How to Calculate Standard Deviation with PySpark in 3 Steps

Authored by

stats writer

January 21, 2026

The Standard deviation is one of the most fundamental measures in descriptive statistics, quantifying the amount of variation or dispersion within a set of data values. Essentially, it tells us how spread out the data points are relative to the average, or mean, value. A low standard deviation indicates that the data points tend to be very close to the mean, while a high standard deviation indicates that the data points are spread out over a wider range of values. Understanding this metric is crucial for tasks ranging from financial risk assessment to quality control.

When working with massive datasets, utilizing a powerful distributed computing framework like PySpark becomes essential. PySpark is the Python API for Apache Spark, designed to handle large-scale data processing in a scalable and fault-tolerant manner. Calculating statistical metrics, including standard deviation, must be optimized for this distributed environment. PySpark handles this complexity seamlessly, using optimized functions within its SQL module to manage the parallel calculations across various worker nodes.

Within PySpark, the standard deviation calculation is implemented through functions available in the **pyspark.sql.functions** module. This function accepts a column of numerical data from a DataFrame as its argument and returns the resultant standard deviation. The underlying calculation follows the traditional statistical formula: determining the variance by summing the squared differences from the mean, and then taking the square root. Crucially, PySpark manages this computation across partitioned data, ensuring accuracy and efficiency by aggregating the results from all executors to produce the final, definitive statistic.

Calculating Standard Deviation in PySpark DataFrames

PySpark offers specialized functions tailored for calculating standard deviation, allowing users to choose between two main approaches: calculating the sample standard deviation or the population standard deviation. The most commonly used function, **stddev** (or **stddev_samp**), calculates the sample standard deviation, which is appropriate when your dataset represents a sample drawn from a larger population. This distinction is vital in accurate statistical analysis.

The standard deviation functions are highly versatile and can be applied in several contexts within a PySpark workflow. Whether you need to find the dispersion for a single variable across the entire dataset, or calculate it for multiple variables simultaneously, the PySpark SQL functions provide clean and efficient syntax. The two primary patterns for implementation often depend on whether the user is aggregating a result for the whole DataFrame or selecting computed columns alongside existing data.

Below, we will demonstrate the two most common procedural methods for applying these calculations within a PySpark DataFrame. These methods utilize the built-in functions provided by the library, ensuring that the computation is handled optimally within the Spark engine.

Understanding both the single-column aggregation method and the multi-column selection method is key to mastering statistical analysis in PySpark.

You can use the following methods to calculate the standard deviation of a column in a PySpark DataFrame:

Method 1: Calculating Standard Deviation for a Single Column

When the objective is to determine the standard deviation of just one specific column and retrieve that single numerical result, the recommended approach involves using the **agg** function combined with the standard deviation function imported as an alias, typically **F**. The **agg** method is designed for applying aggregate functions across the entire dataset or group, efficiently condensing the results into a single row or value.

This method is particularly useful in data profiling or summary statistics generation, where you only require the final statistical measure rather than a new column appended to the existing DataFrame. The syntax requires specifying the column name as a string argument within the **stddev** function. To extract the raw float value in a non-distributed environment (like a local script), the use of **collect()** is often employed to pull the result from the distributed structure back to the driver program.

from pyspark.sql import functions as F

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

Method 2: Calculating Standard Deviation for Multiple Columns

If the requirement is to calculate the standard deviation for several columns simultaneously and view the results in a formatted output, the **select** transformation is highly effective. Using **select**, you can apply the **stddev** function to multiple columns, treating each resulting statistic as a newly calculated column in the output DataFrame. This produces a concise, single-row DataFrame showing all calculated standard deviations.

This approach is preferred when comparing the dispersion across different variables within the dataset. By explicitly calling the **stddev** function on each column reference (e.g., **df.game1**), we instruct Spark to compute the statistic for those specific variables. The output is easily visualized using the **show()** action, which prints the results directly to the console.

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()
```

Note: The **stddev** function uses the sample standard deviation formula to calculate the standard deviation.

Sample vs. Population Standard Deviation in PySpark

It is crucial for accurate statistical modeling to differentiate between the sample standard deviation and the population standard deviation. PySpark provides dedicated functions for both. The default function, **stddev** (or its explicit alias **stddev_samp**), calculates the sample standard deviation, dividing by (N - 1) in the variance calculation to provide an unbiased estimate of the population standard deviation based on the sample data. This is suitable for most analytic tasks where the data is derived from a larger population.

Conversely, if your dataset represents the entire population (i.e., you have data for every single entity you are interested in), you must use the **stddev_pop** function. This function employs the population formula, which involves dividing the sum of squared differences by N (the total count of observations). Using the wrong formula can lead to systemic bias in statistical inference, especially in smaller datasets. Therefore, data scientists must confirm the nature of their data--sample or complete population--before selecting the appropriate PySpark function.

If you would instead like to use the population standard deviation formula, then use the **stddev_pop** function instead.

Setting Up the PySpark DataFrame for Examples

To illustrate the practical application of these standard deviation methods, we must first establish a working DataFrame. This example uses simulated sports data, where teams have numerical scores across three hypothetical games. The initialization process involves creating a Spark Session, defining the schema (column names), and converting the local Python data structure into a distributed PySpark DataFrame. This setup ensures a clean environment for executing the subsequent statistical calculations.

The following code snippet demonstrates the required imports and steps necessary to generate the sample dataset. This dataset will serve as the foundation for both the single-column and multi-column standard deviation calculations demonstrated in the subsequent sections, allowing us to verify the output results effectively.

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

Example 1: Calculating Standard Deviation for a Single Column

In this practical demonstration, we apply Method 1 to calculate the standard deviation for the scores recorded in the **game1** column. This approach uses the **agg** function imported from **pyspark.sql.functions** as the alias **F**. Aggregation is the most direct way to calculate summary statistics for an entire column, returning a concise result.

The code first imports the necessary functions. It then calls the **agg** function on the DataFrame, supplying the **F.stddev** function targeted at the **game1** column. Finally, **collect()** is used to extract the raw floating-point value from the single-cell resulting DataFrame, making it immediately available for use in Python.

from pyspark.sql import functions as F

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

```
7.5806771905065755
```

The calculated standard deviation of values in the **game1** column is approximately **7.5807**. This high value suggests a considerable spread in the scores recorded across the teams for this particular game.

Example 2: Calculating Standard Deviation Across Multiple Columns

For scenarios requiring simultaneous calculation and comparison of dispersion across multiple variables, Method 2 provides a tabular output that is easy to read and interpret. We utilize the **select** function to project the results of the **stddev** function applied individually to **game1**, **game2**, and **game3**.

This method generates a new temporary DataFrame consisting solely of the calculated standard deviation values. Notice that the resulting column names are automatically generated by Spark (e.g., **stddev_samp(game1)**), reflecting the function used. The **show()** command then renders this result, allowing for immediate comparison of the variability in performance across the three different games.

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

Analyzing the output from the multi-column calculation provides clear comparative statistics:

The standard deviation for **game1** scores is approximately **7.5807**, confirming the previous single-column result.
The standard deviation for **game2** scores is **5.7417**, indicating that scores in this game were the least spread out among the three.
The standard deviation for **game3** scores is the highest at **9.5446**, signifying the greatest degree of dispersion and variability in team performance for this game.

Handling Null Values and Further Applications

An important operational detail when calculating statistics in PySpark is how the statistical functions manage missing or null data. By default, the **stddev** function, like most aggregation functions in PySpark, employs a robust mechanism to handle null values: they are automatically ignored during the calculation process. This behavior ensures that valid data points are used correctly without requiring explicit filtering steps prior to calculating the statistic, simplifying the data preparation pipeline significantly.

This default behavior (ignoring nulls) is generally desirable for producing accurate statistical summaries, as including nulls would skew the sample size calculation and potentially lead to inaccurate results. However, users should always be aware of the presence of nulls if they intend to analyze the completeness of their data. If strict filtering or imputation is required, those steps should be performed explicitly before invoking the standard deviation functions.

Note: If there are null values in the column, the **stddev** function will ignore these values by default.

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