How to Easily Analyze Data with the SAS LAG Function

Authored by

stats writer

December 1, 2025

The LAG function in SAS is one of the most powerful and fundamental tools available for data analysis and transformation, particularly when dealing with sequential or time-dependent data. This function specializes in manipulating the order of values within a dataset, allowing analysts to reference data points that occurred prior to the current observation.

Understanding how to properly deploy the LAG function is critical for tasks such as calculating period-over-period differences, generating moving averages, or preparing data for complex time series modeling. By shifting the data stream, LAG effectively creates a new variable that holds the value of a specified variable from an earlier row. This capability enables powerful comparisons and trend detection within your structured data.

The primary purpose of the LAG function is to retrieve lagged values of a specific variable within the current Data Step iteration. When utilized, the function operates on a queue principle, remembering the values it has encountered sequentially. It is important to note that the LAG function retains the value from the input argument based on the position in the data stream, not necessarily based on a time interval unless the data is sorted appropriately.

Understanding the Basic LAG Syntax

The LAG function is remarkably straightforward in its basic implementation. By default, it retrieves the value corresponding to the immediately preceding observation (a lag of 1). The fundamental syntax requires specifying the name of the new variable you wish to create and the variable from which you want to retrieve the lagged value.

lag1_value = lag(value);

While `LAG()` inherently represents a lag of one position, SAS provides numbered variations to easily access values further back in the sequence. For instance, to retrieve the value from the observation two positions prior, you would use `LAG2()`, or for `n` positions prior, `LAGn()`. These powerful extensions allow analysts to simultaneously calculate multiple lagged metrics for advanced comparisons.

The syntax supports multiple levels of lookback, making it highly flexible for various analytical needs:

`LAG1` (or simply `LAG`): Retrieves the value from the previous observation (1-lagged).
`LAG2`: Retrieves the value from the observation two rows back (2-lagged).
`LAGn`: Retrieves the value from the observation n rows back (n-lagged).

The following detailed examples illustrate how to implement these variations effectively in practical scenarios, first for simple lagging and then for conditional lagging by group.

Example 1: Calculating Multiple Lagged Variables

In this initial example, we demonstrate the straightforward application of the LAG function to a simple time-series-like dataset. We begin with a record of daily sales for a single store over 12 consecutive days. Our goal is to create three new variables that capture the sales values from the preceding day, two days prior, and three days prior.

First, we must create the source dataset using the DATA step and populate it with the daily sales figures. This foundation allows us to visualize the raw data before applying the transformation. Notice the distinct sequential nature of the data, which is essential for the LAG function to operate correctly.

```
/*create dataset*/
data original_data;
input day $ sales;
datalines;
1 14
2 19
3 22
4 20
5 16
6 26
7 40
8 43
9 29
10 30
11 35
12 33
;
run;

/*view dataset*/
proc print data=my_data;
```

Obs	day	sales
1	1	14
2	2	19
3	3	22
4	4	20
5	5	16
6	6	26
7	7	40
8	8	43
9	9	29
10	10	30
11	11	35
12	12	33

Implementing 1-Day, 2-Day, and 3-Day Lags

To generate the necessary lagged variables, we execute another DATA step, reading from the `original_data`. Within this step, we simply assign the results of `LAG()`, `LAG2()`, and `LAG3()` to newly created variables: `lag1_sales`, `lag2_sales`, and `lag3_sales`. This concise coding effectively shifts the values across the observations.

It is vital to recognize how the LAG function handles the initial observations. Since there is no preceding value for the first observation, the lagged variables will automatically be assigned a missing value (represented by a period `.` in SAS) for the rows where the lag distance exceeds the available history. For instance, `lag3_sales` will be missing for the first three rows because there are insufficient preceding observations.

```
/*create new dataset that shows lagged values of sales*/
data new_data;
set original_data;
lag1_sales = lag(sales);
lag2_sales = lag2(sales);
lag3_sales = lag3(sales);
run;

/*view new dataset*/
proc print data=new_data;
```

Obs	day	sales	lag1_sales	lag2_sales	lag3_sales
1	1	14	.	.	.
2	2	19	14	.	.
3	3	22	19	14	.
4	4	20	22	19	14
5	5	16	20	22	19
6	6	26	16	20	22
7	7	40	26	16	20
8	8	43	40	26	16
9	9	29	43	40	26
10	10	30	29	43	40
11	11	35	30	29	43
12	12	33	35	30	29

Interpreting the Lagged Output

The resulting dataset clearly demonstrates the shifting mechanism. For any given row (or observation), the newly calculated columns reference values from earlier in the sequence:

The `lag1_sales` value corresponds exactly to the `sales` value from the previous row.
The `lag2_sales` value matches the `sales` value from two rows above.
The `lag3_sales` value corresponds to the `sales` value from three rows above.

For example, looking at the fifth row (Day 5, Sales 16), `lag1_sales` is 20 (sales from Day 4), `lag2_sales` is 22 (sales from Day 3), and `lag3_sales` is 19 (sales from Day 2). This transformation is the foundation for calculating crucial metrics like day-over-day growth rates, which can be easily derived by subtracting the lagged value from the current value.

Example 2: Applying LAG Function within Grouped Data

While the standard LAG function is effective for continuous sequential data, real-world data often involves multiple groups (e.g., different products, regions, or stores) where the lagging operation must restart for each group. Failing to account for group boundaries leads to inaccurate comparisons, where the last observation of one group might be incorrectly lagged against the first observation of the next group.

To illustrate this challenge and its solution, consider an expanded dataset containing daily sales for

two distinct entities, Store A and Store B. The data must be sorted first by the grouping variable (store) and then sequentially by the time variable (implied by the input order):

```
/*create dataset*/
data original_data;
input store $ sales;
datalines;
A 14
A 19
A 22
A 20
A 16
A 26
B 40
B 43
B 29
B 30
B 35
B 33
;
run;

/*view dataset*/
proc print data=original_data;
```

Obs	store	sales
1	A	14
2	A	19
3	A	22
4	A	20
5	A	16
6	A	26
7	B	40
8	B	43
9	B	29
10	B	30
11	B	35
12	B	33

Controlling Lagging with the BY Statement

To ensure that the LAG function resets its queue whenever a new store begins, we must incorporate the BY statement in our SAS code. The BY statement generates two crucial temporary variables: `FIRST.variable` and `LAST.variable`. These variables are binary flags indicating the start and end of a defined group, respectively.

We combine the use of `LAG()` with a conditional statement that checks for `FIRST.store`. When SAS encounters the first observation of a new store, `FIRST.store` is true (1). At this point, we explicitly set the lagged variable to a missing value (`.`), effectively clearing the lag queue for the new group and preventing the data spillover from the previous group. Note that the BY statement requires the input data to be pre-sorted by the variable specified (in this case, `store`).

```
/*create new dataset that shows lagged values of sales by store*/
data new_data;
set original_data;
by store;
lag1_sales = lag(sales);
if first.store then lag1_sales = .;
run;

/*view new dataset*/
proc print data=new_data;
```

Obs	store	sales	lag1_sales
1	A	14	.
2	A	19	14
3	A	22	19
4	A	20	22
5	A	16	20
6	A	26	16
7	B	40	.
8	B	43	40
9	B	29	43
10	B	30	29
11	B	35	30
12	B	33	35

Analyzing the Grouped Output

Examining the output reveals the successful application of grouped lagging. For Store A (rows 1-6), the `lag1_sales` values correctly reflect the previous day's sales within that store's sequence. A crucial observation occurs at row 7, which marks the transition from Store A to Store B (Sales = 40).

The code ensures that the `lag1_sales` for row 7 is missing (`.`) because the `IF FIRST.store` condition is met. This conditional reset is fundamental. Without it, the value for `lag1_sales` in row 7 would have been 26 (the last sales value from Store A), erroneously creating a comparison across group boundaries. This technique is essential for preserving the integrity of group-specific time series analysis.

Technical Deep Dive: How the LAG Function Operates

Unlike standard mathematical functions in SAS that operate solely on values within the current observation, the LAG function operates via an internal queue or FIFO (First-In, First-Out) buffer. When the DATA step executes, the LAG function does not immediately look back at a previous row in the physical dataset. Instead, it maintains a dynamic memory buffer that is updated iteratively.

When SAS processes an observation, the following sequence occurs for `lag1_value = lag(value);`:

The function returns the value that was stored in its queue during the processing of the **previous** observation.
After returning the lagged value, the function takes the current observation's `value` and places it into the queue, ready to be retrieved in the next iteration.

This queue mechanism explains why, on the very first observation of the DATA step, the LAG function returns a missing value--the queue has not yet been populated. For higher lags, like `LAG3()`, the queue holds three preceding values, and three missing values are generated at the start of the data stream.

Common Pitfalls and Best Practices

While powerful, the LAG function must be used carefully to avoid generating misleading results. Here are critical best practices to ensure accurate lagging operations:

Sorting is Mandatory: If your data involves time series or sequential comparisons (e.g., daily sales, stock prices), the dataset must be correctly sorted chronologically (or by the required sequence) before the LAG function is applied. If data is unsorted, the lag calculation will reflect physical row order, which may not correspond to the required time sequence.
Handling Missing Values: The LAG function treats missing values like any other valid data point. If the lagged observation contains a missing value, that missing value will be returned by the function. You may need additional conditional logic (e.g., `IF` statements) if you require imputation or specific handling for missing lagged inputs.
Group Boundaries: Always use the `BY` statement combined with `IF FIRST.group THEN lag_variable = . ;` when calculating lags within distinct groups to reset the internal queue and prevent cross-group contamination, as demonstrated in Example 2.

Mastery of the LAG function is essential for anyone performing advanced analytical tasks in SAS, offering the fundamental building block for sequential data manipulation and time series modeling.

The following tutorials explain how to perform other common tasks in SAS: