Spark Assignment – Climate

Global Land and Temperatures by Major City

Appendix: Spark.ipynb

Yayuan Zhang

ISM 6362 Big Data and Cloud-Based Tools

Contents

1. Introduction

1.1 Overall Findings

The project provides insights into different aggregation operations, window functions, pivot tables, and multi-level aggregations on the dataset, showcasing the versatility of Apache Spark for data analysis.

1.2 Tool

This project was completed using the Anaconda Jupyter environment. Since I had previously set up both PyCharm Community Edition and Anaconda Jupyter on my computer during prior classes, I proceeded to work directly in Anaconda Jupyter. I began by installing the necessary packages using 'pip install pyspark'.

1.3 Importing Library

Subsequently, I imported the required libraries:

```
from pyspark.context import SparkContext
from pyspark.sql.session import SparkSession
from pyspark.sql import *
from pyspark.sql.types import *
```

I then executed the following code to initiate the Spark context and session, which are foundational components to execute Spark operations:

```
sc = SparkContext('local')
spark = SparkSession(sc)
```

The below codlings are initializing the Spark environment for subsequent operations. Spark runs locally on my computer and provides a single point of entry for Data Frame and SQL API functionalities, and is used for reading data, transforming it, and executing SQL queries.

2. Data Information

2.1 Source

The dataset comes from Kaggle: https://www.kaggle.com/datasets/berkeleyearth/climate-change-earth-surface-temperature-data.

2.2 Data

The analysis was performed on the GlobalLandTemperaturesByMajorCity.csv dataset, which contains temperature readings from major cities across the world. The dataset includes the following seven variables:

- dt: Date of the temperature reading.
- AverageTemperature: Mean temperature for the specified date.
- AverageTemperatureUncertainty: The margin of error or uncertainty associated with the average temperature reading.
- City: The city where the temperature was recorded.

- Country: The country of the specified city.
- Latitude: Geographical latitude of the city.
- Longitude: Geographical longitude of the city.

According to the following codes, I've determined that the dataset contains 8 columns and 239,177 data points. It's worth noting that spark.sql doesn't have a direct SQL statement to calculate the number of columns. However, I used the len() function from the PySpark DataFrame API combined with the columns property to obtain the result.

```
spark.sql("SELECT COUNT(*) FROM GlobTemp").show()
```

```
column_count = len(spark.table("GlobTemp").columns)
```

2.3 Data preprocessing

2.3.1 Checking data type

Firstly, I called the data frame name to check the data type for each column. The dt is data type, Average Temperature and Average Temperature Uncertainty are both double types, the remaining columns are all string. According to the interpretation of variables, these data types are normal.

Output:

DataFrame[dt: date, AverageTemperature: double, AverageTemperatureUncertainty: double, City: string, Country: string, Latitude: string, Longitude: string]

The following SQL query checks the dt column and the output shows all the dates in the dt column adhere to the expected 'YYYY-MM-DD' format.

```
spark.sql("""
    SELECT dt
    FROM GlobTemp
    WHERE LENGTH(dt) != 10 OR dt NOT LIKE '____-__-__'
        """).show()
```

Output:

```
+---+
| dt|
+---+
+---+
```

2.3.2 Checking missing values

The following SQL query checks the missing values and there is approximately 4.6% missing data in the attribute of AverageTemperature, which is 11002.

```
spark.sql("""
        SELECT COUNT(*) as missing_values_count
        FROM GlobTemp
        WHERE AverageTemperature IS NULL
```

```
""").show()
```

Output:

```
+--------------------+
|missing_values_count|
+--------------------+
|               11002|
+--------------------+
```

Given that the proportion of missing values is not significant, and since I'm not conducting time series analysis, but rather general descriptive statistics or other non-sequential analyses, I've decided that it's feasible to remove these missing values. I used the following code to filter out missing values and replace the original view:

```
cleaned_df = spark.sql("""
    SELECT * FROM GlobTemp
    WHERE AverageTemperature IS NOT NULL
""")
cleaned_df.createOrReplaceTempView("GlobTemp")
```

After work, I check the missing values again and result shows it work:

Output:

```
+--------------------+
|missing_values_count|
+--------------------+
|                   0|
+--------------------+
```

I continuously check other attributes and they do not have any missing values.

For dt:

```
spark.sql("""
    SELECT COUNT(*) as missing_values_count
    FROM GlobTemp
    WHERE dt IS NULL
""").show()
```

```
+--------------------+
|missing_values_count|
+--------------------+
|                   0|
+--------------------+
```

For AverageTemperature:

```
spark.sql("""
    SELECT COUNT(*) as missing_values_count
    FROM GlobTemp
    WHERE AverageTemperature IS NULL
""").show()
```

```
+--------------------+
|missing_values_count|
+--------------------+
|                   0|
+--------------------+
```

Spark Assignment

For AverageTemperatureUncertainty:

```
spark.sql("""
    SELECT COUNT(*) as missing_values_count
    FROM GlobTemp
    WHERE AverageTemperatureUncertainty IS NULL
""").show()
```

```
+--------------------+
|missing_values_count|
+--------------------+
|                   0|
+--------------------+
```

For City:

```
spark.sql("""
    SELECT COUNT(*) as missing_values_count
    FROM GlobTemp
    WHERE City IS NULL
""").show()
```

```
+--------------------+
|missing_values_count|
+--------------------+
|                   0|
+--------------------+
```

For Country:

Spark Assignment

```
spark.sql("""
    SELECT COUNT(*) as missing_values_count
    FROM GlobTemp
    WHERE Country IS NULL
""").show()
```

```
+--------------------+
|missing_values_count|
+--------------------+
|                   0|
+--------------------+
```

For Latitude:

```
spark.sql("""
    SELECT COUNT(*) as missing_values_count
    FROM GlobTemp
    WHERE Latitude IS NULL
""").show()
```

```
+--------------------+
|missing_values_count|
+--------------------+
|                   0|
+--------------------+
```

For Longitude:

```
spark.sql("""
    SELECT COUNT(*) as missing_values_count
    FROM GlobTemp
    WHERE Longitude IS NULL
""").show()
```

```
+--------------------+
|missing_values_count|
+--------------------+
|                   0|
+--------------------+
```

2.3.3 Checking outlier

The following SQL query checks the outlier values in the attribute of Average Temperature by using IQR methods, firstly I calculated the values of Q1 and Q3.

```
spark.sql("""
        SELECT
            percentile_approx(AverageTemperature, 0.25) as Q1,
            percentile_approx(AverageTemperature, 0.75) as Q3
        FROM GlobTemp
        """).show()
```

Output:

```
+-----+------------------+
|   Q1|                Q3|
+-----+------------------+
|12.71|25.918000000000003|
+-----+------------------+
```

Based on the IQR calculation being Q3 minus Q1, I get an IQR value of 13.208 (= 25.918 - 12.71). Therefore, I learn the lower bound (LB) and upper bound (UB) by following function:

- IQR = 13.208
- Lower Bound (LB) = Q1 - 1.5 * IQR = 12.71 - 1.5 * 13.208 = -7.602

- Upper Bound (UB) = Q3 + 1.5 * IQR = 25.918 + 1.5 * 13.208 = 46.230

Thus, the following SQL query checks the outlier values:

```
spark.sql("""
        SELECT dt, AverageTemperature, City
        FROM GlobTemp
        WHERE AverageTemperature < -7.602 OR AverageTemperature > 46.230
""").show()
```

Output:

```
+----------+------------------+---------+
|        dt|AverageTemperature|     City|
+----------+------------------+---------+
|1767-01-01| -8.252999999999998|   Berlin|
|1776-01-01|            -8.336|   Berlin|
|1788-12-01| -7.790000000000001|   Berlin|
|1795-01-01|            -8.019|   Berlin|
|1823-01-01|            -9.809|   Berlin|
|1829-12-01|            -8.242|   Berlin|
|1830-01-01|            -7.604|   Berlin|
|1838-01-01|            -9.813|   Berlin|
|1848-01-01|            -9.276|   Berlin|
|1855-02-01|             -7.67|   Berlin|
|1893-01-01|            -8.071|   Berlin|
|1929-02-01|           -10.125|   Berlin|
|1940-01-01|            -9.689|   Berlin|
|1942-01-01| -7.867000000000001|   Berlin|
|1947-02-01|            -8.272|   Berlin|
|1956-02-01|            -9.646|   Berlin|
|1963-01-01|            -8.026|   Berlin|
|1820-12-01|           -15.398|Changchun|
|1821-01-01|           -15.507|Changchun|
|1821-02-01|-11.039000000000001|Changchun|
+----------+------------------+---------+
only showing top 20 rows
```

Outliers Analysis

- Berlin has had very low temperatures in January for many years, probably due to the fact that it was winter.
- Changchun's winter temperature is also very low, which is in line with the actual situation of its geographical location.

In this case the temperature given seems reasonable as these cities can indeed experience such low temperatures in winter. Therefore, none of them are defined as outlier and there is no need to handle them.

2.3.4 Checking duplicates

The following SQL query checks duplicates if have, the good news is no duplicates in the dataset.

```
spark.sql("""
    SELECT dt, AverageTemperature, City, COUNT(*) as duplicates
    FROM GlobTemp
    GROUP BY dt, AverageTemperature, City
    HAVING duplicates > 1
""").show()
```

Output:

```
+---+------------------+----+----------+
| dt|AverageTemperature|City|duplicates|
+---+------------------+----+----------+
+---+------------------+----+----------+
```

2.3.5 Checking data consistency

The following SQL query checks data consistency, the good news is data consistency in the dataset.

```
spark.sql("""
        SELECT DISTINCT City
        FROM GlobTemp
        WHERE City NOT RLIKE '^[A-Z][a-z]*$'
        """).show()
```

Output:

```
+----------------+
|            City|
+----------------+
|Ho Chi Minh City|
|     Los Angeles|
|   Santo Domingo|
|   Dar Es Salaam|
|     Addis Abeba|
|       Cape Town|
|  Belo Horizonte|
|       São Paulo|
|      Umm Durman|
|Saint Petersburg|
|        Brasília|
|       New Delhi|
|        New York|
|  Rio De Janeiro|
|          Bogotá|
+----------------+
```

3. Spark Program Implementation

3.1 Task 1: Aggregate by Key

Objective: To group and aggregate temperature data by year and compute total temperature, maximum temperature, and minimum temperature.

Implementation Details: The data was aggregated using SQL queries in Apache Spark, grouping by the year and computing the sum, max, and min of the average temperature.

Coding:

```
spark.sql("""SELECT YEAR(dt) AS Year, SUM(AverageTemperature) AS totalTemp,
Max(AverageTemperature) AS maxTemp, Min(AverageTemperature) AS minTemp
        FROM GlobTemp
        GROUP BY Year
        ORDER BY Year DESC""") \
    .show(10)
```

Results: Top 10 years of aggregated temperature data displayed, including their total, maximum, and minimum average temperature.

Year	totalTemp	maxTemp	minTemp
2013	16352.24599999999	37.12600000000001	-21.106
2012	23601.887000000002	37.859	-20.079
2011	23459.036	37.184	-22.029
2010	23894.08100000002	37.899	-18.555
2009	23800.501999999986	36.607	-17.855999999999998
2008	23530.534000000043	37.143	-18.649
2007	23825.123999999985	36.429	-12.937000000000001
2006	23752.792000000012	37.041	-18.862000000000002
2005	23528.687000000013	36.512	-17.621000000000006
2004	23606.644999999968	36.542	-16.813000000000002

only showing top 10 rows

3.2 Task 2: Window Functions

Objective: To showcase the use of window functions by calculating a cumulative sum of average temperature over time for each city.

Implementation Details: Utilized the window functions in Apache Spark to partition data by the city and order it by date, followed by a cumulative sum of the average temperature.

Coding:

```
from pyspark.sql import functions as F
windowval = (Window.partitionBy('City').orderBy('dt')
             .rangeBetween(Window.unboundedPreceding, 0))
df_w_cumsum = df.withColumn('cum_sum', F.sum('AverageTemperature').over(windowval))
df_w_cumsum.show(10)
```

Results: Displayed the aggregated results for the first 10 rows, showcasing cumulative sum for each city.

```
+----------+------------------+--------------------------+-------+-------------+--------+---------+-----------------+
|        dt|AverageTemperature|AverageTemperatureUncertainty|   City|      Country|Latitude|Longitude|          cum_sum|
+----------+------------------+--------------------------+-------+-------------+--------+---------+-----------------+
|1849-01-01|            26.704|                     1.435|Abidjan|Côte D'Ivoire|   5.63N|    3.23W|           26.704|
|1849-02-01|            27.434|                     1.362|Abidjan|Côte D'Ivoire|   5.63N|    3.23W|54.13800000000005|
|1849-03-01|            28.101|                     1.612|Abidjan|Côte D'Ivoire|   5.63N|    3.23W|           82.239|
|1849-04-01|             26.14|        1.3869999999999998|Abidjan|Côte D'Ivoire|   5.63N|    3.23W|          108.379|
|1849-05-01|            25.427|                       1.2|Abidjan|Côte D'Ivoire|   5.63N|    3.23W|          133.806|
|1849-06-01|            24.844|                     1.402|Abidjan|Côte D'Ivoire|   5.63N|    3.23W|           158.65|
|1849-07-01|24.058000000000003|                     1.254|Abidjan|Côte D'Ivoire|   5.63N|    3.23W|          182.708|
|1849-08-01|            23.576|                     1.265|Abidjan|Côte D'Ivoire|   5.63N|    3.23W|          206.284|
|1849-09-01|            23.662|                     1.226|Abidjan|Côte D'Ivoire|   5.63N|    3.23W|          229.946|
|1849-10-01|            25.263|                     1.175|Abidjan|Côte D'Ivoire|   5.63N|    3.23W|          255.209|
+----------+------------------+--------------------------+-------+-------------+--------+---------+-----------------+
only showing top 10 rows
```

3.3 Task 3: Pivot Tables

Objective: To create a pivot table based on cities and countries, showcasing summed average temperature.

Implementation Details: The groupBy and pivot functions in Apache Spark were used to group by city and pivot on the country.

Coding:

```
df.groupBy("City").pivot("Country").sum("AverageTemperature") \
    .show(10)
```

Results: Displayed a pivot table for the first 10 rows.

City	Afghanistan	Angola	Australia	Bangladesh	Brazil	Burma	Canada	Chile	China	Colombia	Congo (Democratic Republic Of The)	Côte D'Ivoire	Dominican Republic	Egypt	Ethiopia	France	Germany	India	Indonesia	Iran	Iraq	Italy	Japan	Kenya	Mexico	Morocco	Nigeria	Pakistan	Peru	Philippines	Russia	Saudi Arabia	Senegal	Singapore	Somalia	South Africa	South Korea	Spain	Sudan	Syria	Taiwan	Tanzania	Thailand	Turkey	Ukraine	United Kingdom	United States	Vietnam	Zimbabwe
Bangalore	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	62338.58699999996	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null
Cairo	null	null	null	null	null	null	null	null	null	null	null	null	null	51822.31500000013	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null
Casablanca	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	50074.63600000004	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null
Guangzhou	null	null	null	null	null	null	null	null	45032.49800000004	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null
Ho Chi Minh City	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	56264.35200000006	null
Fortaleza	null	null	null	null	44807.33300000005	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null
Lima	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	25505.83100000001	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null
Madrid	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	36246.596999999936	null	null	null	null	null	null	null	null	null	null	null
Mashhad	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	27092.64300000001	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null
Singapore	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	54425.406999999905	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null

only showing top 10 rows

3.4 Task 4: Multi-Level Aggregation

Objective: To compute multi-level aggregation on temperature data, specifically, the average of sum temperatures per date for each city.

Implementation Details: Leveraged nested SQL queries in Apache Spark to aggregate data at multiple levels.

Spark Assignment

Coding:

```
spark.sql("""SELECT City, AVG(ds.sumTemp) AS avgTemp
            FROM (SELECT dt,
                    City,
                    SUM(AverageTemperature) AS sumTemp
                FROM GlobTemp
                GROUP BY dt, City) AS ds
            GROUP BY City""") \
    .show(10)
```

Results: The top 10 cities with their average of summed temperatures across different dates were displayed.

City	avgTemp
Bangalore	24.855895933014303
Cairo	21.22125921375925
Casablanca	17.184157858613595
Guangzhou	21.60868426103644
Fortaleza	27.008639541892705
Ho Chi Minh City	27.193983566940563
Lima	16.76911965811967
Madrid	11.448704042956397
Mashhad	12.571992111368898
Singapore	26.523102826510698

only showing top 10 rows

4. Challenges and Solutions

4.1 Challenge 1: Forgotten the syntax of SQL language

Solution: I took a substantial amount of time to review because I had forgotten the syntax of SQL language.

Feedback: SQL is such a practical querying language, and it's understandable that if not used for some time, some details might be forgotten. Reviewing and practicing are great ways to reacquaint oneself with the skills. Additionally, for quick references on specific syntax or commands, considering online resources like official documentation or relevant community discussions can be invaluable.

4.2 Challenge 2: Handling missing values

Solution: I first identified that 4.6% of the data had missing values and removed them. However, I then encountered the challenge of how to replace the original DataFrame view. After some research, I used the createOrReplaceTempView("GlobTemp") method successfully.

Feedback: Handling missing values is a common task in data preprocessing, especially when working with real-world data. Your approach to simply delete these values is straightforward, but it's always good to ensure that it doesn't negatively impact subsequent analyses. As for replacing the DataFrame view, createOrReplaceTempView is indeed a good method, allowing you to update the temporary view after data manipulations so that you can continue your subsequent queries using the same view name.

Overall, I'm delighted to see how I faced challenges head-on and found solutions. Continuous learning and practice are key to improving, and I hope to achieve great results in future projects as well!

5. Reflection

5.1 Strengths of Using Apache Spark

Apache Spark allowed for efficient and quick data manipulation and analysis. The ability to perform SQL-like queries on data frames and the added flexibility of window functions and pivot operations made data aggregation tasks straightforward.

Normally, I lean towards Python for most of my projects. Before embarking on this particular assignment, I couldn't quite grasp why I shouldn't just stick with Python. However, it all became clear once I tried replicating the task in a Python Jupyter environment. This experience highlighted the stark contrast between spark.sql and Python Jupyter. The spark.sql is designed to handle large-scale data with impressive speed. On the other hand, when I ran the dataset in Python Jupyter, I experienced significant lag. In stark contrast, the process was remarkably smooth with spark.sql. Therefore, it bring me other reflection for the strengths of using Apache Spark, Spark's in-memory processing ensured fast computations, even with a substantial amount of data.

5.2 Limitations of Using Apache Spark

One potential limitation of employing Apache Spark for this project is the inherent learning curve that comes with understanding its intricate functions and operations. I found myself revisiting SQL principles to ensure that my codes were achieving the desired outcomes. This was evident in the challenges I faced during the project. Furthermore, while Spark is undeniably proficient in managing vast datasets, its capabilities might be excessive for smaller datasets, where more straightforward tools would be more than adequate. For medium or small-sized datasets, I would naturally gravitate towards Python for processing.