Learning dplyr: Summarizing DataFrames While Preserving All Columns in R

Authored by

Mohammed looti

June 15, 2026

RECOMMENDED CITATION

Mohammed looti (2026). *Learning dplyr: Summarizing DataFrames While Preserving All Columns in R*. PSYCHOLOGICAL STATISTICS. Retrieved from https://statistics.arabpsychology.com/?p=3783

Introduction to Data Summarization in R and the Tidyverse

Effective data manipulation forms the backbone of modern statistical analysis. Analysts frequently need to condense large, raw datasets into concise, meaningful summaries to uncover patterns, calculate performance metrics, or prepare data for visualization. Within the statistical computing environment R, the dplyr package--a foundational element of the tidyverse ecosystem--is the definitive tool for efficient data transformation. It provides a highly consistent and readable syntax, utilizing a core set of "verbs" that simplify complex data operations such as filtering, selecting, and, most importantly, summarizing data.

The primary function used for data aggregation in **dplyr** is **summarise()** (or **summarize()**). This powerful function excels at calculating descriptive statistics--such as means, standard deviations, or counts--often applied across defined subsets of the data using **group_by()**. However, analysts often encounter a specific requirement: calculating these group-wise statistics while simultaneously preserving the original, granular structure of the data frame, retaining all rows and columns.

The default behavior of **summarise()** is designed to reduce the dataset's dimensionality, resulting in a new table that contains only the grouping variables and the calculated statistics. Any columns not explicitly used in the grouping or summarizing step are discarded. This article will first clarify why this happens and then introduce the elegant solution: utilizing the **mutate()** function in conjunction with **group_by()** to enrich the dataset with summary statistics without losing any of the original contextual information.

The Intentional Granularity Shift of dplyr::summarise()

To understand the challenge of column retention, we must first recognize the fundamental objective of the **summarise()** function. Its core design principle is to perform aggregation, thereby reducing the number of observations (rows) in the dataset. When paired with **group_by()**, the function calculates one result per group. For example, summarizing a dataset of sales transactions by 'Region' to find the 'Total Revenue' inherently changes the dataset's granularity from individual transactions to regional totals.

Consider a detailed dataset containing player metrics like 'points,' 'assists,' and 'rebounds.' If we apply **group_by(Team)** and then **summarise(MeanPoints = mean(points))**, the resulting output will contain one row per team and two columns: 'Team' and 'MeanPoints.' The individual player columns like 'assists' and 'rebounds' are necessarily dropped because they cannot be meaningfully aggregated into a single team row unless they were also summarized. This behavior is typically desired when creating clean reports or high-level dashboards where granular data is irrelevant.

However, data analysis frequently requires "context enrichment." This occurs when an analyst needs to compare each observation (e.g., an individual player's points) against a group metric

(e.g., their team's average points) within the same row. Using standard **summarise()** makes this comparison impossible, as it fundamentally alters the data's structure, sacrificing the individual observations. Understanding this limitation is crucial before adopting the alternative, row-preserving methods detailed in the following sections.

Setting Up Our Example: Basketball Player Data

To concretely illustrate the difference between row aggregation and row enrichment, we will utilize a simple, yet practical, dataset detailing basketball player statistics. This dataset structure, often referred to as a data frame in R, tracks individual performance metrics for players belonging to various teams.

We begin by constructing this sample dataset in the **R** environment. The dataset, named `df`, includes three essential columns: **team** (the grouping variable), **points** (the metric to be summarized), and **assists** (a contextual column we wish to retain).

```
#create data frame
df <- data.frame(team=rep(c('A', 'B', 'C'), each=3),
points=c(4, 9, 8, 12, 15, 14, 29, 30, 22),
assists=c(3, 3, 2, 5, 8, 10, 4, 5, 12))

#view data frame
df

team points assists
1 A 4 3
2 A 9 3
3 A 8 2
4 B 12 5
5 B 15 8
6 B 14 10
7 C 29 4
8 C 30 5
9 C 22 12
```

The resulting data frame contains nine distinct rows, representing nine individual player performances distributed across three teams (A, B, and C). This structure provides a perfect test environment for demonstrating how group-wise calculations affect the overall data structure depending on the chosen **dplyr** verb.

Demonstrating Standard dplyr::summarise() for Grouped Data

We will now apply the conventional aggregation approach using the **summarise()** function to calculate the average points scored for each team. This operation requires specifying the grouping variable using **group_by()**. We chain these operations efficiently using the pipe operator (`%>%`), which passes the result of one function directly into the first argument of the next, significantly enhancing the readability of the data workflow.

library(dplyr)

#summarize mean points values by team
df %>%
group_by(team) %>%
summarise(mean_pts = mean(points))

A tibble: 3 x 2
team mean_pts

1 A 7
2 B 13.7
3 C 27

The resulting output is a concise tibble containing only three rows--one for each team--and two columns: **team** and **mean_pts**. This calculation confirms that the average points for Team A is **7**, Team B is approximately **13.7**, and Team C is **27**.

The crucial observation here is the reduction in rows (from nine to three) and the automatic exclusion of the **assists** column. While this aggregated view is excellent for summary reports, it eliminates the possibility of directly comparing an individual player's performance against their team average within the same table structure. To overcome this limitation and retain the original data's richness, we must turn to the powerful combination of **group_by()** and **mutate()**.

The Solution: Retaining All Columns with dplyr::mutate()

When the objective is to calculate a group-level statistic and append it as a new variable to the original dataset without altering the row count, the **mutate()** function is the appropriate tool. Unlike **summarise()**, **mutate()** is designed to add new columns or transform existing ones while strictly preserving the dimensionality of the input data frame--the output will always have the same number of rows as the input.

When a dataset is first grouped using **group_by()**, any subsequent operation performed by

mutate() is executed independently within those groups. Critically, if the calculation results in a single value per group (like a mean or a sum), **mutate()** automatically "broadcasts" that value back to every single row belonging to that respective group. This mechanism allows us to calculate a team average and assign that team average back to every individual player on that team.

After performing the group-wise mutation, it is considered best practice to explicitly remove the grouping structure using the **ungroup()** function. This prevents unintended group operations in subsequent steps and ensures the data frame returns to its default, ungrouped state.

Let's apply this robust technique to our basketball data to calculate `mean_pts` while retaining all player-level details:

library(dplyr)

#summarize mean points values by team and keep all columns
df %>%
group_by(team) %>%
mutate(mean_pts = mean(points)) %>%
ungroup()

A tibble: 9 x 4
team points assists mean_pts

1 A 4 3 7
2 A 9 3 7
3 A 8 2 7
4 B 12 5 13.7
5 B 15 8 13.7
5 B 14 10 13.7
7 C 29 4 27
8 C 30 5 27
9 C 22 12 27

The resulting tibble now perfectly meets the requirement: it contains all nine original rows, preserving the **points** and **assists** columns, and includes the new **mean_pts** column, which displays the calculated team average alongside each player's individual statistics.

Understanding the Power of mutate() for Data Transformation

The ability of **mutate()** to perform group-wise calculations and then broadcast the result is invaluable, but its utility extends to nearly all aspects of data transformation within dplyr. It is the

primary verb for creating, modifying, and transforming variables while ensuring the dataset remains dimensionally consistent.

Beyond calculating group averages, **mutate()** enables complex operations such as calculating deviations from the group mean (e.g., `points - mean(points)`), standardizing variables, or deriving ratios (e.g., assists per point). Because it respects the row count, it is the function of choice for feature engineering--creating new variables required for modeling or advanced analytical comparison--without aggregating or discarding observational data.

Mastering the distinction between **summarise()** (which reduces rows) and **mutate()** (which preserves rows) is a cornerstone of advanced data manipulation in R. This technique is particularly critical in analyses where individual performance must be benchmarked against a larger group context, ensuring that both the granular detail and the aggregated context are available simultaneously for every observation.

Further Explorations in dplyr

The **dplyr** package provides a comprehensive toolkit for managing tabular data efficiently. Proficiency in the core verbs--**summarise()**, **group_by()**, and **mutate()**--is essential, but the package offers much more functionality to streamline the entire data preparation pipeline.

To expand your expertise in R data wrangling, explore the other fundamental operations provided by **dplyr**. These include **filter()** for subsetting rows based on logical conditions, **select()** for managing columns, **arrange()** for sorting data, and the powerful family of **join** functions for combining multiple data frames based on shared keys.

By integrating these functions, you can create clean, readable, and highly efficient data pipelines for virtually any data manipulation task. The resources below offer further documentation and tutorials on leveraging the full power of the **tidyverse** for data science projects.

Official documentation for the **tidyverse** packages.

Tutorials on advanced joining techniques in **dplyr**.

Guidance on using the **magrittr** package for enhanced piping.