Pyspark for Large Datasets

By : Anand

Leveraging PySpark for NYC Taxi Fare Predictions

Context :

- The NYC Taxi dataset is vast and complex, representing millions of trips.
- Includes key variables: trip distances, durations, fares, and timestamps.
- Analyzing this data can help optimize operations and improve fare predictions.

Challenge:

- **Scale:** How to process millions of records efficiently?
- **Insights:** How to extract meaningful insights and make accurate predictions?

Solution:

- Utilize **PySpark**, a distributed computing framework, designed for large-scale data processing.
- Leverage PySpark's **DataFrame APIs**, **SQL**, and **MLlib** for seamless data handling and predictive modeling.

Dataset & Approach:

- **Dataset Features**: Trip distance, trip duration, fare amount.
- **Approach**:
  - Explore and clean data.
  - Demonstrate PySpark features (lazy evaluation, fault tolerance, SQL).
  - Build and evaluate a **Gradient Boosting Model** to predict trip fares.

Features of Pyspark

- Built on RDDs(Resilient Distributed Dataset)
- Supports Lazy Evaluation and Fault Tolerance
- Offers SQL Integration
- Machine Learning Pipelines (MLlib)

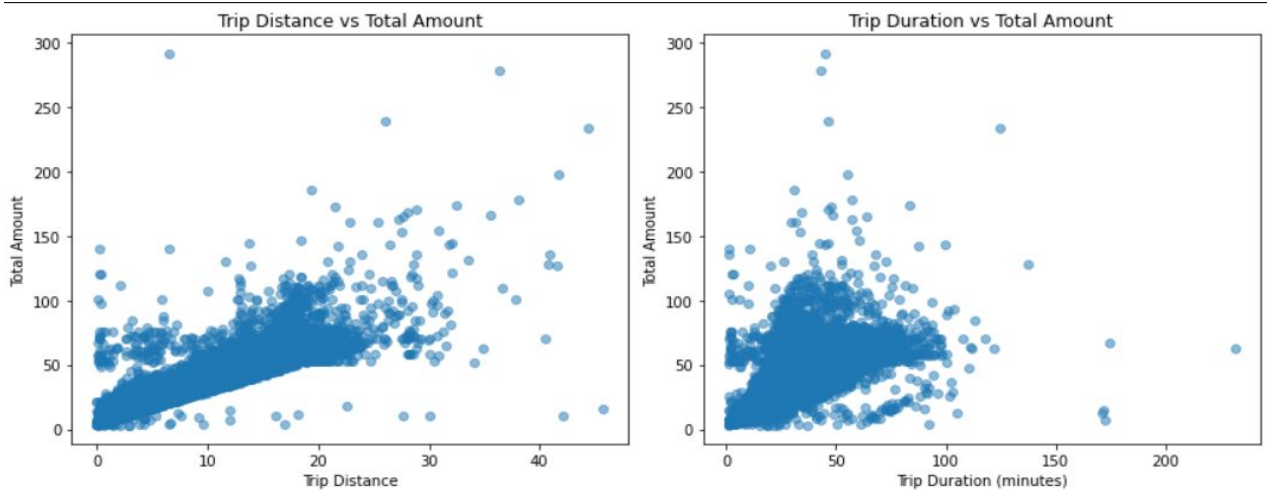
  Gradient Boosted Trees(GBT)

Relationship Between Trips Features and Total Fare

Scatter plot : Trip Distance and Trip Duration vs Total Fare

Correlation Result: Showed strong relationships.

```
Correlation between trip_distance and total_amount: 0.9466871476932208
Correlation between trip_duration_minutes and total_amount: 0.8382511516121054
```



Lazy Evaluation

Lazy Evaluation for Efficient Processing

- Lazy evaluation means that transformations in PySpark, like filtering or adding columns, are not executed immediately. Instead, Spark builds a logical execution plan and waits until an action (e.g., `.show()` or `.count()`) is called to trigger execution. This optimization minimizes unnecessary computations and enhances performance.

```
lazy_filtered_data = df.filter(
    (col("trip_distance") > 10) & (col("total_amount") > 0)
).withColumn(
    "fare_category",
    when(col("total_amount") < 50, "Low Fare")
    .when(col("total_amount") < 100, "Medium Fare")
    .otherwise("High Fare")
)
```

Fault Tolerance:

Fault tolerance ensures that data processing jobs continue running even if a hardware failure or node crash occurs. In distributed computing, where data is split across multiple partitions, failures in one part of the system shouldn't require recomputing the entire dataset. PySpark achieves this by recomputing only the failed partition using its RDD lineage..All Data Frame transformations (filtering, grouping, etc.) inherit this feature, as DataFrames are ultimately converted to RDD operations

Lazy Evaluation vs SQL Integration

SQL Integration: PySpark supports SQL queries for seamless data manipulation.

Example:

- DataFrame API: Filter rows and group by fare categories.
- SQL Query: Same logic but written in SQL syntax.

PySpark's flexibility by performing the same data transformation using both Lazy Evaluation and SQL queries. We can see how PySpark offers multiple approaches to achieve the same result—one emphasizing programmatic transformations (Lazy Evaluation) and the other leveraging declarative syntax (SQL

12:05 PM (50s)

```
df.createOrReplaceTempView("taxi_data")
sql_query = """
SELECT
    fare_category,
    AVG(trip_duration_minutes) AS avg_duration,
    AVG(total_amount) AS avg_fare
FROM (
    SELECT
        trip_distance,
        total_amount,
        trip_duration_minutes,
        CASE
            WHEN total_amount < 50 THEN 'Low Fare'
            WHEN total_amount < 100 THEN 'Medium Fare'
            ELSE 'High Fare'
        END AS fare_category
    FROM taxi_data
    WHERE trip_distance > 10 AND total_amount > 0
) GROUP BY fare_category
"""
sql_result = spark.sql(sql_query)
sql_result.show()
```

12:05 PM (<1s)

```
lazy_filtered_data = df.filter(
    (col("trip_distance") > 10) & (col("total_amount") > 0)
).withColumn(
    "fare_category",
    when(col("total_amount") < 50, "Low Fare")
    .when(col("total_amount") < 100, "Medium Fare")
    .otherwise("High Fare")
)
result = lazy_filtered_data.groupBy("fare_category").agg(
    avg("trip_duration_minutes").alias("avg_duration"),
    avg("total_amount").alias("avg_fare")
)
```

```
+-------------+------------------+------------------+
|fare_category|      avg_duration|          avg_fare|
+-------------+------------------+------------------+
|     Low Fare|30.512717546939705|42.384234546623006|
|  Medium Fare| 44.33087667006116| 63.61749977082756|
|    High Fare| 49.12821303962922|125.18851507576966|
+-------------+------------------+------------------+
```

Gradient Boosted Trees (GBT)

Gradient Boosted Trees is a powerful machine learning algorithm that builds an ensemble of decision trees in a sequential manner.

Each tree corrects the errors of the previous tree, making it well-suited for regression tasks like predicting total_amount in our dataset.

The GBTRegressor predicts the total_amount using the combined feature vector.

MLlib: Gradient Boosting and Pipelines

Feature Engineering: Used `VectorAssembler` to combine trip features.

Model: Gradient Boosting Tree (GBT) Regressor to predict fares.

Pipeline:

- Integrated feature transformation and model training.
- Simplifies workflow and ensures reproducibility.

DAG Visualization in PySpark

DAG (Directed Acyclic Graph):

Tracks stages and tasks for distributed execution.

Shows logical execution plans derived from transformations

Project Examples :

Filtering and repartitioning steps visualized in DAG.

GBT Model Training as another DAG.

Conclusion and Key Takeaways

PySpark Revolutionizes Big Data with its scalable and distributed computing framework, PySpark makes handling large datasets seamless and efficient.

Core Strengths of PySpark:

- **Fault Tolerance**: Ensures reliability even in the event of failures by recomputing failed partitions.
- **Lazy Evaluation**: Optimizes performance by delaying computations until actions are triggered.

Versatile APIs:

- Combines DataFrame, SQL, and MLlib capabilities to simplify complex workflows.
- Supports transformations, aggregations, and advanced machine learning tasks

In the NYC Taxi Fare Prediction Project:

- We implemented PySpark's features end-to-end.
- Demonstrated distributed data cleaning, SQL integration, and machine learning pipelines for Gradient Boosted Trees.
- Achieved strong predictive accuracy with an R^2 of 90.2% and RMSE of 4.05.

Final Takeaway: PySpark’s distributed architecture and robust API ecosystem empower efficient big data processing, making it ideal for modern machine learning workflows.