How to Easily Remove Rows with dplyr’s filter() Function

Authored by

stats writer

December 4, 2025

The process of cleaning and preparing data is foundational to any robust statistical analysis or data science project. In the R programming environment, the management of large datasets is significantly simplified through the use of the **dplyr** package, a core component of the tidyverse ecosystem. **dplyr** provides a cohesive and highly optimized set of functions for data manipulation, designed around intuitive verbs that make code both readable and efficient. One of the most common requirements in data preprocessing is the removal of unwanted observations or rows, which can include handling missing values, eliminating duplicate entries, or filtering based on complex logical criteria.

To effectively remove rows from an R **data frame**, we primarily utilize the **filter()** function provided by **dplyr**. While the name suggests selecting rows, we can easily achieve row removal by negating the selection criteria. The **filter()** function takes logical expressions applied to the columns of the **data frame**, and only retains rows where the condition evaluates to `TRUE`. By leveraging the negation operator (`!`) or defining conditions that exclude specific rows, we transform this powerful selection tool into a precise instrument for observation exclusion. This approach is highly flexible, allowing for the definition of conditions based on column values, comparisons, or even positional index.

This comprehensive guide will detail five essential methods for removing rows from a dataset using **dplyr**. We will explore techniques ranging from the removal of rows containing **NA** (Not Applicable) values--a common data cleaning task--to removing rows based on specific index positions or complex logical combinations involving multiple columns. Mastering these techniques is crucial for ensuring the integrity and quality of your dataset before proceeding with statistical modeling or visualization. All examples will be demonstrated using R's pipe operator (`%>%`), which promotes fluid, sequential data operations typical of the tidyverse style.

Fundamental Techniques for Row Removal

The **dplyr** package offers distinct syntax patterns tailored to different row removal scenarios. Understanding these fundamental structures is the first step toward efficient data cleaning. The following list outlines the core command structures used to eliminate specific types of unwanted observations, such as missing data, duplicates, or rows at specific locations within the **data frame**.

The standard syntax employs the pipeline operator (`%>%`) to sequentially pass the **data frame** (`df`) to the relevant **dplyr** function, streamlining the manipulation process.

1. Remove any row with NA's

This method uses the built-in R function `na.omit()`, which is often faster for blanket removal of observations containing any missing data across all columns.

```
df %>%
na.omit()
```

2. Remove any row with NA's in specific column

Using `filter()` combined with the negation operator (`!`) and `is.na()` allows for targeted removal of missing values only within specified variables.

```
df %>%
filter(!is.na(column_name))
```

3. Remove duplicates

The `distinct()` function efficiently identifies and keeps only unique rows, effectively removing all subsequent duplicates from the dataset.

```
df %>%
distinct()
```

4. Remove rows by index position

This technique leverages the `row_number()` function within `filter()` to identify specific row indices (positions) for exclusion.

```
df %>%
filter(!row_number() %in% c(1, 2, 4))
```

5. Remove rows based on condition

This is the most flexible method, allowing the application of complex logical criteria (using operators like `|` for OR and `&` for AND) across multiple columns simultaneously.

```
df %>%
filter(column1=='A' | column2 > 8)
```

Setting Up the Demonstration Dataset

To effectively illustrate the row removal techniques discussed above, we will utilize a small, representative **data frame** named `df`. This dataset simulates real-world data challenges by including common anomalies, specifically **NA** values (missing data) in the `points` and `assists`

columns, and duplicate entries. We must first ensure the **dplyr** package is loaded into the R session using the `library()` function.

The construction of this **data frame** uses the standard `data.frame()` base R function, defining three variables: `team` (character), `points` (numeric, containing one NA), and `assists` (numeric, containing one NA). Notice that rows 5 and 6, representing Team C, are identical, which will be utilized when demonstrating duplicate removal.

The structure of the initial dataset is shown below. Subsequent examples will reference the row indices (1 through 6) and the presence of missing data and duplicates for clarity.

library(dplyr)

#create data frame
df <- data.frame(team=c('A', 'A', 'B', 'B', 'C', 'C'),
points=c(4, NA, 7, 5, 9, 9),
assists=c(1, 3, 5, NA, 2, 2))

#view data frame
df

team points assists
1 A 4 1
2 A NA 3
3 B 7 5
4 B 5 NA
5 C 9 2
6 C 9 2

Example 1: Eliminating Rows Containing Any Missing Value (NA)

Missing data, represented by **NA** (Not Applicable) in R, can severely complicate statistical modeling and analysis. If an analysis requires complete cases, the simplest approach is to remove any row where at least one variable contains an **NA**. Although `na.omit()` is a base R function, it works seamlessly within the **dplyr** pipeline, offering a concise method for achieving this common data cleaning objective. When `na.omit()` is applied to the **data frame**, it performs a complete case analysis, retaining only those observations that possess non-missing values across all defined columns.

In the context of our `df` dataset, rows 2 (missing `points`) and 4 (missing `assists`) both contain at least one **NA** value. Running the `na.omit()` command effectively removes these two rows from

the resulting dataset. It is important to note that this operation produces a new **data frame**; the original `df` remains unchanged unless the result is explicitly assigned back to `df` or a new object.

This method is highly efficient for general cleaning but should be used cautiously, especially with datasets having many variables and sparse missingness, as it can drastically reduce the sample size. The resulting output clearly shows that only rows 1, 3, 5, and 6--the complete cases--have been preserved.

#remove any row with NA
df %>%
na.omit()

team points assists
1 A 4 1
3 B 7 5
5 C 9 2
6 C 9 2

Example 2: Targeted Removal of Missing Values in Designated Columns

Often, missingness in certain variables is critical, while missingness in other, less important variables can be tolerated. In these situations, removing rows only if they contain **NA** values within a specific set of columns is necessary. The **filter()** function, combined with the base R function `is.na()` and the logical negation operator (`!`), provides the necessary precision for this targeted removal. The `is.na()` function returns `TRUE` for missing values; applying `!` negates this, so the filter only keeps rows where `is.na()` is `FALSE`--meaning the value is present.

Consider the scenario where we must ensure all observations have a recorded value for `points`, but we can accept missing values in the `assists` column. The command `filter(!is.na(points))` checks the `points` column for missingness. Row 2 in the original `df` contains `NA` for `points`, thus `is.na(points)` evaluates to `TRUE` for that row. The negation operator `!` turns this into `FALSE`, resulting in the exclusion of row 2.

Crucially, row 4, which had a missing value in `assists` but a valid value (5) in `points`, is retained in the filtered output. This distinction highlights the power of **filter()** for conditional data cleaning. When reviewing the output, we observe that row 2 (Team A, NA points) is gone, but the row containing NA assists (now row 3 in the output) remains because its `points` value was valid.

#remove any row with NA in 'points' column:
df %>%

filter(!is.na(points))

team points assists
1 A 4 1
2 B 7 5
3 B 5 NA
4 C 9 2
5 C 9 2

Example 3: Identifying and Removing Entirely Duplicate Observations

Data collection errors or faulty merges can often introduce redundant observations into a **data frame**. Identifying and removing duplicate rows is a critical step in ensuring that statistical analyses are not biased by inflated sample sizes or repeated information. The **distinct()** function from **dplyr** is the most straightforward tool for this purpose. When used without any arguments inside the parentheses, `distinct()` operates on all columns, retaining only the first instance of a unique row combination and discarding all subsequent duplicates.

In our working example, the original `df` contains two identical rows for Team C (rows 5 and 6): {'C', 9, 2}. When `distinct()` is applied, it processes the dataset and recognizes that rows 5 and 6 are perfect matches across all three columns (`team`, `points`, and `assists`). It preserves the first occurrence (row 5) and discards the second (row 6), effectively removing the redundancy.

It is important to understand that `distinct()` can also be applied to a subset of columns if the goal is to ensure uniqueness based on key identifiers, rather than the entire row content. For instance, using `df %>% distinct(team)` would retain only one row per team, regardless of the `points` or `assists` values. However, using `distinct()` alone, as demonstrated here, ensures that every remaining row represents a unique combination of all variable values. The resulting **data frame** contains five rows, having successfully dropped the sixth duplicate row.

#remove duplicate rows
df %>%
distinct()

team points assists
1 A 4 1
2 A NA 3
3 B 7 5
4 B 5 NA
5 C 9 2

Example 4: Excluding Observations Based on Positional Index

While most data manipulation focuses on conditional removal (based on values), there are instances where specific rows must be excluded simply because of their sequential position within the dataset. This might be necessary if the first few rows are headers or metadata, or if certain observations were flagged manually for exclusion based on visual inspection. The **dplyr** function `row_number()` provides the current row index within the **data frame**, allowing us to perform positional filtering using `filter()`.

To exclude multiple specific row indices, we combine `row_number()` with the logical operator `%in%` and the negation operator (`!`). The expression `row_number() %in% c(1, 2, 4)` generates a logical vector that is `TRUE` for rows 1, 2, and 4. Applying the negation operator (`!`) reverses this, ensuring that only rows for which the index is **not** 1, 2, or 4 are retained. This means that rows 3, 5, and 6 will be kept in the resulting dataset.

This method is crucial for precise control over observation removal based purely on order. However, caution is advised when using positional filtering in workflows that involve prior sorting or rearrangement, as the index mapping might change. In our demonstration, the code successfully removes rows 1 (A, 4, 1), 2 (A, NA, 3), and 4 (B, 5, NA), leaving the remaining three rows. Notice that the resulting row indices are automatically reset starting from 1 in the output view.

#remove rows 1, 2, and 4
df %>%
filter(!row_number() %in% c(1, 2, 4))

team points assists
1 B 7 5
2 C 9 2
3 C 9 2

Example 5: Conditional Exclusion Using Complex Logical Criteria

The true power of the **filter()** function lies in its ability to handle complex logical conditions spanning multiple variables. This method allows analysts to define precise criteria for inclusion, thereby implicitly excluding any rows that fail to meet these standards. We utilize standard logical operators in R, such as the comparison operator (== for exact match) and the logical OR operator (|), or the logical AND operator (&).

In this example, our goal is to retain rows that meet at least one of two conditions: either the `team` is 'A', or the `points` value is greater than 8. If a row satisfies either condition, it is kept; if it satisfies

neither, it is removed. The crucial aspect here is remembering that **dplyr's** `filter()` keeps rows that return `TRUE` for the combined logical expression.

Applying this logic to our dataset: rows 1 and 2 are kept because `team == 'A'` is `TRUE`. Rows 5 and 6 (Team C) are kept because `points > 8` is `TRUE` (9 > 8). Rows 3 and 4 (Team B) satisfy neither condition (Team is 'B', Points are 7 and 5, neither > 8), and are therefore removed. Note that even the row with missing `points` (row 2 of the original df) is retained because the first condition (`team == 'A'`) was met.

The resulting **data frame** demonstrates the successful execution of this compound filter. It is common practice when removing rows based on complex criteria to first define the criteria for inclusion, and then use the resulting dataset for further analysis.

#only keep rows where team is equal to 'A' or points is greater than 8
df %>%
filter(column1=='A' | column2 > 8)

team points assists
1 A 4 1
2 A NA 3
3 C 9 2
4 C 9 2

Summary and Best Practices for Data Wrangling

Effective data cleaning relies on understanding the context of the data and applying the appropriate removal technique. Whether dealing with missing values using `na.omit()` or `filter(!is.na())`, ensuring uniqueness with **distinct()**, or applying highly specific logical tests via **filter()**, **dplyr** provides the efficient, expressive tools necessary for high-quality data preparation in R. Always remember that data manipulation functions in the tidyverse generally return new data objects, ensuring that the original data structure remains protected unless explicitly overwritten.

For advanced data wrangling, exploring other common **dplyr** verbs--such as `select()` for column manipulation, `mutate()` for variable creation, and `group_by()` paired with `summarize()` for aggregations--will further enhance your ability to preprocess complex datasets efficiently.

The following tutorials explain how to perform other common functions in **dplyr**: