How to Easily Select Numeric Columns in R with dplyr

Authored by

stats writer

November 28, 2025

The process of data subsetting is fundamental to effective data analysis in R. Analysts often need to focus only on specific types of variables--such as those that contain measurable quantities--while excluding descriptive or categorical features. The dplyr package, a core component of the tidyverse ecosystem, provides highly efficient and readable tools for these manipulations. Historically, analysts relied on functions like `select_if()` to manage column selection based on data type. While this approach was functional, the **tidyverse** community has consolidated and improved this functionality, leading to a more streamlined and powerful approach using the `select()` function combined with `where()`.

The core benefit of using **dplyr** lies in its intuitive syntax and its foundation in functional programming principles. By leveraging the pipe operator (`%>%`), complex data transformations can be chained together in a logical sequence, drastically improving code clarity and maintainability. When dealing with large or messy datasets--which often contain a mixture of text, dates, factors, and numbers--the ability to swiftly and accurately isolate the **numeric columns** becomes critical for subsequent statistical modeling or visualization tasks. This isolation step ensures that mathematical operations are applied only to appropriate data types, preventing errors and meaningless results during analysis.

This guide will explore the precise methodology for selecting only the **numeric columns** from an R dataframe using the modern **dplyr** syntax. We will focus on the combination of `select()` and the helper function `where()`, applying the necessary predicate, `is.numeric()`. Understanding this technique is essential for any R user looking to master data preparation, ensuring that your data workflows are both robust and up-to-date with current best practices in the **tidyverse** framework.

Why Isolate Numeric Data?

Numeric data forms the backbone of quantitative analysis. Variables like height, temperature, count, or score are typically represented numerically and are required for operations such as calculating means, standard deviations, correlations, or performing regression analysis. If a column that is intended for analysis is stored incorrectly--perhaps as a character string due to missing values or mixed inputs--statistical functions will either fail or produce erroneous outputs. Therefore, identifying and isolating truly **numeric columns** is a necessary data cleaning step before commencing inferential statistics.

In many real-world datasets, especially those imported from external sources like CSV files or databases, column types are often misclassified. For instance, an ID number might be read as an integer, even though it should function as a categorical variable, or a measurement might be stored as a factor. By specifically targeting only columns that satisfy the `is.numeric()` condition, we create a safeguard against these common data type issues. This targeted selection simplifies the workflow, allowing analysts to immediately proceed with calculations without manually checking the

data type of every single column.

Furthermore, selecting only **numeric columns** is beneficial for certain machine learning applications. Many algorithms, particularly those based on distance metrics (like K-Nearest Neighbors) or linear models (like Ridge or Lasso Regression), require all input features to be numerical. Before training a model, data scientists frequently create subsets containing only the features suitable for the chosen algorithm, often separating continuous numeric features from discrete numeric or categorical variables. The **dplyr** approach provides the most efficient way to handle this initial filtering requirement within the R environment.

The Evolution of Numeric Column Selection in dplyr

For some time, the standard method for conditional column selection in dplyr was the `select_if()` function. This function was powerful because it accepted a function (a predicate) that returns a logical value (TRUE or FALSE) for each column, deciding whether to include it in the resulting data frame. To select numeric columns, the syntax was straightforward: `df %>% select_if(is.numeric)`. This method was widely adopted due to its simplicity and expressiveness.

However, as the **tidyverse** matured, developers aimed for greater consistency and flexibility across the package ecosystem. The `select_if()` and related functions like `select_at()` and `select_all()` were eventually superseded by a single, unified function: `select()`, enhanced by the use of selection helpers, most notably `where()`. The transition reflects a move towards non-standard evaluation where selection criteria are handled more consistently, regardless of whether you are selecting by name, position, or data type. While `select_if()` might still function in older versions or specific setups, it is officially deprecated, and the recommended practice is to use the modern syntax.

The modern, robust syntax involves passing the conditional logic through the `where()` helper function inside `select()`. This structure makes the intent clearer: "select columns where the following condition is true." The condition we use is `is.numeric`, which is an existing base R function designed to check if an object is of type numeric (integer or double). By embracing this updated syntax, analysts ensure their code remains compatible with future versions of **dplyr** and adheres to current tidyverse standards.

You can use the following syntax from the **dplyr** package to select only numeric columns from an R data frame:

df %>% select(where(is.numeric))

The following detailed example demonstrates how to implement this function in a practical scenario, including the necessary setup and validation steps.

Step-by-Step Example: Defining the Data Frame

To demonstrate the practical application of `select(where(is.numeric))`, we must first define a sample dataframe that contains a mix of data types, including character/string variables and various numeric types (integers or doubles). For this example, we will simulate a dataset containing information about various basketball players, tracking their team affiliation (character), and several performance statistics (numeric).

We create the data frame using the base R function `data.frame()`. The resulting structure clearly separates the non-numeric identifier (`team`) from the quantitative metrics (`points`, `assists`, `rebounds`). This initial structure is crucial because it mimics the complexity often found in real-world data, where data preparation begins with mixed data types that need to be categorized and separated before analysis.

The code below outlines the creation of this sample dataset and displays the structure of the original data frame, allowing us to visualize the mix of variables we intend to filter. Notice that the `team` column is of type character, while the three statistical columns are inherently numeric, which is what the dplyr selection process must recognize and isolate.

#create data frame
df <- data.frame(team=c('A', 'B', 'C', 'D', 'E'),
points=c(22, 34, 30, 12, 18),
assists=c(7, 9, 9, 12, 14),
rebounds=c(5, 10, 10, 8, 8))

#view data frame
df

team points assists rebounds
1 A 22 7 5
2 B 34 9 10
3 C 30 9 10
4 D 12 12 8
5 E 18 14 8

Implementing the Selection: Using select(where(is.numeric))

Once the data frame is established, the application of the dplyr selection logic is highly efficient.

The first requirement is to ensure the **dplyr** package is loaded into the R session using the `library(dplyr)` command. Following this, we employ the pipe operator (`%>%`) to feed the data frame `df` directly into the `select()` function. This streamlined approach minimizes temporary variables and makes the data transformation chain easy to follow.

Inside the `select()` function, the key to the entire operation is the `where(is.numeric)` argument. The `where()` function evaluates the specified condition--in this case, whether the column's data type is numeric--for every column in the input data frame. If the function `is.numeric()` returns `TRUE` for a column, that column is included in the output. If it returns `FALSE` (as it would for the character column `team`), the column is dropped from the resulting subset.

The resulting output is a new **dataframe** that contains only the columns that successfully passed the numeric check. This process is instant and highly memory-efficient, regardless of the size of the original dataset. The following code executes this selection process, demonstrating how quickly the non-numeric variable `team` is excluded, leaving only the quantitative performance statistics.

library(dplyr)

#select only the numeric columns from the data frame
df %>% select(where(is.numeric))

points assists rebounds
1 22 7 5
2 34 9 10
3 30 9 10
4 12 12 8
5 18 14 8

Upon reviewing the output above, it is clear that only the three **numeric columns** have been retained: **points**, **assists**, and **rebounds**. The categorical variable `team` has been successfully excluded, yielding a clean subset ready for quantitative analysis.

Verifying Data Types: A Crucial Validation Step

While the **dplyr** output visually confirms the successful selection of the correct columns, it is always best practice in data science to programmatically verify the data types. This validation step confirms that R has correctly identified the variables and ensures that downstream operations will not encounter unexpected class mismatches. For this purpose, we can use the base R function `str()`, which provides a concise, internal summary of an object, displaying the structure and the class of each variable within the data frame.

Applying `str(df)` to our original data frame `df` allows us to inspect the underlying storage type for each column before selection. The output below confirms that `team` is stored as a character variable (`chr`), while `points`, `assists`, and `rebounds` are all stored as numeric variables (`num`). This verifies the initial assumption that guided our use of the `is.numeric()` predicate.

Understanding the output of `str()` is vital, as it differentiates between various numeric types, such as integers (`int`) and double-precision floating-point numbers (`num` or `dbl`). The function `is.numeric()` correctly captures both of these numeric classes, treating them appropriately for quantitative analysis. If we had only wanted to select strictly integer columns, we would instead use a different predicate, such as `is.integer()`, further demonstrating the flexibility afforded by the `where()` helper in **dplyr**.

#display data type of each variable in data frame
str(df)

'data.frame': 5 obs. of 4 variables:
$ team : chr "A" "B" "C" "D" ...
$ points : num 22 34 30 12 18
$ assists : num 7 9 9 12 14
$ rebounds: num 5 10 10 8 8

From this detailed output, we definitively see that `team` is a character variable while `points`, `assists`, and `rebounds` are all numeric, confirming why the `select(where(is.numeric))` command successfully isolated exactly these three columns.

Alternative Methods and Advanced Predicates

While selecting all **numeric columns** is a common requirement, **dplyr** offers immense flexibility by allowing users to select columns based on virtually any condition. The power of the `where()` function lies in its ability to accept any function that returns a logical value for each column. This opens the door to highly specialized subsetting operations that go beyond simple data type checks.

For instance, instead of `is.numeric()`, one might need to select columns that are factors (`is.factor()`), logical variables (`is.logical()`), or character strings (`is.character()`). This versatility allows for the separation of features into distinct groups required for different parts of a modeling pipeline.

Furthermore, advanced users can combine multiple conditions using logical operators within the `where()` helper. Suppose an analyst needed to select columns that are numeric OR columns that contain the word "score" in their name. This compound selection could be achieved using functions

provided by the dplyr package itself, such as `starts_with()` or `contains()`, coupled with standard logical operators like `|` (OR) or `&` (AND). The structure would look something like `df %>% select(where(is.numeric) | contains("score"))`. This level of granular control is what makes **dplyr** an essential tool for sophisticated data manipulation.

Summary and Next Steps

Mastering the selection of columns based on data type is a foundational skill in R data wrangling. By utilizing the modern dplyr syntax--specifically `select(where(is.numeric))`--data analysts can efficiently and reliably isolate all quantitative variables within a data frame. This method supersedes the older `select_if()` approach, ensuring compatibility and alignment with the evolving tidyverse standards.

The detailed example demonstrated how to create a mixed-type data frame and then apply the selection logic to extract `points`, `assists`, and `rebounds`, while excluding the character variable `team`. We also emphasized the importance of validation using the `str()` function to confirm that the underlying data types align with the selection criteria, thereby building confidence in the results of the transformation.

To continue building expertise in data manipulation using the **tidyverse**, we recommend exploring how to perform other common tasks, such as filtering rows based on conditions (using `filter()`), creating new variables (using `mutate()`), and summarizing data (using `summarise()`). These functions, when combined with the column selection techniques discussed here, form the basis of effective and reproducible data analysis workflows in **R**.

The following tutorials explain how to perform other common tasks using **dplyr**:

How to Filter Rows Based on Multiple Conditions
Understanding the Mutate Function in R
Grouping and Summarizing Data with Group_by