lab3_rdd

August 28, 2020

[1]:
```
from datetime import datetime
register_input_path = "/data/students/bigdata_internet/lab3/register.csv"
stations_input_path = "/data/students/bigdata_internet/lab3/stations.csv"
register_w_header_rdd = sc.textFile(register_input_path)
stations_w_header_rdd = sc.textFile(stations_input_path)
outputPath = "lab3RDD_2"
```

[2]:
```
# Removing the header from the rdd created by reading the csv file
register_w_erroneous_lines_rdd = register_w_header_rdd.filter(lambda line: line␣
 ↪!= "station\ttimestamp\tused_slots\tfree_slots")
```

[3]:
```
# We can see the format of the input rdd now after removing the header line
print(register_w_erroneous_lines_rdd.first())
print('Number of reading before erroneous␣
 ↪removal',register_w_erroneous_lines_rdd.count())
```

```
1       2008-05-15 12:01:00     0       18
Number of reading before erroneous removal 25319028
```

[4]:
```
# need to remove erroneous lines with used == 0 and free == 0
def checkIfErroneousLine(line):
    station_id,timestamp,used,free = line.split("\t")
    if int(used) == 0 and int(free) == 0:
        return False
    else:
        return True
register_rdd = register_w_erroneous_lines_rdd.filter(checkIfErroneousLine)
print(register_rdd.first())
print('Number of reading after removal of erroneous ones',register_rdd.count())
```

```
1       2008-05-15 12:01:00     0       18
Number of reading after removal of erroneous ones 25104121
```

[5]:
```
# function that takes as input the register_rdd
# 1 2008-05-15 12:01:00 0 18 (the values are seperated by tab)
# splits the input line into station_id, timestamp (ymd,hms) , used_slots and␣
 ↪free_slots
# it then cheks if the free slots are zero which is the critical condition
# if so it assigns a 1 value, else it assignes zero
def createkv(line):
    station_id,timestamp,used,free = line.split("\t")
    # create a datetime object in order to first parse the input string␣
 ↪(strptime string parse time)
    dtobj = datetime.strptime(timestamp, "%Y-%m-%d %H:%M:%S")
    # using the datetime object and formatting it with strftime (string format␣
 ↪time)
    weekday = dtobj.strftime('%A')
    hour = dtobj.strftime("%H")
    timeslot = (weekday,hour)
    critical = 0
    if int(free) == 0:
        critical = 1
    else :
        critical = 0
    return ( (station_id,timeslot) , (critical,1) )
# in this way i can count the total occurences by giving as values the␣
 ↪criticality 1/0 and the instances aka 1 each
```

[6]:

```
# rdd with station_id,time slot as keys and (1/0, 1) as the values that␣
 ↪represent if critical or not and the 1 to count the occurances
station_timeslot_values_rdd = register_rdd.map(createkv)
print(station_timeslot_values_rdd.top(2))
```

```
[(('99', ('Wednesday', '23')), (0, 1)), (('99', ('Wednesday', '23')), (0, 1))]
```

[7]:

```
# function to calculate total number of critical readings and total number of␣
 ↪readings for a Si,Tj key pair
def functt(v1,v2):
    crit1 = v1[0]
    count1 = v1[1]
    crit2 = v2[0]
    count2 = v2[1]
    return (crit1+crit2,count1+count2)
```

[11]:

```
# Using reduceByKey in order to calculate for each unique key pair of Si,Tj
# the values necessary for the criticality, which are the total numer of␣
 ↪critical readings
# and the total readings for that pair
criticality_instances_rdd = station_timeslot_values_rdd.reduceByKey(functt)
print(criticality_instances_rdd.top(2))
```

```
[(('99', ('Wednesday', '23')), (0, 568)), (('99', ('Wednesday', '22')), (15,
568))]
```

[12]:

```
# Using mapValues in order to apply a function on the values for each line of␣
 ↪the input rdd
# since we have reduced by key every line will have a unique key
# the lambda function calculates the fraction of critical values over all␣
 ↪readings (for a certain key Si,Tj)
criticality_value_rdd = criticality_instances_rdd.mapValues(lambda␣
 ↪critical_and_total: critical_and_total[0]/critical_and_total[1])
print(criticality_value_rdd.top(2))
```

```
[(('99', ('Wednesday', '23')), 0.0), (('99', ('Wednesday', '22')),
0.026408450704225352)]
```

[13]:

```
# Select only the pair that have criticality over a certain threshold defined␣
 ↪by user
THRESHOLD = 0.6
# Function to pass to the filter transformation in order to
# select only those lines that satisfy the threshold contraint on criticality
def filterForThreshold(line,THRESHOLD):
    # (('99', ('Wednesday', '23')), 0.0017574692442882249)
    criticality_val = line[1]
    if criticality_val>= THRESHOLD:
        return True
    else:
        return False
# Filtering
filtered_criticality_rdd = criticality_value_rdd.filter(lambda line:␣
 ↪filterForThreshold(line,THRESHOLD))
print("Number of elements that satisfy the filter␣
 ↪condition",filtered_criticality_rdd.count())
# Examples of the filtered RDD
print(filtered_criticality_rdd.take(2))
```

```
Number of elements that satisfy the filter condition 5
[(('9', ('Friday', '22')), 0.6258389261744967), (('58', ('Monday', '00')),
0.6323119777158774)]
```

[22]:

```
def mapForSorting(line):
    criticality = line[1]
    return␣
 ↪((criticality,int(line[0][0]),line[0][1][0],int(line[0][1][1])),1)#criticality,sId,day,hour
 ↪
newKeysValue = filtered_criticality_rdd.map(mapForSorting)
sortedRdd = newKeysValue.sortByKey(False)
sortedRdd.take(5)
```

[22]:

```
[((0.6323119777158774, 58, 'Monday', 0), 1),
 ((0.6258389261744967, 9, 'Friday', 22), 1),
 ((0.6239554317548747, 58, 'Monday', 1), 1),
 ((0.622107969151671, 10, 'Saturday', 0), 1),
 ((0.6129032258064516, 9, 'Friday', 10), 1)]
```

[23]:

```
def orderForDF(line):
    return line[0][1],line[0][2],line[0][3],line[0][0]␣
 ↪#station|weekday|hour|criticality|
reordered_rdd = sortedRdd.map(orderForDF)
print(reordered_rdd.take(2))
```

```
[(58, 'Monday', 0, 0.6323119777158774), (9, 'Friday', 22, 0.6258389261744967)]
```

[27]:

```
df = spark.
 ↪createDataFrame(reordered_rdd,["station","weekday","hour","criticality"])
```

```
+-------+-------+----+------------------+
|station|weekday|hour|       criticality|
+-------+-------+----+------------------+
|     58| Monday|   0|0.6323119777158774|
+-------+-------+----+------------------+
only showing top 1 row
```

[28]:

```
stations_df = spark.read.load(stations_input_path,
                     format="csv", header=True, inferSchema=True, sep="\t")
sorted_criticality_w_coordinates_df = df.join(stations_df,df.station ==␣
 ↪stations_df.id, "inner")
final_df = sorted_criticality_w_coordinates_df.select("station", ␣
 ↪"weekday","hour", "criticality", "longitude" ,"latitude")
final_df.show(2)
```

```
+-------+-------+----+------------------+---------+---------+
|station|weekday|hour|       criticality|longitude| latitude|
+-------+-------+----+------------------+---------+---------+
|     58| Monday|   0|0.6323119777158774| 2.170736|41.377536|
|      9| Friday|  22|0.6258389261744967| 2.185294|41.385006|
+-------+-------+----+------------------+---------+---------+
only showing top 2 rows
```

[43]:

```
final_df.write.csv(outputPath, header=True,sep = "\t")
```

[30]:

```
####bonus#####
stations_noheader = stations_w_header_rdd.filter(lambda line:line!
 ↪='id\tlongitude\tlatitude\tname')
```

[32]:

```
station_usedOne=register_rdd.map(lambda line:(line.split("\t")[0], (int(line.
 ↪split("\t")[2]),1)))
```

[33]:

```
def sumUsedOne(v1,v2):
    u1 = v1[0]
    i1 = v1[1]
    u2 = v2[0]
    i2 = v2[1]
    return (u1+u2,i1+i2)
station_totUsedTotInst = station_usedOne.reduceByKey(sumUsedOne)
station_usedAvg = station_totUsedTotInst.mapValues(lambda val: val[0]/val[1])
```

[34]:

```
from math import radians, cos, sin, asin, sqrt
def haversine(line):
    lat1 = 41.386904
    lon1 = 2.169989
    stationid,lon,lat,name = line.split("\t")
    lon2=float(lon)
    lat2=float(lat)
    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    # haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371
    d =  c * r
    return (stationid,d)
```

[35]:

```
station_distToCentre = stations_noheader.map(haversine)
```

[36]:

```
Lessthan1km = station_distToCentre.filter(lambda line: line[1]<1)
Morethan1km = station_distToCentre.filter(lambda line: line[1]>=1)
```

[37]:

```
station_usedAvg_distLessThanOne=station_usedAvg.join(Lessthan1km)
station_usedAvg_distMoreThanOne=station_usedAvg.join(Morethan1km)
```

[38]:

```
avgUsed_distMoreThanOne = station_usedAvg_distMoreThanOne.map(lambda line:␣
 ↪('more',line[1][0]))
avgUsed_distLessThanOne = station_usedAvg_distLessThanOne.map(lambda line:␣
 ↪('more',line[1][0]))
```

[39]:

```
total_avgUsed_distMoreThanOne = avgUsed_distMoreThanOne.reduceByKey(lambda␣
 ↪acc,n:acc+n)
total_avgUsed_distLessThanOne = avgUsed_distLessThanOne.reduceByKey(lambda␣
 ↪acc,n:acc+n)
```

```
[40]: total_avgUsed_distMoreThanOne.first()[1]/station_usedAvg_distMoreThanOne.
       ↪count() # U1
```

```
[40]: 7.905908792496716
```

```
[41]: total_avgUsed_distLessThanOne.first()[1]/station_usedAvg_distLessThanOne.
       ↪count() # U2 further away stations are more used
```

```
[41]: 8.223426232628029
```

lab3_df

August 28, 2020

[1]:

```
from datetime import datetime
register_input_path = "/data/students/bigdata_internet/lab3/register.csv"
stations_input_path = "/data/students/bigdata_internet/lab3/stations.csv"
outputPath = "lab3DF"
```

[2]:

```
# Create a DataFrame from register.csv
register_df = spark.read.load(register_input_path,
                          format="csv", header=True, inferSchema=True, sep="\t")
```

[3]:

```
# Create a DataFrame from stations.csv
stations_df = spark.read.load(stations_input_path,
                          format="csv", header=True, inferSchema=True, sep="\t")
```

[ ]:

```
# total number of lines in the register data frame
register_df.count()
```

[10]:

```
# I need to remove the lines of the register_df that have errors
# removing the erroneous lines where both used and free slots are zero
filtered_register_df = register_df.filter("not(used_slots=0 and free_slots=0)")
```

[11]:

```
filtered_register_df.count()
```

[11]:

```
25104121
```

[12]:

```
def create_timeslot(timestamp):
    # '2008-05-15 12:20:00' is the timestamp format and is a timestamp object␣
 ↪of class datetime
    weekday = (timestamp.strftime('%A'))
    hour = (timestamp.strftime("%H"))
    timeslot = str(weekday) +' '+ str(hour)
    return timeslot
```

[42]:

```
spark.udf.register("toWeekday", lambda timestamp: (timestamp.strftime('%A')) )
```

[43]:

```
spark.udf.register("toHour", lambda timestamp: timestamp.strftime('%H'))
```

[44]:

```
spark.udf.register("toTimeslot", lambda timestamp: create_timeslot(timestamp))
```

[16]:

```
register_w_weekday_hour_df = filtered_register_df.
 ↪selectExpr("station","toWeekday(timestamp) as weekday","toHour(timestamp) as␣
 ↪hour", "used_slots","free_slots" )
```

[17]:

```
timestamp_df = filtered_register_df.selectExpr("station","toTimeslot(timestamp)␣
 ↪as timeslot", "used_slots","free_slots" )
```

[18]:

```
def checkcriticality(free_slots):
    critical = 0
    if free_slots == 0: # do not have to cast to int since df infer schema␣
 ↪already makes it integer
        critical = 1
    else :
        critical = 0
    return int(critical)
```

[19]:

```
def countreading(station):
    return int(1)
```

[45]:

```
spark.udf.register("CountReading", lambda station: countreading(station)␣
 ↪,"integer")
```

[46]:

```
spark.udf.register("isCritical", lambda free_slots:␣
 ↪checkcriticality(free_slots) ,"integer")
```

[22]:

```
criticality_df = timestamp_df.
 ↪selectExpr("station","timeslot","isCritical(free_slots) as␣
 ↪critical","CountReading(station) as reading")
```

[23]:

```
statio_timeslot_criticaltotal_df = criticality_df.groupBy("station","timeslot").
 ↪sum("critical","reading").
 ↪withColumnRenamed("sum(critical)","total_critical")\
                                                                              ␣
 ↪                                                  .
 ↪withColumnRenamed("sum(reading)","total_readings")
```

[24]:

```
statio_timeslot_criticaltotal_df.show(2)
```

```
+-------+-----------+--------------+--------------+
|station|   timeslot|total_critical|total_readings|
+-------+-----------+--------------+--------------+
|    180|  Friday 18|             0|           597|
|    180|Thursday 08|             3|           531|
+-------+-----------+--------------+--------------+
only showing top 2 rows
```

[25]:

```
def calculateCriticality(total_critical,total_readings):
    criticality = float(total_critical/total_readings)
    return criticality
```

[47]:

```
spark.udf.register("CalculateCriticality", lambda total_critical,total_readings:
 ↪ calculateCriticality(total_critical,total_readings) ,"float")
```

[27]:

```
station_timeslot_criticality_df = statio_timeslot_criticaltotal_df.
 ↪selectExpr("station","timeslot","CalculateCriticality(total_critical,total_readings)␣
 ↪as criticality")
```

[28]:

```
#station_timeslot_criticality_df.show(5)
```

[29]:

```
# Defining a threshold to filter the values with
THRESHOLD = 0.6
```

[30]:

```
filtered_criticality_df = station_timeslot_criticality_df.
 ↪filter("criticality>{}".format(THRESHOLD))
```

[31]:

```
filtered_criticality_df.show(2)
```

```
+--------+-----------+-----------+
|station|   timeslot|criticality|
+--------+-----------+-----------+
|      9|  Friday 10| 0.61290324|
|     10|Saturday 00|   0.622108|
+--------+-----------+-----------+
only showing top 2 rows
```

[32]:

```
filtered_criticality_df.count()
```

[32]: 5

[33]:

```
sorted_criticality_df = filtered_criticality_df.sort("criticality",ascending =␣
 ↪False)
```

[34]:

```
sorted_criticality_df.show(5)
```

```
+--------+-----------+-----------+
|station|   timeslot|criticality|
+--------+-----------+-----------+
|     58|  Monday 00|   0.632312|
|      9|  Friday 22| 0.62583894|
|     58|  Monday 01|  0.6239554|
|     10|Saturday 00|   0.622108|
|      9|  Friday 10| 0.61290324|
+--------+-----------+-----------+
```

```
[35]: #  ta.join(tb, ta.name == tb.name, 'inner').
      sorted_criticality_w_coordinates_df = sorted_criticality_df.
       ↪join(stations_df,sorted_criticality_df.station == stations_df.id, "inner")
```

```
[41]: #sorted_criticality_w_coordinates_df.show(2)
```

```
[37]: final_df = sorted_criticality_w_coordinates_df.select("station",  "timeslot",␣
       ↪"criticality",  "longitude" ,"latitude")
```

```
[38]: final_df.show(5)
```

```
+--------+-----------+-----------+---------+---------+
| station|   timeslot|criticality|longitude| latitude|
+--------+-----------+-----------+---------+---------+
|      58|  Monday 00|   0.632312| 2.170736|41.377536|
|       9|  Friday 22| 0.62583894| 2.185294|41.385006|
|      58|  Monday 01|  0.6239554| 2.170736|41.377536|
|      10|Saturday 00|   0.622108| 2.185206|41.384875|
|       9|  Friday 10| 0.61290324| 2.185294|41.385006|
+--------+-----------+-----------+---------+---------+
```

```
[40]: final_df.write.csv(outputPath, header=True,sep = "\t")
```