruct `across()` to execute both the `mean()` and `sd()` functions on the selected columns. The names provided within the list (e.g., `mean` and `sd`) are automatically concatenated with the original column names to form the descriptive labels of the resulting summary columns, ensuring output clarity.

```
library(dplyr)

#calculate mean and standard deviation for points and rebounds columns
df %>%
summarise(across(c(points, rebounds), list(mean=mean, sd=sd), na.rm=TRUE))

points_mean points_sd rebounds_mean rebounds_sd
1 23.5 6.156298 8.833333 2.562551
```

The resulting data frame now contains four new, informative columns: `points_mean`, `points_sd`, `rebounds_mean`, and `rebounds_sd`. This output beautifully encapsulates the central tendency and standard deviation for both specified variables. This single, elegant `across()` command delivers rich descriptive statistics, making it an invaluable technique for efficient data summarization.

## Conclusion and Next Steps

The `across()` function in `dplyr` is unequivocally an essential tool for any data practitioner utilizing R. Its core capability--applying functions and calculating statistics across multiple columns with a concise, uniform syntax--dramatically enhances code readability, eradicates redundancy, and boosts analytical throughput.

Whether performing necessary data transformations, such as scaling and standardizing, or generating complex multi-statistic summaries for reporting, `across()` offers a flexible and powerful solution to common data manipulation challenges. By mastering the techniques demonstrated here, you are equipped to write significantly cleaner, more robust, and easier-to-maintain R code.

The seamless integration of `across()` with other fundamental `dplyr` verbs, including `mutate()` and `summarise()`, unlocks a vast array of possibilities for sophisticated data wrangling and detailed exploratory analysis. We strongly encourage further experimentation with different column selection helpers (like `starts_with()` or `matches()`) and custom functions to fully tailor the power of `across()` to your unique analytical requirements. The resulting efficiency gains will undoubtedly optimize your R workflow.

## Additional Resources

For a more comprehensive understanding and exploration of advanced use cases for the `across()` function, please consult the complete official documentation. This resource offers in-depth details on all available arguments, selection helpers, and further examples to enhance your proficiency.

The following tutorials provide valuable insights into executing other foundational data manipulation

tasks using `dplyr`, complementing the knowledge gained from mastering `across()`:

Official `dplyr` website documentation for `across()`.

Tutorials explaining effective data filtering and grouping using `dplyr`.

Guides on reshaping data with Tidyverse packages.