Learning to Select Rows with Minimum Values Using dplyr's `slice_min()` Function in R

Authored by

Mohammed looti

November 13, 2025

Mastering Data Subset Selection with slice_min() in R's dplyr Package

In the dynamic field of **data science** and statistical computing, the R programming language remains an essential tool for sophisticated data manipulation and analysis. Analysts frequently encounter the requirement to identify and isolate specific records based on extreme values--a task that involves pinpointing the rows associated with the lowest or highest numerical entries in a given column. While core R functionality offers methods to address this, the contemporary and highly efficient approach, championed by the modern dplyr package, provides solutions that are both highly readable and computationally effective for these common filtering tasks.

The core challenge in such data handling is not simply calculating the minimum value of a variable, but rather retrieving the entire associated observation, or row, from the underlying data frame. For example, a quality control team might need to locate the batch with the lowest recorded impurity level, or a financial analyst might seek the transaction with the smallest recorded fee. These operations demand robust and clear filtering tools that uphold data integrity while ensuring optimal processing speed. The **dplyr** package, an integral part of the larger Tidyverse ecosystem, directly addresses this need through a specialized family of slicing functions designed for positional subsetting.

Specifically, when the objective is to extract the single row or multiple rows corresponding to the absolute smallest values within a chosen numeric variable, the function of choice is slice_min(). This powerful utility is purpose-built to streamline the process of subsetting data based on rank order. It offers a significant advantage in terms of conciseness and clarity when compared to relying on complex indexing mechanisms or traditional conditional filtering methods. By integrating slice_min() into their workflow, analysts can quickly focus on observations representing the lower bounds of a distribution, a process invaluable for tasks such as outlier identification, performance benchmarking, or identifying minimum resource consumption within complex datasets.

Deconstructing the slice_min() Function and Its Essential Syntax

The **slice_min()** function serves as a foundational element within the data transformation toolkit provided by the dplyr library. Its mechanism involves evaluating a specified column, internally sorting the data based on that column's values, and subsequently returning only the rows that occupy the lowest positions in the resulting ordered list. This behavior fundamentally differentiates it from functions like `filter()`, which rely on logical conditions (e.g., value > 10); **slice_min()** focuses purely on positional ranking relative to the provided metric.

To utilize this tool effectively, a clear understanding of its fundamental syntax is essential. The function's design adheres closely to the typical Tidyverse structure: it accepts the data object first, followed by the variable used for ordering, and then optional parameters that allow for fine-tuning the selection process. This intuitive structure is particularly advantageous when constructing

sophisticated data workflows by chaining operations together using the pipe operator (`%>%`), enabling sequential and highly logical data processing scripts.

The basic structure of the **slice_min()** function requires three primary arguments, although standard usage often necessitates specifying only the first two. This structured approach ensures clarity and reduces the potential for coding errors. The formal syntax is formally defined as follows:

slice_min(.data, order_by, n, ...)

Each component plays a distinct role in governing the function's execution:

.data: This parameter specifies the input object, which is typically an R data frame or a tibble, upon which the slicing operation will be performed.
order_by: This is the critical variable (column name) or expression that establishes the basis for ordering and subsequent selection. The function identifies the smallest values within this designated column.
n: This is an optional yet highly powerful argument that dictates the precise number of rows to be returned. If this argument is omitted entirely, slice_min() defaults to returning only the single row (or all rows, in case of ties) corresponding to the absolute minimum value observed in the **order_by** column across the entire dataset.

Demonstration: Selecting the Absolute Minimum Observation

To effectively illustrate the simplicity and power inherent in the slice_min() function, we will commence with a straightforward, common data manipulation scenario. Imagine we have assembled performance statistics for various athletes and organized this information into a structured R data frame. Our immediate objective is to isolate the single observation--the specific player row--that recorded the lowest value in a chosen performance metric, such as the number of assists.

We first establish the foundational sample data frame. This synthetic dataset includes critical information such as player team affiliations, total points scored, the number of assists recorded, and rebounds collected. Creating this clear, reproducible foundation is essential for executing and visualizing the precise impact of the **slice_min()** function within a typical data analysis context.

```
#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, 45, 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 45 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

If our goal is specifically to determine which observation corresponds to the minimum number of assists recorded across the entire dataset, the process is streamlined by chaining the **slice_min()** function directly to the data frame using the pipe operator, specifying the column `assists` as the ordering variable. It is important to note that because the crucial `n` argument is intentionally omitted in this instance, the function automatically defaults to seeking out only the single lowest value, returning the corresponding row or rows if ties exist at the absolute minimum.

library(dplyr)

#select row with smallest value in assists column
df %>% slice_min(assists)

team points assists rebounds
1 A 99 22 30

The resulting output confirms that the row where the value in the `assists` column is **22** is the absolute minimum observation within the entire dataset. This operation is fundamentally cleaner, more readable, and far less error-prone than attempting to achieve the same result using equivalent base R methods that typically involve complex interactions between functions like `which.min()` and subsetting brackets, reinforcing the preference for the **dplyr** approach in contemporary R programming workflows.

Customizing Selection: Utilizing the n Argument for Subsets

While identifying the single, absolute minimum observation is certainly valuable, advanced data analysis frequently necessitates retrieving a precisely defined subset of the smallest observations--for example, isolating the bottom 10 percent of recorded sales figures or retrieving the three lowest measured temperatures. This is precisely where the critical flexibility provided by the **n** argument

within **slice_min()** becomes truly indispensable. By explicitly assigning a numerical value to **n**, we provide a direct instruction to the function to return exactly that number of rows corresponding to the smallest values found in the specified metric column.

Continuing with our basketball player data frame example, let us hypothesize that we need to identify the players who recorded the five lowest values in the `assists` column. To accomplish this specific task, we simply pass the value `n=5` into the function call. A key advantage of the dplyr framework is its internal efficiency: it handles all necessary internal sorting without requiring the user to manually sort the entire data frame beforehand. This dramatically simplifies the required syntax and enhances execution speed.

The following code implementation clearly demonstrates the technique for selecting the five observations that possess the lowest assist counts. The function efficiently identifies the five smallest values (22, 28, 28, 34, 31, if we consider the unique values and ties) and returns the full corresponding rows, thereby ensuring the complete context of those specific observations is maintained for subsequent analysis.

library(dplyr)

#select rows with 5 smallest values in assists column
df %>% slice_min(assists, n=5)

team points assists rebounds
1 A 99 22 30
2 A 68 28 28
3 B 78 28 30
4 A 88 35 24
5 B 93 31 29

A crucial and often-overlooked aspect of the **n** argument is its default interaction with tied values, a behavior designed for comprehensive data retrieval. If the value specified for **n** causes the selection boundary to fall on a tie (meaning the nth smallest value is shared by multiple rows), **slice_min()**, by default, returns all tied rows. This feature is important because it prevents the accidental exclusion of relevant data points simply because they share a minimum metric. While this default behavior might result in returning slightly more than **n** rows, it ensures analytical completeness. For specialized scenarios where an analyst strictly requires only **n** rows, the function provides additional arguments, such as `with_ties = FALSE`, allowing for explicit control over how ties are resolved.

Addressing Data Integrity: The Importance of Tie Handling

When utilizing ranking and slicing functions like **slice_min()**, one of the most critical design considerations is how the function manages observations that possess identical values, commonly referred to as ties. The default behavior of slice_min() in dplyr is specifically implemented to be robust and comprehensive: if multiple rows are tied for the minimum value (or tied at the positional boundary established by the **n** argument), all of the tied rows are automatically included in the final result set. This proactive measure is essential for preventing the arbitrary exclusion of relevant data points that share the minimum observed metric.

To clearly demonstrate this default tie-handling mechanism, we will examine the `rebounds` column within our sample data frame. A quick visual inspection of the data reveals that the minimum value for rebounds is **24**, and this value is present in two distinct rows (rows 3 and 4). If we execute the function requesting only the single absolute minimum row (by choosing to omit the **n** argument), the function correctly identifies and returns both tied observations, providing a complete record of the minimum performance.

We run the following code to select the row corresponding to the smallest value in the `rebounds` column:

library(dplyr)

#select row with smallest value in rebounds column
df %>% slice_min(rebounds)

team points assists rebounds
1 A 86 31 24
2 A 88 35 24

As anticipated, the resulting output includes two rows, both registering **24** rebounds. This automatic inclusion of ties ensures that no critical information is lost and presents a complete analytical picture of all observations that achieved the minimum metric. Analysts who rely on the **slice_min()** function must consistently bear this default tie-handling mechanism in mind, particularly when they are expecting a fixed number of output rows. If, in contrast, an analyst absolutely requires only a single observation (or exactly **n** rows) and is willing to accept an arbitrary selection among tied values, they have the option to employ the `with_ties = FALSE` argument. However, for the majority of scientific, statistical, and business applications, preserving all tied observations remains the recommended best practice to maintain robust statistical integrity. This commitment to handling ties gracefully is a defining feature that significantly elevates the utility of **slice_min()** above many less sophisticated ranking methods available within the R programming

language environment.

Hierarchical Analysis: Combining slice_min() with group_by()

The true analytical potential of dplyr functions is fully unlocked when they are seamlessly combined with the `group_by()` function. While **slice_min()**, when used alone, determines the overall minimum across the entire data frame, pairing it with `group_by()` enables the calculation and retrieval of minimums *independently within each subgroup*. This capability addresses an exceptionally common requirement in real-world data analysis, such as identifying the lowest recorded cost product for every defined market segment or determining the minimum waiting time for customers in each regional call center.

Returning to our basketball scenario, let us assume we need to locate the player who recorded the lowest point score (minimum `points`) specifically within the context of each distinct `team` (A and B). The solution involves a two-step process: we first group the data frame based on the `team` variable, and then we apply **slice_min()** to the `points` column. The function is designed to be context-aware, executing the slicing operation independently and accurately within the bounds of each defined group.

The sequential logic facilitated by the pipe operator is highly intuitive and easy to audit: first, aggregate the data by group, and subsequently, slice the minimum value from the grouping variable. This powerful technique significantly simplifies complex hierarchical queries that would traditionally demand multiple steps, manual splits, or cumbersome conditional loops in base R.

library(dplyr)

Find the player with the minimum points scored for each team
df %>%
group_by(team) %>%
slice_min(points)

Results:
Team A minimum points: 68 (Row 2)
Team B minimum points: 74 (Row 6)

team points assists rebounds
1 A 68 28 28
2 B 74 45 36

The resulting output accurately presents one row for Team A (points=68) and one row for Team B (points=74), correctly representing the minimum point total achieved within their respective teams.

This grouped slicing operation profoundly demonstrates the efficiency and analytical flexibility that **slice_min()** contributes to the R programming language ecosystem, establishing it as a cornerstone for both grouped summary and sophisticated filtering tasks.

Conclusion and Overview of Related Slicing Functions

The **slice_min()** function delivers a powerful, concise, and highly readable solution for the essential task of identifying and retrieving rows corresponding to the smallest values in a specified column within an R data frame. Its ability to manage ties gracefully through inclusive defaults and its seamless integration with the `group_by()` function cement its position as an absolutely essential tool in any data analyst's dplyr toolkit. By gaining proficiency with its straightforward syntax, users can dramatically enhance the efficiency, clarity, and reproducibility of their data manipulation scripts.

It is important for users to recognize that slice_min() belongs to a broader family of related slicing functions within dplyr, all designed for subsetting data based on positional or ranked criteria:

slice_max(): Performs the exact inverse operation to **slice_min()**, selecting rows that correspond to the largest values observed in the specified column.
slice_head(): Selects the initial **n** rows of a data frame (highly useful for quick dataset inspection).
slice_tail(): Selects the final **n** rows of a data frame.
slice_sample(): Selects random rows from the data frame, useful for bootstrapping or quick sampling.

Familiarity with this comprehensive set of related functions empowers data scientists to choose the most appropriate and optimized tool for virtually any given subsetting task, ensuring that the resulting code is consistently clean, efficient, and reproducible. For those seeking exhaustive details on all parameters, tie-handling options, and advanced uses, consulting the complete official documentation for the **slice_min()** function within the **dplyr** package is strongly recommended.

Note: You can find the complete documentation for the **slice_min()** function from the **dplyr** package here.

Additional Resources for R Data Manipulation Mastery

To further expand your expertise in powerful data manipulation techniques using the R programming language and the Tidyverse ecosystem, consider engaging with tutorials on these related common tasks:

Filtering Data based on Multiple Logical Conditions in R
Using `mutate()` for Efficient Column Creation and Data Transformation

Summarizing Data Effectively using the `summarise()` function in conjunction with `group_by()`

These advanced concepts build logically upon the foundational skills established by mastering slicing functions and are crucial for developing fully optimized, end-to-end data analysis workflows.