group3_vspark

January 20, 2023

1 Group project : PySpark

Bilal Kostet, Antoine Somerhausen, Pierre Hosselet, Pacome Van Overschelde, Romain Vandepopeliere - Group 3

1.1 Index

1.2 Starting PySpark

[1]:
```
#This is needed to start a Spark session from the notebook
#You may adjust the memory used by the driver program based on your machine's␣
↪settings
import os
os.environ['PYSPARK_SUBMIT_ARGS'] ="--conf spark.driver.memory=2g ␣
↪pyspark-shell"

from pyspark.sql import SparkSession
```

[2]:
```
# --------------------------------
# Start Spark in LOCAL mode
# --------------------------------

#The following lines are just there to allow this cell to be re-executed␣
↪multiple times:
#if a spark session was already started, we stop it before starting a new one
#(there can be only one spark context per jupyter notebook)
try:
    spark
    print("Spark application already started. Terminating existing application␣
↪and starting new one")
```

```
    spark.stop()
except:
    pass

# Create a new spark session (note, the * indicates to use all available CPU␣
 ↪cores)
spark = SparkSession \
    .builder \
    .master("local[*]") \
    .appName("demoRDD") \
    .getOrCreate()

#When dealing with RDDs, we work the sparkContext object. See https://spark.
 ↪apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkContext
sc=spark.sparkContext
```

```
23/01/19 21:45:32 WARN Utils: Your hostname, romain-XPS-13-9300 resolves to a
loopback address: 127.0.1.1; using 192.168.0.20 instead (on interface wlp0s20f3)
23/01/19 21:45:32 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
address
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
```

```
23/01/19 21:45:32 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform… using builtin-java classes where applicable
```

[3]:
```
# check that we have a working spark context, print its configuration
sc._conf.getAll()
```

[3]:
```
[('spark.driver.extraJavaOptions',
  '-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED
--add-opens=java.base/java.lang.invoke=ALL-UNNAMED --add-
opens=java.base/java.lang.reflect=ALL-UNNAMED --add-opens=java.base/java.io=ALL-
UNNAMED --add-opens=java.base/java.net=ALL-UNNAMED --add-
opens=java.base/java.nio=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED
--add-opens=java.base/java.util.concurrent=ALL-UNNAMED --add-
opens=java.base/java.util.concurrent.atomic=ALL-UNNAMED --add-
opens=java.base/sun.nio.ch=ALL-UNNAMED --add-opens=java.base/sun.nio.cs=ALL-
UNNAMED --add-opens=java.base/sun.security.action=ALL-UNNAMED --add-
opens=java.base/sun.util.calendar=ALL-UNNAMED --add-
opens=java.security.jgss/sun.security.krb5=ALL-UNNAMED'),
 ('spark.app.startTime', '1674161132836'),
 ('spark.executor.id', 'driver'),
 ('spark.driver.host', '192.168.0.20'),
 ('spark.rdd.compress', 'True'),
 ('spark.executor.extraJavaOptions',
  '-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED
--add-opens=java.base/java.lang.invoke=ALL-UNNAMED --add-
opens=java.base/java.lang.reflect=ALL-UNNAMED --add-opens=java.base/java.io=ALL-
UNNAMED --add-opens=java.base/java.net=ALL-UNNAMED --add-
opens=java.base/java.nio=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED
--add-opens=java.base/java.util.concurrent=ALL-UNNAMED --add-
opens=java.base/java.util.concurrent.atomic=ALL-UNNAMED --add-
opens=java.base/sun.nio.ch=ALL-UNNAMED --add-opens=java.base/sun.nio.cs=ALL-
UNNAMED --add-opens=java.base/sun.security.action=ALL-UNNAMED --add-
opens=java.base/sun.util.calendar=ALL-UNNAMED --add-
opens=java.security.jgss/sun.security.krb5=ALL-UNNAMED'),
 ('spark.app.id', 'local-1674161134539'),
 ('spark.driver.memory', '2g'),
 ('spark.serializer.objectStreamReset', '100'),
 ('spark.master', 'local[*]'),
 ('spark.driver.port', '45325'),
 ('spark.submit.pyFiles', ''),
 ('spark.submit.deployMode', 'client'),
 ('spark.app.submitTime', '1674161132628'),
 ('spark.ui.showConsoleProgress', 'true'),
 ('spark.app.name', 'demoRDD')]
```

back to Index

1.3 Loading and preparing data

Let us import the packages and the functions that we'll need throughout this work

[4]:
```
import json
import time
import geopy.distance
import pandas as pd
import shapely.geometry as sg
from datetime import datetime
from pyspark.sql.functions import udf, col, lead, mean , sum
from pyspark.sql.types import IntegerType, FloatType
from pyspark.sql.window import Window
```

Spark can read .csv files and put them in a dataframe

[5]:
```
drivers =  spark.read.csv('drivers.csv', header=True, inferSchema=True)  ␣
 ↪#inferSchema = True, in order not to have strings only in the dataframe
```

[6]:
```
drivers.show()
drivers.printSchema()
```

```
+--------+-------------------+----------+----------+
|  driver|          timestamp|  latitude| longitude|
+--------+-------------------+----------+----------+
|c473205b|2017-08-31 14:24:25|-12.106778|-76.998078|
|a0f3b4e1|2017-08-31 14:24:26|-12.103913|-76.963727|
|1236f9fe|2017-08-31 14:24:26|-12.133777|-77.004266|
|ae4a06a2|2017-08-31 14:24:26|-12.085963|-76.987582|
|ab7a6c63|2017-08-31 14:24:26|-12.072973|-77.061448|
|5ee73484|2017-08-31 14:24:26|-12.067694|-77.068442|
|4fff90cb|2017-08-31 14:24:26|-12.144308|-76.989234|
|d892d208|2017-08-31 14:24:26|-16.401221|-71.499513|
|e9f90dfb|2017-08-31 14:24:10|-12.063665|-76.963254|
|c1719f8d|2017-08-31 14:24:10|-12.070187|-76.994167|
|4c299505|2017-08-31 14:24:11| -11.96459|-77.015983|
|49f033bd|2017-08-31 14:24:11|  -12.0906|-77.069808|
|749df32a|2017-08-31 14:24:11|-12.109964| -76.97535|
|f8d5c453|2017-08-31 14:24:11|-12.080683|-77.036188|
|4314c58a|2017-08-31 14:24:11|-16.404571|-71.519775|
|914f7296|2017-08-31 14:24:11|-12.218541|-76.928484|
|ee744228|2017-08-31 14:24:30|-12.120202|-77.035704|
|353c20b9|2017-08-31 14:24:31|-12.119177|-76.997462|
|483e84f9|2017-08-31 14:24:31|-12.109511|-76.995862|
|e26ff40f|2017-08-31 14:24:31|-12.130376|-77.017779|
+--------+-------------------+----------+----------+
only showing top 20 rows

root
 |-- driver: string (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- latitude: double (nullable = true)
 |-- longitude: double (nullable = true)
```

We also have to create the zones dataframe, based on the JSON file. This file is really small, and will be used in one function only. We assume that reading via PySpark is not necessary here.

[7]:
```python
with open('zones.json') as f:
    d = json.load(f)

zones = pd.json_normalize(d['zones'])
```

We will need to apply a defined function to the driver dataframe. Let us define the following function, that returns the zone in which a coordinate point is :

[8]:
```python
def find_zone(latitude,longitude):

    #create a shapely Point object from the driver's coordinates
    point = sg.Point(latitude,longitude)
    #checking if the point is in a given zone
    for i in range(len(zones)):
        polygon= sg.Polygon( (x['lat'],x['lng']) for x in zones['polygon'].
 ↪iloc[i])
        if polygon.contains(point):
            return int(zones['id_zone'].iloc[i])

    #if no zone contains the point
    return None
```

In order to be used in a PySpark dataframe, this function has to be converted into a User Defined Function (= udf) following :

[9]:
```
find_zone_udf = udf(find_zone, IntegerType())  #The output is an integer
```

It is now time to simply generate the new column 'zone'. We will drop the null values since the points that do not belong to the zone dataset are not of interest for us

[10]:
```
driversz = drivers.withColumn('zone', find_zone_udf(col('latitude'),␣
 ↪col('longitude'))).na.drop()
```

In practice, we will save this dataframe in a .csv file in order to use it as a new starting point. Indeed, in PySpark, operations that do wide transformations (such as groupBy() and join(), that we will mainly use in this work) are really expensive to use on a dataframe on which some function that is not part of the pyspark.sql library has been applied (such as our udf find_zone). Note that there is no way to generate the column zone without using a custom function, or in other words, there is no way to generate this column using functions of pyspark.sql only. Since we will use the resulting dataframe a lot, this will save us precious time, even if the csv writing takes about 30min. For instance, the Question 1 takes 35 min to run if it must compute the zones and then groupby, whereas it takes 12s to groupby a brand new csv file with the additional column zone.

[11]:
```
#driversz.write.option("header", True).csv('drivers_zones')
```

[12]:
```
driversZ =  spark.read.csv('drivers_zone.csv', header=True, inferSchema=True)\
                 .select(['driver','timestamp','latitude','longitude','zone'])\
                 .withColumn('zone', col('zone').cast(IntegerType())).na.drop()
```

[13]:
```
driversZ.show()
driversZ.printSchema()
```

```
+--------+-------------------+----------+----------+----+
|  driver|          timestamp|  latitude| longitude|zone|
+--------+-------------------+----------+----------+----+
|c473205b|2017-08-31 14:24:25|-12.106778|-76.998078|  21|
|a0f3b4e1|2017-08-31 14:24:26|-12.103913|-76.963727|  21|
|1236f9fe|2017-08-31 14:24:26|-12.133777|-77.004266|  14|
|ae4a06a2|2017-08-31 14:24:26|-12.085963|-76.987582|  21|
|ab7a6c63|2017-08-31 14:24:26|-12.072973|-77.061448|  22|
|5ee73484|2017-08-31 14:24:26|-12.067694|-77.068442|  22|
|4fff90cb|2017-08-31 14:24:26|-12.144308|-76.989234|  14|
|e9f90dfb|2017-08-31 14:24:10|-12.063665|-76.963254|  21|
|c1719f8d|2017-08-31 14:24:10|-12.070187|-76.994167|  21|
|4c299505|2017-08-31 14:24:11| -11.96459|-77.015983|  25|
|49f033bd|2017-08-31 14:24:11|  -12.0906|-77.069808|  22|
|749df32a|2017-08-31 14:24:11|-12.109964| -76.97535|  21|
|f8d5c453|2017-08-31 14:24:11|-12.080683|-77.036188|  22|
|914f7296|2017-08-31 14:24:11|-12.218541|-76.928484|  15|
|ee744228|2017-08-31 14:24:30|-12.120202|-77.035704|  22|
|353c20b9|2017-08-31 14:24:31|-12.119177|-76.997462|  21|
|483e84f9|2017-08-31 14:24:31|-12.109511|-76.995862|  21|
|e26ff40f|2017-08-31 14:24:31|-12.130376|-77.017779|  13|
|3697a724|2017-08-31 14:24:31|-12.097054|-77.032223|  22|
|45a9882d|2017-08-31 14:24:31|-12.016618|-77.004344|  20|
+--------+-------------------+----------+----------+----+
only showing top 20 rows

root
 |-- driver: string (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- latitude: double (nullable = true)
 |-- longitude: double (nullable = true)
 |-- zone: integer (nullable = true)
```

back to Index

1.4 Question 1

The goal of this question is to know what are the ten zones that are visited by the most drivers. The strategy is simply to exhibit the different pairs (driver, zone) that are in the dataframe and afterwards to group them by zone while counting how many unique drivers have been in each zone. It's a good idea to .cache() the result, since we will use it again later on in Question 2.

[14]:
```
driversByZone = driversZ.dropDuplicates(['driver','zone']).groupBy('zone').
 ↪count().cache()
```

[15]:
```
%%time
driversByZone.orderBy('count', ascending=False).show(10)
```

```
[Stage 8:==================================================>      (175 + 9) / 200]

+----+-----+
|zone|count|
+----+-----+
|  22|10823|
|  21| 8039|
|  14| 4150|
|  25| 3638|
|  13| 3178|
|  20| 1591|
|  23| 1160|
|  24| 1064|
|  26|  747|
|  15|  512|
+----+-----+
only showing top 10 rows

CPU times: user 20.8 ms, sys: 4.49 ms, total: 25.3 ms
Wall time: 11.8 s
```

back to Index

1.5 Question 2

In this question, we want to adress some other features of the dataset.

Firstly, we would like to know what are the average speed, the average travel time and the average travel distance for the drivers in a given zone. For this purpose, we will need to compute some differences between data of distinct rows. In PySpark, there is no notion of index for a row. However, a pyspark.sql function such as lead (with offset $= j$) allows us to access the would-be $(n+j)^{\text{th}}$ row when computing the would-be n^{th} row, accounting that the dataset is ordered and partitioned. All in all, this can be done by defining a Window and by specifying how to partition and to order it. In this way, Spark can have some notion of index and position.

Secondly, what about the exchanges between the zones ?

1.5.1 2.1 Dynamics

Let us define the functions that we will need :

[16]:
```python
#Distance function
def get_distance(lat1,long1,lat2,long2):
    return geopy.distance.geodesic((lat1,long1),(lat2,long2)).m

#Conversion to udf
get_distance_udf = udf(get_distance, FloatType())


#Time function
def get_time(t1,t2):
    if (t1 == None) or (t2 == None) :
        return None
    else:
        return (t2-t1).seconds

#Conversion to udf
get_time_udf = udf(get_time, IntegerType())
```

In order the compute the "instantaneous" speeds, we have to compute the distance and the time gap between two successive GPS datapoints of the same driver. Hence, the natural way to specify the Window in which the lead function will run is the following :

```
[17]: #Spec of the Window, in order to run the lead function
windowSpec = Window.partitionBy('driver').orderBy('timestamp')
```

In this way, the function lead will see each driver subset as a finite area beyond which it cannot pass. In particular, it will return a null value for the last row of a driver subset, where the ordering by timestamp ensures that the last row correponds to the latest GPS recording of the driver. It is again a good idea to get rid of the null values that can occur in several ways : either it's the last row of a driver hence there is no next quantites, or the speed is a null value due to a vanishing Δt. The latter comes from some bugs of the GPS recordings (as we will explain just below) : indeed, it is unphysical to have 2 datapoints that share the same timestamp but that are 100m apart.

```
[18]: dynamics = driversZ.withColumn('next_lat', lead('latitude', offset=1).
  ↪over(windowSpec)) \
    .withColumn('next_lng', lead('longitude', offset=1).over(windowSpec)) \
    .withColumn('next_time', lead('timestamp', offset=1).over(windowSpec)) \
    .withColumn('deltaT', get_time_udf('timestamp', 'next_time') ) \
    .withColumn('deltaX', get_distance_udf('latitude',␣
  ↪'longitude','next_lat','next_lng') ) \
    .withColumn('speed', col('deltaX')/col('deltaT')).na.drop()
```

By inspecting the results, one realizes that some GPS recordings are outliers. Indeed, an .orderBy('speed') in decreasing order tells us that some instantaneous speeds are around 1500 m/s, and a non-negligible number of other speeds are impossible to reach by car. This feature is not a computation mistake nor a code mistake, it really belongs to the dataset : these are just bugs of the GPS. In order to obtain some average speeds which are realistic, we put a cut-off and we will throw away all the datapoints that have a speed bigger than 200 km/h. Working in m/s units, this upper bound is chosen to be 56 m/s.

Also, we will also choose a cut-off for the Δt. Sometimes there is a huge time gap ($\sim$ hours) between two consecutive timestamps for the same driver. We assume this is not due to his travel and that the speed computed during this big time gap is not a speed of travel : the driver simply stopped the recording and restarted it somewhere else, hours later. Hence, one should get rid of this step. We choose 20min (=1200s) as an upper bound for the difference between two consecutive timestamps, in order to be sure that they are kinematically meaningful.

Note that we chose to apply this filtering in this dynamical section only, the other sections are processed with the whole dataset (except that we dropped out the datapoints that are not in the zones).

```
[19]: dynamicsFinal = dynamics.filter((dynamics['speed'] <56.0) & (dynamics['deltaT']␣
      ↪< 1200))
```

```
[20]: %%time
      dynamicsFinal.show()
```

```
[Stage 11:>                                                         (0 + 1) / 1]

+--------+-------------------+----------+----------+----+----------+----------+-
------------------+------+----------+--------------------+
|  driver|          timestamp|  latitude| longitude|zone|  next_lat|  next_lng|
next_time|deltaT|    deltaX|                speed|
+--------+-------------------+----------+----------+----+----------+----------+-
------------------+------+----------+--------------------+
|001b6172|2017-08-31 17:32:06|-12.004618| -77.06468|
25|-12.004617|-77.064681|2017-08-31 17:32:33|    27|0.15523106|
0.005749298466576471|
|001b6172|2017-08-31 17:32:33|-12.004617|-77.064681|
25|-12.004626|-77.064683|2017-08-31 17:34:19|   106|
1.0191461|0.009614585705523222|
|001b6172|2017-08-31 17:34:19|-12.004626|-77.064683|
25|-12.004626|-77.064683|2017-08-31 17:34:25|     6|       0.0|
0.0|
|001b6172|2017-08-31 17:34:25|-12.004626|-77.064683|  25|
-12.00542|-77.068807|2017-08-31 17:35:50|    85| 457.61472|   5.383702536190257|
|001b6172|2017-08-31 17:35:50| -12.00542|-77.068807|
25|-12.005483|-77.069245|2017-08-31 17:36:02|    12|  48.20485|
4.017070770263672|
|001b6172|2017-08-31 17:36:02|-12.005483|-77.069245|
25|-12.005496|-77.069302|2017-08-31 17:36:08|     6|  6.371734|
1.0619556903839111|
|001b6172|2017-08-31 17:36:08|-12.005496|-77.069302|  25|
-12.00562|-77.069628|2017-08-31 17:36:20|    12| 38.059425|  3.1716187795003257|
|001b6172|2017-08-31 17:36:20| -12.00562|-77.069628|
25|-12.006175|-77.069554|2017-08-31 17:36:32|    12| 61.922016|
5.160168011983235|
|001b6172|2017-08-31 17:36:32|-12.006175|-77.069554|
25|-12.006576|-77.069476|2017-08-31 17:36:38|     6| 45.165497|
7.5275828043619795|
|001b6172|2017-08-31 17:36:38|-12.006576|-77.069476|
25|-12.008546|-77.069153|2017-08-31 17:36:56|    18| 220.74648|
12.263693067762587|
|001b6172|2017-08-31 17:36:56|-12.008546|-77.069153|
25|-12.009867|-77.068935|2017-08-31 17:37:08|    12| 148.04793|
12.337327321370443|
|001b6172|2017-08-31 17:37:08|-12.009867|-77.068935|
25|-12.010338|-77.069187|2017-08-31 17:37:18|    10| 58.888237|
5.888823699951172|
|001b6172|2017-08-31 17:37:18|-12.010338|-77.069187|
25|-12.009556|-77.070849|2017-08-31 17:37:42|    24| 200.60068|
8.3583615620931|
|001b6172|2017-08-31 17:37:42|-12.009556|-77.070849|
25|-12.007902|-77.071388|2017-08-31 17:38:06|    24| 192.15382|
8.006409327189127|
|001b6172|2017-08-31 17:38:06|-12.007902|-77.071388|  25|
-12.00828|-77.071834|2017-08-31 17:38:24|    18|  64.08959|   3.560532887776693|
|001b6172|2017-08-31 17:38:24| -12.00828|-77.071834|  25|
-12.00726|-77.072739|2017-08-31 17:39:12|    48| 149.81522|   3.121150334676107|
|001b6172|2017-08-31 17:39:12| -12.00726|-77.072739|  25|
-12.00726|-77.072739|2017-08-31 17:39:24|    12|       0.0|                 0.0|
|001b6172|2017-08-31 17:39:24| -12.00726|-77.072739|
25|-12.007261|-77.072739|2017-08-31 17:39:36|
12|0.11062235|0.009218528866767883|
|001b6172|2017-08-31 17:39:36|-12.007261|-77.072739|
25|-12.006872|-77.072967|2017-08-31 17:40:06|    30| 49.681465|
1.656048838297526|
|001b6172|2017-08-31 17:40:06|-12.006872|-77.072967|
25|-12.006414|-77.073051|2017-08-31 17:40:19|    13|  51.48421|
3.960323920616737|
+--------+-------------------+----------+----------+----+----------+----------+-
------------------+------+----------+--------------------+
only showing top 20 rows

CPU times: user 9.43 ms, sys: 6.63 ms, total: 16.1 ms
Wall time: 17.5 s
```

The first things that we want to compute are the *average travel time* and *average travel distance* for each zone. For each driver, one should sum all the Δt belonging to the same zones. Same for the Δx. Using a groupBy([‘driver’,‘zone’]), we will exhibits all the unique pairs (driver,zone) and we will associate the total time and distance to them with the agg function.

[21]:
```
dzoneDistTime = dynamicsFinal.groupBy(['driver','zone']).agg( sum('deltaX').
 ↪alias('dtot') , sum('deltaT').alias('ttot')).cache()
```

Now we simply have to groupBy the zones while summing all the distances and times, then to divide these by the number of drivers in each zone that we computed in Question 1. For this purpose, we join the column ‘count’ of Question 1.

[22]:
```
averageQuantities = dzoneDistTime.groupBy('zone').agg(sum('ttot').
 ↪alias("total_time"), sum('dtot').alias('total_dist')) \
    .join(driversByZone, 'zone') \
    .withColumn('avg_time/driver (min)', col('total_time')/(60*col('count'))) \
    .withColumn('avg_dist/driver (km)', col('total_dist')/(1000*col('count')))
```

```
averageQuantities = averageQuantities.select(['zone','avg_time/driver␣
↪(min)','avg_dist/driver (km)'])
```

[23]:
```
%%time
averageQuantities.orderBy('avg_time/driver (min)', ascending = False).show(30)
```

```
[Stage 17:===================================================>   (188 + 8) / 200]

+----+---------------------+--------------------+
|zone|avg_time/driver (min)|avg_dist/driver (km)|
+----+---------------------+--------------------+
|  22|    75.92330530660014|   14.95825744395522|
|  21|   52.470497988970436|  11.178773470081994|
|  14|   29.755004016064255|    6.89719232403659|
|  27|   29.624458874458874|   6.292309179868985|
|  11|   29.066666666666666|   8.745267377084742|
|  16|   28.298816029143897|   6.543657695698078|
|  25|   26.751736301997436|   6.696107607515276|
|  26|    26.23119143239625|   5.479586937049127|
|   8|              26.0625|   8.501636286875232|
|  28|    23.62638888888889|   5.159186138508842|
|  19|   22.100751366120218|   4.951896950200414|
|  38|    20.24848484848485|  3.4727615119801327|
|  23|    19.50846264367816|   4.777424736315237|
|  20|   19.442698512465956|  4.1675496402618215|
|  13|   19.344435703796936|   4.616003906233819|
|  24|   18.449733709273183|   2.922989659491128|
|  18|     18.0785641025641|   3.958002401680419|
|  37|   17.429931972789117|  4.8839770184007225|
|  15|   13.444889322916667|  3.6580007328134525|
|  30|    12.61111111111111|    3.59692679082850 6|
|  40|   12.326470588235294|   4.128118237722008|
|  31|    12.17542735042735|  1.4988695824008722|
|  17|    11.75906432748538|   2.083281552488605|
|  29|   11.335576923076923|  2.8555806200687703|
|  49|    8.684722222222222|  2.6786293389927596|
|  12|   6.7994047619047615|   2.118051070667803|
|  35|                6.595|  1.4005737886466085|
|   7|    5.833333333333333| 0.09301459789276123|
|   2|    4.294444444444444|   3.904337541739146|
+----+---------------------+--------------------+

CPU times: user 288 ms, sys: 117 ms, total: 405 ms
Wall time: 7min 42s
```

Afterwards, one can compute the *average speed by zone*. One must be careful : the timestamps are not equally distributed, so that the average speed of a given driver in a given zone is not equal to the average of the “instantaneous” speeds that we computed. Instead, it is obviously given by the total travel distance divided by the total travel time of a driver in the given zone. We already computed it and cached it in the dzoneDistTime dataframe.

[24]:
```
averageSpeeds = dzoneDistTime.withColumn('dzoneSpeed', col('dtot')*3.6/
 ↪col('ttot')) \
            .groupBy('zone').agg( mean('dzoneSpeed').alias('avg_speed (km/h)'))
```

[25]:
```
%%time
averageSpeeds.orderBy('avg_speed (km/h)', ascending = False).show(30)
```

```
[Stage 24:=============================================>         (165 + 8) / 200]

+----+------------------+
|zone|  avg_speed (km/h)|
+----+------------------+
|   2| 70.19798352784623|
|  11| 59.53952607743003|
|   8|36.896887496332376|
|  30|26.484407724141303|
|  12| 25.27155985417999|
|  15| 25.17200485499606|
|  27|24.636710630752123|
|  29| 22.82129088318582|
|  20| 22.10889838694594|
|  16| 20.99577766424281|
|  40|20.345577486434095|
|  28|20.187964746304303|
|  37| 19.74989393166856|
|  17|18.008349792442996|
|  25|17.698093068482624|
|  49| 17.51972791634245|
|  19|17.134066348741296|
|  13|16.805070382999883|
|  18|16.777034642167383|
|  24|16.686885571654198|
|  14|15.711401011118559|
|  26| 15.65981682608462|
|  23|15.566055160976825|
|  21|14.067287275992708|
|  38|13.140226659040211|
|  22|12.429955215420618|
|  35|11.379321170849513|
|  31| 11.10770715159875|
|   7| 0.956721578325544|
+----+------------------+

CPU times: user 32.3 ms, sys: 8.08 ms, total: 40.4 ms
```

```
Wall time: 1.54 s
```

back to Index

1.5.2 2.2 Exchanges

Here we would like to have a better understanding of how the zones are connected. For instance, let us take the i^{th} zone. We could ask ourselves : in average, how many different zones are visited by the drivers which pass through zone i ? The second question that we want to adress is closely related but take the back and forth between zones into account. Now the question would be : in average, how many zone exchanges are made by the drivers which pass through zone i ?

Let us start with the amount of *visited zones* related to a given zone :

[26]:
```
pairsDriverZone = driversZ.dropDuplicates(['driver','zone']).select(['driver',␣
  ↪'zone'])
HowManyZonesByDriver = pairsDriverZone.cache().groupBy('driver').count()
```

[27]:
```
%%time
pairsDriverZone.join(HowManyZonesByDriver, 'driver') \
    .groupBy('zone').agg( mean('count').alias('NumberOfZones')).
  ↪orderBy('NumberOfZones', ascending = False).show(30)
```

```
+----+------------------+
|zone|     NumberOfZones|
+----+------------------+
|  12|              6.25|
|   8| 6.166666666666667|
|  35|               5.3|
|  49|              5.25|
|  11|               5.0|
|  40| 4.705882352941177|
|  30| 4.333333333333333|
|   2| 4.333333333333333|
|  29| 4.288461538461538|
|  16| 4.251366120218579|
|  15|        4.08203125|
|  27| 3.987012987012987|
|  28| 3.966666666666667|
|  37|3.9591836734693877|
|  17|3.9298245614035086|
|  24|3.9097744360902253|
|  19|3.7950819672131146|
|  13|3.7174323473882946|
|  23| 3.603448275862069|
|  20|3.5392834695160276|
|  14| 3.399277108433735|
|  25|3.2825728422210005|
|  18| 3.236923076923077|
|  31| 3.217948717948718|
|  21| 3.055852717999751|
|  26|2.9036144578313254|
|  22|2.8590039730204193|
|  38|2.3636363636363638|
|   7|               2.0|
+----+------------------+

CPU times: user 35 ms, sys: 5.23 ms, total: 40.2 ms
Wall time: 13.5 s
```

Now, the *exchanges* between a given zone and the other zones. Here, we have to keep track of people coming back in the same zones. We will use the lead function as for the speeds, in order to spot the rows in which there is a change of zone.

[28]:
```
driversZd = driversZ.select(['driver','timestamp','zone']).
↪withColumn('next_zone', lead('zone', offset=1).over(windowSpec)).na.drop() \
        .withColumn('diffzone', (col('next_zone') - col('zone'))/
↪(col('next_zone') - col('zone')))

HowManyZonesExchByDriver = driversZd.groupBy('driver').agg(sum('diffzone').
↪alias('SumDiffZone')) \
        .withColumn('SumDiffZone+1', col('SumDiffZone')+1)  #to count the first␣
↪zone in the process
```

[29]:
```
%%time
pairsDriverZone.join(HowManyZonesExchByDriver, 'driver') \
    .groupBy('zone').agg(mean('SumDiffZone+1').alias('NumberOfZonesExch')).
↪orderBy('NumberOfZonesExch', ascending = False).show(30)
```

```
[Stage 45:==========================>                              (4 + 5) / 9]

+----+------------------+
|zone| NumberOfZonesExch|
+----+------------------+
|  12|10.074074074074074|
|   8|               9.5|
|  11| 8.333333333333334|
|  35| 7.444444444444445|
|  30| 7.333333333333333|
|  49|7.2727272727272725|
|  15| 7.246913580246914|
|  13| 7.241500962155228|
|  16| 7.226993865030675|
|  17|7.0754716981132075|
|  24| 6.805443548387097|
|  40| 6.733333333333333|
|  14| 6.712426978226235|
|  23|6.4586330935251794|
|  29|            6.3125|
|  27|6.2615384615384615|
|  20| 6.237264480111654|
|  28| 6.083333333333333|
|  19| 5.873303167420814|
|  37| 5.738095238095238|
|  21| 5.724942975982826|
|  25|  5.67685723020483|
|  18| 5.473856209150327|
|  22|  5.45071474290591|
|  31| 5.166666666666667|
|  26| 4.949832775919733|
|   2| 4.333333333333333|
|  38|             3.375|
|   7|               2.0|
+----+------------------+

CPU times: user 16.6 ms, sys: 6.18 ms, total: 22.8 ms
Wall time: 13.9 s
```

back to Index

1.6 Cleaning up

[30]:
```
sc.stop()
```

back to Index