Spark DataFrame-based programming

February 22, 2024

1 Analyze bike-sharing system of Barcelona - Spark DataFrame-based programming

[ ]:
```
# In this analysis, I am going to consider the occupancy of the stations where␣
↪users can pick up or drop off
# bikes in order to identify the most "critical" timeslots (day of the week,␣
↪hour) for each station.
```

[ ]:
```
# data is located on the big data cluster and I am going to read data from␣
↪there.
# there are two types of data:
# 1. register.csv: This contains the historical information about the number of␣
↪used and free slots for
#    ~3000 stations from May 2008 to September 2008. Each line of register.csv
#    corresponds to one reading about the situation of one station at a␣
↪specific timestamp.
# 2. stations.csv: It contains the description of the stations (station_id,␣
↪latitude, longitude, name).
```

[3]:
```
registerPath = "/data/students/bigdata_internet/lab3/register.csv"
```

[4]:
```
stationPath = "/data/students/bigdata_internet/lab3/stations.csv"
```

[ ]:
```
'''In this analysis, PySpark was utilized for its robust distributed computing␣
↪capabilities,
ideal for handling large datasets efficiently.
If you're using the PySpark shell, no additional setup is necessary.
However, for those working in a Python environment, setting up PySpark involves␣
↪the following steps:
1. Install PySpark: Begin by installing PySpark using pip:
pip install pyspark
2. Configure PySpark.sql: In your Python script or interactive session, include␣
↪the following configuration
to initialize PySpark.sql:
```python
from pyspark.sql import SparkSession
```
```

```
spark = SparkSession.builder.getOrCreate()
```
Ensure to execute this configuration before performing any PySpark operations.
For comprehensive installation and configuration instructions, refer to the␣
↪official PySpark documentation:
PySpark Installation Guide
'''
```

[5]:
```
spark = SparkSession.builder.getOrCreate()
```

[ ]:
```
The file is separated with tab so I will pu the separator \t.
```

[6]:
```
registerDF = spark.read.load(registerPath, format="csv", header=True, sep='\t')
```

[6]:
```
registerDF.count()
```

[6]: 25319028

[ ]:
```
To clean the data, I am going to filter data that their used_slot != 0 or␣
↪their free_slot != 0
because whether there are some bicycles in station or not and it is not␣
↪possible to have 0 for both.
```

[7]:
```
registerDF_clean = registerDF.filter("used_slots != 0 OR free_slots != 0")
```

[8]:
```
registerDF_clean.count()
```

[8]: 25104121

[ ]:
```
There are 25,319,028 rows (without the header) in the original file and it␣
↪decreases to 25,104,121
(without the header) after we did the filter and deleted wrong data.
```

1.2 stations.csv

[ ]:
```
Reading station data as a DataFrame
```

[8]:
```
stationDF = spark.read.load(stationPath, format="csv", header=True, sep='\t')
```

[ ]:
```
Write a Spark application that selects the pairs (station, timeslot) that are␣
↪characterized
by a high "criticality" value
```
```

[7]:
```
# Convert timestamp to timeslot (weekday, hour)
```

[9]:
```
registerDF_timeslot = registerDF_clean.selectExpr("station",␣
↪"(date_format(timestamp,'EEEE'), hour(timestamp)) as timeslot",␣
↪"used_slots", "free_slots")
```

……………………………………………………………………….

[ ]:
```
# Computes the criticality value C(Si, Tj) for each pair (Si, Tj)
```

[ ]:
```
# Filter only those data that have free_slot = 0 which means that all of the␣
↪slots were used
```

[10]:
```
zeroFreeSlots = registerDF_timeslot.filter("free_slots = 0")
```

[ ]:
```
# Group the dataframe based on (station_id, timeslot) and count the number of␣
↪free_slots = 0 readings
```

[11]:
```
zeroFreeSlotsGroup = zeroFreeSlots.groupBy("station", "timeslot").count()
```

[ ]:
```
# Rename the columns
```

[12]:
```
zeroFreeSlotsGroup = zeroFreeSlotsGroup.withColumnRenamed("count",␣
↪"count_zero_free_slots")
```

[ ]:
```
# Group the original dataframe based on (station_id, timeslot) and count the␣
↪number of all readings
```

[13]:
```
registerDFTotalGroup = registerDF_timeslot.groupBy("station", "timeslot").
↪count()
```

[ ]:
```
# Rename the columns
```

[14]:
```
registerDFTotalGroup = registerDFTotalGroup.withColumnRenamed("count",␣
↪"count_total_free_slots")
```

[ ]:
```
# Join two previous dataframes
```

[15]:
```
joinedRegister = zeroFreeSlotsGroup.join(
    registerDFTotalGroup,
    (zeroFreeSlotsGroup.station == registerDFTotalGroup.station) &
    (zeroFreeSlotsGroup.timeslot == registerDFTotalGroup.timeslot)
).select(
    zeroFreeSlotsGroup['station'],
    zeroFreeSlotsGroup['timeslot'],
    zeroFreeSlotsGroup['count_zero_free_slots'],
    registerDFTotalGroup['count_total_free_slots']
```

```
)
```

```
23/12/23 09:02:30 WARN sql.Column: Constructing trivially true equals predicate,
'station#10 = station#10'. Perhaps you need to use aliases.
23/12/23 09:02:30 WARN sql.Column: Constructing trivially true equals predicate,
'timeslot#36 = timeslot#36'. Perhaps you need to use aliases.
```

[ ]:
```
# Calculate the criticality value
```

[16]:
```
criticalityRegister = joinedRegister.selectExpr("station", "timeslot",␣
 ↪"count_zero_free_slots/count_total_free_slots as criticality")
```

[ ]:
```
# Now, I will select only the critical pairs (Si, Tj) having a criticality␣
 ↪value C(Si, Tj) greater than
# a minimum threshold (0.6).
```

[17]:
```
criticalRegister = criticalityRegister.filter("criticality >= 0.6")
```

[ ]:
```
# Order the results by increasing criticality.
```

[18]:
```
orderedCriticalRegister = criticalRegister.sort("criticality", ascending=True)
```

[ ]:
```
# Show the most critical (station_id, timeslot) in Barcelona
```

[ ]:
```
orderedCriticalRegister.show()
```

[ ]:
```
# Store the sorted critical pairs C(Si, Tj) in the output folder (also an␣
 ↪argument of the application),
# by using a csv files (with header), where columns are separated by "tab".␣
 ↪Store exactly the following
# attributes separated by a "tab":
# station / station longitude / station latitude / day of week / hour /␣
 ↪criticality value
```

[ ]:
```
# Join critical stations dataframe from register data with station data
```

[19]:
```
criticalOutput = orderedCriticalRegister.join(
    stationDF, orderedCriticalRegister.station == stationDF.id).select(
    orderedCriticalRegister["station"],
    stationDF["longitude"],
    stationDF["latitude"],
    orderedCriticalRegister["timeslot"],
    orderedCriticalRegister["criticality"])
```

[20]:
```
orderedCriticalOutput = criticalOutput.sort("criticality", ascending=True)
```

[ ]:
```
# Register the function for week
```

[21]:
```
spark.udf.register('week', lambda x: x[0])
```

[21]: <function __main__.<lambda>(x)>

[ ]:
```
# Register the function for hour
```

[22]:
```
spark.udf.register('hour', lambda y: y[1])
```

```
23/12/23 09:02:46 WARN analysis.SimpleFunctionRegistry: The function hour
replaced a previously registered function.
```

[22]: <function __main__.<lambda>(y)>

[ ]:
```
# Select the desired columns
```

[23]:
```
finalDF = orderedCriticalOutput.selectExpr("station", "longitude", "latitude",␣
 ↪"week(timeslot) as week", "hour(timeslot) as hour", "criticality")
```

[ ]:
```
# Save the output
```

[23]:
```
finalDF.write.csv('critical-stations-Barcelona-DataFrame', header=True,␣
 ↪sep='\t')
```

[24]:
```
finalDF.show()
```

```
[Stage 13:=============================>                           (2 + 2) / 4]

+-------+---------+---------+--------+----+------------------+
|station|longitude| latitude|    week|hour|       criticality|
+-------+---------+---------+--------+----+------------------+
|      9| 2.185294|41.385006|  Friday|  10|0.6129032258064516|
|     10| 2.185206|41.384875|Saturday|   0| 0.622107969151671|
|     58| 2.170736|41.377536|  Monday|   1|0.6239554317548747|
|      9| 2.185294|41.385006|  Friday|  22|0.6258389261744967|
|     58| 2.170736|41.377536|  Monday|   0|0.6323119777158774|
+-------+---------+---------+--------+----+------------------+
```

[ ]:
```
# In this section, I am going to compute the distance between each station and␣
 ↪the city center.
# The city center has coordinates:
```

```
# latitude = 41.386904
# longitude = 2.169989
# To compute the distance implement the Haversine function (use the formula
# in https://en.wikipedia.org/wiki/Haversine_formula).
# Then, compute the average number of used_slots per station
```

[ ]:
```
# Turn the latitude and longitude columns to double type
```

[25]:
```
from pyspark.sql.functions import col
```

[27]:
```
stationDF = stationDF.withColumn("latitude", stationDF["latitude"].
 ↪cast("double"))
stationDF = stationDF.withColumn("longitude", stationDF["longitude"].
 ↪cast("double"))
```

[ ]:
```
# Define the function to compute the haversine
```

[28]:
```
import math
def haversine(lat, lon):
    # City center coordination
    lat1 = 41.386904
    lon1 = 2.169989
    # Radius of the Earth in kilometers
    R = 6371.0
    # Convert latitude and longitude from degrees to radians
    lat1, lon1, lat, lon = map(math.radians, [lat1, lon1, lat, lon])
    dlat = lat - lat1
    dlon = lon - lon1
    hav = math.sin(dlat / 2) ** 2 + math.cos(lat1) * math.cos(lat) * math.
 ↪sin(dlon / 2) ** 2
    distance = 2 * R * math.asin(math.sqrt(hav))
    return distance
```

[ ]:
```
# Register the haversine function
```

[29]:
```
spark.udf.register('hav', haversine)
```

[29]:
```
<function __main__.haversine(lat, lon)>
```

[ ]:
```
# Calculate the distance
```

[30]:
```
dinstanceStationDF = stationDF.selectExpr("id", "hav(latitude, longitude) as␣
 ↪distance")
```

[ ]:
```
# Join the distanceStationDF dataframe with cleaned register dataframe and␣
 ↪select the desired columns
```

[31]:
```
joinedRegisterStation = registerDF_clean.join(
    dinstanceStationDF, registerDF_clean.station == dinstanceStationDF.id).
 ↪select(
        registerDF_clean['station'],
        registerDF_clean['used_slots'],
        dinstanceStationDF['distance'])
```

[ ]:
```
# Now, I want to find the stations that are closer than 1.5 km from the center
```

[ ]:
```
# Filter distance closer than 1.5 km.
```

[32]:
```
closerStations = joinedRegisterStation.filter("distance < 1.5")
```

[ ]:
```
# Turn the used_slots column to float type
```

[33]:
```
closerStations = closerStations.withColumn("used_slots",␣
 ↪closerStations["used_slots"].cast("float"))
```

[ ]:
```
# Calculate the average of used_slots for closer stations
```

[34]:
```
avgCloserStations = closerStations.agg({"used_slots": "avg"})
```

[35]:
```
avgCloserStations.show()
```

```
[Stage 15:================================================>        (6 + 1) / 7]

+----------------+
|avg(used_slots)|
+----------------+
| 8.174875311666|
+----------------+
```

[ ]:
```
# Now, I am going to find the stations that are farther than 1.5 km from the␣
 ↪center
```

[ ]:
```
# Filter distance further than 1.5 km.
```

[36]:
```
furtherStations = joinedRegisterStation.filter("distance >= 1.5")
```

[ ]:
```
# Calculate the average of used_slots for further stations
```

[37]:
```
avgFurtherStations = furtherStations.agg({"used_slots": "avg"})
```

[38]:
```
avgFurtherStations.show()
```

```
[Stage 18:===================================================>          (6 + 1) / 7]

+-----------------+
|  avg(used_slots)|
+-----------------+
|7.913817257872483|
+-----------------+
```

[ ]:
```
# The result shows that the average of used slots of closer ones is higher than
↪further ones.
# The average of used slots for closer stations is 8.17 and it is 7.91 for
↪further ones.
```