भारतीय प्रौद्योगिकी संस्थान जोधपुर
Indian Institute of Technology Jodhpur

Software And Data Engineering(CSL7090)

Code Documentation

for

IPL Data Analysis using Apache Spark

by

Mehul Sharma (M24CSE013)

Shivender Thapa (M24CSE023)

Shreyank Jaiswal(M24CSE024)

Contents

Dataset Overview

Five main datasets are utilized to accomplish a comprehensive overview of the data for the IPL season. Each dataset provides specific information that enables different types of analysis, leading to insights regarding the performance of teams as well as player performances, match outcomes, and the nature of the venues themselves.

1.1 match.csv

1.1.1 Description

The dataset contains information at the match level for teams, date, season, venue, decision taken on toss, and the result of the match. It further marks "Man of the Match" and also captures margin details if the match was a win scenario.

1.1.2 Analytics Enabled

The data set supports seasonal and venue wise analysis in respect of match outcome. This will help us to investigate match-relevant decisions, analyze different teams' performances at different times and contrast margin-based win statistics across matches and venues as well.

1.2 player.csv

1.2.1 Description

This file contains the player-specific information which has the player's name, birthdate, batting and bowling abilities along with the country.

1.2.2 Analyses Enabled

This data set will allow player profiling, experience-based analysis as well as international player trends. In addition, the data is used to perform performance trend analysis along the dimensions of player age and skill set, thereby making the comparison of players even more specific.

1.3 team.csv

1.3.1 Description

The data set provides a basic mapping of team IDs to team names.

1.3.2 Analyses Enabled

This data for team.csv is largely utilized to make join easy with other datasets as well as support the statistics of teams, such as win-loss ratios, home vs. away performances and scoring patterns of teams.

1.4 ball_by_ball.csv

1.4.1 Description

The data captures ball-by-ball records of each match, runs scored and extras, wicket types and information about the bowler with respect to each delivery.

1.4.2 Analyses Enabled

With such detail in the data set, every nuance in batting and bowling can be analyzed, strike rates gauged, wicket-taking patterns assessed, and scoring trends interpreted. Economy rates can be calculated, and performance in powerplay vs. non-powerplay overs can be compared.

1.5 player_match.csv

1.5.1 Description

This file combines the player and match data but with the additional information regarding how each player played, whether he was captain or keeper or "Man of the Match." This includes information about opposing teams as well with player-specific performance metrics.

1.5.2 Analyses Enabled

Multiple analyses are possible from the dataset, such as how well a player performs in games and across different teams, whether or not captaincy impacts performances, role-based contributions of players, and indeed consistent players. For instance, it is possible to compare the achievements and contributions of various players to team outcomes.

Each of the datasets plays a unique role in the IPL analysis, providing the opportunity for a multi-faceted examination of performance, trends, and outcomes across IPL seasons. Together, they complement a wide analysis perspective that is well-supported by Apache Spark capabilities in terms of handling structured and relational data in extracting meaningful insights from complex data sets.

Data Preprocessing

In this chapter, we address the most critical preprocessing operations one performs to prepare data for analysis. The operations are geared toward consistency, completeness, and smooth handling of large datasets. They comprise schema definition, null value handling, and feature engineering. Each step in preprocessing enhances the quality of the data, hence more efficient for analytical and modeling purposes on Apache Spark.

2.1 Schema Definition and Data Loading

One of the first steps in working with big data sets is to define a schema explicitly to make handling the data efficient. Schema was created by using StructType and StructField classes available in PySpark. This will enable us to specify a data type as well as nullability for each column. It has the added advantage of type inference at runtime with a minuscule runtime overhead on Spark because it infers types dynamically, especially on large datasets containing varied data types.

2.1.1 Implementation Details

IPL ball-by-ball data for the matches was loaded using a schema that captures all the important match events. Columns in this schema include identifiers like match_id and over_id, as well as attributes such as team_batting, runs_scored, and extra_type.

Here's an example of how the schema was defined and data loaded:

```
from pyspark.sql.types import StructField, StructType,
   IntegerType, StringType,
BooleanType, DateType

ball_by_ball_schema = StructType([
    StructField("match_id", IntegerType(), True),
    StructField("over_id", IntegerType(), True),
    StructField("ball_id", IntegerType(), True),
    StructField("innings_no", IntegerType(), True),
    StructField("team_batting", StringType(), True),
    StructField("team_bowling", StringType(), True),
    StructField("runs_scored", IntegerType(), True),
    StructField("extra_type", StringType(), True),
    StructField("wides", IntegerType(), True),
    StructField("byes", IntegerType(), True),
    StructField("noballs", IntegerType(), True),
    StructField("match_date", DateType(), True),
    StructField("season", IntegerType(), True)
])

ball_by_ball_df = spark.read.schema(ball_by_ball_schema)\
    .format("csv")\
    .option("header", "true")\
    .load("Dataset/Ball_By_Ball.csv")
```

By predefining data types, we avoided errors associated with incorrect data interpretation and allowed Spark to apply distributed computations more effectively. This schema was reused across multiple datasets (e.g., match-level data, player-level data), each with its unique attributes. As a result, data loading was streamlined across various files with Spark automatically interpreting the values according to the schema specified, ensuring data integrity and compatibility for analysis.

2.2 Handling Null Values

Handling missing values is critical in preparing data for accurate analysis. In this project, missing values were addressed by either filling them with placeholders or removing rows that could not contribute to meaningful analysis.

2.2.1 Filling and Dropping Null Values

For example, the match_df DataFrame had several columns with missing values. The Win_Margin and Outcome_Type fields, both essential for understanding match outcomes, were populated with default values—0 for Win_Margin and 'Unknown' for Outcome_Type. These substitutions ensured the completeness of data without introducing biases due to null values.

Here's how it was implemented:

```
from pyspark.sql.functions import count, when, isnull

# Fill nulls for specific fields
match_df = match_df.fillna({'Win_Margin': 0, 'Outcome_Type':
    'Unknown'})

# Verify null replacements by checking remaining null counts
match_df.select([count(when(isnull(c), c)).alias(c)
                for c in match_df.columns]).show()
```

Using this approach, we ensured that records with minor missing values remained usable for analysis rather than being discarded. This strategy is particularly useful in sports datasets where some values, such as Outcome_Type, may not be recorded due to certain game conditions (e.g., match abandonment).

2.3 Date Parsing and Feature Engineering

Dates are often a crucial component of time-based data analysis, and transforming these values into a standard format enables efficient querying and aggregation. In this project, the match_date column in match_df was parsed to a consistent date format using the to_date function in PySpark. This process standardized date entries, allowing us to extract seasonal insights from the data.

2.3.1 Date Parsing and Standardization

Some entries in match_date contained null values or inconsistent formats, so we used a fallback date ('1900-01-01') for null entries. This placeholder date enabled us to retain all records in the DataFrame while ensuring date operations would not fail due to null entries:

```
from pyspark.sql.functions import to_date

# Standardize date format and handle nulls with a placeholder
    date
match_df = match_df.withColumn("match_date",to_date("
   match_date","M/d/yyyy"))
match_df = match_df.fillna({'match_date': '1900-01-01'})
```

By filling in missing dates with a placeholder, we could preserve record structure, ensuring date-based calculations were applied consistently across the dataset.

2.4 Feature Engineering for Seasonal Analysis

After date parsing, seasonal attributes were engineered to extract patterns across different time periods. For instance, a new column, season, was derived from match_date, allowing for analysis based on IPL seasons. This feature engineering step was critical for examining trends such as changes in player performance, team wins, and match statistics over the years. Extracting these temporal features added value by enabling seasonal trend analysis that could inform insights on player or team consistency over time.

2.4.1 Example of Seasonal Feature Extraction

This snippet demonstrates how we could derive the season column from the match_date to enable seasonal analyses:

```
from pyspark.sql.functions import year

# Extract season from match_date
match_df = match_df.withColumn("season", year("match_date"))
```

By creating the season column, we facilitated aggregations and analyses specific to each IPL season. This approach enabled the exploration of temporal patterns, including average scores per season or win trends by team, supporting a deeper understanding of performance shifts across seasons.

Key Analysis Components

This chapter explores various analyses conducted on the IPL dataset, providing insights into player performance, bowling efficiency, impact of toss decisions, and venue-based scoring trends.

3.1 Top Scoring Batsmen Per Season

To identify the top-scoring batsmen each season, an analysis was performed on runs scored by each player across different IPL seasons. By aggregating runs scored by each batsman and ranking them within each season, we could see which players consistently dominated and track emerging trends over time.

The following SQL query was used to rank batsmen by season:

```
top_scoring_batsmen_per_season = spark.sql("""
WITH ranked_batsmen AS (
    SELECT
        p.player_name,
        m.season_year,
        SUM(b.runs_scored) AS total_runs,
        ROW_NUMBER() OVER (PARTITION BY m.season_year ORDER
           BY
            SUM(b.runs_scored) DESC) as rank
    FROM ball_by_ball b
    JOIN match m ON b.match_id = m.match_id
    JOIN player_match pm ON m.match_id = pm.match_id AND b.
       striker =
        pm.player_id
    JOIN player p ON p.player_id = pm.player_id
    GROUP BY p.player_name, m.season_year
)
SELECT
    player_name,
    season_year,
    total_runs
FROM ranked_batsmen
WHERE rank <= 3
ORDER BY season_year, total_runs DESC
""")
top_scoring_batsmen_per_season.show(30)
```

This query outputs the top three batsmen per season by total runs, showcasing key players like SE Marsh, G Gambhir, and SR Tendulkar who have led in scoring across multiple seasons. This analysis provides a valuable measure of consistency and high performance over time (IPL DATA ANALYSIS SPARK).

3.2 Economical Bowlers in Powerplay

In T20 cricket, especially the powerplay (first 6 overs), restricting runs while taking wickets is crucial. To evaluate bowling performance in the powerplay, we calculated each bowler's average runs conceded per ball and total wickets taken.

Using the following SQL query, we identified the most economical bowlers:

```
economical_bowlers_powerplay = spark.sql("""
SELECT
    p.player_name,
    AVG(b.runs_scored) AS avg_runs_per_ball,
    COUNT(CASE WHEN b.bowler_wicket = 1 THEN 1 END) AS
       total_wickets
FROM ball_by_ball b
JOIN player_match pm ON b.match_id = pm.match_id AND b.bowler
   =pm.player_id
JOIN player p ON pm.player_id = p.player_id
WHERE b.over_id <= 6
GROUP BY p.player_name
HAVING COUNT(*) >= 1
ORDER BY total_wickets DESC, avg_runs_per_ball
""")
economical_bowlers_powerplay.show()
```

This query yielded bowlers like Z Khan and B Kumar, who demonstrated efficiency in run containment with average runs per ball below 1.1. Such analysis can help teams select bowlers who consistently perform well in the opening overs, enhancing their early-game strategy.

3.3 Impact of Toss Decisions

To analyze the impact of toss decisions, we examined match data to observe whether the team that won the toss also won the match, and how batting or fielding decisions influenced outcomes.

The following transformation was applied to categorize toss outcomes:

```
from pyspark.sql.functions import when
match_df = match_df.withColumn(
    "toss_match_winner",
    when(col("toss_winner") == col("match_winner"), "Yes").
       otherwise("No")
)
match_df.show(2)
```

This step identified a pattern where teams winning the toss tend to have a strategic advantage. A visualization of the toss impact on match results can help teams decide on toss strategies based on match conditions or opposition strength.

3.4 Venue-Averages and High Scores

Different venues can have varying pitch conditions, influencing scoring patterns. To understand these variations, we calculated and compared average and highest scores in the first and second innings across venues. This analysis highlights high and low-scoring grounds, providing teams with insights into likely scoring outcomes and helping them tailor strategies for different locations.

The SQL query used for this analysis is:

```
scores_by_venue_innings = spark.sql("""
SELECT
    venue_name,
    CAST(MAX(CASE WHEN innings_no = 1 THEN average_score ELSE
        NULL
        END) AS INT) AS Inng_1_avg,
    CAST(MAX(CASE WHEN innings_no = 2 THEN average_score ELSE
        NULL
        END) AS INT) AS Inng_2_avg,
    CAST(MAX(CASE WHEN innings_no = 1 THEN highest_score ELSE
        NULL
        END) AS INT) AS Inng_1_highest,
    CAST(MAX(CASE WHEN innings_no = 2 THEN highest_score ELSE
        NULL
        END) AS INT) AS Inng_2_highest
FROM (
    SELECT
        match.venue_name,
        ball_by_ball.innings_no,
        AVG(SUM(runs_scored)) OVER (PARTITION BY match.
           venue_name,
            ball_by_ball.innings_no) AS average_score,
        MAX(SUM(runs_scored)) OVER (PARTITION BY match.
           venue_name,
            ball_by_ball.innings_no) AS highest_score
    FROM ball_by_ball
    JOIN match ON ball_by_ball.match_id = match.match_id
    GROUP BY match.venue_name, ball_by_ball.innings_no,
        ball_by_ball.match_id
) GROUP BY venue_name
ORDER BY venue_name
""")
scores_by_venue_innings.show()
```

This query provided averages and maximum scores per innings at each venue. Visualizations of this data show that certain venues, like Eden Gardens and Wankhede Stadium, tend to be high-scoring grounds, influencing team strategies for batting order and target setting.

Results Interpretation

In this chapter, we interpret the results derived from each analysis, examining patterns and drawing conclusions. We then connect these findings with insights from the research paper on real-time processing, graph-based analysis, and machine learning capabilities in Apache Spark.

4.1 Summary of Key Insights

Top Scoring Batsmen Per Season:

The seasonal analysis of top-scoring batsmen highlighted consistency among players like SE Marsh and SR Tendulkar, showing that certain players maintained high performance across multiple seasons. This trend aligns with the focus in real-time processing on observing evolving patterns and highlights how PySpark's efficient processing can handle yearly breakdowns and trend analysis at scale.

Economical Bowlers in Powerplay:

Identifying bowlers with high efficiency during the powerplay revealed that bowlers like Z Khan and B Kumar were key contributors to their teams' success. This real-time insight allows teams to adjust their strategies based on player efficiency. The research paper emphasizes that Spark's real-time processing supports timely and resource-efficient data processing, making such insights feasible in real-time environments.

Impact of Toss Decisions:

The analysis of toss decisions showed that winning the toss can influence match outcomes, especially when teams choose fielding or batting based on pitch and weather conditions. The research underscores that real-time decisions based on such analyses are vital for competitive advantages.

Venue-wise Score Averages and High Scores:

Venue-based analysis highlighted that certain grounds favor high scores, which has strategic implications for team selection and batting orders. Venues like Eden Gardens and Wankhede Stadium often show higher scores, enabling teams to forecast and plan based on expected scoring ranges at these venues.

4.2 Integration with Research Insights

The findings align with several aspects discussed in the research paper:

- **Real-Time Data Processing with Spark Streaming:** Spark's micro-batch processing enables real-time analytics necessary for analyzing live cricket data.
- **Graph-Based Analysis using GraphX:** Venue-based scoring analysis benefits from GraphX's capabilities, allowing comparisons of interconnected performance metrics across multiple seasons and matches.
- **Machine Learning in Apache Spark:** Analyzing scoring trends or toss decisions and extending to predict match outcomes becomes scalable with MLlib, enhancing predictive analytics in sports.

In summary, this chapter not only provides an interpretation of the key analyses conducted but also emphasizes the role of Apache Spark's advanced data-processing frameworks in enabling robust and scalable analysis in real-time contexts.