Calculate Mean of Multiple Columns in PySpark

Authored by

stats writer

November 16, 2025

RECOMMENDED CITATION

stats writer (2025). *Calculate Mean of Multiple Columns in PySpark*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=92292

ARABPSYCHOLOGY.COM

Calculating statistics like the mean across multiple, specified columns for each row in a DataFrame is a fundamental requirement in large-scale data analysis pipelines. In PySpark, performing this operation efficiently requires leveraging the powerful built-in functions library and dynamic expression generation. This dynamic approach is crucial for maintaining performance across the massive datasets typical of distributed computing environments like Apache Spark.

The following syntax provides a clean, scalable solution for calculating the arithmetic mean value across multiple columns simultaneously within a PySpark DataFrame:

```
from pyspark.sql import functions as F

#define columns to calculate mean for
mean_cols =

#define function to calculate mean
find_mean = F.expr('+'.join(mean_cols))/len(mean_cols)

#calculate mean across specific columns
df_new = df.withColumn('mean', find_mean)
```

This powerful snippet effectively creates a new column, typically named **mean**, which holds the arithmetic average of the values found in the specified input columns, such as **game1**, **game2**, and **game3**. The methodology relies on dynamically generating a SQL expression using Python's string joining capabilities, which is then passed to the F.expr function. This is critical because it ensures that the summation and division are performed efficiently row-wise, meaning the average is computed for each individual observation based on the selected fields.

Understanding the mechanism behind this calculation is vital for efficient PySpark development. We are essentially concatenating the list of column names with the addition operator, calculating the sum of these values for the row, and then dividing that sum by the count of the columns provided (found using the standard Python function `len()`). The subsequent sections will walk through a detailed, practical example demonstrating how to implement and verify this calculation in a real-world scenario.

Introduction to Column-Wise Mean Calculation in PySpark

When working with large-scale data processing using the PySpark framework, a frequent analytical task involves computing row-level metrics derived from multiple features. Unlike standard SQL aggregation, which typically calculates the column mean (averaging all values within one column across all rows), here we focus on the row mean--the average of several columns for a single record. This operation is critical in scenarios like calculating average performance metrics,

ARABPSYCHOLOGY.COM

normalizing feature vectors, or summarizing repeated measurements for an individual entity.

Traditional data processing libraries might handle this using explicit iteration or specialized map functions, but in the distributed context of Apache Spark, we must rely on optimized Spark SQL functions to ensure maximum performance and scalability. The method we employ utilizes the `pyspark.sql.functions` module, specifically leveraging the F.expr function, which allows us to execute complex SQL expressions directly against the DataFrame columns. This strategy prevents costly data serialization and movement, keeping the computation within the highly optimized Spark engine.

The primary benefit of the `F.expr` approach, coupled with dynamic string generation, is its flexibility. If the number of columns to be averaged changes--for instance, if we start tracking 10 games instead of 3--the code structure remains the same, requiring only an update to the initial `mean_cols` list. This adaptability makes the solution highly valuable in dynamic analytical environments where data schema frequently evolves.

The Challenge of Row-Wise Operations in Spark

Spark is fundamentally optimized for columnar operations, making row-wise calculations slightly less intuitive than standard aggregations like calculating the average of an entire column (e.g., `F.avg('column_name')`). To compute the sum of values across an arbitrary list of columns for every row efficiently, we must generate a dynamic expression that Spark can process natively. Manually writing out the entire summation expression--such as `df.withColumn('mean', (df + df + df) / 3)`--becomes impractical, tedious, and error-prone when dealing with dozens or hundreds of columns.

The solution presented here cleverly bypasses this manual effort by dynamically generating the summation string required by the F.expr function. By defining the list of column names (`mean_cols`) and using Python's `'+'.join(mean_cols)` command, we construct the necessary arithmetic expression string, which will look like `'game1+game2+game3'`. This string is the core component passed into `F.expr`, allowing Spark SQL to interpret and execute the summation across the selected fields simultaneously.

The final step involves dividing this summation expression by the count of the columns provided (`len(mean_cols)`). This crucial division completes the calculation, resulting in the accurate mean value for each row. This dynamic method ensures that the calculation is always mathematically sound regardless of the number of columns involved, preserving computational integrity across various data sizes.

ARABPSYCHOLOGY.COM

Prerequisites and Setup for PySpark Operations

Before implementing the dynamic mean calculation, it is essential to establish a Spark session and prepare the initial DataFrame. This section outlines the standard setup required in any PySpark environment, including importing the necessary classes and defining the input data structure. We begin by importing `SparkSession` to initialize the Spark context, which serves as the essential gateway to utilizing all Spark functionality.

The example dataset provided simulates basketball game scores, where we have team identifiers and points scored across three different games. This realistic setup is crucial for visualizing exactly how the row-wise operation impacts each record individually. The initial data preparation step also ensures that the resulting calculation is performed on numeric data types, which is a necessary prerequisite for accurate arithmetic operations.

The code below demonstrates the creation of the sample DataFrame. This structure will serve as our input for the subsequent mean calculation, allowing us to accurately track the transformation process from raw data to derived metric.

Practical Example: Defining the Input Data

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players during three different games:

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()

ARABPSYCHOLOGY.COM

```
+-------+-----+-----+-----+
| team|game1|game2|game3|
+-------+-----+-----+-----+
| Mavs| 25| 11| 10|
| Nets| 22| 8| 14|
| Hawks| 14| 22| 10|
| Kings| 30| 22| 35|
| Bulls| 15| 14| 12|
|Blazers| 10| 14| 18|
+-------+-----+-----+-----+
```

The generated DataFrame, `df`, contains six rows representing six teams, and four columns, where the last three (`game1`, `game2`, and `game3`) are the numeric fields we are interested in averaging. Our objective is to add a new column called **mean** that contains the mean of points scored by each player across all three games.

This setup clearly illustrates the need for a dynamic solution. We require a single average score per team, derived horizontally across the score columns. If the number of games tracked were to increase, the definition of the target columns would only need a minor adjustment to the `mean_cols` list, proving the scalability of the chosen method.

Executing the Row-Wise Mean Calculation

To implement the desired row-wise mean calculation, we utilize the imported functions and the dynamic expression generation method discussed earlier. The core logic is encapsulated in the variable `find_mean`, which does not store a fixed value but rather a Spark Column object representing the pending calculation across every row.

The definition of `find_mean` ensures that the sum of the specified columns is calculated and immediately divided by the total count of those columns. This approach is highly performant because it leverages Spark SQL's optimization engine to process the arithmetic calculation in parallel across all data partitions, minimizing latency even on massive distributed datasets. This ability to execute complex string-based SQL expressions is one of the most powerful features of `pyspark.sql.functions`.

Once the calculation logic is defined, the withColumn transformation is applied to the original `df`. This results in the creation of a new DataFrame, `df_new`, which includes the calculated **mean** column, preserving the immutability of the original data object.

from pyspark.sql import functions as F

ARABPSYCHOLOGY.COM

```
#define columns to calculate mean for
mean_cols =

#define function to calculate mean
find_mean = F.expr('+'.join(mean_cols))/len(mean_cols)

#calculate mean across specific columns
df_new = df.withColumn('mean', find_mean)

#view new DataFrame
df_new.show()

+-------+-----+-----+-----+-----------------+
| team|game1|game2|game3| mean|
+-------+-----+-----+-----+-----------------+
| Mavs| 25| 11| 10|15.333333333333334|
| Nets| 22| 8| 14|14.666666666666666|
| Hawks| 14| 22| 10|15.333333333333334|
| Kings| 30| 22| 35| 29.0|
| Bulls| 15| 14| 12|13.666666666666666|
|Blazers| 10| 14| 18| 14.0|
+-------+-----+-----+-----+-----------------+
```

Verifying and Interpreting the Results

Notice that the new **mean** column has been successfully added to the resulting DataFrame. This column provides the average score across the three games for each corresponding team. It is essential to perform a quick verification to ensure that the complex, dynamically generated calculation aligns perfectly with fundamental arithmetic principles.

The values in the **mean** column represent the division of the sum of `game1`, `game2`, and `game3` by 3. This successful execution demonstrates the elegance and efficiency of using `F.expr` for dynamic, row-wise computations in PySpark. Let us examine a few specific records to confirm the arithmetic mean was computed accurately:

The mean of points for the **Mavs** player is calculated as (25 + 11 + 10) / 3, resulting in approximately **15.333**.

The mean of points for the **Nets** player is calculated as (22 + 8 + 14) / 3, resulting in approximately **14.667**.

ARABPSYCHOLOGY.COM

The mean of points for the **Hawks** player is calculated as (14 + 22 + 10) / 3, resulting in approximately **15.333**.

The mean of points for the **Kings** player is calculated as (30 + 22 + 35) / 3, resulting in exactly **29.0**.

The successful matching between manual calculation and the PySpark output validates the methodology, confirming that `F.expr` correctly interpreted the dynamically generated summation expression.

Importance of the withColumn Function

The overall integrity and success of this operation relies fundamentally on the use of the withColumn transformation. This standard PySpark method is the conventional tool for adding a new column or replacing an existing column based on the results of a given expression. In our context, it takes the complex calculation logic defined in `find_mean` and maps it directly to the new **mean** column across every row of the distributed dataset.

It is vital to reiterate that Spark DataFrames are inherently immutable structures. The withColumn function does not modify the original `df` in place; instead, it returns an entirely new DataFrame (`df_new`) that incorporates the result of the transformation. This immutable design principle is crucial for ensuring data integrity, simplifying debugging, and enabling reliable distributed processing, as operations can be tracked and rerun deterministically.

For developers seeking deeper insights into how column-level transformations are handled in a distributed cluster, or for information on advanced features like chaining multiple transformations, consulting the official documentation for the PySpark withColumn function is highly recommended. Mastering this function is key to performing almost any transformation or feature engineering task efficiently within the Spark ecosystem.

ARABPSYCHOLOGY.COM