Learning Guide: How to Select Numeric Columns in PySpark DataFrames

Authored by

Mohammed looti

November 11, 2025

In the realm of modern data engineering and statistical analysis, the ability to efficiently process and filter massive datasets is paramount. When utilizing distributed computing frameworks like Apache Spark, specifically through its Python API, PySpark DataFrames serve as the central structure for data manipulation. A frequently encountered and essential preparatory step in this workflow is isolating columns that contain only numerical data, while systematically excluding categorical identifiers or free-form text fields. This precise filtering is critical because downstream processes--such as calculating aggregates, training machine learning models, or performing complex mathematical operations--depend entirely on clean, quantitative inputs.

PySpark manages diverse information using specialized data types, encompassing everything from high-precision floating-point numbers (`double`) to large integers (`bigint`) and various string formats. To effectively identify and select only the non-string-based columns, we leverage Python's foundational capabilities, such as powerful list comprehension, coupled with the detailed schema information provided by the DataFrame object itself. This combination allows for a concise, scalable, and highly performant solution for data preparation in a distributed environment.

Mastering the Technique for Numeric Column Identification

The most dependable and standard methodology for programmatically selecting all numerical columns hinges on rigorous inspection of the DataFrame schema. The schema is accessed via the `df.dtypes` attribute, which yields a comprehensive list of tuples. Each tuple in this list consists of two essential components: the column's name and its assigned PySpark data type string (e.g., `('age', 'int')`). By systematically iterating over this list, we can apply a conditional filter that retains only the columns whose data types correspond to numerical formats.

A particularly robust strategy in PySpark utilizes negative filtering. Instead of attempting to enumerate every possible numerical type (`int`, `long`, `float`, `double`, `decimal`, etc.), which can be cumbersome and prone to error if new types are introduced, we simply exclude the singular primary textual type: `string`. Since PySpark treats virtually all non-string types as inherently numerical (or complex structures built upon numbers), excluding the 'string' prefix effectively isolates all quantitative fields. This method is highly resilient and ensures future compatibility with evolving Spark versions.

The following compact and elegant Python syntax illustrates how to efficiently implement this negative filtering technique, generating a list of all column names ready for subsequent selection:

 # Find all numeric columns in DataFrame using list comprehension
numeric_cols =

# Select only numeric columns in DataFrame using the list
df.select(*numeric_cols).show()

Grasping this fundamental filtering logic is crucial for any efficient PySpark DataFrame manipulation. The subsequent sections of this guide will walk through a detailed, hands-on example, demonstrating the practical application of this technique using a realistic sample dataset.

Establishing the Environment and Sample Data

To effectively demonstrate the process of numeric column selection, we must first initialize the distributed environment and create a representative sample PySpark DataFrame. Our example utilizes basketball player statistics, which typically feature a necessary mix of categorical data (e.g., player `team` and `position`) and quantitative measurements (e.g., `points` scored, `assists`, and `minutes` played). This hybrid structure perfectly simulates the challenges encountered with raw, unstructured input data, making it an ideal candidate for demonstrating data cleaning and filtering.

The preparatory phase begins with the initialization of the SparkSession. This object acts as the essential entry point for all interactions with the core Spark functionality, especially when utilizing the powerful DataFrame API in Python. Once the session is properly established, we define the raw data list and the corresponding column names. Finally, the `spark.createDataFrame()` method is invoked to ingest this data and construct the distributed, immutable DataFrame object. This rigorous setup ensures that the necessary foundation is laid for subsequent schema inspection and column filtering operations.

The following code block meticulously details the steps required for initializing the Spark environment and creating our practical, mixed-type example DataFrame:

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

# Define sample data for basketball players
data = ,
,
,
,
,
,
,
]

Define column names for the dataset
columns =

Create the DataFrame using the defined data and schema

```
df = spark.createDataFrame(data, columns)

# Display the initial DataFrame content
df.show()

+----+--------+------+-------+-------+
|team|position|points|assists|minutes|
+----+--------+------+-------+-------+
| A| Guard| 11| 4| 22.4|
| A| Guard| 8| 5| 34.1|
| A| Forward| 22| 6| 35.1|
| A| Forward| 22| 7| 18.7|
| B| Guard| 14| 12| 20.2|
| B| Guard| 14| 8| 15.6|
| B| Forward| 13| 9| 20.9|
| B| Center| 7| 9| 4.8|
+----+--------+------+-------+-------+
```

Validating the Schema: The Crucial Inspection Step

Prior to implementing any data types-based filtering, it is absolutely essential to confirm how PySpark has interpreted the underlying structure of the data. Even though Spark is sophisticated in its type inference, misinterpretation can occur, particularly if data contains mixed values or is loaded from generic sources. Knowing the precise schema--the inferred or explicitly defined types--is the only way to guarantee the filtering operation will execute without error and produce accurate results.

We achieve this confirmation by executing the simple yet powerful `df.dtypes` command. As previously mentioned, this command returns the list of column name and type tuples. In the context of our basketball statistics example, this inspection will explicitly delineate which columns Spark recognizes as textual (`string`) and which are recognized as various numerical forms (like `bigint` for whole numbers or `double` for decimals). This output is the definitive source of truth for our filtering targets.

Analyzing the output below clearly shows that the categorical fields (`team` and `position`) are correctly identified as strings, while the performance metrics (`points`, `assists`, and `minutes`) are stored using appropriate numerical types (`bigint` and `double`). This confirmation validates our understanding of the dataset and prepares us for the final selection logic:

 # Display data type of each column using df.dtypes

df.dtypes

Executing the Selection Logic with List Comprehension

With the schema verified and the specific column types confirmed, we proceed to implement the core filtering logic using list comprehension. This Pythonic idiom provides a concise and highly efficient way to construct the final list of desired columns. The process involves iterating through every tuple in `df.dtypes`, which are temporarily unpacked into column name (`c`) and type (`t`) variables. The crucial conditional statement, `if t.startswith('string') == False`, ensures that only column names associated with non-string data types are captured and stored in the `numeric_cols` list.

Generating this intermediate list of column names is a vital step, offering a clean verification point. By printing `numeric_cols`, we can immediately confirm that the filtering logic successfully isolated `points`, `assists`, and `minutes` before attempting the final DataFrame transformation. This verification minimizes the risk of executing resource-intensive operations on an incorrectly filtered dataset.

The following code block executes this filtering and demonstrates the resulting list of isolated numeric column names:

 # Find all numeric columns in DataFrame using the negative filter
numeric_cols =

# View list of numeric columns for confirmation
print(numeric_cols)

The final action involves using the `df.select()` transformation. We employ the Python splat operator (`*`) before `numeric_cols` to unpack the list of column names, passing each name as an individual argument to the `select` function. This produces the final, desired DataFrame, containing only the quantitative fields necessary for analytical tasks.

 # Select only numeric columns in DataFrame and display results
df.select(*numeric_cols).show()

+------+-------+-------+
|points|assists|minutes|
+------+-------+-------+
| 11| 4| 22.4|

8	5	34.1
22	6	35.1
22	7	18.7
14	12	20.2
14	8	15.6
13	9	20.9
7	9	4.8
+------+-------+-------+

Synthesizing the Logic and Results

The resulting DataFrame, displayed above, unequivocally confirms the success of our filtering operation. It contains exclusively the `points`, `assists`, and `minutes` columns, exactly matching our requirements. The true strength of this approach lies not just in this successful execution, but in its inherent adaptability and dynamism. This script will seamlessly handle DataFrames with hundreds of columns, always ensuring that only those columns recognized by Spark as numerical are retained, making it an indispensable tool for large-scale, automated data pipelines.

To summarize the key technical takeaway, we deliberately utilized a **negative filter** based on the `startswith('string')` condition. This avoids the fragility of listing specific numerical data types. By excluding any column whose data type description begins with 'string', we automatically and robustly capture all other quantitative data types, including `int`, `float`, `decimal`, `double`, and `bigint`. This methodology represents the most straightforward and reliable path to isolating numeric fields within the complex distributed environment of PySpark.

For any data scientist or engineer engaging with distributed processing, mastering this technique is a foundational skill. It guarantees that initial data cleaning and preparation stages are executed with maximum efficiency, directing computational resources only toward the data required for quantitative tasks, thereby optimizing overall workflow performance and accuracy.

Further Resources for PySpark Data Manipulation

Successfully selecting numeric columns is often the first step in a much larger data transformation process. Once the quantitative data is isolated, subsequent tasks might involve aggregations, feature scaling, or merging with other datasets. The PySpark DataFrame environment offers a rich set of tools for these advanced operations.

To continue building expertise in distributed data processing, consider exploring the following essential topics:

Understanding advanced aggregation techniques and complex analytic functions, such as utilizing PySpark's powerful window functions for row-based calculations.

The necessity of feature engineering, including converting categorical string columns into a numerical format using specialized encoders like `StringIndexer` or `OneHotEncoder` from the MLlib library.

Implementing custom User-Defined Functions (UDFs) to apply complex, non-standard logic and row-wise transformations not inherently supported by standard DataFrame APIs.

Strategies for optimizing performance when managing exceptionally large datasets, focusing on crucial factors like managing partitioning, caching intermediate results, and minimizing shuffle operations.