# How to find the max value across multiple columns in R?

Authored by

**stats writer**

December 10, 2025

In the world of statistical computing and data analysis using the R programming language, a frequent requirement involves calculating summary statistics not just for entire columns, but across specific subsets of columns on a row-by-row basis. Identifying the maximum value across multiple variables for each observation (or row) is a fundamental technique for tasks ranging from performance scoring to quality control. Historically, solutions often involved looping or generalized functions like the apply() function.

The apply() function is highly versatile, designed to operate on the margins (rows or columns) of a matrix or array. It accepts the data structure, a margin argument (1 for rows, 2 for columns), and a function (like `max()`) to execute. While effective for column-wise operations, utilizing `apply()` for row-wise maximums across only selected columns often requires subsetting the data first. However, for efficiently comparing corresponding elements from multiple vectors or columns directly, R provides a specialized, vectorized solution: the `pmax()` function. This tutorial focuses on leveraging the power of pmax() function to rapidly determine the maximum value among several specified columns within a data frame.

## The Role of pmax() in Row-Wise Comparisons

When analyzing quantitative data, researchers frequently need to establish a benchmark or find the peak measurement achieved by a single entity across various metrics. For example, in sports analytics, we might want to know the highest statistical contribution (points, rebounds, or assists) a player made in a specific game. This row-wise comparison is critical for summarizing performance where individual columns represent comparable but distinct variables. The standard `max()` function, when applied to a vector, returns a single scalar value: the overall maximum of that vector. In contrast, `pmax()` performs parallel maximum calculations, returning a vector of maximums derived from comparing elements at the same position across all input vectors. This fundamental difference makes `pmax()` the optimal tool for solving our specific problem.

The pmax() function stands out because it is highly optimized for element-wise operations, offering superior performance compared to manually iterating through rows or using less specialized functions. It takes two or more vectors as input and returns a resulting vector whose elements are the largest values found at the corresponding positions across all input vectors. This vectorized approach is central to efficient data processing in the R programming language, promoting concise and readable code while maximizing computational speed, especially when dealing with large datasets.

We can use the **pmax()** function to find the maximum value across multiple columns in an R data frame. Its syntax is straightforward, primarily requiring the columns (vectors) you wish to compare. Understanding the arguments is essential for handling real-world data, particularly concerning missing values, which are ubiquitous in data analysis.

## Syntax and Key Arguments of pmax()

The structure of the `pmax()` function is simple and elegant, designed for direct comparison of parallel inputs. It allows for any number of input vectors or column references. The syntax is defined as:

**pmax(..., na.rm = FALSE)**

This structure ensures flexibility. We can pass two column vectors, three, or twenty, and the function will process them efficiently. The result will always be a new vector that aligns perfectly with the number of rows in the original data frame.

The arguments include:

**...** : A list of vectors (typically columns referenced using the `$` notation or list elements from a data frame). These are the vectors whose elements are compared in parallel.
**na.rm:** A logical indicator specifying whether missing values (`NA`) should be removed or ignored during the calculation. The default setting is **FALSE**, meaning that if any element being compared is `NA`, the resulting element in the output vector will also be `NA`. Setting this to **TRUE** is crucial when working with incomplete data, as it allows the function to find the maximum among the non-missing values.

The default behavior where `NA` inputs result in `NA` outputs is a careful design choice in R, ensuring that the presence of missing data is always acknowledged. However, in many practical scenarios, especially when data is known to be partially missing but we still want the maximum of available data points, overriding this default by setting `na.rm = TRUE` is necessary. This tutorial, for simplicity, uses a clean dataset, but users should be aware of the importance of the `na.rm` argument in production environments.

## Setting Up the Demonstration Data Frame

To illustrate the application of pmax() function, we will utilize a sample data frame representing performance statistics for several athletes. This structure allows us to clearly see how row-wise operations compare specific metrics (points, rebounds, assists) for each player. Defining a standard dataset ensures that the results of our code examples are reproducible and easy to verify. We will focus on finding the maximum performance metric for each player across the quantitative columns.

We initialize the data frame below, containing player names and three performance statistics. Note the simple command structure used to generate this foundational data structure in the R

programming language:

```
#create data frame
df <- data.frame(player=c('A', 'B', 'C', 'D', 'E', 'F', 'G'),
points=c(28, 17, 19, 14, 23, 26, 5),
rebounds=c(5, 6, 4, 7, 14, 12, 9),
assists=c(10, 13, 7, 8, 4, 5, 8))

#view DataFrame
df

player points rebounds assists
1 A 28 5 10
2 B 17 6 13
3 C 19 4 7
4 D 14 7 8
5 E 23 14 4
6 F 26 12 5
7 G 5 9 8
```

This data frame, named `df`, serves as the foundation for the subsequent examples. Each row represents a single observation (a player), and we are interested in calculating the peak value achieved between their `points`, `rebounds`, and `assists` scores. This is the classic scenario where a row-wise maximum calculation is required, allowing us to generate new metrics summarizing overall player contribution.

## Example 1: Finding the Max Across Specific Columns

Our first objective is to calculate the maximum value achieved by each player across just two specific statistical categories: `points` and `rebounds`. This calculation generates an output vector that is parallel in length to the original data frame, where each element represents the greater of the two corresponding input values. This is often useful for intermediate calculations before integrating the result back into the main dataset.

The code below demonstrates the direct application of `pmax()` by explicitly referencing the columns of interest using the standard R subsetting notation (`df$columnName`). The result is a simple vector that can be immediately printed to the console or stored in a variable for later use. This approach is highly efficient because it avoids the overhead of iterating through the dataset manually or creating temporary subsets.

The following code shows how to find the max value across the `points` and `rebounds` columns in

our demonstration data frame:

**#find max values in each row across points and rebounds columns**
**pmax(df$points, df$rebounds)**

28 17 19 14 23 26 9

Analyzing the output, we can verify that the function successfully compared the `points` and `rebounds` for each player: Player A had 28 points and 5 rebounds, resulting in a maximum of 28. Player G had 5 points and 9 rebounds, resulting in a maximum of 9. This confirmation highlights the power of the pmax() function in performing parallel comparisons across multiple input vectors, providing the highest value for each row index. This simplicity and speed are why `pmax()` is the preferred method for this specific task in R.

## Example 2: Adding A New Column Containing the Max Value

While viewing the output vector is helpful, the true value often lies in integrating this new calculated metric directly back into the data frame. By creating a new column, we persist the row-wise maximum and make it available for subsequent analysis, sorting, filtering, or visualization tasks. This process is standard practice in data manipulation: calculate a derived variable and append it to the existing structure.

To achieve this, we simply assign the result of the `pmax()` calculation to a new column name within the existing data frame using the assignment operator (`<-`). This operation modifies the data structure in place, ensuring that the derived maximum value is permanently associated with the corresponding player observation. We choose a descriptive name, `max_points_rebs`, to clearly indicate the source of the calculated maximum.

The following code shows how to add a new column to the data frame that contains the maximum value across the `points` and `rebounds` columns:

**#add new column that contains max values across points and rebounds columns**
**df$max_points_rebs <- pmax(df$points, df$rebounds)**

#view data frame
df

player points rebounds assists max_points_rebs
1 A 28 5 10 28
2 B 17 6 13 17
3 C 19 4 7 19

4 D 14 7 8 14
5 E 23 14 4 23
6 F 26 12 5 26
7 G 5 9 8 9

As displayed in the output, the `df` now includes the new column `max_points_rebs`. This column contains the row-wise maximum derived from the comparison of only the `points` and `rebounds` columns. This structured approach is highly readable and ensures data integrity, as the calculated metric is permanently bound to the source data. This transformation is crucial for downstream analysis, such as identifying players whose performance is dominated by scoring or rebounding.

## Example 3: Adding Several New Columns Containing Max Values

Data analysis often requires calculating multiple derived metrics simultaneously. Instead of just finding the maximum between points and rebounds, we might also be interested in the maximum between rebounds and assists, or even the overall maximum across all three variables. The pmax() function easily supports creating multiple derived columns in rapid succession, demonstrating its versatility and efficiency in complex data preparation workflows.

In this example, we perform two distinct row-wise maximum calculations: first between `points` and `rebounds`, and second between `rebounds` and `assists`. Each calculation is assigned to its own unique column name (`max_p_r` and `max_r_a`, respectively). This allows us to capture different comparative insights within the same data frame structure, making the resulting dataset richer for exploration.

The following code shows how to add several new columns to the data frame that contain the max values across different groups of columns:

**#add new column that contains max values across points and rebounds columns**
**df$max_p_r <- pmax(df$points, df$rebounds)**

#add new column that contains max values across rebounds and assists columns
df$max_r_a <- pmax(df$rebounds, df$assists)

#view data frame
df

player points rebounds assists max_p_r max_r_a
1 A 28 5 10 28 10
2 B 17 6 13 17 13
3 C 19 4 7 19 7

4 D 14 7 8 14 8
5 E 23 14 4 23 14
6 F 26 12 5 26 12
7 G 5 9 10 9 10

Observing the final output, Player A's maximum for points/rebounds (`max_p_r`) is 28 (from points), while their maximum for rebounds/assists (`max_r_a`) is 10 (from assists, as 5 rebounds < 10 assists). Player G's `max_r_a` is 10 (from assists, as 5 points < 9 rebounds < 10 assists). This sequential assignment process demonstrates a clean and effective methodology for expanding a data frame with multiple comparative metrics using base R functionality.

## Advanced Techniques: Applying pmax() to Many Columns

While the previous examples demonstrated comparing two columns, data frames often contain dozens or even hundreds of columns where a row-wise maximum might be desired across a large subset. Manually typing out `pmax(df$col1, df$col2, ..., df$coln)` becomes tedious and error-prone. For such situations, we must use more programmatic methods to select and pass vectors to the `pmax()` function.

One common technique in base R programming language involves using the `do.call()` function in conjunction with `pmax()`. First, we identify the specific columns by index or name that need to be compared. Then, we extract those columns into a list structure. Finally, `do.call()` takes the function name (`pmax`) and the list of arguments (the selected column vectors) and executes the function, effectively unpacking the list of vectors directly into the pmax() function. This prevents the need for manual argument listing and scales seamlessly to any number of columns.

Alternatively, users familiar with the `tidyverse` ecosystem may prefer using `dplyr::rowwise()` combined with `c_across()`. The `rowwise()` function fundamentally changes how operations are executed, ensuring that subsequent commands operate on an observation-by-observation basis. Within a `mutate()` call, one can use `c_across(c(points, rebounds, assists))` to select the relevant columns, and then apply the standard `max()` function to this collected vector of values for that specific row. While this approach is often more readable and integrated into modern R workflows, the base R `pmax()` remains the fastest and most direct method when only the maximum of parallel vectors is needed.

## Conclusion: The Efficiency of Vectorized Operations

Finding the maximum value across multiple columns on a row-by-row basis is a foundational data preparation task in R. While generalized functions like the apply() function can be adapted for this purpose, the specialized **pmax()** function provides a highly optimized, vectorized solution that is

both concise and computationally efficient. Its design allows for direct comparison of corresponding elements across multiple input vectors, seamlessly generating a resulting vector of row maximums.

This tutorial demonstrated how to use pmax() function to extract specific row-wise maximums and integrate these new metrics back into the data frame, whether for a single calculation or multiple comparative analyses. Mastering vectorized functions like `pmax()` is key to writing high-performance and scalable code in the R programming language, ensuring your data manipulation workflows are optimized for large datasets and complex analytical requirements.