A Comprehensive Guide to Descriptive Statistics with PySpark DataFrames

Authored by

Mohammed looti

November 11, 2025

RECOMMENDED CITATION

Mohammed looti (2025). *A Comprehensive Guide to Descriptive Statistics with PySpark DataFrames*. PSYCHOLOGICAL STATISTICS. Retrieved from https://statistics.arabpsychology.com/?p=16786

In the high-stakes environment of big data processing, the ability to rapidly generate accurate and insightful summary statistics is paramount for effective Exploratory Data Analysis (EDA). When dealing with petabyte-scale datasets, relying on tools engineered for distributed computation, like **PySpark**, is no longer optional--it is a necessity. PySpark offers highly scalable and robust methodologies for computing descriptive statistics across diverse data types housed within a PySpark DataFrame. These essential metrics enable data scientists to quickly grasp the core characteristics of their data, including central tendency, dispersion, and the overall shape of the underlying distributions.

This comprehensive guide focuses specifically on mastering the powerful, built-in `.summary()` method, which is accessible directly on the PySpark DataFrame object. This function is designed for flexibility, providing analysts with the capability to retrieve either a predefined, comprehensive battery of descriptive statistics or a precisely customized selection of measures tailored to specific analytical needs. Critically, we will also explore practical strategies for filtering the analysis to focus exclusively on relevant **numeric columns**, a technique essential for streamlining the interpretation process and ensuring that statistical results remain mathematically sound by preventing the calculation of meaningless averages or standard deviations on categorical fields. Proficiency in these methods is fundamental for rigorous data quality assurance and preliminary statistical modeling within any distributed computing architecture.

Core Methods for Generating PySpark Summary Statistics

The `.summary()` function stands as the foundational utility for deriving descriptive statistics within the PySpark ecosystem. Its architecture is explicitly optimized for efficiency across **distributed nodes**, facilitating rapid calculations even when processing datasets spanning multiple terabytes. To cater to varied analytical requirements, we will illustrate three distinct paradigms for invoking this function. These methods progress from providing a broad, default snapshot of the data to executing a highly focused, statistically rigorous analysis exclusively on numeric fields. Understanding these variations grants the user necessary control over the scope, precision, and granularity of the statistical output generated.

The three techniques outlined below are structured to offer progressive refinement over the output data profile. Initially, the default invocation provides a standard, broad overview of all columns. The second method grants analysts the power to specify exact distribution markers, such as specific quantiles or boundary values. Finally, the third and most sophisticated approach utilizes **data type filtering** to ensure only mathematically meaningful results are returned for quantitative fields. These techniques showcase the flexibility of PySpark's DataFrame API in addressing common data profiling requirements encountered in modern data science workflows.

Method 1: Calculate Comprehensive Summary Statistics for All Columns: This approach

utilizes the function without arguments, triggering the calculation of PySpark's standard set of descriptive measures across every feature.

df.summary().show()

Method 2: Calculate Specific Distribution Markers for All Columns: This involves explicitly providing a sequence of desired metrics (such as minimum, maximum, or specific quantiles) as string arguments to the function.

df.summary('min', '25%', '50%', '75%', 'max').show()

Method 3: Calculate Summary Statistics Exclusively for Numeric Columns: This advanced technique requires programmatically identifying and selecting only the numeric columns before applying the summary function, ensuring statistical purity.

numeric_cols =

df.select(*numeric_cols).summary().show()

Prerequisite: Setting Up the PySpark Environment and Sample Data

Before we can execute the descriptive statistics methods, the foundational step involves initializing a SparkSession. This session acts as the essential entry point and orchestrator for all operations conducted within the Apache Spark framework. Following the successful activation of the session, we will construct a small but illustrative dataset. This sample data is designed to model basketball player performance metrics, purposefully incorporating a mix of data types: categorical features like `'team'` and `'conference'`, alongside truly quantitative fields such as `'points'` and `'assists'`. Utilizing this mixed-type PySpark DataFrame is crucial for effectively demonstrating the divergent behavior of the `.summary()` function based on the specific data type associated with each column.

The provided code snippet below encapsulates the entire setup process. It systematically imports the required PySpark modules, establishes the `SparkSession`, defines the raw data structure and corresponding column names, and ultimately generates and displays the distributed PySpark DataFrame. This meticulous preparation ensures that our testing environment is robust and ready for applying the subsequent statistical methods. Working with this verifiable, small-scale dataset allows us to clearly validate both the accuracy and the structural nuances of the output produced by the PySpark summary functions.

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|conference|points|assists|
+----+----------+------+-------+
|  A|      East|    11|      4|
|  A|      East|     8|      9|
|  A|      East|    10|      3|
|  B|      West|     6|     12|
|  B|      West|     6|      4|
|  C|      East|     5|      2|
+----+----------+------+-------+
```

The resulting DataFrame structure, as visualized above, distinctly features both **string-based categorical variables** (`team`, `conference`) and **integer-based numeric variables** (`points`, `assists`). Recognizing this mixed structure is crucial because the default `.summary()` invocation attempts to compute a full range of standard descriptive metrics for every column presented. While basic measures like the minimum and maximum can be meaningfully retrieved for string fields (based on lexical or alphabetical ordering), statistical metrics demanding numerical computation--specifically the arithmetic mean, the standard deviation, and precise percentile points--will invariably return `null` values for non-numeric columns. This behavior is a fundamental design feature of PySpark, ensuring statistical integrity by not reporting mathematically nonsensical results, and it forms a key aspect of interpreting the standard summary output.

Example 1: Generating a Comprehensive Default Summary

The most straightforward implementation of the `.summary()` function involves calling it with no arguments. This default command instructs PySpark to calculate a fixed, predefined set of comprehensive summary statistics across every column contained within the distributed data structure. This default method is invaluable for initial data profiling, offering a rapid, holistic overview of several critical data dimensions: **data completeness** (measured by count), the **data range** (minimum and maximum values), **central tendency** (mean and median), and overall **data spread** (quantified by the standard deviation). It serves as an excellent starting point for any preliminary data quality check.

Upon execution, the default summary function produces the detailed output shown below. A crucial observation here is the contrast between the valid numerical results reported for the quantitative fields (`points` and `assists`) and the deliberate presence of `null` values for specific metrics within the categorical columns (`team` and `conference`). This is PySpark's intended behavior, as it prevents the reporting of mathematically nonsensical results for non-quantitative features.

#calculate summary statistics for each column in DataFrame
df.summary().show()

```
+-------+----+----------+-----------------+------------------+
|summary|team|conference| points| assists|
+-------+----+----------+-----------------+------------------+
| count| 6| 6| 6| 6|
| mean|null| null|7.666666666666667| 5.666666666666667|
| stddev|null| null|2.422120283277993|3.9327683210007005|
| min| A| East| 5| 2|
| 25%|null| null| 6| 3|
| 50%|null| null| 6| 4|
| 75%|null| null| 10| 9|
| max| C| West| 11| 12|
+-------+----+----------+-----------------+------------------+
```

This output table delivers eight core descriptive statistics. Focusing on the numeric columns, we can extract valuable insights: the average points scored is approximately 7.67, accompanied by a relatively low standard deviation of 2.42. This suggests that the observed point totals are clustered quite closely around the mean. In contrast, the `assists` column displays a higher standard deviation (3.93), which signals greater inherent variability in player assists performance across the dataset. The percentile values, specifically the 25%, 50%, and 75% markers, provide crucial details regarding the internal data distribution. For instance, the median (50%) for points is 6, confirming

that precisely half of the players in this sample scored 6 points or fewer. It is essential to remember that the utility of these statistics for categorical columns is inherently limited; while `min` and `max` simply return the first and last entries based on lexical ordering ('A' and 'C' for `team`), the mean and standard deviation rows are correctly and importantly designated as `null`, underscoring the necessity of statistical rigor when interpreting data types.

count: Represents the total number of non-missing observations present in the column, serving as a check on data completeness.
mean: The arithmetic average, which is a key measure of central tendency but is only calculable for **numeric data types**.
stddev: The standard deviation, a fundamental measure of the dispersion or variability of the data points around the mean.
min: The minimum value observed, determined lexically for string data and numerically for quantitative data.
25%: The first percentile (Q1), marking the value below which 25% of the data falls.
50%: The 50th percentile, commonly known as the **median**, representing the exact center point of the distribution.
75%: The 75th percentile (Q3), marking the value below which 75% of the data falls.
max: The maximum value observed in the column.

Example 2: Targeted Analysis Using Specific Distribution Markers

In many analytical scenarios, the data scientist may only require a subset of metrics, focusing primarily on **distribution boundaries** and measures of central location, such as the quartiles and the overall data range. PySpark's `.summary()` method accommodates this need for precision by enabling explicit customization: instead of relying on the verbose default output, analysts can pass a defined sequence of desired statistical indicators as string arguments. This targeted approach is exceptionally valuable when the goal is to prepare data streams for visualization, or when conducting detailed **outlier detection** analysis based on measures like the Interquartile Range (IQR).

When we explicitly define the metrics `'min'`, `'25%'`, `'50%'`, `'75%'`, and `'max'` within the `.summary()` function call, we compel PySpark to return a much cleaner, highly focused table containing only these specific distribution markers. This practice significantly reduces output clutter and shifts the analytical focus entirely onto how the data is spread across its range. Although this methodology continues to scan all columns in the PySpark DataFrame, it efficiently filters the resulting summary rows to display only the requested metrics, as demonstrated in the output below.

#calculate specific summary statistics for each column in DataFrame

df.summary('min', '25%', '50%', '75%', 'max').show()

```
+-------+----+----------+------+-------+
|summary|team|conference|points|assists|
+-------+----+----------+------+-------+
| min| A| East| 5| 2|
| 25%|null| null| 6| 3|
| 50%|null| null| 6| 4|
| 75%|null| null| 10| 9|
| max| C| West| 11| 12|
+-------+----+----------+------+-------+
```

The resulting table is notably more concise and focused compared to the comprehensive default summary, highlighting the minimum, maximum, and quartile points exclusively for the numeric fields. For a practical example, the `assists` column exhibits a range from a minimum of 2 to a maximum of 12, with the median (50th percentile) situated at 4. Furthermore, we can easily calculate the **Interquartile Range (IQR)**--a robust measure of statistical dispersion--by subtracting Q1 (3) from Q3 (9), yielding an IQR of 6. This figure reliably quantifies the spread of the central 50% of the assist data. Crucially, even though we specifically requested these percentile markers, PySpark diligently preserves data integrity by automatically returning `null` for the quartile rows associated with categorical columns, while still providing the minimum and maximum lexical bounds for essential context.

Example 3: Isolating and Analyzing Only Numeric Features

Adhering to best practices in rigorous statistical analysis necessitates isolating **quantitative variables** to guarantee that every calculated metric is statistically meaningful. Attempting to compute metrics like the mean or standard deviation on string-based columns is not only redundant but also introduces noise and potentially obscures crucial patterns within the data. The most efficient and clean solution within PySpark is to dynamically identify all appropriate numeric columns and apply the `.summary()` function exclusively to that refined subset of the distributed PySpark DataFrame.

This precise targeting is achieved through a **list comprehension** that systematically traverses the DataFrame's schema, which is accessible via `df.dtypes`. The filtering logic ensures that we select only those columns whose associated data types do not begin with the string identifier `'string'`. Once the list of numeric column names (`numeric_cols`) is successfully generated, we leverage the `df.select()` method, combined with the Python splat operator (`*`), to dynamically create a temporary, focused DataFrame. It is upon this streamlined structure that we execute the default `.summary()` call, yielding a pure statistical output.

#identify numeric columns in DataFrame
numeric_cols =

#calculate summary statistics for only the numeric columns
df.select(*numeric_cols).summary().show()

+-------+-----------------+------------------+
|summary| points| assists|
+-------+-----------------+------------------+
count	6	6
mean	7.666666666666667	5.666666666666667
stddev	2.422120283277993	3.9327683210007005
min	5	2
25%	6	3
50%	6	4
75%	10	9
max	11	12
+-------+-----------------+------------------+

The resulting output is the cleanest and most informative for quantitative analysis, displaying summary statistics only for the `points` and `assists` columns. Crucially, this table delivers all necessary measures--including count, mean, standard deviation, minimum, maximum, and the key quartiles--without the intrusion of any `null` values derived from categorical inputs. This approach is highly recommended when generating reports or feeding descriptive statistics into downstream **machine learning pipelines**, as it guarantees that all metrics are mathematically sound and relevant to the data features being studied. Analysts can confidently use these figures to assess normality, identify skewness, and detect potential outliers based on spread and distribution.

Conclusion and Recommendations for Further Study

The PySpark `.summary()` function proves itself to be an exceptionally versatile and indispensable utility for conducting rapid and scalable data profiling within a distributed computing architecture. We have demonstrated that PySpark provides the requisite flexibility to handle varied analytical demands: whether the requirement is a swift, all-inclusive data quality verification (Method 1), a focused investigation into crucial distribution markers such as the percentile ranges (Method 2), or a rigorous, statistically pure analysis dedicated exclusively to quantitative features (Method 3). Mastery of these three core techniques empowers both developers and data scientists to efficiently perform preliminary exploratory data analysis on **large-scale datasets**, thereby establishing a robust and accurate foundation for subsequent complex modeling, data transformations, and reporting activities.

For individuals seeking deeper technical understanding concerning specific function parameters, exploring advanced use cases, or investigating other related descriptive statistics functions available within the framework, we strongly endorse consulting the comprehensive official documentation. **Official resources**, particularly those detailing the PySpark **summary** function, consistently offer the most reliable and authoritative source for information regarding ongoing feature developments, performance optimizations, and best practices for production environments.

Additional PySpark Resources

To further expand your proficiency in PySpark, the following resources and tutorials illustrate how to execute other crucial and common tasks within the distributed environment: