How to Calculate the Conditional Mean in PySpark: A Step-by-Step Guide

Authored by

stats writer

February 5, 2026

The concept of the Conditional Mean is fundamental in statistical computing and is frequently required when performing advanced Data Analysis. Calculating the conditional mean in PySpark refers to the specialized process of determining the average value of a specific numerical variable within a dataset, contingent upon meeting a defined condition or set of criteria applied to other columns. This capability is efficiently realized through the utilization of PySpark's powerful distributed computing framework and built-in SQL functions, notably groupBy and agg. Understanding how to execute this operation allows data scientists to derive significant insights into variable relationships, supporting effective decision-making in complex Machine Learning preprocessing and analytical tasks.

Calculate Conditional Mean in PySpark

Understanding the Conditional Mean in PySpark

While simple aggregation methods calculate the overall average across an entire column, the Conditional Mean restricts this calculation to a relevant subset of the data. This approach is invaluable when dealing with large-scale datasets, such as those processed by PySpark, because it allows for granular inspection without needing to permanently partition the data. Instead of computing the mean of all basketball player points, for example, we might only want the mean points scored by players on 'Team A' or by players who scored more than 10 points. This conditional filtering is the key differentiator and essential for nuanced Data Analysis.

The operational mechanism in PySpark relies heavily on the DataFrame API, which provides optimized, declarative methods for data manipulation. Although the standard groupBy method can achieve similar results, defining a conditional mean often involves combining the filter transformation with the agg action. This combination first isolates the rows that satisfy the condition and then applies the aggregation function, such as the mean, exclusively to those filtered rows. This method ensures computational efficiency and code clarity, especially in complex data pipelines.

The conditional aggregation pattern demonstrated here is a versatile tool, moving beyond simple averages to complex statistical summaries. Whether the condition is based on categorical data (strings) or continuous numerical values, PySpark handles both scenarios seamlessly. We will explore two primary methods for applying this concept: one utilizing a string variable for segregation and the other utilizing a numeric threshold, showcasing the flexibility of the DataFrame API.

Setting Up the PySpark Environment and Sample Data

Before demonstrating the calculation methods, it is essential to establish a working **SparkSession**

and define the sample DataFrame that will be used throughout the examples. This DataFrame simulates a typical dataset containing performance statistics for basketball players across different teams and positions. We will use this structure to test both string and numeric conditions effectively.

The dataset contains four critical columns: **'team'** (string/categorical), **'position'** (string/categorical), **'points'** (numeric), and **'assists'** (numeric). These variables allow us to calculate the average of a metric (e.g., points or assists) based on conditions applied to either a categorical field (like 'team') or a numerical field (like 'points'). The creation process involves importing the necessary modules, defining the raw data, specifying column names, and finally creating the DataFrame using the **spark.createDataFrame()** method.

The following code snippet demonstrates the necessary steps to initialize the environment and display the resulting DataFrame structure. This setup is crucial for reproducing the conditional mean calculations successfully.

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()

+----+--------+------+-------+
|team|position|points|assists|
+----+--------+------+-------+
| A| Guard| 11| 4|

A	Guard	8	5
A	Forward	22	5
A	Forward	22	9
B	Guard	14	12
B	Guard	14	3
B	Forward	13	5
B	Forward	7	2
+----+--------+------+-------+

Method 1: Conditional Mean Based on String Criteria (Categorical Variables)

This method is employed when the condition for calculating the mean depends on a categorical variable, which is usually stored as a string. A common use case is calculating a performance metric (e.g., points) exclusively for records belonging to a specific group (e.g., team 'A'). The comparison operator used in the filter function will be the equality check (**==**) against the target string value.

The operational sequence here is to use **df.filter()** to isolate rows matching the criteria (**df.team == 'A'**) and then apply the aggregation. We rely on **pyspark.sql.functions**, typically aliased as **F**, to access the F.mean function. The structure is clean and powerful, allowing for complex data slicing.

The following generalized structure demonstrates calculating the mean of the **points** column specifically for players on Team A. Note the use of alias to ensure the output column is clearly labeled.

from pyspark.sql import functions as F

df.filter(df.team=='A').agg(F.mean('points').alias('mean_points')).show()

This particular example calculates the mean value of the **points** column only for the rows where the value in the **team** column is equal to A.

Detailed Example 1: Filtering by Team 'A' (String Variable)

Applying the string-based conditional mean methodology to our sample basketball dataset allows us to answer the question: "What is the average number of points scored by players on Team A?" This requires filtering the **df** where **team** equals 'A', and then calculating the mean of the **points** column using the agg function.

Executing the PySpark code confirms that the total points for Team A (11, 8, 22, 22) sum to 63,

which, divided by four players, yields 15.75. The distributed computation handles this arithmetic efficiently.

We can use the following syntax to calculate the mean value in the **points** column only for the rows where the corresponding value in the **team** column is equal to A:

```
from pyspark.sql import functions as F

#calculate mean value in points column for rows where team column is equal to 'A'
df.filter(df.team=='A').agg(F.mean('points').alias('mean_points')).show()

+-----------+
|mean_points|
+-----------+
| 15.75|
+-----------+
```

We can see that the mean value in the **points** column among players on team A is **15.75**. We utilized the alias function to rename the column in the resulting DataFrame to **mean_points** for enhanced readability.

Method 2: Conditional Mean Based on Numeric Criteria (Thresholds)

The second powerful application of conditional mean involves filtering based on a numerical threshold or range, rather than a fixed categorical label. This technique is extremely useful for performance stratification or when analyzing data based on intensity metrics. For instance, we might want to calculate the average assists only for players who meet a specific scoring benchmark, such as scoring above 10 points.

In this scenario, the filter function utilizes standard numerical comparison operators such as greater than (**>**) or less than (**<**). The structural pattern remains the consistent **df.filter().agg()** workflow, providing uniformity across different comparison types.

The following code structure outlines how to calculate the mean of the **assists** column conditioned on the **points** column exceeding 10.

```
from pyspark.sql import functions as F

df.filter(df.points>10).agg(F.mean('assists').alias('mean_assists')).show()
```

This particular example calculates the mean value of the **assists** column only for the rows where

the value in the **points** column is greater than 10. This method is vital in Machine Learning preprocessing where feature engineering often requires calculating statistics for high-performing subsets.

Detailed Example 2: Filtering by Points Threshold (Numeric Variable)

We now apply the numeric filtering technique to determine the average number of assists provided by players who scored more than 10 points. This condition selects six specific rows from our sample DataFrame. The aggregation function, agg, then computes the mean assists (38 total assists / 6 players) within this high-performance subset.

We can use the following syntax to calculate the mean value in the **assists** column only for the rows where the corresponding value in the **points** column is greater than 10:

from pyspark.sql import functions as F

#calculate mean value in assists column for rows where points is greater than 10
df.filter(df.points>10).agg(F.mean('assists').alias('mean_assists')).show()

```
+-----------------+
| mean_assists|
+-----------------+
|6.333333333333333|
+-----------------+
```

The output reveals that the players scoring above 10 points have an average of 6.333 assists. This demonstrates the precision achievable when applying conditional logic using PySpark DataFrames.

Summary and Further Learning

The methods discussed--filtering by string equality and filtering by numeric thresholds--are foundational approaches to calculating the Conditional Mean in PySpark. By leveraging the distributed processing power of Spark combined with the concise syntax of the DataFrame API's filter and agg functions, complex statistical summaries can be generated quickly and reliably from vast datasets.

For advanced scenarios, it is also possible to integrate complex SQL conditions directly within the filter clause, or even use the more flexible **when().otherwise()** structure within the agg function itself to create conditional sums or counts which can then be used to derive means. However, for straightforward conditional mean calculations based on a single condition, the

df.filter().agg(F.mean()) pipeline remains the clearest and most idiomatic PySpark solution.

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

Performing complex joins on DataFrames.
Handling missing values using fillna and dropna.
Implementing window functions for rolling calculations.
Optimizing data partitioning for performance gains.