Learning to Create New Variables in R with mutate() and case_when()

Authored by

Mohammed looti

November 7, 2025

RECOMMENDED CITATION

Mohammed looti (2025). *Learning to Create New Variables in R with mutate() and case_when()*. PSYCHOLOGICAL STATISTICS. Retrieved from https://statistics.arabpsychology.com/?p=12440

In the realm of data analysis using R, the ability to transform raw data into meaningful derived variables is paramount. Analysts frequently encounter scenarios where they must categorize observations, calculate performance metrics, or assign specific statuses based on complex, multi-layered conditions applied to existing columns. While base R provides tools for this transformation, the modern data science ecosystem overwhelmingly favors the dplyr package, a core component of the Tidyverse. This powerful library offers a consistent, readable, and highly efficient grammar for data manipulation tasks, dramatically simplifying what could otherwise become cumbersome code.

When the goal is to define a new column whose values are determined by checking a sequence of logical tests against other columns, the combination of **`mutate()`** and **`case_when()`** proves indispensable. The **`mutate()`** function serves the structural purpose of adding or modifying a column within a data frame, acting as the container for the new variable. Concurrently, **`case_when()`** handles the heavy lifting of sequential conditional logic. It allows for the definition of mutually exclusive conditions that are evaluated in a specific order, offering a clean, vastly superior alternative to deeply nested `if-else` or `ifelse()` statements prevalent in base R.

This comprehensive tutorial is designed to equip you with mastery over this dynamic duo of dplyr functions. We will systematically explore how to generate derived variables, beginning with simple binary classifications and escalating to advanced scenarios involving the simultaneous evaluation of multiple numeric and categorical variables. By the end of this guide, you will be proficient in structuring robust, scalable, and easy-to-interpret conditional statements, thereby streamlining your data preparation pipeline and ensuring your resulting variables are perfectly aligned with your statistical or business requirements.

The Foundation of Transformation: `mutate()` and `case_when()`

The dplyr package is lauded for its intuitive grammar, making standard data operations straightforward and predictable. The **`mutate()`** function is central to this paradigm, specializing in non-destructive modifications to datasets. Its core role is either to introduce an entirely new column to the existing data frame or to recalculate the values of an existing one. The syntax maintains the Tidyverse philosophy: the data object is piped into **`mutate()`**, followed by a named argument where the new variable (or calculation) is defined.

While **`mutate()`** sets the stage, **`case_when()`** is where complex conditional assignment logic resides. Traditional conditional logic often forces the analyst into using nested `ifelse()` structures when faced with more than two potential outcomes. Such nesting quickly degrades code readability and significantly increases the probability of introducing logical errors. **`case_when()`** resolves this by providing an expressive, multi-case structure that is evaluated sequentially. It accepts pairs of arguments separated by a tilde (~): the logical test (the condition) comes first,

followed by the value to assign if that condition evaluates to **`TRUE`**.

Understanding the **sequential evaluation** of **`case_when()`** is absolutely critical to defining accurate logic. The function processes the conditions in the exact order they are written, stopping and assigning the value once the first condition is met for a given row. This means that if conditions overlap, the most restrictive or specific conditions must be placed earlier in the sequence, moving toward broader conditions later. For instance, if you categorize "points greater than 20" before "points greater than 10," the data points exceeding 20 will be correctly caught by the first, more specific category, ensuring hierarchical classification is maintained.

Furthermore, **`case_when()`** offers a robust mechanism for handling rows that fail all specified conditions. This is achieved by including a final, universally true condition, typically written as **`TRUE ~ 'default_value'`**. If this default condition is omitted and a row does not satisfy any of the predefined criteria, the resulting value for that row in the new column will be assigned as **`NA`** (Not Available). Including a default case is strongly recommended practice for writing durable code, as it prevents unexpected missing values and clearly documents how unclassified data points are managed during the transformation phase.

Establishing the Environment: The Sample Data Frame

To effectively illustrate the practical application of **mutate()** and **case_when()**, we will utilize a small, focused data frame representing performance metrics for a cohort of players. This simplified dataset structure is ideal for clearly mapping conditional logic from input variables to new classification schemes. The dataset incorporates variables detailing player identification (`player`), their designated role (`position`), and key statistical outputs (`points` and `rebounds`).

The variables are defined as follows: `player` is a unique categorical identifier; `position` classifies the player as either a Guard (G) or Forward (F); and `points` and `rebounds` are essential numeric metrics reflecting on-court performance. The subsequent examples will demonstrate how to transform these raw numeric and categorical columns into highly informative, derived variables that categorize performance or define role complexity based on predefined logical thresholds.

We start by executing the standard R code necessary to create and display this foundational data frame. Note that this initial step uses base R functions for creation, setting the stage for the powerful transformation capabilities provided by dplyr that follow:

```
#create data frame
df <- data.frame(player = c('a', 'b', 'c', 'd', 'e'),
position = c('G', 'F', 'F', 'G', 'G'),
points = c(12, 15, 19, 22, 32),
rebounds = c(5, 7, 7, 12, 11))
```

#view data frame
df

player position points rebounds
1 a G 12 5
2 b F 15 7
3 c F 19 7
4 d G 22 12
5 e G 32 11

Example 1: Discretization Using a Single Numeric Variable

One of the most common data preparation tasks is the conversion of a continuous numeric variable, such as `points`, into a discrete categorical variable representing performance tiers. This process, known as binning or discretization, significantly aids in interpretation and subsequent grouping for statistical models. In this inaugural example, we leverage **mutate()** to introduce a new variable, `scorer`, which classifies players into 'low', 'med', or 'high' tiers based solely on their total points score.

The conditional assignment is handled entirely within the **case_when()** function. Notice the sequential order of the conditions as they test against the `points` column. Since **case_when()** proceeds in order, we define the tiers using exclusive upper bounds (e.g., `points < 15`). A player scoring 12 points immediately meets the first condition and is assigned 'low'. A player scoring 19 points fails the first test, and the evaluation proceeds to the second condition (`points < 25`), resulting in the assignment 'med'. This ordered evaluation is key to ensuring that observations are cleanly slotted into the appropriate, non-overlapping bracket.

This approach dramatically improves clarity compared to complex nested `if-else` structures. The explicit pairing of condition and outcome (e.g., `points < 15 ~ 'low'`) renders the transformation logic transparent and easily verifiable. Furthermore, the use of the pipe operator (`%>%`) allows us to pass the data frame seamlessly into the **mutate()** function, maintaining the clean, linear workflow characteristic of the R Tidyverse environment.

Examine the resulting code block below. The newly generated `scorer` column is appended to the original data frame, demonstrating the effective application of the conditional assignment across the entire dataset:

library(dplyr)

#define new variable 'scorer' using mutate() and case_when()
df %>%

mutate(scorer = case_when(points < 15 ~ 'low',
points < 25 ~ 'med',
points < 35 ~ 'high'))

player position points rebounds scorer
1 a G 12 5 low
2 b F 15 7 med
3 c F 19 7 med
4 d G 22 12 med
5 e G 32 11 high

Example 2: Grouping Based on Multiple Categorical Variables

While the initial example focused on binning a single numeric field, real-world data transformation often requires combining criteria from multiple categorical columns. This is essential when assigning abstract roles, statuses, or classifications based on specific combinations of underlying attributes. In this second example, we introduce a new variable named `type`, which defines a player's role (starter, backup, or reserve) by simultaneously evaluating both the `player` identifier and their `position`.

Within the **case_when()** structure, we employ the logical OR operator (`|`) to group multiple criteria that lead to the same outcome. For instance, the first condition, `player == 'a' | player == 'b' ~ 'starter'`, checks if the player is 'a' OR if the player is 'b'. If either condition is true, the player is immediately assigned the 'starter' role. This technique is invaluable for establishing groups where membership is based on meeting any one of several defined criteria.

The final condition in this example again underscores the importance of sequential evaluation. The assignment for 'reserve' is defined by `position == 'G'`. This condition is intentionally placed last because players 'a' and 'd' are also Guards. However, player 'a' is already classified as a 'starter' by the first condition, and player 'd' is classified as a 'backup' by the second condition. Consequently, the 'reserve' classification only applies to players who are Guards but failed to meet the criteria established earlier in the sequence--in this specific case, player 'e'. This hierarchical structure confirms that the desired order of classification is strictly maintained, despite overlapping criteria.

The application of **mutate()** effortlessly integrates this complex categorical assignment, producing the new `type` column that accurately reflects the custom classification rules based on player identity and position:

library(dplyr)

```
#define new variable 'type' using mutate() and case_when()
df %>%
mutate(type = case_when(player == 'a' | player == 'b' ~ 'starter',
player == 'c' | player == 'd' ~ 'backup',
position == 'G' ~ 'reserve'))
```

```
player position points rebounds type
1 a G 12 5 starter
2 b F 15 7 starter
3 c F 19 7 backup
4 d G 22 12 backup
5 e G 32 11 reserve
```

Example 3: Complex Logic Combining Numeric and Categorical Data

The maximum utility of **`case_when()`** is realized when you must define conditions that simultaneously evaluate multiple variables, often necessitating the mixing of both numeric and categorical data types. This functionality is vital for creating highly nuanced scoring systems or sophisticated classification models. In this final, advanced example, we calculate a new variable, `valueAdded`, which is a numeric score derived from complex interactions between `points` (numeric) and `rebounds` (numeric).

To define these intricate, multi-faceted conditions, we utilize the logical AND operator (**`&`**). A condition employing **`&`** will only evaluate to **`TRUE`** if all constituent logical tests are satisfied simultaneously. For example, the condition `points <= 15 & rebounds <= 5 ~ 2` mandates that a player must score 15 points or fewer AND achieve 5 rebounds or fewer, in order to be assigned a `valueAdded` score of 2. This multi-criteria conditional assignment is fundamental when calculating composite metrics that rely on meeting multiple thresholds.

Observe how the criteria are painstakingly structured to logically cover the entire performance space without introducing ambiguity. The combination of ranges (e.g., `points < 25`) coupled with specific threshold checks on the secondary variable (`rebounds < 8` or `rebounds > 8`) ensures that every row falls into a defined category, resulting in a robust, multi-dimensional classification. The final condition, `points >=25 ~ 9`, operates as a high-performance catch-all, guaranteeing that any player exceeding the maximum point threshold is automatically assigned the highest score, regardless of where they fell in the rebound categories of the earlier conditions.

This application powerfully demonstrates how **mutate()** and **case_when()** collaborate to produce sophisticated, calculated variables that are essential for advanced statistical modeling and reporting. The resulting `valueAdded` score effectively summarizes player performance based on

the precise, predefined weighted logic:

library(dplyr)

#define new variable 'valueAdded' using mutate() and case_when()
df %>%
mutate(valueAdded = case_when(points <= 15 & rebounds <=5 ~ 2,
points <=15 & rebounds > 5 ~ 4,
points < 25 & rebounds < 8 ~ 6,
points < 25 & rebounds > 8 ~ 7,
points >=25 ~ 9))

player position points rebounds valueAdded
1 a G 12 5 2
2 b F 15 7 4
3 c F 19 7 6
4 d G 22 12 7
5 e G 32 11 9

Conclusion: Streamlining Data Preparation in R

The pairing of **mutate()** and **case_when()** represents an indispensable toolset for modern data transformation within R. By harnessing the expressive and clean syntax of **case_when()**, analysts can successfully transition away from complex, error-prone nested conditional statements. This allows for the creation of clean, sequential, and highly readable rules for any variable derivation task, whether it involves discretizing a numeric score, assigning a categorical role based on multiple attributes, or calculating a complex composite metric.

To ensure accuracy and robustness in your code, always remember two fundamental rules: First, meticulously prioritize the order of your conditions within `case_when()`, as only the first true condition will be executed for a given row. Second, it is highly recommended practice to include a final default condition (e.g., `TRUE ~ default_value`) to guarantee that all rows are accounted for and to mitigate the propagation of unintended `NA` values into your resulting dataset. Mastery of these techniques will significantly enhance the efficiency, clarity, and reproducibility of your data preparation scripts within the dplyr framework.

Additional Resources for R Data Manipulation

How to Rename Columns in R
How to Remove Columns in R

How to Filter Rows in R