Spark RDD-based programming

February 22, 2024

1 Analyze bike-sharing system of Barcelona - Spark RDD-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).
```

[1]:
```
registerPath = "/data/students/bigdata_internet/lab3/register.csv"
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: In your Python script or interactive session, include the␣
 ↪following configuration
to initialize PySpark:
```python
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("MyApp")
```
```

```
sc = SparkContext(conf=conf)
```
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
'''
```

[ ]:
```
Reading register data as a RDD
```

[2]:
```
registerRDD = sc.textFile(registerPath)
```

[ ]:
```
The file is separated with tab so I will split each row by \t.
```

[3]:
```
registerRDDList = registerRDD.map(lambda l: l.split('\t'))
```

[5]:
```
registerRDD.count()
```

[5]: 25319029

[ ]:
```
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.
```

[4]:
```
registerRDDFiltered = registerRDDList.filter(lambda l: l[2] != "0" or l[3] !=
↪"0")
```

[7]:
```
registerRDDFiltered.count()
```

[7]: 25104122

[ ]:
```
There are 25,319,029 rows (one row is for the header) in the original file
↪and
it decreases to 25,104,122 (one row for the header) after we did the filter
↪and deleted wrong data.
```

………………………………………………………………………….

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

[5]:
```
stationRDD = sc.textFile(stationPath)
```
```

[6]:
```
stationRDDList = stationRDD.map(lambda l: l.split('\t'))
```

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

[ ]:
```
# In this section I am going to find critical stations which have the most used␣
↪bicycles
```

[ ]:
```
# Because the file is csv, there is header that I have to remove it because I␣
↪will analyze by RDD-based programming
```

[7]:
```
headerR = registerRDDFiltered.first()
```

[8]:
```
registerCleanRDD = registerRDDFiltered.filter(lambda x: x != headerR)
```

[9]:
```
headerS = stationRDDList.first()
```

[10]:
```
stationCleanRDD = stationRDDList.filter(lambda x: x != headerS)
```

[ ]:
```
# For this analysis I will use "day of week" and "hour" to find critical␣
↪timeslots
# So, I am changing the timestamp into this format
```

[11]:
```
from datetime import datetime as dt
```

[12]:
```
def format_timestamp(l):
    timestamp = dt.strptime(l[1], "%Y-%m-%d %H:%M:%S")
    formatted_timestamp = dt.strftime(timestamp, "%A, %H")
    l[1] = formatted_timestamp
    return l
```

[13]:
```
registerTimeslot = registerCleanRDD.map(format_timestamp)
```

…………………………………………………………………………….

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

[ ]:
```
# Turn register data into (k, v) pairs of (station_id, timslot) and (used_slot,␣
↪free_slot)
```

[14]:
```
registerKeyValue = registerTimeslot.map(lambda l: ((l[0], l[1]), [l[2], l[3]]))
```

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

[15]:
```
zeroFreeSlots = registerKeyValue.filter(lambda t: t if t[1][1] == '0' else None)
```

[ ]:
```
# Turn this data into (k, v) pairs of (station_id, timeslot) and 1 in order to
↪be able to find the number of
# (station_id, timeslot) with zero free_slot (all bicycles were used)
```

[16]:
```
zeroNumber = zeroFreeSlots.map(lambda x: (x[0], 1))
```

[17]:
```
numberZero = zeroNumber.reduceByKey(lambda a,b: a+b)
```

[ ]:
```
# Turn the register data into (k, v) pairs of (station_id, timeslot) and 1 in
↪order to be able to find
# the number of all pairs (station_id, timeslot) readings.
```

[18]:
```
registerNumber = registerKeyValue.map(lambda l: (l[0], 1))
```

[19]:
```
numberTotal = registerNumber.reduceByKey(lambda a,b: a+b)
```

[ ]:
```
# Join two previous data (number of free_slots = 0 and all readings) for each
↪pair (station_id, timeslot)
```

[20]:
```
joinedZeroTotal = numberZero.join(numberTotal)
```

[ ]:
```
# The ration between these two data will give us the criticality value of each
↪pair (station_id, timeslot)
```

[21]:
```
criticalityRDD = joinedZeroTotal.map(lambda l: (l[0], int(l[1][0])/
↪int(l[1][1])))
```

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

[22]:
```
criticalPointsRDD = criticalityRDD.filter(lambda x: float(x[1])>=0.6)
```

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

[23]:
```
orderedCriticalPointsRDD = criticalPointsRDD.sortBy(lambda x: float(x[1]), True)
```

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

[25]:
```
orderedCriticalPointsRDD.collect()
```

[25]:
```
[(('9', 'Friday, 10'), 0.6129032258064516),
 (('10', 'Saturday, 00'), 0.622107969151671),
 (('58', 'Monday, 01'), 0.6239554317548747),
 (('9', 'Friday, 22'), 0.6258389261744967),
 (('58', 'Monday, 00'), 0.6323119777158774)]
```

[ ]:
```
# 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
```

[24]:
```
orderedCriticalSeparated = orderedCriticalPointsRDD.map(lambda x: (x[0][0],␣
↪(x[0][1], x[1])))
```

[25]:
```
stationPairRDD = stationCleanRDD.map(lambda s: (s[0], (s[1], s[2])))
```

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

[26]:
```
joinedCriticalStationsRDD = stationPairRDD.join(orderedCriticalSeparated)
```

[27]:
```
finalRDD = joinedCriticalStationsRDD.map(lambda s: [s[0], s[1][0][0],␣
↪s[1][0][1], s[1][1][0].split(',')[0], s[1][1][0].split(',')[1], s[1][1][1]])
```

[28]:
```
finalSortedRDD = finalRDD.sortBy(lambda x: float(x[5]), True)
```

[ ]:
```
# Add the header to RDD
```

[29]:
```
headerList = [['station', 'station_longitude', 'station_latitude',␣
↪'day_of_week', 'hour', 'criticality_value']]
```

[30]:
```
headerRDD = sc.parallelize(headerList)
```

[31]:
```
csvFinal = headerRDD.union(finalSortedRDD)
```

[32]:
```
def to_string(x):
    x[5] = str(x[5])
    return x
```

[33]:
```
csvFinal = csvFinal.map(to_string)
```

[34]:
```
finalTSVRDD = csvFinal.map(lambda x: '\t'.join(x))
```

[ ]:
```
# save the result
```

[35]:
```
finalTSVRDD.saveAsTextFile('critical-stations-Barcelona-RDD')
```

[36]:
```
finalTSVRDD.collect()
```

[36]:
```
['station\tstation_longitude\tstation_latitude\tday_of_week\thour\tcriticality_v
alue',
 '9\t2.185294\t41.385006\tFriday\t 10\t0.6129032258064516',
 '10\t2.185206\t41.384875\tSaturday\t 00\t0.622107969151671',
 '58\t2.170736\t41.377536\tMonday\t 01\t0.6239554317548747',
 '9\t2.185294\t41.385006\tFriday\t 22\t0.6258389261744967',
 '58\t2.170736\t41.377536\tMonday\t 00\t0.6323119777158774']
```

[ ]:
```
# 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
```

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

[66]:
```
import math
def haversine(x):
    lat1 = 41.386904
    lon1 = 2.169989
    # Radius of the Earth in kilometers
    R = 6371.0
    x[2] = float(x[2])
    x[1] = float(x[1])
    # Convert latitude and longitude from degrees to radians
    lat1, lon1, x[2], x[1] = map(math.radians, [lat1, lon1, x[2], x[1]])
    dlat = x[2] - lat1
    dlon = x[1] - lon1
    hav = math.sin(dlat / 2) ** 2 + math.cos(lat1) * math.cos(x[2]) * math.
↪sin(dlon / 2) ** 2
    distance = 2 * R * math.asin(math.sqrt(hav))
    return x + [distance]
```

[ ]:
```
# Add the distance to RDD
```

[43]:
```
stationDistanceRDD = stationCleanRDD.map(haversine)
```

[ ]:
```
# Create pair-RDD for (stations, distance)
```

[44]:
```
stationDistancePair = stationDistanceRDD.map(lambda x: (x[0], x[4]))
```

[ ]:
```
# Create pair-RDD for (stations, used_slots)
```

[45]:
```
registerPair = registerCleanRDD.map(lambda l: (l[0], int(l[2])))
```

[ ]:
```
# Group stations
```

[46]:
```
registerPairGrouped = registerPair.groupByKey()
```

[ ]:
```
# Calculate the average of used slaots in each station
```

[47]:
```
registerPairReduced = registerPairGrouped.map(lambda x: (x[0], sum(x[1])/
 ↪len(x[1])))
```

[ ]:
```
# Join station pair-RDD and register pair-RDD
```

[48]:
```
distanceJoinedRDD = stationDistancePair.join(registerPairReduced)
```

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

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

[49]:
```
closeStations = distanceJoinedRDD.filter(lambda x: x[1][0] < 1.5)
```

[ ]:
```
# calculate the number of stations closer than 1.5 km.
```

[50]:
```
closeStations.count()
```

[50]: 64

[ ]:
```
# Calculate the sum of used_slots of stations closer than 1.5 km to city center
```

[51]:
```
closeStationsUsedSlots = closeStations.map(lambda x: float(x[1][1]))
```

[52]:
```
closeStationsUsedSlotsSum = closeStationsUsedSlots.reduce(lambda a, b: a + b)
```

[53]:
```
print(closeStationsUsedSlotsSum)
```

523.2437013187326

[ ]:
```
# The average of used_slots of stations closer than 1.5 km from city center
```

[54]:
```
avgCloseStationsUsedSlots = closeStationsUsedSlotsSum/closeStations.count()
```

[55]:
```
print(avgCloseStationsUsedSlots)
```

8.175682833105197

[ ]:
```
# 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.
```

[56]:
```
furtherStations = distanceJoinedRDD.filter(lambda x: x[1][0] >= 1.5)
```

[ ]:
```
# calculate the number of stations
```

[57]:
```
furtherStations.count()
```

[57]: 220

[ ]:
```
# Calculate the sum of used_slots of stations further than 1.5 km from city␣
↪center
```

[58]:
```
furtherStationsUsedSlots = furtherStations.map(lambda x: float(x[1][1]))
```

[59]:
```
furtherStationsUsedSlotsSum = furtherStationsUsedSlots.reduce(lambda a, b: a +␣
↪b)
```

[60]:
```
print(furtherStationsUsedSlotsSum)
```

1731.242401514143

[ ]:
```
# The average of used_slots of stations further than 1.5 km from city center
```

[61]:
```
avgFurtherStationsUsedSlots = furtherStationsUsedSlotsSum/furtherStations.
↪count()
```

[62]:
```
print(avgFurtherStationsUsedSlots)
```

7.869283643246105

[26]:
```
# The result shows that the number of stations further than 1.5 km from city␣
↪center is approximately 3 times more
# than those which are closer than 1.5 km from city center. Also, 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 7.87 for further
# ones that shows there are a little more free_slots for further stations.
```