How do I remove rows with some or all NAs in R?

Authored by

stats writer

December 23, 2025

Handling missing data is one of the most crucial steps in any data cleaning and preparation pipeline. In the statistical programming environment known as R, missing data is represented by the special value, NA (Not Available). When working with structured datasets, particularly a data frame, analysts frequently encounter scenarios where they must eliminate entire rows that contain one or more of these missing values. While row deletion is often considered a last resort, it is a necessary procedure when observations are incomplete, especially if the data is assumed to be missing completely at random (MCAR).

There are several robust methods available in R for addressing this common challenge. The primary built-in function is na.omit(), which offers a straightforward way to return an object stripped of any rows containing NAs. It is vital to remember that this function returns a new object, meaning you must explicitly assign the result to a new variable or overwrite the original data structure if you wish to persist the changes. Furthermore, for greater control and precision, R provides the powerful logical function complete.cases(), which identifies rows that are entirely free of NAs, enabling fine-tuned subsetting using bracket notation or the `subset()` function.

This tutorial aims to provide an expert guide on effectively removing rows with some or all NA values, utilizing both the foundational methods available in **Base R** and the efficient, modern approach offered by the tidyr package, a key component of the Tidyverse ecosystem. Understanding these techniques is fundamental for ensuring the integrity and usability of your datasets prior to statistical modeling or visualization. We will explore each method in detail, providing clear code examples to illustrate implementation and output interpretation.

Understanding NA Values and the Sample Data

Before diving into the methods for removal, it is essential to appreciate what NAs represent in the context of R programming. NAs signify that a value is missing or undetermined. Unlike zero or an empty string, NA is a placeholder for data that should exist but does not or was not recorded. Ignoring these values can lead to inaccurate summary statistics, biased model coefficients, and errors in analysis, as many R functions default to returning NA if any NA is present in the input. Therefore, preprocessing to handle these entries is a necessary precursor to reliable statistical work.

Our focus in this guide is on listwise deletion, which is the process of removing any observation (row) where data is absent in one or more variables (columns). This approach is simplest but can lead to a significant loss of statistical power if the proportion of missing data is high. To demonstrate the various techniques discussed, we will utilize a small, representative data frame containing intentional missing entries across multiple columns. This structure allows us to observe how different functions handle rows with partial versus total missingness.

The following code block generates the sample data frame, `df`, which simulates typical sports

statistics where certain metrics (points, assists, rebounds) might be unavailable for specific players or games. Note the strategic placement of NAs: Row 2 is completely missing, Row 4 is missing an assist value, and Row 5 is missing a rebound value.

#create data frame with some missing values
df <- data.frame(points = c(12, NA, 19, 22, 32),
assists = c(4, NA, 3, NA, 5),
rebounds = c(5, NA, 7, 12, NA))

#view data frame
df

points assists rebounds
1 12 4 5
2 NA NA NA
3 19 3 7
4 22 NA 12
5 32 5 NA

As observed in the output, only two rows (Row 1 and Row 3) are completely intact. Our goal is to develop methods that efficiently isolate and retain only these complete observations, or selectively remove NAs based on specific column criteria. The subsequent sections will detail the code required to achieve these data manipulation outcomes using both Base R and the tidyr package.

Base R Simplification: Utilizing na.omit()

The simplest and often fastest way to perform listwise deletion in R is through the `na.omit()` function, which is part of the standard Base R distribution. This function is incredibly convenient because it requires minimal input--simply the name of the data structure you wish to clean--and it automatically scans all columns for any missingness. If an observation contains even a single NA value, the entire row is discarded from the resulting output.

The primary mechanism behind `na.omit()` is its ability to identify and exclude incomplete cases across the dataset. While its ease of use is a major advantage, analysts should be cautious, as this global approach offers no opportunity for handling missing values differently based on the column context. For instance, if a missing value in one column is deemed harmless but missingness in another column is critical, `na.omit()` treats both scenarios identically, potentially leading to unnecessary data loss.

To use `na.omit()` on our sample data frame, `df`, the command is straightforward. The resulting data frame will only contain observations 1 and 3, as observations 2, 4, and 5 all contained at least

one NA value. If you plan to continue working with the cleaned data, remember to store the output, perhaps by assigning it to a new variable like `df_clean <- na.omit(df)`, thereby preserving the original structure for reference or alternative processing methods.

Advanced Base R Technique: Leveraging complete.cases()

For data cleaning scenarios demanding granular control, the complete.cases() function provides a powerful and flexible Base R alternative. Instead of directly deleting rows, `complete.cases()` returns a logical vector--a series of `TRUE` and `FALSE` values--indicating whether each row in the specified data frame or vector is entirely free of NAs. A value of `TRUE` signifies a complete row, while `FALSE` marks an incomplete row containing one or more NAs.

This logical vector is then perfectly suited for use in R's subsetting syntax (square brackets). By passing the logical vector generated by `complete.cases(df)` into the row index position of the data frame, we effectively filter the data, retaining only those rows marked as `TRUE` (i.e., rows with no missing values in any column). This approach is often preferred by experienced R users because it makes the filtering mechanism explicit and readable, demonstrating exactly how the subsetting criteria are being applied.

The following code snippet demonstrates how to use `complete.cases()` to achieve the same listwise deletion result as `na.omit()`, but with greater transparency in the underlying mechanism. The function checks all variables in `df` and generates a logical index, which is then used to select only the non-missing rows. This results in the removal of any row exhibiting a missing value in *any* column, leaving us with only the two complete observations.

#remove all rows with a missing value in *any* column
df

points assists rebounds
1 12 4 5
3 19 3 7

Refining NA Removal in Base R: Targeting Specific Columns

One of the major advantages of using `complete.cases()` over the simpler `na.omit()` function is the ability to restrict the check for missingness to a specified subset of columns. In real-world data analysis, missing data in ancillary variables might be tolerable, whereas missingness in core variables (like an outcome measure or a primary predictor) might necessitate row deletion. By targeting specific columns, we avoid discarding valuable observations simply because data is absent in a less critical field.

To implement this targeted removal, we supply `complete.cases()` not with the entire data frame, but with a subset of the data frame containing only the columns of interest. The logical vector returned will only flag a row as `FALSE` if an NA exists within the selected columns. This ensures that observations that are missing data only in unselected columns are retained in the final output, thereby minimizing unnecessary data loss and preserving observations that are complete for the crucial analytic variables.

The following examples illustrate this powerful capability. The first example focuses the check exclusively on the third column (`rebounds`). Only rows where the rebound value is missing (Row 2 and Row 5) are identified as incomplete, resulting in the preservation of Row 4, which had a missing value in the `assists` column but a complete value in the targeted `rebounds` column. The second example demonstrates selecting multiple columns by index, showcasing how to impose stricter completeness requirements across a critical set of variables (Columns 1 and 3, `points` and `rebounds`).

#remove all rows with a missing value in the *third* column
df),]

points assists rebounds
1 12 4 5
3 19 3 7
4 22 NA 12

#remove all rows with a missing value in either the *first* or *third* column
df),]

points assists rebounds
1 12 4 5
3 19 3 7
4 22 NA 12

Modern Data Wrangling with tidyr: Introducing drop_na()

While Base R methods like `na.omit()` and `complete.cases()` are effective, the R community has increasingly adopted the Tidyverse suite of packages for data manipulation due to its emphasis on readability and chained operations using the pipe operator (`%>%`). Within the Tidyverse, the tidyr package provides `drop_na()`, a dedicated function specifically designed for row deletion based on missing values.

The `drop_na()` function is conceptually equivalent to `na.omit()`, but it integrates seamlessly into a piped workflow, making it a highly preferred choice for modern data preparation scripts. Its

syntax is clean, intuitive, and designed to improve the flow of data cleaning operations. By using `drop_na()`, analysts can easily string together steps like grouping, filtering, selecting, and dropping missing data in a single, coherent sequence of commands, which drastically improves script maintainability and comprehension.

To utilize `drop_na()`, the tidyr package must first be loaded into the R session using `library(tidyr)`. When called without any arguments, `drop_na()` defaults to examining *all* columns in the data frame, mimicking the behavior of `na.omit()` and `complete.cases(df)`. The following example demonstrates this implementation, confirming that only the two complete rows are retained after the function executes within the piped structure.

#load tidyr package
library(tidyr)

#remove all rows with a missing value in *any* column
df %>% drop_na()

points assists rebounds
1 12 4 5
3 19 3 7

tidyr::drop_na() for Global and Targeted Removal

Just like `complete.cases()` allows for column selection, `drop_na()` also supports targeted removal, allowing the user to specify exactly which columns should be checked for the presence of NAs. This capability is managed by simply listing the column names as arguments within the function call. If a row is found to be missing data only in columns that were *not* specified in the `drop_na()` argument list, that row will be preserved. This feature underscores the function's utility in complex data cleaning pipelines where variable importance dictates data retention strategy.

The use of column names directly within `drop_na()` is generally considered more readable than indexing columns by position (e.g., `df`) as is often required in Base R subsetting. This consistency with the data dictionary makes the code self-documenting. If columns are renamed or reordered, the `drop_na()` operation remains robust, provided the column names themselves remain consistent, thus offering better long-term stability for scripts.

As illustrated below, if we only provide the column name `rebounds` to the function, the function restricts its check solely to that column. Since Row 4 has a complete value for `rebounds` (despite missing an `assists` value), it is retained. This demonstrates how efficiently `drop_na()` handles targeted listwise deletion, ensuring that data loss is minimized based on the analytic requirements

tied to specific variables.

#load tidyr package
library(tidyr)

#remove all rows with a missing value in the *third* column
df %>% drop_na(rebounds)

points assists rebounds
1 12 4 5
3 19 3 7
4 22 NA 12

Comparison of Methods and Performance Considerations

When choosing the appropriate method for NA removal, analysts typically weigh simplicity, performance, and integration into existing code structures. For simple, quick listwise deletion on small to medium-sized datasets where the entire environment relies on Base R, `na.omit()` remains the fastest and most concise option. It requires the least typing and executes very efficiently, making it suitable for quick cleaning tasks.

However, when flexibility is paramount, such as needing to perform targeted deletion or integrating the operation into a complex indexing scheme, `complete.cases()` combined with subsetting provides maximum control. This method is slightly more verbose but gives the user explicit control over row selection through the logical vector, which can also be stored and reused for other filtering tasks, thus enhancing its utility beyond simple NA removal.

For modern R projects built around the Tidyverse philosophy, `drop_na()` is the recommended standard. While its performance on extremely large datasets might occasionally be marginally slower than optimized Base R functions due to package overhead, its benefits in terms of code readability, maintainability, and seamless integration with the pipe operator (`%>%`) far outweigh minor performance differences for most analytical tasks. The ability to specify columns by name further solidifies its status as the contemporary best practice for data wrangling.

Best Practices and Alternatives to Deletion

While deleting rows with missing values is the most straightforward handling method, it is crucial to recognize that listwise deletion should often be treated as a last resort. The primary drawback is the potential for bias if the data is not missing completely at random (MCAR). If the probability of missingness depends on the value of the missing data itself, deletion introduces systematic bias into the remaining sample, potentially skewing statistical inferences and model training.

Therefore, before executing any drop operation, analysts should first quantify the extent of missingness. If the data loss is minimal (e.g., less than 5% of observations), deletion is often acceptable. If data loss is substantial, more sophisticated techniques should be considered. The most common alternative is data imputation, which involves estimating and filling in the missing values based on existing data patterns. R offers many robust imputation packages, such as `mice` or functions like `tidyr::replace_na()`.

In conclusion, whether you opt for the simplicity of na.omit(), the precision of `complete.cases()`, or the workflow integration of `drop_na()`, the choice depends heavily on the specific analytical goals, the scale of the data frame, and the overall coding environment. Mastering these techniques ensures that your data preparation process is both clean and highly controllable, leading to more reliable downstream analysis.

You can find more detailed R tutorials and guides for advanced statistical analysis on our dedicated resources page.