PharmaSUG China 2024 - Paper PO-10040

An R Function to Split Text Strings Exceeding 200 Characters into Multiple Variables

Longfei LI, Sanofi

ABSTRACT

In the SDTM (Study Data Tabulation Model), we often encounter the need to handle text string exceeding 200 characters, such as TS, CO, etc. When it comes to splitting these long text string into different variables, programmers' first instinct might be to use loop statements to process the texts individually. However, in R, applying such a loop-based approach to large datasets can lead to inefficient performance or even fail to run properly in some cases. To address this issue, this article introduces an R function that aims to handle such text splitting tasks without resorting to traditional loop statements, instead utilizing a more efficient programming approach.

INTRODUCTION

In current clinical data submissions, the XPT file (SAS Transport File Format) remains the primary file format for data delivery. However, since the XPT file format does not support storing long text strings natively, the SDTM (Study Data Tabulation Model) standard defines conventions for storing long text strings using multiple variables. As SAS programmers, when faced with such a requirement, combining arrays and loops to develop macros is often the preferred and efficient solution. This approach has been widely proven to be both efficient and reliable in SAS programming.

However, unlike SAS, the R language is designed to be more suitable for rapid analysis and data manipulation. When dealing with large datasets, relying excessively on ***for-loops*** or other iterative statements can lead to increased computational costs, thus reducing work efficiency. To address this issue, R language developers have created a series of tools, such as the apply family of functions (e.g., lapply, sapply, tapply, mapply) and the purrr package, that mimic the principles of parallel computing. These tools enable more efficient data processing and avoid performance bottlenecks caused by explicit loops.

Regarding the problem of separating text strings exceeding 200 characters into multiple variables, we need a solution that is both efficient and capable of creating new variables in the dataset to store these separated texts. In this study, we will explore how to achieve this goal without using explicit loops. Specifically, we will design a dataset-based function *(function_name(data, other_arguments))* that can efficiently process long text fields in the dataset, split them into multiple parts, and create new variables in the dataset to store these parts.

R FUNCTION – SPLIT_VAR

As previously mentioned, we will be designing a dataset-based function named ***function_name(data, other_arguments)***. Along with the ***data*** argument (the input dataset), we have included the following additional arguments as outlined in Table 1.

Argument	Statement
data	Input dataset. Typically, a data frame or tibble.
target_var	A character variable in the input data set that needs to be split.
outvar_name	A common prefix for variable names like 'CMDECOD'.
sep	a sepcified separator.

Argument	Statement
len	A scalar integer (a numeric value of length one) specifying the maximum length that should not be exceeded by any variable. The default value is 200L.

Table 1. Function Argument List

ALGORITHM

The function, ***split_var(data, target_var, outvar_name, sep, len)***, does not utilize explicit loops to solve problems but instead simplifies the process into four steps: ***split***, ***cumsum***, ***combine***, and ***transpose***. Among these, "cumsum" is a new vocabulary term, which is a compound word of "Cumulative Sum." It is worth mentioning that although we do not use explicit loops like in SAS, the concept of looping is pervasive as R's data processing is often based on vectorized operations.

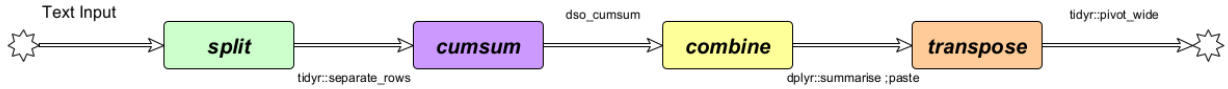

Figure 1. Algorithm Flow Chart

1. Split

As the literal meaning suggests, when we talk about "split", we refer to the process of breaking a text string into multiple substrings based on a delimiter. To achieve this, we introduce the ***separate_rows()*** function from the ***tidyr*** package. This function splits the variable to be separated using a specified separator and, while keeping other variables in the same observation unchanged, adds rows corresponding to the separated strings. Please refer to Figure 2 for a demonstration of this process.

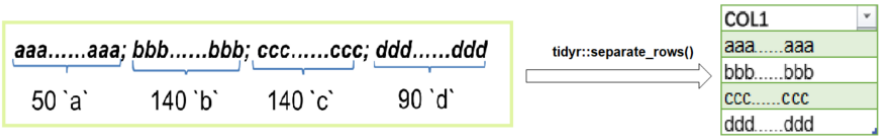

Figure 2. Separating Rows with Semicolon using separate_rows()

2. Cumsum

Unlike ordinary cumulative calculations, in our cumulative calculation, we need to consider the length of the separator, which is an additional +1, and we need to start the next cumulative count when the accumulated value reaches 200. This operation is designed to ensure that the string length of each group to be combined next will not exceed 200. To this end, we have defined a new cumulative summation function called ***dso_cumsum()***. An important note to mention is that Figures 3, 4, and 5 are continuations of Figure 2.

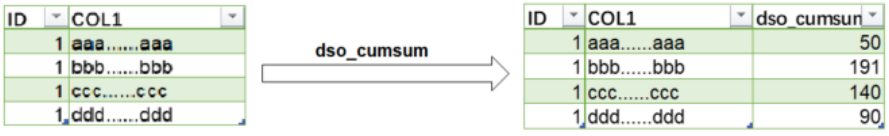

Figure 3. Custom function dso_cumsum()

Display 1 displays the source code for the dso_cumsum function as well as the test results. If you are interested, you can try to copy and run this code.

```
dso_cumsum <- function(num, cond) {
  .cond <- rlang::quo_squash(rlang::enquo(cond))

  Reduce(function(x1, x2) {
    x1 <- sum(x1, (x2 + 1L), na.rm = TRUE)
    if (eval(.cond)) {
      x1 <- 0L
      x1 <- sum(x1, x2, na.rm = TRUE)
    }
    return(x1)
  }, num, accumulate = TRUE)
}

test_string <- paste(strrep("a", 50), strrep("b", 140), strrep("c", 140), str
rep("d", 90))

test_data <- data.frame(ID = 1, COL1 = test_string) |>
  tidyr::separate_rows(COL1) |>
  dplyr::mutate(dso_cumsum = dso_cumsum(nchar(COL1), x1 > 200L))

test_data[, c("ID", "dso_cumsum")]

## # A tibble: 4 × 2
##       ID dso_cumsum
##    <dbl>      <int>
## 1     1          50
## 2     1         191
## 3     1         140
## 4     1          90
```

Display 1. demonstration code for the dso_cumsum() function

3. Combine

By using the custom function ***dso_cumsum*** to calculate the cumulative sum and comparing it with the length of each string calculated by the **nchar** function, we can obtain a column **"**nchar = dso_cumsum". By performing cumulative calculations on this column, we can get new groups. As a result, you can see that "aaa...aaa" and "bbb...bbb" should be grouped together, "ccc...ccc" as the second group, and "ddd...ddd" as the third group. The remaining task becomes relatively simple and we only need to concatenate the strings within the same group into a single string.

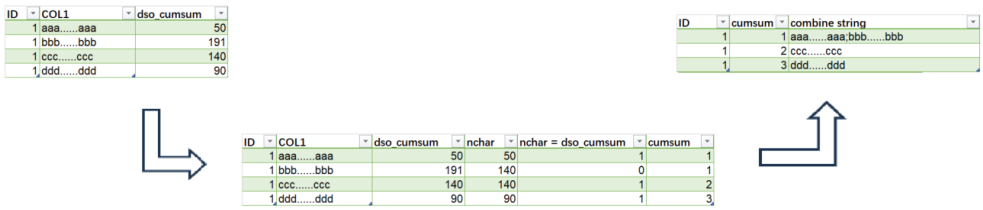

Figure 4. Combine Strings within Each Subgroup

4. Transpose

By reaching the third step, which is called "combine", we have successfully split a long string and combined it into individual cells with lengths not exceeding 200. However, we have not yet completed the process of creating variables and storing the separated and re-combined string values into the variables we have created. The "transpose" step introduces the ***pivot_wider()*** function from the **tidyr** package.

Figure 5. Transposing with Function pivot_wider()

SOURCE CODES

Display 2 shows all source codes, and you can use the debugging process in R to run and examine the results of each step in detail based on the source codes.

```
split_var <- function(data, target_var, outvar_name, sep = ";", len = 200L) {

  .need_split_var <- rlang::enquo(target_var)

  stopifnot(is.data.frame(data),
            length(rlang::quo_get_expr(.need_split_var)) == 1L,
            rlang::is_string(outvar_name),
            rlang::is_string(sep),
            rlang::is_integer(len))

  # Remove a series of variable named outvar_name with numeric suffix
  .to_keep <- names(data)[!grepl(paste0(outvar_name, "([0-9]+)$"), names(dat
a))]
  .new_data <- data %>%
    dplyr::mutate(dso_unique_id = if (nrow(.) == 0L) {
      numeric(0L)
    } else {
      dplyr::row_number()
    }) %>%
    dplyr::select(dplyr::all_of(.to_keep), dso_unique_id)


  # if (is.na(sep)) sep <- "\\;"
  # separate string
  .split_data <- .new_data %>%
    dplyr::mutate(dso_split_var = !!.need_split_var) %>%
    tidyr::separate_rows(dso_split_var, sep = sep) %>%
    dplyr::group_by(dso_unique_id, !!!rlang::syms(.to_keep)) %>%
    dplyr::mutate(
      dso_var1 = if (is.null(dso_cumsum(nchar(dso_split_var), x1 > {{len}})))
 {
        numeric(0L)
      } else {
```

```
        dso_cumsum(nchar(dso_split_var), x1 > {{len}})
      },
      dso_var2 = ifelse(nchar(dso_split_var) == dso_var1, TRUE, FALSE),
      dso_subid = dplyr::coalesce(cumsum(dso_var2), 1L),
      dso_text = dplyr::coalesce(dso_split_var, "")
    ) %>%
    dplyr::ungroup() %>%
    dplyr::group_by(dso_unique_id, !!!rlang::syms(.to_keep), dso_subid) %>%
    dplyr::summarise(dso_text = paste(dso_text, collapse = sep)) %>%
    dplyr::ungroup() %>%
    dplyr::mutate_at("dso_text", ~ dplyr::if_else(stringr::str_squish(.) == "
", NA_character_, .)) %>%
    # Avoid intermediate separator causing redundant symbols at the end
    dplyr::mutate(dso_text = gsub(paste0("(", sep, "+)$"), "", dso_text)) %>%
    tidyr::pivot_wider(names_from = dso_subid, values_from = dso_text, names_
prefix = "dso_text") %>%
    dplyr::select(-dso_unique_id)

  # if the name of target_var and .out_var is same, then rename target_var to
 paste(., target_var)

  if (identical(rlang::as_label(.need_split_var), outvar_name)) {
    names(.split_data)[names(.split_data) == outvar_name] <- paste0(".", outv
ar_name)
  }

  # modify the name of series variable calling outvar_name
  .text_name <- names(dplyr::select(.data = .split_data, dplyr::starts_with("
dso_text")))

  if (length(.text_name) == 1L) {
    names(.split_data)[names(.split_data) %in% .text_name] <- outvar_name
  } else {
    names(.split_data)[names(.split_data) %in% .text_name] <- c(outvar_name,
paste0(outvar_name, 1L:(length(.text_name) - 1)))
  }

  invisible(.split_data)
}


dso_cumsum <- function(num, cond) {
  .cond <- rlang::quo_squash(rlang::enquo(cond))

  Reduce(function(x1, x2) {
    x1 <- sum(x1, (x2 + 1L), na.rm = TRUE)
    if (eval(.cond)) {
      x1 <- 0L
      x1 <- sum(x1, x2, na.rm = TRUE)
```

```
    }
    return(x1)
  }, num, accumulate = TRUE)
}
```

Display 2. Source Codes of the Funcion split_var()

USAGE

Display 3 shows the usage of the function. After running the above source codes, you can directly copy the following codes into your RStudio to use the split_var function, but before doing so, please ensure that your RStudio has the tidyverse packages installed.

```
test_string <- paste(strrep("b", 50), strrep("b", 140), strrep("c", 140), str
rep("d", 90), sep = ";")

test_data <- data.frame(ID = 1, COL1 = test_string)

new_test_data <- test_data %>% split_var(COL1, "new_var", "\\;")
```

Display 3. Usage of the Funcion split_var()

CONCLUSION

In developing the split_var function for R, we adhered to the programming habits and logic of the R language, abandoning the explicit loop approach commonly used in SAS, to solve the problem of how to split strings exceeding 200 characters into multiple variables. The function cleverly addresses this issue through four steps: **split**, **cumsum**, **combine**, and **transpose**. Of course, R is a highly flexible language, and I believe there are even more elegant ways to achieve the same result.

However, regardless of the approach taken, we should always consider the speed of R as a programming language. We should utilize R's vectorized operations and implicit loop logic flexibly, and combine efficient algorithms to avoid using explicit loops, especially when dealing with datasets. This is particularly crucial for improving code efficiency and data processing speed, especially for large datasets.

REFERENCES

CDISC Submission Data Standards Team. (2018). Study Data Tabulation Model Implementation Guide: Human Clinical Trials (SDTM IG) Version 3.3. Available at https://www.cdisc.org/standards/foundational/sdtmig/sdtmig-v3-3/html#Text+Strings+Greater+than+200+Characters+in+Other+Variables

ACKNOWLEDGMENTS

Here, we sincerely express our gratitude to the developers of **RStudio**, **tidyverse** packages for creating such an efficient and convenient working environment for R users. Additionally, I would like to extend my special thanks to my colleagues, Echo Yan and Fan Zhang, for their encouragement that gave me the courage to take the first step in submitting my article. While this article still has room for improvement, it marks an important milestone in my journey of academic exploration. Finally, I am grateful to PharmaSUG for providing us with such a valuable platform.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

<Name: Longfei LI>
<E-mail: Longfei2.Li@Sanofi.com>