Paper CT01

SAS Arrays in R Way

Jagadish Katam, Princeps Technologies, Toronto, Canada

ABSTRACT

This paper will demonstrate a comparative analysis of data from an Array between R and SAS. This is a better way of learning any new programming language. The industry has been using SAS for quite some time and recently there is a new trend of R in clinical trials. Considering the same, we are slowing moving to use R parallelly in generating datasets and outputs, so for data manipulation we need to know how to use R for Array type of programming. A SAS Array is a temporary grouping of SAS variables under a single name. The use of Arrays typically reduces the number of lines of code needed to perform a task, resulting in code that is less error-prone and easy to understand by programmers. In this paper, we will explore the different ways of doing Arrays in R vs SAS with some examples.

INTRODUCTION

Arrays are useful when we need to perform similar operations on multiple variables or when we want to avoid writing similar code for multiple variables. SAS arrays provides a simple, efficient way to process a list of variables in a SAS DATA step. R provides several ways to work with arrays and multidimensional data structures, which can be quite helpful for various data analysis and manipulation tasks. Here's how R can assist in working with arrays:

ARRAYS

It provides a simple, appropriate way to process a group of variables in a DATA step/script. In this paper we will see how we can do the arrays in R in comparison with SAS. Here is the sample dataset / data frame we are going to use in this paper.

Dataset / Data frame:

x1	x2	x3	x4	x5
1	1	3	AA	AB
2	3	4	AB	CC
3	4	5	AC	DD
4	5	6	AD	EE
5	6	7	AE	FF
6	7	8	AF	GG

Example 1: Numeric variables having value greater than 3 need to be replaced with missing value

The provided script imports data from the 'temp' dataset and proceeds to inspect each data point. It assesses whether the values within the 'x1,' 'x2,' and 'x3' variables surpass the threshold of 3. If any of these values exceed 3, they are substituted with missing values. The modified dataset, denoted as 'test,' now contains these updated values.

In SAS, missing values are represented by a period (".") in numeric variables. For example, if a numeric variable has no value, it is assigned a period. In R, missing values are represented by the special value "NA." This applies to both numeric and character variables. In R, if a value is missing, it is represented as "NA." For numeric data, "NA" is used to indicate the absence of a value.

SAS

```
data test;
set temp;
array var1 (3) _numeric_;
do i = 1 to dim(var1);
if var1{i}>3 then var1{i}=.;
end;
run;
```

R

Approach 1:

```
test <- temp %>% mutate(across(x1:x3, ~
ifelse(.>3,NA_real_, .)))
```

Approach 2:

```
var <- names(which(sapply(temp,
is.numeric)))


test <- temp %>%
mutate(across(all_of(var), ~
ifelse(.>3,NA_real_, .x)))
```

SAS Output:

x1	x2	x3	x4	x5
1	1	3	AA	AB
2	3		AB	CC
3			AC	DD
			AD	EE
			AE	FF
			AF	GG

R Output:

x1	x2	x3	x4	x5
1	1	3	AA	AB
2	3	NA	AB	CC
3	NA	NA	AC	DD
NA	NA	NA	AD	EE
NA	NA	NA	AE	FF
NA	NA	NA	AF	GG

Approach 1:

This code is using the dplyr package in R for data manipulation. Let us break it down step by step:

1. `test <- temp`: This line creates a new data frame called 'test' and assigns it the values from the 'temp' data frame. In essence, it duplicates the 'temp' data frame. The pipe operator (`%>%`) is used to chain together operations. It takes the output from the previous step and passes it as the input to the next step.

2. `mutate()`: This is a dplyr function used for creating or modifying columns in a data frame. In this case, it will modify the 'test' data frame.

3. `across(x1:x3, ~ ifelse(.>3, NA_real_, .))`: This part of the code is using the `across()` function, which applies the specified operation to multiple columns. Here, it is being used to target columns 'x1' through 'x3'.

- `x1:x3` specifies the range of columns you want to apply the operation to.

- `~ ifelse(.>3, NA_real_, .)` is a function that checks if a value in the selected columns is greater than 3. If it is greater than 3, it replaces that value with `NA_real_` (a way to represent missing or undefined data in R), otherwise, it leaves the value unchanged.

So, in summary, this code creates a new data frame 'test' based on the 'temp' data frame. It then goes through the columns 'x1' to 'x3' and replaces any values greater than 3 with missing values (NA), leaving the other values unchanged in the 'test' data frame.

Approach 2:

let's break down the provided R code step by step:

1. `var <- names(which(sapply(temp, is.numeric)))`: This line extracts the names of numeric variables in the 'temp' data frame.

- `sapply(temp, is.numeric)` checks which columns in the 'temp' data frame are of numeric data type. It returns a logical vector with `TRUE` for numeric columns and `FALSE` for non-numeric ones.

- `which()` is used to find the indices where the logical vector is `TRUE`, corresponding to numeric columns.

- `names()` extracts the names of the numeric columns, and these names are stored in the 'var' variable.

2. `across(all_of(var), ...)` applies a function to multiple columns specified in 'var'. In this case, it will apply the function defined in the next part to all numeric columns identified in 'var'.

Example 2: Extract first letter of all the character variables

This code extracts data from the 'temp' dataset and proceeds to iterate through each character variable. It retains solely the initial letter of each variable, discarding the remainder. Consequently, a fresh dataset named 'test' is created, comprising character variables shortened to their initial letters.

SAS

```
data test;
set temp;
array cvars (*) _character_;
do i = 1 to dim(cvars);
cvars{i} = substr(cvars{i},1,1);
end;
drop i;
run;
```

R

```
varc <- names(which(sapply(temp,
is.character)))


test <- temp %>%
mutate(across(all_of(varc),
~stringr::str_sub(.x,1,1)))
```

Output:

x1	x2	x3	x4	x5
1	1	3	A	A
2	3	4	A	C
3	4	5	A	D
4	5	6	A	E
5	6	7	A	F
6	7	8	A	G

Certainly, let's break down the provided R code step by step:

1. `test <- temp %>% ...`: This line creates a new data frame called 'test' and assigns it the values from the 'temp' data frame. The `%>%` operator is used to pipe the 'temp' data frame into the subsequent operation.

2. `mutate(...)`: This is a dplyr function used for creating or modifying columns in a data frame. In this case, it will modify the 'test' data frame.

3. `across(all_of(varc), ...)`: The `across()` function is used to apply a function to multiple columns. Here, it's applied to the columns specified in 'varc' which represents character column names.

4. `~stringr::str_sub(.x, 1, 1)`: This is the function that is applied to each column in the 'varc' selection. It uses the 'stringr' package to extract a substring from each column.

- `stringr::str_sub(.x, 1, 1)` takes the first character (letter) from the input column represented by `.x`. The `1, 1` arguments specify the starting and ending positions for the substring, effectively extracting only the first letter.

So, in summary, this code creates a new data frame 'test' based on the 'temp' data frame. It then goes through the columns specified in 'varc' and extracts the first letter from each character variable using the 'stringr' package's `str_sub()` function. The result is a data frame with character variables truncated to their first letters.

Example 3: Assign Initial Values in an Array

The following code reads data from the "temp" dataset and then goes through each numeric variable in "temp" and multiplies it by a corresponding percentage increase factor from the temporary "val" array. The results are assigned to the "x1", "x2", and "x3" variables in the "test" dataset.

SAS

```
data test;
set temp;
array nvars (*) _numeric_;
array pctinc {3} _temporary_ (1, 2 , 3); do i = 1 to dim(nvars);
nvars{i} = nvars{i} * pctinc{i};
end;
drop i;
run;
```

R

```
Vector:
val <- c(x1=1, x2=2, x3=3)


Approach 1:
test <- temp %>%
mutate(across(all_of(var),
~.x*val[cur_column()]))


Approach 2:
temp1 <- purrr::map2_dfc(temp[,var],
val, `*`)
test <- cbind(temp1,temp[,c('x4','x5')])
```

Output:

x1	x2	x3	x4	x5
1	2	9	AA	AB
2	6	12	AB	CC
3	8	15	AC	DD
4	10	18	AD	EE
5	12	21	AE	FF
6	14	24	AF	GG

Approach 1:

Let's break down the provided R code step by step:

1. `val <- c(x1=1, x2=2, x3=3)`: This line creates a vector named 'val' with named elements 'x1', 'x2', and 'x3', each having corresponding numeric values (1, 2, and 3).

2. `test <- temp %>% ...`: This line creates a new data frame 'test' and assigns it the values from the 'temp' data frame. The `%>%` operator is used to pipe the 'temp' data frame into the subsequent operation.

3. `mutate(...)`: The `mutate()` function is part of the dplyr package and is used for creating or modifying columns in a data frame. In this case, it will modify the 'test' data frame.

4. `across(all_of(var), ...)`: The `across()` function is used to apply a function to multiple columns. Here, it's applied to the columns specified in 'var', which are presumably numeric variables.

5. `~.x*val[cur_column()]`: This is the function that is applied to each column in 'var'. Let's break it down:

- `.x` represents the values in each column.
- `val[cur_column()]` is used to fetch the corresponding value from the 'val' vector. `cur_column()` returns the current column name, and this is used to index the 'val' vector. For example, if the current column name is 'x1', it will fetch the value 1 from 'val'.

So, in summary, this code creates a new data frame 'test' based on the 'temp' data frame. It then goes through the columns specified in 'var', multiplies the values in each column by the corresponding value from the 'val' vector, and stores the results in 'test'. This is a way to perform column-wise multiplication of values in 'temp' by specific factors specified in the 'val' vector.

Approach 2:

Let's break down the provided R code step by step:

1. `temp1 <- purrr::map2_dfc(temp[,var], val, `*`)`: This line uses the `purrr` package's `map2_dfc` function to perform element-wise multiplication between the columns in 'temp[,var]' and the 'val' vector. Here's the breakdown:

- `temp[,var]`: This extracts the columns in the 'temp' data frame that correspond to the variable names stored in 'var'. `val`: This is a vector defined earlier, which presumably contains numeric values associated with the 'var' columns.

- `*`: The backticks (`) around the asterisk (*) are used to specify that multiplication should be performed element-wise between the columns in `temp[,var]` and the 'val' vector.

- `map2_dfc()`: This function is used to apply this element-wise multiplication to all columns and returns a data frame as the result.

So, 'temp1' will contain the result of element-wise multiplication between 'temp[,var]' and 'val'.

2. `test <- cbind(temp1, temp[,c('x4','x5')])`: This line combines the data frames 'temp1' and specific columns from 'temp' into a new data frame 'test'. Let's break it down:

- `temp1` contains the result of the element-wise multiplication from step 1. `temp[,c('x4','x5')]` selects columns 'x4' and 'x5' from the 'temp' data frame.

- `cbind()` combines the two data frames, with 'temp1' on the left and the selected 'x4' and 'x5' columns on the right.

So, in summary, 'test' contains the results of element-wise multiplication between specific columns from 'temp' and corresponding values in the 'val' vector (stored in 'temp1') combined with columns 'x4' and 'x5' from 'temp'.

Example 4: Find the Minimum Value from a list of variables in an Array

In this example, we assume that you have three variables x1, x2, and x3 for which you want to find the minimum value. You can adapt the code by changing the array size and variable names to match your specific use case. The minval variable will contain the minimum value from the list of variables in the array.

SAS

```
data test;
set temp;
array nvars (*) _numeric_;
do i = 1 to dim(nvars);
minval = min(of nvars{*});
end;
drop i;
run;
```

R

```
test <- temp %>% rowwise() %>%
mutate(minval=min(across(all_of(var))))
```

Output:

x1	x2	x3	x4	x5	minval
1	1	3	AA	AB	1
2	3	4	AB	CC	2
3	4	5	AC	DD	3
4	5	6	AD	EE	4
5	6	7	AE	FF	5
6	7	8	AF	GG	6

This R code snippet performs the following operations using the dplyr package:

1. `test <- temp %>% rowwise()`: This line begins by creating a new data frame named 'test' that is initially identical to the 'temp' data frame. The `%>%` operator is used to pipe the 'temp' data frame into the subsequent operations. The `rowwise()` function is applied, which indicates that the following operations will be performed row-wise. This is important because it means that for each row, calculations will be done independently.

2. `mutate(minval = min(across(all_of(var)))`: Within the `mutate()` function, a new variable called 'minval' is created. This variable will store the minimum value among a specific set of columns. The `across()` function is used to select these columns. The `all_of(var)` part specifies a list of variable names, which are assumed to be stored in the 'var' variable. The `min()` function is then applied to these selected columns to calculate the minimum value for each row. The result is stored in the 'minval' column.

In summary, this code creates a new data frame 'test' based on 'temp' and calculates the minimum value for a set of specified columns ('var') for each row. This is done row-wise, ensuring that the minimum value is calculated independently for each row in the 'test' data frame.

Example 5: Find the Sum of Values from a list of variables in an Array

In this example, we assume that you have three variables x1, x2, and x3 for which we want to find the sum of values. You can adapt the code by changing the array size and variable names to match your specific use case. The sum variable will contain the sum of values from the list of all numeric variables in the array.

SAS

```
data test;
set temp;
array nvars (*) _numeric_;
do i = 1 to dim(nvars);
sum = sum(of nvars{*});
end;
drop i;
run;
```

R

```
test <- temp %>%
mutate(sum=rowSums(across(all_of(var))))
```

Output:

x1	x2	x3	x4	x5	sum
1	1	3	AA	AB	5
2	3	4	AB	CC	9
3	4	5	AC	DD	12
4	5	6	AD	EE	15
5	6	7	AE	FF	18
6	7	8	AF	GG	21

This R code snippet, which uses the dplyr package, performs the following operations:

1. `test <- temp %>%`: This line starts by creating a new data frame called 'test' that initially mirrors the 'temp' data frame. The `%>%` operator is used to pipe the 'temp' data frame into the subsequent operations.

2. `mutate(sum = rowSums(across(all_of(var)))`: Within the `mutate()` function, a new variable named 'sum' is created. This variable is designed to store the sum of values from a specified set of columns.

- `across(all_of(var))`: The `across()` function is employed to select a list of columns, which are determined by the variable names stored in 'var'.

- `rowSums()`: The `rowSums()` function is applied to calculate the sum of values across the selected columns for each row in the data frame.

In summary, this code creates a new data frame 'test' based on 'temp' and computes the sum of values across a specified set of columns ('var') for each row. The results are stored in a new column named 'sum' in the 'test' data frame.

Example 6: How to compare one set of columns with another set of columns

Expecting to derive a new flag variable which is derived by comparing the 1 set of columns (jan feb mar) with another set of columns (apr may jun), if the values of 1 set match the values of another set then flag='Y', else 'N'.

jan	feb	mar	apr	may	jun
10	20	30	5	0	10
7	9	6	5	9	8
1	2	3	4	5	6

SAS

```
data temp;
set temp;
array var1 (3) jan feb mar;
array var2 (3) apr may jun;
array var3 (3)$ apr1 may1 jun1;
do i = 1 to dim(var1);
do j = 1 to 3;
if var1{i} = var2{j} then var3{i}='Y';
end;
end;
flag=coalescec(of var3{*}, 'N');
drop apr1 may1 jun1 i j;
run;
```

R

```
temp$flag <- sapply(1:nrow(temp), \(x)
ifelse(any(unlist(temp[x, 1:3]) %in%
unlist(temp[x, 4:6])), "Y", "N"))
```

Output:

jan	feb	mar	apr	may	jun	flag
10	20	30	5	0	10	Y
7	9	6	5	9	8	Y
1	2	3	4	5	6	N

The provided R code adds a new column 'flag' to the 'temp' data frame based on a condition that checks for the presence of common elements between two sets of columns within each row. Let's break it down step by step:

1. `temp$flag <-`: This part of the code creates a new column 'flag' in the 'temp' data frame to store the results of the conditional check.

2. `sapply(1:nrow(temp), \(x) ...`: This section applies a function to each row in the 'temp' data frame.

   - `sapply(1:nrow(temp), ...)` applies the subsequent function to each integer from 1 to the number of rows in 'temp'.

   - `\(x)` is a lambda function that takes an argument 'x,' which represents the row index.

3. `ifelse(any(unlist(temp[x, 1:3]) %in% unlist(temp[x, 4:6])), "Y", "N")`:

   - `unlist(temp[x, 1:3])` extracts and unlists the values from columns 1 to 3 for the current row 'x'.

   - `unlist(temp[x, 4:6])` extracts and unlists the values from columns 4 to 6 for the same row 'x'.

   - `%in%` is used to check if there are any common elements between the two unlisted sets of values.

   - `any(...)` checks if there is at least one common element, returning a logical value.

   - `ifelse(..., "Y", "N")` assigns "Y" if there is at least one common element (the condition is true), and "N" if there are no common elements.

In summary, this code iterates through each row in the 'temp' data frame and checks whether there are any common elements between columns 1 to 3 and columns 4 to 6 for each row. If there is at least one common element, it assigns "Y" to the 'flag' column; otherwise, it assigns "N." This is a way to determine whether there is an intersection between the values in these column groups for each row.

CONCLUSION

In conclusion, both SAS and R provide array functionality that is essential for data manipulation and analysis. The choice between the two depends on our specific needs, our programming preferences, and the ecosystem in which we work. SAS is preferred in enterprise environments with a budget for licensing, while R is popular among data scientists, statisticians, and academics due to its open-source nature and extensive statistical capabilities. Our choice should align with our project requirements and the tools our team is most comfortable using.

REFERENCES

Listen Data. (2016, January 1). SAS Arrays and Do Loop Made Easy. Listen Data. https://www.listendata.com/2016/01/sas-arrays-and-do-loop-made-easy.html

Hadley Wickham, Romain François, Lionel Henry, Kirill Müller, & R Core Team. (2021, December 7). dplyr: A Grammar of Data Manipulation. The Comprehensive R Archive Network (CRAN). https://CRAN.R-project.org/package=dplyr

R Core Team. (2021). R: A language and environment for statistical computing. R Foundation for Statistical Computing. https://www.R-project.org/

CONTACT INFORMATION

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

Jagadish Katam
Princeps Technologies
Toronto, Canada
Email: Jagadish.katam@princepstechnologies.com
LinkedIn: Jagadish Katam | LinkedIn