Calculate a Rolling Mean in PySpark

Authored by

stats writer

November 16, 2025

Introduction to Time Series Analysis and Rolling Means

The calculation of a Rolling Mean, also known as a moving average, is a fundamental technique in time series analysis and data smoothing. This statistical operation helps analysts identify underlying trends by reducing short-term fluctuations and noise within sequential data. When working with large datasets, especially within a distributed computing environment, utilizing tools like PySpark becomes essential for scalable processing across clusters.

A rolling mean is derived by calculating the average of a specific number of preceding data points, subsequently sliding this computational window forward through the dataset. This approach is particularly valuable in financial analysis, sales forecasting, and monitoring sensor data, where understanding the underlying long-term movement is more critical than instantaneous values. Achieving this efficiently in a distributed DataFrame structure requires leveraging PySpark's specialized tools, specifically the powerful Window function framework, designed for ordered analytical calculations.

We will demonstrate the precise syntax required to compute a rolling mean over ordered data within a PySpark environment. The efficiency of this method ensures that complex statistical calculations are handled swiftly across clustered resources, adhering to the best practices for scalable data engineering and analytical processing.

The Core Syntax for Calculating a Rolling Mean in PySpark

To effectively calculate a rolling mean in PySpark, we must define a specific computational window over the data. This definition uses the Window function, which partitions and orders rows relative to the current row, allowing aggregate functions (like average) to operate on a bounded subset of data. The following concise PySpark code illustrates the necessary imports and logic for defining a 4-day trailing average:

```
from pyspark.sql import Window
from pyspark.sql import functions as F

#define window for calculating rolling mean
w = (Window.orderBy('day').rowsBetween(-3, 0))

#create new DataFrame that contains 4-day rolling mean column
df_new = df.withColumn('rolling_mean', F.avg('sales').over(w))
```

This implementation snippet performs two critical actions: first, it defines the window specification `w`, ensuring the calculation is ordered by the `day` column. Second, it applies an aggregate function, `F.avg('sales')`, over this defined window using `.over(w)`. This specific configuration calculates

a new column named `rolling_mean` that holds the 4-day rolling average of values found in the **sales** column of the DataFrame.

It is important to note the definition of the window boundaries using `rowsBetween(-3, 0)`. This configuration specifies that for any given row (day), the calculation should include the current row (represented by `0`) and the preceding three rows (represented by `-3`), totaling four data points. This methodological approach ensures that the rolling average computation is always accurate relative to the sequence of events defined by the ordering column, which is essential for accurate time series analysis.

Understanding the PySpark Window Function Parameters

The core complexity in calculating sophisticated metrics like the rolling mean lies in correctly defining the window specification object. The `Window` class provides the necessary tools to define how aggregate calculations should operate on subsets of data. The three primary components of a typical window definition are `partitionBy`, `orderBy`, and boundary definitions (like `rowsBetween` or `rangeBetween`).

For a standard time series rolling average, where data is simply sequential and not grouped by different entities (e.g., store ID or product type), the `partitionBy` clause is often omitted. However, the `orderBy` clause is mandatory to establish the sequence in which the data points are processed. If the data were not ordered by a sequential metric like `day`, the concept of "the last four days" would be meaningless, potentially leading to inconsistent or incorrect analytical results.

The rowsBetween function is pivotal here. It sets physical boundaries relative to the current row. Using `rowsBetween(start, end)`, where `start` and `end` are offsets from the current row (0), allows for precise control over the lookback period. Specifically, negative indices denote preceding rows, and positive indices denote following rows. This powerful capability allows users to define not only trailing means but also centered means or leading indicators, depending on the specific requirements of the data analysis project.

Practical Implementation: Setting up the PySpark Environment

To illustrate the application of this essential syntax, we must first establish a sample PySpark DataFrame that represents daily sales data. This initial setup requires initializing a `SparkSession` and defining the raw input data structure, which mirrors the typical data ingestion process in a real-world scenario where data often originates from databases or external storage systems.

The following code block demonstrates how to create a structured DataFrame containing ten consecutive days of sales figures from a hypothetical grocery store. The `day` column serves as our explicit ordering variable, vital for time-series operations, while the `sales` column is the metric we

wish to smooth using the rolling mean calculation.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = ,
,
,
,
,
,
,
,
,
]

#define column names
columns =

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+---+-----+
|day|sales|
+---+-----+
1	11
2	8
3	4
4	5
5	5
6	8
7	7
8	7
9	6
10	4
+---+-----+

The resulting DataFrame, `df`, provides a clean tabular view of the sales metrics over time. This foundational dataset is now perfectly structured and prepared for the application of the window function logic, allowing us to proceed with the calculation of the rolling average. Understanding the initial structure is critical, as the rolling mean calculation relies entirely on the sequential integrity of the data points defined by the `day` column.

Defining and Applying the Rolling Window Calculation

With the sales DataFrame prepared, we can now proceed to calculate the 4-day rolling mean for the **sales** column. This involves re-importing the necessary PySpark modules (`Window` and `functions`), defining the window specification (`w`), and using the `withColumn` transformation combined with the `avg` function to generate the new feature column. This procedure ensures the calculation is executed efficiently in a distributed manner across the Spark cluster.

The window configuration `rowsBetween(-3, 0)` dictates that when PySpark processes the record for Day N, it looks backward to include Days N-3, N-2, N-1, and N itself in the averaging calculation. This specific boundary setting is the classic definition of a trailing four-period moving average. The subsequent code block applies this logic, creates the new DataFrame `df_new`, and displays the final, enriched DataFrame output:

```
from pyspark.sql import Window
from pyspark.sql import functions as F

#define window for calculating rolling mean
w = (Window.orderBy('day').rowsBetween(-3, 0))

#create new DataFrame that contains 4-day rolling mean column
df_new = df.withColumn('rolling_mean', F.avg('sales').over(w))

#view new DataFrame
df_new.show()

+---+-----+-----------------+
|day|sales| rolling_mean|
+---+-----+-----------------+
| 1| 11| 11.0|
| 2| 8| 9.5|
| 3| 4|7.666666666666667|
| 4| 5| 7.0|
| 5| 5| 5.5|
| 6| 8| 5.5|
```

7	7	6.25
8	7	6.75
9	6	7.0
10	4	6.0
+---+-----+-----------------+

The output `df_new` successfully includes the new column called **rolling_mean**. This column effectively smooths the short-term fluctuations present in the raw **sales** data, thereby providing a clearer trend line for analysts. Examining these results is crucial for validating that the window definition was correctly interpreted and applied by PySpark across the entire dataset.

Detailed Analysis of the Calculated Results

The newly created `rolling_mean` column requires careful inspection, particularly concerning the initialization phase (the first few rows) and the consistency of the calculation throughout the dataset. It is a known feature of rolling mean calculations that they behave differently at the start because there are insufficient preceding data points to fill the full window size defined by the user.

For instance, on **day 1**, only one sales figure (11) is available, so the rolling mean calculation is simply 11.0. On **day 2**, two sales figures are available (11 and 8), yielding an average of (11 + 8) / 2 = 9.5. This partial calculation continues until **day 4**, at which point the full 4-day window is utilized for the first time.

Let us mathematically verify the calculation for two specific subsequent rows where the full window size is consistently engaged:

For the entry corresponding to **day 4**, the rolling mean calculation incorporates the sales figures from days 1 through 4 (11, 8, 4, 5).

Total Sales within Window: 11 (Day 1) + 8 (Day 2) + 4 (Day 3) + 5 (Day 4) = 28

Rolling Mean = 28 / 4 = **7.0**

Similarly, for the entry corresponding to **day 5**, the rolling window slides forward by one period, now incorporating sales from days 2 through 5 (8, 4, 5, 5).

Total Sales within Window: 8 (Day 2) + 4 (Day 3) + 5 (Day 4) + 5 (Day 5) = 22

Rolling Mean = 22 / 4 = **5.5**

This detailed verification confirms that the PySpark `Window` function correctly interprets the `rowsBetween(-3, 0)` command as a trailing 4-day average, dynamically adjusting the divisor

based on the availability of preceding rows at the beginning of the time series.

Customizing the Rolling Mean Period and Lookback

One of the greatest advantages of using PySpark's Window functions is the ease with which the period of the moving average can be modified. The desired length of the lookback period is controlled entirely by the negative index provided in the rowsBetween function, offering high flexibility in statistical modeling.

To generalize this concept, if an analyst requires an N-day rolling average, they must define the window boundaries as `rowsBetween(-(N-1), 0)`. The `-1` adjustment is necessary because the calculation inherently includes the current row (represented by 0), meaning that a 4-day average requires a lookback of 3 preceding rows (resulting in 4 total rows).

For example, should the analysis require a 5-day rolling average instead of the 4-day average demonstrated, the only required modification is changing the start boundary of the window definition. The updated window specification would become `Window.orderBy('day').rowsBetween(-4, 0)`. This inherent flexibility allows data scientists to quickly iterate through different smoothing parameters to find the optimal fit for trend identification and noise reduction. This method of calculating rolling metrics is far more computationally efficient than manual iteration loops, especially when dealing with high-volume datasets processed on an Apache Spark cluster.

Advanced Considerations: Grouped Rolling Means

While the primary example focused on a single time series, real-world data often involves multiple independent series that require parallel rolling mean calculations (e.g., calculating the 4-day rolling sales mean for Store A, Store B, and Store C simultaneously). In such scenarios, the `partitionBy` clause of the Window function becomes an absolutely crucial element of the definition.

If our DataFrame included an additional descriptive column such as `store_id`, the window definition would be expanded to include partitioning: `w = Window.partitionBy('store_id').orderBy('day').rowsBetween(-3, 0)`. This ensures that the rolling mean calculation restarts independently for every unique store identifier, preventing data leakage between groups and ensuring the analytical integrity across separate entities. This capability is a common and necessary requirement in large-scale data aggregation tasks within PySpark environments, enabling complex multi-dimensional time series analysis.

Mastering the effective combination of `partitionBy`, `orderBy`, and boundary functions like `rowsBetween` allows data practitioners to execute highly complex time series operations directly within the PySpark ecosystem, thereby facilitating robust, scalable, and reproducible data

processing workflows.