UNIVERSITY OF LIVERPOOL

Big Data Analytics: Assignment 1

COMP 336: Big Data Analytics

Prepared by
Matthew Paver
201582813

Table of Contents

Introduction

For this assignment, I will analyse a Geolife GPS trajectory dataset using PySpark. This involves assessing time-stamped GPS data points that were gathered over five years from over 100 individuals. I will be focusing on the methods and analytical procedures applied to tackle each task and ensuring scalability and efficiency are at the forefront of the solution.

Dataset Description

The dataset given is a portion of the Geolife project dataset from Microsoft Research Asia and consists of GPS paths, from users, spanning a duration of time. Each point in the path carries the following details;

- **UserID**: Identifies individual users, allowing analysis on a per-user basis.
- **Latitude**: Latitude coordinate of the GPS point.
- **Longitude**: Longitude coordinate of the GPS point.
    - These columns allow for the calculation of user movements and filtering by geographic location.
- **AllZero**: Column with zeros, which was ignored in this analysis.
- **Altitude**: Used to analyse changes in altitude across user activities.
- **Timestamp**: Number of days (with fractions)
- **Date**: Date of the GPS record.
- **Time**: Time of the GPS record.
    - These columns are beneficial for converting local time and grouping by day/week.

Executing the Code

The solution runs in a VirtualBox environment with PySpark 3.5.3 and Python 3.10.12 to ensure full compatibility. In addition, please ensure that dataset.txt is in the same directory as PART-1.py.

To run the code, open the script in your desired IDE and select the run button to execute the code.

Methodology

Part-1.py uses PySpark's DataFrames and RDD (Resilient Distributed Dataset) transformations, which break down large data tasks into smaller, manageable parts. These transformations allow for efficient filtering, grouping, and joining of data. This is to handle large datasets without overloading memory.

Part 1.py uses the DataFrames and Resilient Distributed Dataset (RDD) transformations in PySpark to divide data tasks into manageable parts. These transformations enable filtering, grouping and joining of data. This is to handle the dataset without overloading memory.

Decomposition of Tasks

Task 1: Timestamp Adjustment

To convert all timestamps from GMT to the local time zone based on the longitude of each GPS point. The Earth is divided into 24 equal time zones, each covering 15 degrees of longitude. This adjustment ensures accurate time representation for GPS data collected across multiple time zones, enabling a more consistent time-based analysis.

Method

1. I created a user-defined function, *adjust_timezone*, to calculate the timezone offset by dividing the longitude by 15 degrees and rounding down to the nearest integer. This function then applied the offset to adjust each timestamp efficiently based on longitude.
2. The calculated offset was added to the original GMT timestamp and converted as a fraction of a day (offset hours/24) to create an *Adjusted_Timestamp* in local time. This function was applied to each row using PySpark's *withColumn* method.

Code:

```
# Task 1: Convert timestamps to local time zone based on longitude
from pyspark.sql.functions import col, udf
from pyspark.sql.types import DoubleType, IntegerType

# Define the function to calculate timezone offset based on longitude
def calculate_offset(longitude):
    # Calculate the timezone offset based on the longitude (15 degrees per hour)
    return int(longitude // 15)

# Define the function to adjust timestamp using the calculated offset
def adjust_timestamp(longitude, timestamp):
    offset = calculate_offset(longitude)
    return timestamp + offset / 24  # Adjust timestamp by offset in days

# Register User Defined Functions (UDFs)
calculate_offset_udf = udf(calculate_offset, IntegerType())
adjust_timestamp_udf = udf(adjust_timestamp, DoubleType())

# Apply the UDFs to add offset and adjusted timestamp columns
df = df.withColumn("Offset_Hours", calculate_offset_udf(col("Longitude"))) \
      .withColumn("Adjusted_Timestamp", adjust_timestamp_udf(col("Longitude"), col("Timestamp")))

# Show a limited number of rows with relevant columns to verify adjustments
df.select("UserID", "Latitude", "Longitude", "Timestamp", "Offset_Hours", "Adjusted_Timestamp").show(10)  # Show only 10 rows
```

Output:

The output below shows a sample of records with adjusted timestamps based on their longitude. Each record displays the original timestamp in GMT, the calculated timezone offset in hours, and the new Adjusted_Timestamp column. This adjustment aligns timestamps with the local time zone of each GPS point, enhancing the dataset's accuracy in representing local time.

```
+------+---------+----------+----------------+------------+-------------------+
|UserID| Latitude| Longitude|       Timestamp|Offset_Hours|Adjusted_Timestamp|
+------+---------+----------+----------------+------------+-------------------+
|   130|39.975088| 116.33269|40000.7996759259|           7| 40001.09134259256|
|   130| 39.97504|116.332806|   40000.7996875|           7| 40001.09135416667|
|   130|39.975009|116.332997|40000.7996990741|           7|40001.091365740765|
|   130|39.975048|116.332932|40000.7997106481|           7| 40001.09137731476|
|   130|39.974977| 116.33305|40000.7997337963|           7| 40001.09140046297|
|   130|39.974967|116.333116| 40000.799849537|           7| 40001.09151620366|
|   130|39.974931|116.333188|40000.7999652778|           7|40001.091631944466|
|   130|39.974927|116.333249|40000.8000347222|           7| 40001.09170138886|
|   130|39.974953| 116.33331|40000.8001388889|           7| 40001.09180555557|
|   130|39.974988|116.333359|40000.8001967593|           7|40001.091863425965|
+------+---------+----------+----------------+------------+-------------------+
only showing top 10 rows
```

Pro

- Provides local time context with a simple approach. It also makes timestamps more meaningful in relation to their geographic location.

Con

- The timezone adjustment is approximate, as it only considers longitude-based divisions rather than country-specific time zones. This means that local time zones, like daylight saving time or region-specific offsets, are not accounted for. This could lead to slight inaccuracies in some cases if individuals are on the boundaries.

Task 2: Beijing Records Filtering

To filter records within the approximate geographic borders of Beijing. The area includes points with latitude between 39.5 and 40.5 and longitude between 115.5 and 117.5, which are the main areas of Beijing. This step focuses the analysis on Beijing-specific data highlighted through the latitude and longitude. Even though Beijing's specific area isn't exactly rectangular, it provides an efficient approximation.

Method

1. Used PySpark's filter() function to select records meeting the specified latitude and longitude bounds.
2. A new DataFrame, *df_beijing*, was created to store the filtered records, which can be referenced for any analysis specific to Beijing.

Code:

```
# Task 2: Filter records within Beijing's borders
df_beijing = df.filter(
    (col("Latitude") >= 39.5) & (col("Latitude") <= 40.5) &
    (col("Longitude") >= 115.5) & (col("Longitude") <= 117.5)
)

# Output count of records in Beijing
print(f"Number of records in Beijing: {df_beijing.count()}")
```

Output:

The output below shows the count of records that fall within Beijing's specified latitude and longitude bounds. This count represents the total number of GPS points available in the dataset for this region to be data specific to Beijing.

```
Number of records in Beijing: 2776254
```

Pro

- Quick to implement with some approximation for Beijing's boundaries.

Con

- For further accurate analysis, Beijing's boundaries could be established more distinctly. This could alter the number of GPS points on the boundary of Beijing.

Task 3: Days with More Than 10 Data Points

Calculate the number of days each user recorded more than 10 data points. This analysis helps identify active days for each user and provides insights into user behaviour and movement patterns.

Method:

1. The dataset was grouped by UserID and Date, counting the data points for each day per user.
2. Days with more than 10 data points were kept, filtering out days with less activity. This helps identify users with significant daily activities, indicating possibly longer or more frequent GPS sessions.
3. I then grouped by UserID to count each user's total number of active days.

4. Finally, the DataFrame was sorted by day count in descending order, displaying the top 6 users with the highest number of active days.

Code:

```
# Task 3: Calculate the number of days each user recorded more than 10 data points
df_grouped_by_user_day = df.groupBy("UserID", "Date").count().filter(col("count") > 10)
df_days_per_user = df_grouped_by_user_day.groupBy("UserID").count()

# Output top 6 users ranked by the number of days
df_days_per_user.orderBy(col("count").desc(), col("UserID")).show(6)
```

Output:

The output below shows the top 6 users with the highest count of days containing more than 10 data points, highlighting users with frequent activity. These users have demonstrated the most consistent engagement, making them ideal subjects for further analysis.

```
+------+-----+
|UserID|count|
+------+-----+
|   153| 1272|
|   144|  338|
|   140|  241|
|   142|  103|
|   134|   54|
|   147|   51|
+------+-----+
only showing top 6 rows
```

Pro

- Highlights users with significant daily activity

Con

- The threshold may not account for shorter trips or single day records.

Task 4: Weeks with More Than 100 Data Points

For each user, calculate the number of weeks they recorded more than 100 data points. This helps to assess each user's periods of sustained activity and identify users with consistent weekly engagement.

Method:

1. The weekofyear() function was applied to extract the week number from each date, creating a "Week" column to allow grouping by UserID and Week.
2. I then grouped them by UserID and Week to count the data points for each week per user.
3. Only weeks with more than 100 data points were retained to highlight significant weekly GPS activity.
4. I then calculated the total qualifying weeks per user and ordered the results in descending order, showing users with the highest counts.

Code:

```
# Task 4: Calculate the number of weeks each user recorded more than 100 data points
df_with_week = df.withColumn("Week", weekofyear(col("Date")))

df_grouped_by_user_week = df_with_week.groupBy("UserID", "Week").count().filter(col("count") > 100)
df_weeks_per_user = df_grouped_by_user_week.groupBy("UserID").count()

# Output user ID along with the count of weeks
df_weeks_per_user.orderBy(col("count").desc(), col("UserID")).show(6)
```

Output:

The output below displays the count of weeks with more than 100 data points per user. Users with higher counts demonstrate consistent activity over weeks, indicating sustained engagement. These users are valuable for studying long-term behaviour patterns.

```
+------+-----+
|UserID|count|
+------+-----+
|   153|   53|
|   144|   47|
|   140|   45|
|   142|   23|
|   134|   15|
|   147|   12|
+------+-----+
only showing top 6 rows
```

Pro

- Identifies users with high weekly GPS engagement

Con

- A high threshold may exclude infrequent users.

Task 5: Northernmost Point and Beijing Presence Check

To identify the northernmost latitude reached by each user and verify if each of the top 6 users recorded data in the Beijing region. Tracking the maximum latitude per user provides insight into their range of travel, and checking Beijing presence reveals if their paths intersected with this area.

Method:

1. The *spark_max()* function was applied to the Latitude column to identify the maximum latitude for each user, indicating their northernmost point.
2. The highest latitude ,*MaxLatitude,* reached by each user helps identify users who travelled the farthest north in the dataset. Additionally, joining the data with the filtered Beijing data allows us to confirm if these top users recorded any points within Beijing
3. The date associated with each maximum latitude was also extracted to capture when this point was reached.
4. I then performed a left join with the Beijing DataFrame, adding a Boolean column, *InBeijing,* which is True if the user has recorded data within the Beijing bounds and False otherwise.

Code:

```
# Task 5: Find the northernmost point (greatest Latitude) for each user
df_northernmost = df.groupBy("UserID").agg(
    spark_max("Latitude").alias("MaxLatitude"),
    spark_max("Date").alias("Date")
)

# Output top 6 users based on MaxLatitude
df_northernmost.orderBy(col("MaxLatitude").desc(), col("UserID")).show(6)

# Check if these top users have any records within Beijing
df_northernmost_beijing = df_northernmost.join(
    df_beijing.select("UserID").distinct(),
    "UserID", "left_outer"
).withColumn("InBeijing", col("UserID").isNotNull())

# Output whether the top users have been to Beijing
df_northernmost_beijing.select("UserID", "InBeijing").show(6)
```

Output:

The output below displays the *MaxLatitude* for the top 6 users and indicates whether each user has data recorded within Beijing. This provides insight into users' overall geographic reach and possible travels to Beijing.

```
+------+----------------+----------+
|UserID|     MaxLatitude|      Date|
+------+----------------+----------+
|   142|       64.751993|2011-01-31|
|   153|       55.976195|2012-06-17|
|   140|      47.7119616|2009-09-28|
|   144|      44.1550933|2009-09-24|
|   149|      40.7217849|2009-09-12|
|   139|40.6649833333333|2007-10-09|
+------+----------------+----------+
only showing top 6 rows
```

```
+------+---------+
|UserID|InBeijing|
+------+---------+
|   137|     true|
|   133|     true|
|   140|     true|
|   132|     true|
|   139|     true|
|   130|     true|
+------+---------+
only showing top 6 rows
```

Pro

- Allows regional analysis by revealing the geographic reach of users.

Con

- It does not account for complex travel patterns and only shows the top latitude.

Task 6: Daily Altitude Span

To calculate the daily altitude span (the difference between the maximum and minimum altitude) for each user and find the highest daily span for each user. This analysis helps to capture variations in elevation throughout each user's travels, with the maximum span highlighting the most significant elevation change observed on a single day.

Method:

1. The data was grouped by UserID and Date to isolate each user's altitude records per day.
2. For each day, the altitude span was calculated by subtracting the minimum altitude from the maximum altitude, representing the variation in elevation.
3. I then calculated the maximum daily altitude span for each user across all recorded days.
4. The top 6 users with the highest maximum daily altitude span were displayed.

Code:

```
# Task 6: Calculate daily altitude span for each user (Max Altitude - Min Altitude per day)
df_altitude_span = df.groupBy("UserID", "Date").agg(
    (spark_max("Altitude") - spark_min("Altitude")).alias("AltitudeSpan")
)

# Find the maximum altitude span for each user across all days
df_max_altitude_span = df_altitude_span.groupBy("UserID").agg(
    spark_max("AltitudeSpan").alias("MaxAltitudeSpan")
)

# Output top 6 users based on max altitude span
df_max_altitude_span.orderBy(col("MaxAltitudeSpan").desc(), col("UserID")).show(6)
```

Output:

The output below lists the top 6 users with the highest daily altitude spans, representing the maximum elevation changes they experienced on any single day. Users with high altitude spans may have travelled through areas with significant elevation differences, such as hilly regions or mountains.

```
+------+------------------+
|UserID|   MaxAltitudeSpan|
+------+------------------+
|   144|           86013.8|
|   140|           54826.1|
|   153|25469.668608923836|
|   147| 7020.102349081365|
|   142|            6496.0|
|   143| 6284.338369422574|
+------+------------------+
only showing top 6 rows
```

Pro

- Shows users travelling over varied terrain, highlighting geographical diversity.

Con

- Does not differentiate urban altitude variability from natural terrain changes

---

Task 7: Total Distance Travelled and Longest Distance Day

To calculate each user's total distance travelled per day and identify the longest travel day. This analysis helps to capture daily travel patterns and highlight days with extensive movement.

Method:

1. The Haversine formula was used to calculate distances between consecutive latitude and longitude coordinates, providing accurate measurements of the Earth's surface. The formula was implemented with guidance from *Stack Overflow (n.d.)*.
2. A window function with lag was used to access each user's previous GPS coordinates, enabling the calculation of the distance to the next point.
3. Daily distances were summed for each user, and the maximum distance was determined to identify the day when each user travelled the furthest.
4. The top 6 users with the highest maximum daily distances were displayed.

Code:

```
# Task 7: Calculate total distance travelled per day for each user
def calculate_distance(lat1, lon1, lat2, lon2):
    if lat1 is None or lon1 is None or lat2 is None or lon2 is None:
        return 0.0  # Return 0 distance if any of the coordinates are None
    # Haversine formula to calculate the distance between two points on Earth
    R = 6371  # Earth radius in kilometers
    d_lat = math.radians(lat2 - lat1)
    d_lon = math.radians(lon2 - lon1)
    a = math.sin(d_lat / 2) ** 2 + math.cos(math.radians(lat1)) * math.cos(math.radians(lat2)) * math.sin(d_lon / 2) ** 2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    return R * c

# Register the UDF for distance calculation
distance_udf = udf(calculate_distance)

# Create a window to calculate the distance between consecutive points
window_spec = Window.partitionBy("UserID", "Date").orderBy("Timestamp")

# Calculate distance between consecutive points
df_with_distance = df.withColumn(
    "Distance",
    distance_udf(
        lag("Latitude", 1).over(window_spec), lag("Longitude", 1).over(window_spec),
        col("Latitude"), col("Longitude")
    )
)

# Sum up daily distance travelled by each user
df_daily_distance = df_with_distance.groupBy("UserID", "Date").agg(
    spark_sum("Distance").alias("TotalDistance")
)

# Find the day with the maximum distance for each user
df_max_daily_distance = df_daily_distance.groupBy("UserID").agg(
    spark_max("TotalDistance").alias("MaxDistance")
)

# Output the top 6 users based on maximum distance travelled in a day
df_max_daily_distance.orderBy(col("MaxDistance").desc(), col("UserID")).show(6)

# Calculate the total distance travelled by all users
total_distance = df_daily_distance.agg(spark_sum("TotalDistance")).collect()[0][0]
print(f"Total distance travelled by all users: {total_distance:.2f} km")

# Stop the Spark session
spark.stop()
```

Output:
The output below shows the top 6 users with the longest distances travelled on a single day. Users with significant daily travel distances may have gone across cities or regional journeys, indicating a higher level of mobility.

```
+------+------------------+
|UserID|       MaxDistance|
+------+------------------+
|   142| 9717.989553601705|
|   153| 9270.768789721904|
|   144| 8065.822732693285|
|   137|1629.8245491677008|
|   135|1599.8622067236083|
|   159|1265.0809557863565|
+------+------------------+
only showing top 6 rows
```

```
Total distance travelled by all users: 190292.99 km
```

Pro

- Detailed view of travel distance due to Haversine formula as it accurately calculates distance.

Con

- It does not account for elevation differences, which could affect distance accuracy.

References

- **Stack Overflow. (n.d.)** *Getting distance between two points based on latitude/longitude.* Available at: https://stackoverflow.com/questions/19412462/getting-distance-between-two-points-based-on-latitude-longitude (Accessed: 7 November 2024).

Summary

All tasks were completed using the Geolife GPS trajectory dataset by adjusting timestamps to match time and filtering movement data. This was done within Beijing to identify high mobility users and calculate travel distances, and altitude changes effectively with PySparks to handle a large dataset. The findings offer valuable insights into users' travel habits and movements in various locations.