How to Calculate the Mean of Multiple Columns in a PySpark DataFrame

Authored by

stats writer

February 8, 2026

The ability to efficiently analyze large datasets is paramount in modern data engineering, and calculating the arithmetic mean of numerical features is a fundamental operation. In the context of PySpark, calculating the average across multiple columns in a DataFrame is a common requirement, often needed for feature engineering or descriptive statistics. This process involves determining the central tendency for specific data points, usually resulting in a new derived column representing the average value per row, or an aggregated row representing the average value per column.

While **PySpark** provides the specialized **agg** function for performing group-wise or dataset-wide aggregations--such as calculating the mean of an entire column--a different approach is necessary when the goal is to compute the average **across multiple columns for each individual row**. This technique is especially useful when normalizing features or creating composite metrics from existing variables. Mastering this distinction is essential for leveraging PySpark's parallel processing capabilities effectively when dealing with voluminous data.

This guide focuses on generating a new column that holds the row-wise mean derived from several numerical input columns. We will utilize powerful PySpark functions such as withColumn in conjunction with SQL expression parsing via F.expr. This combination allows for dynamic and scalable operations without resorting to User Defined Functions (UDFs), which typically incur significant performance overhead in distributed computing environments.

Calculate Mean of Multiple Columns in PySpark

Core Methodology for Row-wise Mean Calculation

Calculating the mean value across multiple columns for every record in a **PySpark DataFrame** requires constructing a flexible expression that can sum the values of the target columns and divide by the count of those columns. This methodology avoids iterative loops, ensuring that the operation is executed in a highly optimized manner across the distributed cluster infrastructure provided by Apache Spark. The foundation of this approach relies on the **pyspark.sql.functions** module, aliased typically as **F**, which grants access to essential SQL-like operations.

The crucial step involves defining the list of columns subject to the mean calculation. Once defined, these column names are dynamically joined together using the addition operator within a string expression. This string expression is then parsed and executed by the F.expr function. By leveraging **F.expr**, we are effectively utilizing Spark's highly optimized Catalyst Optimizer to handle the complex arithmetic expression, leading to superior performance compared to native Python operations applied row by row.

The following syntax illustrates the fundamental pattern used to calculate the row-wise mean and

assign it to a new column within your existing **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)
```

As demonstrated in the code snippet above, the resulting expression, stored in the variable **find_mean**, encapsulates the entire calculation logic. This derived value is then seamlessly integrated into the original **DataFrame** (**df**) using the withColumn function. The **withColumn** transformation is non-mutating, meaning it returns a new DataFrame, **df_new**, which includes the newly computed column, typically named **mean**, alongside all existing columns. This design principle ensures data integrity and adherence to functional programming paradigms prevalent in Spark development.

Setting Up the PySpark Environment and Sample Data

To demonstrate the practical application of calculating row-wise means, we first need to initialize a **SparkSession**, which serves as the entry point to all functionality in PySpark. After establishing the session, a sample dataset must be defined to mimic real-world scenarios, such as tracking performance metrics over multiple trials or periods. Our example uses basketball scores, where we track points achieved by different teams across three distinct games.

Defining the data structure involves creating a list of lists (the data rows) and a corresponding list of column names (the schema). This explicit definition ensures that the subsequent creation of the **DataFrame** is structured correctly, assigning appropriate names and ensuring type inference works accurately. Numerical data types are crucial here, as the mean calculation relies entirely on arithmetic operations; thus, column types for the score columns must be integers or floats.

The following code block meticulously sets up the **PySpark DataFrame**, providing a solid foundation for the subsequent mean calculation steps. Notice the use of **spark.createDataFrame()**, which is the standard method for converting local Python data structures into distributed Spark DataFrames:

```
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|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|
+-------+-----+-----+-----+
```


Upon execution, the **df.show()** command provides a visualization of the initial dataset. We can clearly see five columns: a categorical column (**team**) and three numerical columns (**game1**, **game2**, **game3**) representing the scores. The objective now is to derive a new column, provisionally named **mean**, where each row reflects the average score achieved by that specific team across the three games. This metric summarizes the overall performance efficiency of each entry.

Dynamic Calculation using PySpark Functions

The power of PySpark lies in its ability to handle complex operations through concise, declarative syntax. Calculating the mean across dynamic sets of columns leverages this principle. Instead of

hardcoding the summation, we define the list of column names, **mean_cols**, which provides flexibility; if the number of games changes, only this list needs modification.

The core expression creation involves two primary steps. First, the **join** method in Python strings is used to concatenate the elements of **mean_cols** with the addition operator (**+**) as the separator. For our example, this generates the string **'game1+game2+game3'**. Second, this generated string is passed into F.expr, transforming the simple string into a valid, optimized SQL expression that Spark can execute efficiently.

Finally, to complete the arithmetic mean calculation, the total sum derived from the **F.expr** output must be divided by the count of columns being averaged. This count is determined simply by calculating the length of the **mean_cols** list using **len(mean_cols)**. Storing this complete logic in a single variable, **find_mean**, makes the subsequent DataFrame transformation extremely clean and readable, clearly separating the definition of the calculation from its application.

Applying the Mean Calculation with withColumn

The operation to introduce the newly calculated mean into the **DataFrame** is handled by the specialized withColumn function. This function is designed to return a new DataFrame where a specified column has been added or replaced. In our context, we are adding the new **mean** column, using the meticulously constructed expression from the previous step as its value generator.

The syntax **df_new = df.withColumn('mean', find_mean)** is highly declarative. It instructs PySpark to take the original DataFrame **df**, calculate the values based on the expression **find_mean** for every row, and label the resulting column **mean** in the new DataFrame **df_new**. Importantly, this operation is executed lazily, meaning the calculation is only triggered when an action, such as **df_new.show()**, is called.

Executing the transformation and displaying the results confirms the successful integration of the calculated row-wise averages. The resulting output clearly demonstrates how the new column summarizes the performance across the individual game scores for each team:

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)

#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|
+-------+-----+-----+-----+-----------------+
```

Verification and Interpretation of Results

The output reveals the newly appended **mean** column, which contains floating-point numbers representing the calculated average scores. It is crucial to verify these results manually for a few records to confirm the accuracy of the complex **F.expr** calculation executed by **PySpark**. This verification step ensures that the dynamic string construction and the division operation functioned as intended.

For instance, examining the data for the **Mavs** team, the scores were 25, 11, and 10. The summation is (25 + 11 + 10) = 46. Dividing this sum by the count of games (3) yields 46 / 3, which is approximately 15.3333... This matches the value provided in the **df_new.show()** output, confirming the calculation's fidelity. Similarly, for the **Kings** team, the scores are 30, 22, and 35, totaling 87. Dividing 87 by 3 results in exactly 29.0, which is also correctly reflected in the new DataFrame.

This verification demonstrates that the row-wise mean was successfully computed, providing immediate insight into the central performance level of each team across the monitored period. This technique is superior to using iterative Python loops (like Pandas **apply**), as **PySpark** handles the calculation in a distributed, optimized manner, making it suitable for petabyte-scale datasets without performance degradation.

Specific examples of the calculated means for verification are as follows:

The **Mavs** player mean score is calculated as (25 + 11 + 10) / 3 = **15.33**.
The **Nets** player mean score is calculated as (22 + 8 + 14) / 3 = **14.67**.

The **Hawks** player mean score is calculated as (14 + 22 + 10) / 3 = **15.33**.

Alternative: Calculating Column-wise Means (Aggregation)

While the primary focus of the dynamic **F.expr** method is calculating the mean across columns for each row, it is important to acknowledge the standard method for calculating the mean of an entire column across all rows. This is achieved using aggregation functions in **PySpark**, typically involving the **agg** method. If the goal is to summarize the performance of **game1** across all teams, aggregation is the correct path.

To perform column-wise aggregation, one would typically use **df.agg()** and pass a dictionary or list of tuples specifying the aggregation function (**F.mean**) to be applied to the target columns. The output of this operation is a significantly smaller DataFrame, containing only a single row, where each cell represents the calculated mean for the specified column across the entire dataset. This is commonly used for generating summary statistics of the data distribution.

For example, calculating the mean points scored in **game1**, **game2**, and **game3** across all teams requires a distinct approach compared to the row-wise calculation demonstrated earlier. This distinction highlights the flexibility of **PySpark** in addressing different statistical needs, whether summarizing attributes (row-wise) or summarizing features (column-wise aggregation).

Performance Considerations and Best Practices

When dealing with big data, performance is paramount. Using **F.expr** and withColumn for row-wise calculation is considered a best practice because it avoids converting data back to native Python objects. If we had chosen to use a **User Defined Function (UDF)** implemented in Python, the data would need to be serialized out of the JVM (Java Virtual Machine) and into the Python environment for calculation, then serialized back, which introduces considerable performance penalties, especially on wide DataFrames.

Furthermore, defining the list of columns dynamically, as shown with **mean_cols**, ensures that the code remains maintainable. If the schema of the source data changes--for instance, if **game4** is introduced--only the **mean_cols** list needs to be updated. The rest of the calculation logic (**F.expr('+'.join(mean_cols))/len(mean_cols)**) automatically adapts, promoting code robustness and scalability in production environments.

Finally, always ensure that the columns used in the mean calculation are numeric. **PySpark** is generally strict about type consistency. Attempting to apply the summation and division logic to non-numeric columns (like strings or dates) will result in execution errors, as the **F.expr** function relies on the underlying SQL engine to perform valid arithmetic operations. Prior data cleansing and casting steps are often necessary before applying sophisticated transformations like this.

It is important to reiterate that the withColumn function is central to this operation, as it returns a new immutable DataFrame with the new **mean** column added and all existing columns preserved. This approach is highly efficient for large-scale data manipulation in PySpark environments.