A Guide to Splitting Data for Machine Learning Models Using PySpark

Authored by

Mohammed looti

November 11, 2025

The Importance of Data Splitting in Machine Learning

When developing and rigorously evaluating sophisticated **machine learning models**, a crucial preliminary step involves preparing the dataset. It is almost universally necessary to first partition the complete dataset into distinct subsets: typically a training set and a test set. This procedure is fundamental to ensuring that the model we build is not merely memorizing the data it sees, a phenomenon known as **overfitting**, but is truly learning generalizable patterns and relationships that can be applied effectively to new, unseen data. Without proper segregation of data, any performance metric derived from the model will be inherently biased and overly optimistic about real-world applicability.

The designated training set is utilized exclusively to teach the model its parameters. This allows the underlying algorithm to optimize its performance based on the input features and known outcomes through iterative processes. Conversely, the separate test set is reserved strictly for the final, unbiased evaluation of the model's performance. By keeping the test data completely separate during the training phase, we obtain an objective measure of the model's predictive capability and its ability to generalize to real-world scenarios. This rigorous separation is the cornerstone of robust model validation across all computational environments, especially when dealing with high-dimensional data.

In the context of **big data** processing, specifically within the **PySpark** environment, handling data splitting requires specialized tools designed for distributed computation. Traditional single-machine methods are entirely insufficient when dealing with petabytes of data spread across a cluster. **PySpark** provides optimized functions that ensure the split is performed efficiently and randomly across all partitions of the distributed **DataFrame**, preserving the integrity and statistical properties of the original dataset while adhering to the principles of distributed computing. Understanding how to leverage these native functions is critical for any data scientist working with **Apache Spark**.

Introducing randomSplit: PySpark's Core Splitting Function

The most straightforward and preferred method for reliably splitting a dataset into training and testing subsets within **PySpark** is through the utilization of the **randomSplit** function. This powerful function operates directly on the **DataFrame** object and is specifically engineered to perform a truly randomized split across the entire distributed data structure. Crucially, it returns a list of new **DataFrames**, each corresponding precisely to the specified proportions defined by the user. Unlike localized sampling methods common in non-distributed frameworks, **randomSplit** guarantees that the randomization is applied uniformly across the entire cluster, maintaining data consistency and statistical representativeness across all partitions.

The basic syntax for employing **randomSplit** is highly concise and intuitive, requiring only two

essential parameters: the list of weights (proportions) and an optional integer value for the random seed, which is vital for reproducibility. For a standard two-way split, such as separating the data into 70% for training and 30% for testing, the implementation is executed in a single line of Python code, immediately yielding the two desired data structures ready for subsequent model development or exploratory analysis. This elegance in execution is a hallmark of the **PySpark** API design, minimizing boilerplate code when dealing with massive datasets.

Below is the canonical example demonstrating how to invoke the **randomSplit** function, where an existing **DataFrame**, denoted as `df`, is decomposed into `train_df` and `test_df`. Notice the direct unpacking of the returned list of DataFrames into our target variables, simplifying the assignment process significantly and adhering to clean Pythonic practices.

train_df, test_df = df.randomSplit(weights=, seed=100)

It is important to remember that this powerful operation is executed lazily, consistent with Spark's underlying architecture. This means the split is only fully materialized and computations are performed when an action (like `count()` or `show()`) is called on the resulting DataFrames. This efficiency is paramount when working with truly massive datasets, as computational resources are only utilized when absolutely necessary, thereby preventing unnecessary data movement and processing overhead across the distributed cluster nodes.

Configuring the Split: Weights and Ensuring Reproducibility

The critical parameter governing the proportional distribution of data is the **weights** argument. This argument requires a list or tuple of non-negative floating-point numbers that explicitly specify the proportionate size of each resulting **DataFrame** relative to the original source. For instance, the list indicates that 70% of the observations from the source DataFrame will be allocated to the first resulting DataFrame (`train_df`), and 30% will be allocated to the second (`test_df`). A key technical detail is that while these weights do not strictly need to sum exactly to 1.0, **randomSplit** automatically normalizes them so that their sum effectively represents 100% of the input data being distributed among the resulting subsets. This normalization simplifies usage, allowing for ratios like or to be used interchangeably with .

While the split process is inherently randomized across the distributed partitions, ensuring that this randomization is repeatable across different execution sessions is vital for scientific rigor, collaborative development, and production stability. This is precisely where the **seed** argument comes into play. The **seed** is an integer value that initializes the random number generator used by the function. By setting a fixed **seed** (such as `seed=100` in our examples), we guarantee that every time the code is executed, the resulting partition of rows will be exactly the same, irrespective of the cluster configuration or the timing of the run.

Without a defined **seed**, subsequent runs would produce different training and testing subsets, making it impossible to reliably compare model performance results or debug issues related to specific data partitions that might have caused model anomalies. The combination of defined **weights** and a fixed **seed** allows data scientists to maintain complete control over the experimental environment. Using a 70% training and 30% testing split is a common industry practice, as it strikes an effective balance between providing enough data for the **machine learning model** to learn complex patterns and reserving a sufficiently large portion for reliable, statistically significant evaluation.

Practical Demonstration: Setting Up the PySpark Environment

To fully illustrate the practical application of **randomSplit**, we must first establish a representative **DataFrame** within the **PySpark** environment. Our example dataset will track simulated student performance metrics at a university, including the number of hours spent studying, the count of preparatory exams taken, and the final exam score. This structured data is ideal for demonstrating predictive modeling techniques, such as multivariate **linear regression**, where we predict the score based on the other two variables.

Before any data manipulation can occur, we must initialize the Spark context by creating a **SparkSession**. This object serves as the essential entry point for utilizing all Spark functionality and is necessary for working with distributed DataFrames. The following code snippet demonstrates the standard initialization process, followed by the definition of our raw data and column headers. We then instantiate the PySpark DataFrame using `spark.createDataFrame()`, transforming the raw Python list of lists into a structured, distributed data object ready for processing.

The data structure defined here represents twenty individual observations. After creation, we inspect the first five rows using `df.limit(5).show()`, confirming the successful structure and population of our source **DataFrame** before we attempt any partitioning. This preliminary validation step ensures that the data is correctly loaded and schema inference has been successful prior to the resource-intensive splitting operation.

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 first five rows of dataframe
df.limit(5).show()

+-----+----------+-----+
|hours|prep_exams|score|
+-----+----------+-----+
1	1	76
2	3	78
2	3	85
4	5	88
2	2	72
+-----+----------+-----+

Executing the Data Split and Verifying Proportions

The immediate objective is to prepare this newly created data for fitting a predictive model, where the variables **hours** and **prep_exams** serve as predictors, aiming to forecast the student's **score**. Following established data science best practices, we must randomly partition the dataset,

dedicating 70% of the total rows for the training phase and reserving the remaining 30% for unbiased testing and validation. This specific ratio, 70/30, provides the model with sufficient historical data to learn complex patterns while ensuring a meaningful sample size remains for accurate performance assessment, mitigating the risk of **overfitting**.

To achieve this precise 70/30 split, we execute the **randomSplit** function on our base **DataFrame**, `df`. We explicitly provide the proportional weights and, critically, ensure the `seed` parameter is set to a fixed integer, such as `100`. Setting the **seed** guarantees that the exact rows assigned to `train_df` and `test_df` will remain consistent across executions, which is paramount for debugging, model iteration, and maintaining the integrity of the **machine learning pipeline**. The resulting DataFrames are assigned immediately, ready for the next stages.

#split dataset into training and test sets
train_df, test_df = df.randomSplit(weights=, seed=100)

After the split is performed, the next essential step is to verify the resulting row counts to confirm that the desired ratios were accurately applied relative to the original data volume. Since our original DataFrame contained 20 rows, a 70/30 split should ideally result in 14 rows for the training set and 6 rows for the test set. We use the **count()** action, which triggers the execution of the split across the cluster, and then print the totals for verification. The output confirms that the proportional split was successful, with exactly 14 rows allocated to the `train_df` (70%) and exactly 6 rows allocated to the `test_df` (30%), demonstrating the precision of **randomSplit** even on a small dataset.

#view count of rows in train_df
print(train_df.count())

14

#view count of rows in test_df
print(test_df.count())

6

Final Validation and Transition to Model Training

A final, important validation step involves visually inspecting a sample of the resulting DataFrames to ensure that the data structures and content remain intact, and that the randomization was effective. By using the `limit(5).show()` method on both `train_df` and `test_df`, we can confirm that both subsets contain the correct columns (hours, prep_exams, score) and that the rows

displayed are indeed distinct. This process confirms the effective randomization achieved by the **randomSplit** function, verifying that the data preparation phase is complete and accurate.

#view first five rows of training set
train_df.limit(5).show()

```
+-----+----------+-----+
|hours|prep_exams|score|
+-----+----------+-----+
|    1|         1|   76|
|    2|         3|   78|
|    2|         3|   85|
|    4|         5|   88|
|    1|         2|   69|
+-----+----------+-----+
```

#view first five rows of test set
test_df.limit(5).show()

```
+-----+----------+-----+
|hours|prep_exams|score|
+-----+----------+-----+
|    2|         2|   72|
|    2|         0|   88|
|    4|         1|   94|
|    3|         4|   82|
|    4|         4|   85|
+-----+----------+-----+
```

With the original dataset successfully partitioned into a **training set** and a **test set**, we can confidently transition to the core task of model development. The `train_df` is used exclusively for training the selected statistical or **machine learning model** (e.g., fitting a **linear regression** or decision tree), allowing the algorithm to learn the underlying patterns that connect the predictor variables to the response variable. This segregation ensures the integrity of the subsequent evaluation stage.

Upon successful training, the resulting model can then be applied to the reserved `test_df`. The performance metrics derived from this evaluation (such as R-squared, Mean Squared Error, or accuracy) provide an honest, unbiased assessment of how well the model is expected to perform in real-world deployment on data it has never encountered before. This systematic and

reproducible approach, facilitated efficiently by **PySpark**'s distributed capabilities, forms the backbone of reliable predictive modeling in **big data** environments.

Beyond Splitting: Next Steps in the PySpark Workflow

Mastering the art of data preparation in **PySpark** involves proficiency in several fundamental operations that extend beyond simple randomized splitting. Effective data science workflows often require complex transformations, filtering, aggregation, and joining of large distributed **DataFrames** before model training can commence. To continue building expertise in this powerful framework, exploring tutorials on these related data wrangling tasks is highly recommended, as they form the complete data preparation lifecycle.

Common tasks that frequently follow the data splitting stage include crucial steps like feature engineering, handling missing values through imputation, scaling numerical features to prevent bias, and converting categorical variables into numerical formats suitable for algorithm input (such as one-hot encoding). Tools and classes within the `pyspark.ml` library are essential for performing these subsequent, scalable operations, providing robust, distributed solutions for preparing data specifically for advanced **machine learning models** within the Spark ecosystem.

The following list outlines other critical, distributed operations that data professionals commonly execute using PySpark, enabling comprehensive data management and preparation prior to the modeling phase:

Filtering rows based on complex Boolean conditions across large partitions.
Joining multiple **DataFrames** efficiently using various join types (e.g., inner, outer) optimized for distributed computation.
Aggregating data using specialized functions like `groupBy` and `agg` to summarize statistics across the cluster.
Adding new columns or modifying existing ones using the expressive `withColumn` operation.
Handling null or missing values using robust imputation techniques provided by the ML library components.