Learning to Calculate Standard Deviation in PySpark DataFrames

Authored by

Mohammed looti

November 11, 2025

The ability to calculate measures of dispersion is fundamental in data analysis, particularly when working with large datasets processed by frameworks like PySpark DataFrames. The Standard deviation (SD) provides a crucial insight into the volatility or spread of data points around the mean. A low standard deviation indicates that the data points tend to be close to the mean, while a high standard deviation indicates that the data is spread out over a wider range of values. This guide, written for data professionals and engineers, explores the two primary, highly efficient methods available in PySpark SQL functions for computing the standard deviation of numerical columns.

Introduction to Standard Deviation in PySpark

PySpark, the Python API for Apache Spark, offers robust functionalities for statistical analysis. When dealing with numerical features within a PySpark DataFrame, calculating metrics like standard deviation is a routine operation often performed during exploratory data analysis (EDA) or feature engineering. PySpark provides specialized functions, accessible via `pyspark.sql.functions`, that are optimized for distributed computing, ensuring that computations scale effectively across large clusters.

We will demonstrate two distinct programmatic approaches to computing the standard deviation. The first method utilizes the powerful aggregation function `agg()`, which is ideal when you need to calculate the standard deviation for a single column and retrieve the result immediately as a scalar value. The second method leverages the `select()` transformation combined with the stddev function directly, making it highly efficient for calculating this metric simultaneously across multiple columns and returning the results within a new, compact DataFrame.

Choosing the correct method depends primarily on the scope of the calculation (single column vs. multiple columns) and the desired output format (a scalar value versus a resulting DataFrame). Regardless of the choice, both methods rely on vectorized operations, allowing Spark to distribute the workload efficiently and deliver performance superior to traditional Python methods when handling petabyte-scale data.

Setting Up the PySpark Environment and Sample Data

Before performing any calculations, it is essential to establish a SparkSession and define the sample data we will use throughout the examples. Our sample data simulates scores for different teams across three separate games. This setup ensures that our demonstrations are reproducible and easily verifiable.

The following setup code initializes the Spark environment, defines the data structure--which includes four columns: `team`, `game1`, `game2`, and `game3`--and creates the final PySpark DataFrame named `df`. It is crucial to import the necessary components from `pyspark.sql` to begin the

process.

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

Once the DataFrame `df` is created and displayed, we have the necessary structure to proceed with statistical calculations. The three numerical columns--`game1`, `game2`, and `game3`--will serve as the focus for calculating their respective standard deviations, illustrating the spread of scores for each game across the competing teams.

Method 1: Calculating Standard Deviation for a Single Column using `agg()`

The first and often most direct method for calculating a single aggregate statistic like the standard

deviation involves using the `agg()` function. This approach requires importing the PySpark functions module, typically aliased as `F`, to access the optimized statistical functions. The `agg()` method is particularly useful when the analyst requires the resulting standard deviation value to be immediately extracted and used in subsequent non-Spark operations, or when performing a quick check during iterative development.

To retrieve a scalar value (a single number) representing the standard deviation of a column, we chain the `agg()` function with `collect()`. The `collect()` action pulls the result from the distributed Spark environment back to the driver program, and the index access isolates the specific value from the resulting row structure.

from pyspark.sql import functions as F

#calculate standard deviation of values in 'game1' column
df.agg(F.stddev('game1')).collect()

Let us apply this powerful technique to calculate the standard deviation for the `game1` column in our sample dataset.

Example 1: Calculate Standard Deviation for One Specific Column

We apply the `agg()` syntax directly to the DataFrame `df`, specifically targeting the `game1` column using the stddev function. This calculation yields the measure of dispersion for the scores in that particular game.

from pyspark.sql import functions as F

#calculate standard deviation of column named 'game1'
df.agg(F.stddev('game1')).collect()

7.5806771905065755

Upon execution, the standard deviation for the values recorded in the **game1** column is determined to be approximately **7.5807**. This output confirms the method's effectiveness in providing a quick, precise scalar result for focused column analysis.

Method 2: Calculating Standard Deviation Across Multiple Columns using `select()`

When the analytical requirement is to compute the standard deviation for several numerical columns simultaneously, the use of the `select()` transformation combined with the imported

stddev function is generally preferred. This method avoids the need to iterate through columns manually and keeps the computation within the highly optimized Spark SQL framework, returning a new DataFrame containing the results.

By importing `stddev` directly from `pyspark.sql.functions`, we can apply it to multiple column references within a single `select()` statement. This not only streamlines the code but also ensures that the calculation is executed in parallel across the distributed cluster, maximizing performance when dealing with high dimensionality. The resulting DataFrame is typically cleaner for visualization or further programmatic manipulation compared to extracting multiple scalar values via `collect()`.

from pyspark.sql.functions import stddev

#calculate standard deviation for game1, game2 and game3 columns
df.select(stddev(df.game1), stddev(df.game2), stddev(df.game3)).show()

This syntax instructs PySpark to calculate the standard deviation for all specified columns (`game1`, `game2`, and `game3`) and output a resultant DataFrame containing these three calculated metrics, each housed in its own column with an auto-generated descriptive name (e.g., `stddev_samp(game1)`).

Example 2: Calculate Standard Deviation for Multiple Columns

Applying the `select()` method to calculate the standard deviation across all three game columns demonstrates the efficiency of this approach. We can clearly compare the data spread for each game score simultaneously, a critical step in comparative statistical analysis.

from pyspark.sql.functions import stddev

#calculate standard deviation for game1, game2 and game3 columns
df.select(stddev(df.game1), stddev(df.game2), stddev(df.game3)).show()

```
+------------------+------------------+------------------+
|stddev_samp(game1)|stddev_samp(game2)|stddev_samp(game3)|
+------------------+------------------+------------------+
|7.5806771905065755| 5.741660619251774| 9.544631999192006|
+------------------+------------------+------------------+
```

The resulting output DataFrame concisely presents the calculated standard deviations for all three columns. Analyzing this result allows us to draw immediate conclusions about the relative consistency of the scores across the games.

The standard deviation of values in the **game1** column is **7.5807**.

The standard deviation of values in the **game2** column is **5.7417**.

The standard deviation of values in the **game3** column is **9.5446**.

Based on these findings, `game2` exhibits the lowest standard deviation, suggesting that the scores were most tightly clustered around the mean for that game, indicating higher consistency among team performances. Conversely, `game3` shows the highest standard deviation, implying the widest variance in scores.

Understanding Sample vs. Population Standard Deviation

A critical distinction in statistical analysis is whether the data being analyzed represents the entire population or merely a sample drawn from a larger population. This distinction dictates which formula for standard deviation should be employed. PySpark provides two functions to handle these scenarios precisely.

By default, the **`stddev`** function (or `F.stddev`) calculates the Sample standard deviation. This method uses a denominator of $N-1$ (where N is the number of data points) in its variance calculation, a technique known as Bessel's correction, which is necessary to provide an unbiased estimate of the population standard deviation when only a sample is available. This is the most common requirement in data analysis where the dataset in hand is typically considered a subset of all possible observations.

If, however, your PySpark DataFrame truly contains data for the entire population--meaning every single possible observation is present--you must use the stddev_pop function. This function calculates the Population standard deviation, which uses N as the denominator in the variance calculation. Using the correct function is paramount for maintaining statistical integrity.

If you would instead like to use the population standard deviation formula, then use the **`stddev_pop`** function instead. While the difference between the two functions is negligible for very large datasets, using the correct statistical definition is crucial for accuracy, especially with smaller data samples.

Note: The **`stddev`** function, like most aggregation functions in PySpark, handles missing data gracefully. If there are null values in the column being analyzed, the function will automatically ignore these records by default, ensuring that the standard deviation is calculated only on non-null numerical observations.

Additional Resources

Mastering statistical operations like standard deviation calculation is just one facet of effective data processing in a distributed environment. Expanding your knowledge of PySpark's statistical capabilities will significantly enhance your ability to perform complex data analysis at scale.

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

Calculating Mean and Median in PySpark

Handling Skewed Data Distributions using PySpark

Advanced Aggregation Techniques in PySpark SQL