Learning R: Selecting the Top N Rows with dplyr’s top_n() Function

Authored by

Mohammed looti

November 13, 2025

Introduction & The Role of top_n()

In the expansive realm of R programming and sophisticated data manipulation, analysts are perpetually challenged with efficiently managing and summarizing massive datasets. A common and crucial requirement is the ability to subset these large collections of observations by zeroing in on the rows that represent the extremes--either the highest or lowest values--within a specific quantitative column. This process, known as selecting the **top N observations**, is fundamental for myriad analytical tasks, including identifying market leaders, pinpointing critical performance metrics, or studying anomalous outliers that require focused attention.

Achieving such complex data wrangling tasks in R is streamlined tremendously through the utilization of the **dplyr** package, which is arguably the cornerstone of modern data science workflows in the R ecosystem. This powerful framework provides a grammar of data manipulation that is both intuitive and highly performant. Within this suite of functions, the **top_n()** function stands out as the dedicated tool engineered to handle this exact requirement: retrieving the leading (or trailing) rows of a data frame based on a designated weighting variable, offering a syntax that is remarkably clean and conducive to pipeline operations.

This guide will explore the mechanics, syntax, and advanced applications of the **top_n()** function, demonstrating how it serves as an indispensable utility for analysts seeking to extract meaningful insights rapidly. By mastering this function, users can circumvent the complexities associated with manual filtering, sorting, and indexing large tables, thereby significantly enhancing the efficiency and clarity of their analytical code, which is essential for promoting reproducible research.

The Fundamental Need for Data Subsetting

Effective data analysis fundamentally revolves around strategic prioritization and focus. Datasets often contain thousands or even millions of rows, but the most salient conclusions frequently rely on examining a small, high-impact subset of that data. Consider a scenario involving quality control in manufacturing: while all units produced are logged, the quality assurance team primarily needs to analyze the top 5% of units that exhibited the highest defect scores or, conversely, the bottom 10% that required the least maintenance. Without a precise method for extracting these specific observations, the analytical effort becomes diluted and inefficient.

The traditional approach to finding the top N rows--involving a combination of `order()` or `sort()` followed by manual indexing and subsetting--is cumbersome, prone to human error, and lacks the readability required for collaborative projects. The **top_n()** function provides a standardized, single-step solution that abstracts away this underlying complexity. For instance, if an inventory manager needs to identify the top 20 products contributing the most to quarterly profit, applying **top_n()** to the profit column yields the results immediately, ensuring that the focus remains on the interpretation of the data rather than the mechanics of retrieval.

The core principle driving **top_n()** is its ability to perform the necessary sorting and filtering internally, returning only a specified subset of rows. This behavior is distinct from simple sorting, which merely reorders the entire dataset without reducing its size. Because **top_n()** integrates seamlessly into the **dplyr** pipeline structure (using the pipe operator `%>%`), it allows analysts to chain multiple data transformation steps together in a logical, flow-based manner. Before any practical application can commence, the user must ensure that the **dplyr** package is properly installed and loaded into the active R session, confirming its functions are accessible globally.

Installing and Understanding the top_n() Syntax

As **top_n()** is a component of the **dplyr** package, it is not available in the base installation of R. The initial prerequisite for utilizing this powerful function is ensuring that the package is present on the system. If the package has not yet been installed, the standard R package management command must be executed once. Subsequently, the package must be explicitly loaded at the beginning of every R session where the function is required.

The installation process is straightforward and performed using the standard `install.packages()` function. This command fetches the latest stable version of the package from the Comprehensive R Archive Network (CRAN) and places it within the user's R library directory:

install.packages('dplyr')

Once installed, the structure of the **top_n()** function itself is highly structured yet flexible, typically demanding a minimum of two arguments, though a third is often employed to remove any potential ambiguity regarding the ranking criteria. The function is specifically designed to operate within a data pipeline, where the primary data object is passed directly to the function call via the pipe operator.

The generalized syntax for calling **top_n()** is formally defined by three parameters: **top_n(x, n, wt)**. Understanding the role of each parameter is key to successful execution:

x: This argument represents the data object--either a standard **data frame** or a **tibble** (a modernized version of the data frame used by dplyr). When used in a piped sequence (`%>%`), this argument is automatically populated by the output of the preceding command, making the code cleaner as the data argument is implied.
n: This is a crucial numeric value that determines the **number of rows** the function is instructed to return. The sign of this value dictates the direction of the selection: a positive `n` (e.g., `n=10`) selects the largest values (the top), while a negative `n` (e.g., `n=-5`) selects the smallest values (the bottom).
wt: Standing for "weight," this optional but essential parameter defines the specific **variable**

(column) that **top_n()** should use for ordering and ranking the observations. If the `wt` argument is entirely omitted, the function defaults to using the very last column in the provided data object for ranking, a behavior that can sometimes lead to unintended results if the data structure changes.

A critical feature of **top_n()** is its robust handling of ties. The function guarantees that if multiple rows share the exact same value at the cutoff point (the Nth rank value), all tied rows will be included in the final output. This means that the resulting subset might occasionally contain more than N rows, a feature that ensures the selection process is unbiased and comprehensive, preventing the arbitrary exclusion of observations that share the same high (or low) rank.

Practical Application 1: Default Selection and Data Setup

To effectively demonstrate the fundamental utility of **top_n()**, we must first establish a reproducible sample dataset. For this illustration, we will construct a small **data frame** containing hypothetical performance statistics for a group of basketball players. This dataset, which we will name `df`, includes categorical data (team affiliation) and several quantitative metrics that can serve as weighting variables for ranking: `points`, `assists`, and `rebounds`.

The following code block executes the creation of the sample data frame and displays its contents. Note that the structure places the `rebounds` column as the final variable, which is important for demonstrating the default behavior of **top_n()**:

#create data frame
df <- data.frame(team=c('A', 'A', 'A', 'A', 'B', 'B', 'B', 'B'),
points=c(99, 68, 86, 88, 95, 74, 78, 93),
assists=c(22, 28, 31, 35, 34, 45, 28, 31),
rebounds=c(30, 28, 24, 24, 30, 36, 30, 29))

#view data frame
df

team points assists rebounds
1 A 99 22 30
2 A 68 28 28
3 A 86 31 24
4 A 88 35 24
5 B 95 34 30
6 B 74 45 36
7 B 78 28 30
8 B 93 31 29

Now, let us apply **top_n()** to retrieve the top six rows from this dataset. If we choose not to explicitly specify a weighting variable using `wt`, the function automatically reverts to ranking the rows based on the values in the last column. In this setup, the ranking criterion becomes the `rebounds` column. We use the `library(dplyr)` command to load the package and then use the pipe operator (`%>%`) to pass the data frame `df` directly to the **top_n()** function call, specifying only the desired count (`6`):

library(dplyr)

#select top six rows from data frame (defaulting to last column: rebounds)
df %>% top_n(6)

team points assists rebounds
1 A 99 22 30
2 A 68 28 28
3 B 95 34 30
4 B 74 45 36
5 B 78 28 30
6 B 93 31 29

The resulting output confirms that the function successfully identified and returned the six rows corresponding to the highest values in the `rebounds` column (36, 30, 30, 30, 29, 28). While this default mechanism offers a quick way to subset data, analysts are strongly advised to always explicitly specify the `wt` parameter to eliminate reliance on column order, which can change unexpectedly during upstream data transformations.

Practical Application 2: Targeted Selection Using the wt Parameter

In real-world data analysis, reliance on default column selection is rarely acceptable. Precision demands that the analyst explicitly dictates which metric determines the rank. For example, if the objective is to find the top performers based on playmaking ability, the ranking must be based exclusively on the `assists` column, irrespective of where that column is positioned within the **data frame**. This is where the `wt` parameter becomes indispensable.

To achieve this targeted selection, we simply include the `wt` argument in the function call, setting it equal to the desired column name (e.g., `wt=assists`). This action instructs **top_n()** to perform the ranking solely on the values contained within that specified column. Let us now retrieve the top six players based on the highest recorded assists:

library(dplyr)

#select top six rows with highest values in assists column
df %>% top_n(6, wt=assists)

team points assists rebounds
1 A 68 28 28
2 A 86 31 24
3 A 88 35 24
4 B 95 34 30
5 B 74 45 36
6 B 78 28 30
7 B 93 31 29

A crucial observation in the resulting output is that the function returned seven rows, despite the requested parameter being six (`n=6`). This outcome perfectly illustrates the tie-handling mechanism intrinsic to **top_n()**. The sixth highest assist value in the dataset is 28. Since there are two players who recorded 28 assists (row 1 and row 7), both observations are included to ensure that the selection is fair and complete according to the ranking criteria.

Furthermore, the **top_n()** function is not limited to selecting only the maximum values. By supplying a negative integer to the `n` parameter, the function reverses its logic and selects the smallest values, allowing for easy retrieval of the bottom N observations. For example, the command `df %>% top_n(-2, wt=points)` would swiftly return the two players with the lowest total point scores, providing a concise method for identifying underperforming observations or minimum values in a distribution.

Advanced Techniques: Leveraging group_by() for Subgroup Analysis

The true operational efficiency of **top_n()** is unlocked when it is seamlessly integrated with the powerful **dplyr** function `group_by()`. Many complex analytical tasks require conditional subsetting--that is, finding the top N observations not globally across the entire dataset, but locally within predefined categorical subgroups. This technique is indispensable for segmented analysis, such as finding the best-performing employee in each department or the highest-rated product category in each region.

Returning to our basketball data, suppose the goal is to identify the single highest-scoring player (based on `points`) for each individual team (Team A and Team B). To accomplish this, we first use `group_by(team)` to partition the dataset into two distinct analytical groups. The subsequent application of `top_n(1, wt=points)` is then executed independently on each group, yielding one result per team rather than a single global result:

library(dplyr)

#select row with highest value in points column, grouped by team
df %>% group_by(team) %>% top_n(1, wt=points)

A tibble: 2 x 4
Groups: team
team points assists rebounds

1 A 99 22 30
2 B 95 34 30

The output is a summarized tibble containing precisely two rows: the top scorer from Team A (99 points) and the top scorer from Team B (95 points). This methodology exemplifies a highly effective and concise approach to hierarchical data summarization, confirming that the N selections are made locally within the confines of each defined group. This combination of `group_by()` and **top_n()** is a cornerstone technique for any serious R analyst engaging in segmentation and subgroup comparisons.

Modern Alternatives: Transitioning to slice_max() and Conclusion

While **top_n()** remains a fully valid and supported function within the **dplyr** package, it is important for analysts to be aware of the evolution of the package. The maintainers of **dplyr** have introduced newer, more explicit functions--namely `slice_max()` and `slice_min()`--which are recommended for modern code development. These alternatives offer clearer syntax by explicitly stating whether the maximum or minimum values are sought, and they provide more granular control over tie-handling (e.g., specifying whether to keep all ties or only the first N).

Despite the recommendation to transition to `slice_max()` and `slice_min()` for new projects, **top_n()** is deeply embedded in the existing codebase of many organizations and R community scripts. Therefore, understanding its behavior, particularly its default settings and tie-handling logic, is essential for maintaining and interpreting legacy R code. Ultimately, the ability to efficiently subset and rank data remains a fundamental skill, regardless of the specific function used.

To ensure that the most current and optimized methods are applied to data manipulation tasks, analysts should habitually consult the official documentation provided by the package authors. This documentation offers comprehensive details regarding all parameters, edge case handling (such as dealing with missing values, or `NA`s), and the very latest updates on data slicing in R.

Mastering functions like **top_n()** provides the necessary toolkit for transforming raw data into actionable insights, reinforcing R's status as the premier environment for statistical computing and

data science.

<!--

Featured Posts

-->