Learning dplyr: Selecting Columns in R with Multiple String Criteria

Authored by

Mohammed looti

November 13, 2025

Data wrangling and manipulation form the backbone of any analytical project conducted within the R programming language environment. Among the most repetitive, yet critical, tasks is the process of subsetting--specifically, selecting a precise set of columns from a large data frame. While selecting columns by their exact name is trivial, significant complexity arises when the requirement shifts to filtering columns based on patterns, such as requiring columns whose names contain one of several distinct substrings. Historically, relying on traditional base R methods for this fuzzy matching often leads to cumbersome indexing, reliance on complex `grep` logic, and code that is verbose and difficult for collaborators to interpret or maintain. This common challenge--selecting columns based on multiple potential strings--demands a streamlined, highly readable solution that integrates seamlessly into modern data pipelines.

The difficulty of managing column selection is greatly amplified when dealing with large-scale datasets, especially those where variables follow hierarchical or inconsistent naming conventions. Imagine a scenario where an analyst must isolate all variables related to "income," "salary," or "wage" from a dataset containing hundreds of columns. Manually listing every single required column is not only impractical but also introduces a high risk of error, particularly as the dataset evolves. This is precisely the domain where the powerful capabilities of the dplyr package, a fundamental element of the Tidyverse, provide immense organizational and functional value. By transforming complex selection logic into concise, expressive code, `dplyr` drastically improves efficiency.

This guide focuses on mastering this targeted, multi-criteria column selection. We will utilize the core dplyr package functionality, specifically combining the `select()` verb with a specialized helper function designed explicitly for sophisticated pattern matching. This pairing dramatically enhances code clarity and accelerates data preparation scripts. Our primary objective is to implement "OR" logic within column selection criteria, enabling us to define a list of patterns and efficiently retrieve any column name that satisfies at least one of those criteria. This technique is indispensable for effective large-scale data wrangling and reproducible research.

Introducing the dplyr Solution: select() and matches()

The dplyr package provides a robust and highly readable framework for data manipulation in R. At the core of its utility are the five primary "verbs," among which `select()` is dedicated specifically to choosing, renaming, or reordering columns. To effectively handle pattern-based selection, `select()` relies on various helper functions. The most potent tool for implementing complex multi-string criteria is the matches() function. This function is purpose-built for applying Regular Expressions (Regex) directly against column names, facilitating sophisticated filtering logic that extends far beyond simple exact-name checks.

The true power of the matches() function shines when it interprets a single string pattern as a

combined logical criteria. When analysts need to select columns based on multiple potential substrings, they leverage the fundamental properties of Regular Expressions (Regex), notably the vertical bar (**|**) character. Within a Regex pattern, the vertical bar acts as the logical "OR" operator. By simply concatenating all desired strings using this pipe operator, we instruct `matches()` to return any column where the name contains the first string OR the second string OR any subsequent strings listed. This elegant approach allows users to condense logic that might traditionally require multiple conditional statements or iterative loops into a single, clean line of code.

The structure of a typical `dplyr` workflow utilizing this method is highly standardized, benefiting from R's powerful pipe operator (**%>%**), which chains operations together seamlessly. The syntax below illustrates the essential structure required to select columns from a generic data frame (**df**) that match one of two specified patterns (**pattern1** or **pattern2**). This command is both concise and highly expressive, embodying the Tidyverse philosophy of clear data processing.

library(dplyr)

df %>%
select(matches('pattern1|pattern2'))

This concise command efficiently returns all columns from the input data frame, **df**, that successfully contain either the substring **pattern1** or the substring **pattern2** within their respective column headings. It is vital to recognize the scalability of this method; we are not restricted to just two patterns. The pattern string can be extended indefinitely using the **|** operator, allowing for the simultaneous selection based on dozens of potential prefixes, suffixes, or embedded names, thereby greatly simplifying the initial data preparation phase of any analysis.

Understanding the Syntax of matches() and the OR Operator

Effective multi-string column selection relies entirely on the proper construction of the Regular Expressions (Regex) pattern supplied as the argument to the `matches()` function. Regex is a powerful sequence of characters defining a search pattern, and its operators are foundational for complex text searching. In the specific context of selecting column names in R, the vertical bar (**|**) is arguably the most critical operator. Its function is that of a logical "OR," stipulating that a match should be registered if the text contains the expression immediately preceding the bar or the expression immediately following it. For instance, the simple pattern **'North|South|East'** will successfully match any column name containing 'North', 'South', or 'East'.

The flexibility offered by the **|** operator empowers developers and analysts to construct extremely specific inclusion criteria. Consider a large survey dataset where columns track metrics such as

Q1_score_A, **Q2_score_B**, **Q3_time_A**, and **Q4_time_B**. If the analytical goal is to extract all columns related to either 'score' or 'time', the required Regex pattern is simply **'score|time'**. The `matches()` function automatically handles the search across the entire set of column names, returning only those that satisfy this disjunction. This dramatically simplifies scripting compared to the necessity of writing iterative loops or complex nested conditional statements often required in less structured or older programming environments.

A key consideration when using this function is that `matches()` performs case-sensitive searching by default. If the objective requires selecting columns regardless of capitalization (e.g., matching both "Total" and "total"), the user must incorporate the appropriate Regex modifiers (such as `(?i)`) or utilize alternative `dplyr` helpers like `contains()` if appropriate for the specific task. However, for most standard analytical tasks where column naming conventions are consistent (e.g., all lowercase or snake_case), the basic **pattern1|pattern2** structure remains sufficient and highly effective. This function, being part of the Tidyverse ecosystem, ensures consistency and predictability in data manipulation workflows.

Practical Example: Selecting Columns with Multiple Strings

To fully appreciate the utility and clarity of the `select(matches())` combination, we will walk through a concrete, replicable example. We begin by constructing a sample data frame in the R programming language environment. This synthetic dataset simulates performance data collected about various basketball players, including columns that we will specifically target for extraction based on defined string patterns.

The following code block establishes our example data. Note that the data frame includes player team identity and performance statistics labeled with **total_points**, **total_assists**, and **total_rebounds**. This setup provides the necessary context for demonstrating the selective filtering process using `dplyr`.

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

#view data frame
df

team total_points total_assists total_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

Now, let us assume our analytical requirement is to isolate only the columns related to scoring and rebounding performance. Specifically, we want to select columns containing the substrings **"points"** or **"reb"**. We achieve this by applying the `select()` function in combination with the `matches()` function, utilizing the **|** operator to logically combine our search strings within a single Regex pattern. This critical operation effectively targets columns that contain "points" (matching **total_points**) or "reb" (matching **total_rebounds**).

library(dplyr)

#select any columns that contain 'points' or 'reb' in the name
df %>%
select(matches('points|reb'))

total_points total_rebounds
1 99 30
2 68 28
3 86 24
4 88 24
5 95 30
6 74 36
7 78 30
8 93 29

As clearly demonstrated by the resulting output, the executed code successfully extracts only the **total_points** and **total_rebounds** columns. The **total_assists** column is correctly excluded because its name does not contain either of the specified patterns, and the **team** column is also excluded. This outcome confirms that the `matches()` function accurately interpreted the combined Regex pattern, delivering an immediate and clean subset of the data frame based on fuzzy string matching criteria. This powerful, readable method is highly desirable for maintaining reproducible research and dramatically simplifying complex data preparation phases.

Leveraging Advanced Pattern Matching with Regular Expressions

The full potential of the `matches()` function is realized when users move beyond simple substring matching and begin to incorporate the full range of Regular Expressions (Regex) capabilities. Regex provides meta-characters--special characters that define how a pattern must match--allowing for extremely precise control over where and how a match must occur within the column name. For example, while simply searching for 'pattern' finds that string anywhere, analysts often need to specify that a column name must **begin** with a certain prefix, or **end** with a specific suffix, or contain a character from a defined set of options.

For column selection, two of the most commonly used and essential Regex operators are the carat (**^**) and the dollar sign (**$**). The carat (**^**) is used as an anchor, forcing the match to occur only at the very beginning of the string (i.e., the column name must start with the pattern that follows the carat). Conversely, the dollar sign (**$**) anchors the match to the end of the string. Combining these anchoring operators with the logical OR operator (**|**) allows for the creation of sophisticated and highly nuanced logical selections. For instance, an analyst might want to select columns that start with "total" OR columns that contain the substring "mean" anywhere within the name.

Consider an expanded example based on our previous basketball data frame. We might want to select all columns that either **start** with the string "total" OR include the substring "tea" (which would capture the **team** column). To achieve this, we apply the **^** anchor to the "total" pattern, but leave the "tea" pattern unanchored, relying entirely on the OR operator to combine these two distinct criteria within the search string:

library(dplyr)

#select any columns that start with 'total' or contain 'tea' in the name
df %>%
select(matches('^total|tea'))

team total_points total_assists total_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

In this specific instance, the result returns all columns from the data frame. This is because the

condition is met for every column: **total_points**, **total_assists**, and **total_rebounds** all start with "total," satisfying the **^total** pattern, while the **team** column contains "tea," satisfying the second pattern. This advanced usage demonstrates how granular control over pattern matching allows data scientists to precisely define the necessary subset, ensuring that only relevant data fields are carried forward for subsequent analysis or modeling steps. We are entirely free to employ whatever valid Regular Expression (Regex) operators are required to select columns that match specific, complex patterns.

Installation and Pre-requisites

Before any user can leverage the powerful features of the `dplyr` package, including the `select()` and `matches()` function, they must first ensure the package is correctly installed within their R programming language environment. If the package has not yet been installed, R provides a standard command to retrieve and set up the necessary components from CRAN (Comprehensive R Archive Network). This installation step is fast and typically only needs to be performed once per R environment setup.

The required syntax for installation is straightforward and should be executed directly in the R console:

install.packages('dplyr')

Once successfully installed, the package must be loaded into the current R session using the `library(dplyr)` command before any of its functions can be called. Adhering to best practice, the loading of all required libraries should occur at the beginning of any script to ensure reproducibility and clearly document dependencies. Following these initial setup steps, the sophisticated `matches()` function becomes fully available for use within data manipulation pipelines that utilize the pipe operator (**%>%**).

As a final note on best practices, analysts are encouraged to always consult the official Tidyverse documentation for the most precise details regarding function behavior, especially concerning edge cases in string matching or specific Regex implementations. The `matches()` function, being an integral part of the `select` family, adheres to the strict standards and consistent philosophy of the Tidyverse ecosystem.

Conclusion: Mastering Complex Column Selection

Selecting columns based on multiple string patterns is an indispensable requirement in modern data preparation workflows. The combination of the `dplyr` package's `select()` verb and the versatile `matches()` function provides the ideal, highly readable, and powerful solution within the

R environment. By skillfully leveraging the logical OR operator (**|**) within Regular Expressions (Regex), users can define highly specific and complex criteria to filter column names, significantly streamlining otherwise arduous data wrangling tasks. This streamlined approach ensures that data analysts can maintain clear, reproducible, and scalable code, which dramatically boosts productivity when dealing with large or evolving data frame structures.

Mastering these advanced selection techniques is a crucial step towards achieving proficiency in modern R data science workflows. While the Tidyverse ecosystem offers many other convenient selection helpers for simpler tasks--such as **starts_with()**, **ends_with()**, and **contains()**--the `matches()` function remains the most flexible and indispensable option when multiple, complex patterns must be combined using logical operators or when strict Regex rules (like anchoring) are required. By integrating this method, analysts can ensure their code is both efficient and self-documenting.

For those interested in delving deeper into `dplyr` selection helpers or understanding the nuances of R's string matching capabilities, further exploration of the Tidyverse documentation is highly recommended. The principles demonstrated here are foundational for advanced data preparation tasks.

<!--

Featured Posts

-->