group3_vdask_separate

January 20, 2023

1 Group project : Dask

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

1.1 Index

1.2 Starting Dask

[ ]:
```
#We start by loading the Dask client, that allows to connect to a distributed␣
 ↪cluster.
#In our case, since we use a single machine, it is useful to visualize the␣
 ↪progress of the tasks.
from dask.distributed import Client
client = Client()
client
```

```
c:\Users\Bilal\anaconda3\lib\site-packages\distributed\node.py:179: UserWarning:
Port 8787 is already in use.
Perhaps you already have a cluster running?
Hosting the HTTP server on port 58808 instead
  warnings.warn(
```

[ ]:
```
<Client: 'tcp://127.0.0.1:58809' processes=4 threads=12, memory=15.75 GiB>
```

1.3 Loading and preparing data

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

[ ]:
```
import pandas as pd
import numpy as np
import dask
import dask.dataframe as dd
import dask.array as da
import dask.bag as db
from shapely.geometry import Point, Polygon
import json
import time
```

Dask can read .csv files and put them in a dask dataframe

[ ]:
```
%%time
gps=dd.read_csv('ProjectData/drivers.csv', dtype={'latitude' : 'float64',␣
 ↪'longitude' : 'float64', 'driver' : 'string'},parse_dates=['timestamp'])
```

```
Wall time: 11 ms
```

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. Dask advises that in these cases, it is not that useful and Pandas should be used instead. We will proceed to use dask anyway for the sake of scalability, as we may want to apply this analysis to a bigger region.

For this, we will need a custom function to convert the contents of the nested JSON file to something that can be easily interepreted into a dask dataframe.

[ ]:
```
%%time
def polyzone(dbag):
    polygonpts=[]
    for i in range(len(dbag['polygon'])):
        polygonpts.
 ↪append((float(dbag['polygon'][i]['lat']),float(dbag['polygon'][i]['lng'])))
    return {
        'id_zone' : dbag['id_zone'],
        'polygonpts' : polygonpts
    }
```

```
Wall time: 0 ns
```

We apply this custom function to a string corresponding to the contents of our JSON file. This allows to obtain a dataframe with our zones, with the id of the zone as index for the sake of optimization.

[ ]:
```
%%time
zones=db.read_text('ProjectData/zones.json').map(json.loads)
zones=zones.pluck('zones').flatten().map(polyzone)
dzones=zones.to_dataframe().set_index('id_zone',sorted=True)
```

```
Wall time: 583 ms
```

We create a list with all the Polygon structures defining the zones that will be used as an iterator

to check in which zone the data points belong.

[ ]:
```
%%time
polygonlist=[]
for i,j in dzones.iterrows():
    polygonlist.append(Polygon(j.values.tolist()[0]))
```

```
Wall time: 414 ms
```

We apply our custom function to generate the new column 'zone' by returning the zone in which a coordinate point is thanks to the fact that the zones are just indexed from 1 to 51.

[ ]:
```
%%time
def findzone(row):
    point=Point(row.latitude,row.longitude)
    for i in range(len(polygonlist)):
        polygon=Polygon(polygonlist[i])
        if polygon.contains(point):
            return i+1
    return 0

gps['zone'] = gps.apply(findzone, axis=1).astype('int32')
```

```
Wall time: 16 ms
```

```
c:\Users\Bilal\anaconda3\lib\site-packages\dask\dataframe\core.py:5523:
UserWarning:
You did not provide metadata, so Dask is running your function on a small
dataset to guess output types. It is possible that Dask will guess incorrectly.
To provide an explicit output types or to silence this message, please provide
the `meta=` keyword, as described in the map or apply function that you are
using.
  Before: .apply(func)
  After:  .apply(func, meta=(None, 'int64'))

  warnings.warn(meta_warning(meta))
```

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, dropping all the duplicates of this pair of values and then count the number of appearance of each zone identifier. We chose to not exclude points not belonging to any zone, here, to have an idea of how much GPS jitter there was, indicated by zone 0.

[ ]:
```
%%time
gps_mostvisited=gps[['driver','zone']].drop_duplicates().zone.value_counts().
 ↪nlargest(11)
```

```
Wall time: 5.98 ms
```

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 Dask, we can use the dataframe method .diff() to compute these. There are also products between values of distinct rows. This can be accessed by the .shift(period=) method on series that takes a series, and returns its shift by the amount of rows indicated in the key period.

Secondly, what about the exchanges between the zones?

1.5.1 2.1 Dynamics

Let us first clean the dataframe of the drivers belonging to invalid zones:

[ ]:
```
%%time
gpsclean=gps[gps.zone > 0]
```

```
Wall time: 997 µs
```

We now define the function that will be applied to our subdataframe originating from grouping on driver and zone values. This function will compute the total time and distance travelled by each driver in each zone, as well as their mean speed.

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.

Also, we will also choose a cut-off for the delay column. 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: for example, the driver simply stopped the recording and restarted it somewhere else, hours later. Hence, one should get rid of this step. We choose 20 min (=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).

[ ]:
```
%%time
def speed_compute(x):
    x=x.sort_values('timestamp')
    x['delay'] = x.timestamp.diff().dt.total_seconds()
    x['distances'] = da.arcsin(da.sqrt(da.sin(da.radians(x.longitude.diff().
 ↪mul(0.5))).pow(2).mul(da.cos(da.radians(x.latitude))).mul(da.cos(da.
 ↪radians(x.latitude.shift(periods=1)))).add(da.sin(da.radians(x.latitude.
 ↪diff().mul(0.5))).pow(2)))).mul(2*6371)
```

```
    x['speeds'] = x.distances.div(x.delay).mul(3600)
    x=x[~(da.isinf(x.speeds) | (x.speeds > 200) | x.delay > 1200)]
    x['total_time'] = x['delay'].sum()
    x['total_time'].iloc[1:] = np.nan
    x['total_distance'] = x['distances'].sum()
    x['total_distance'].iloc[1:] = np.nan
    x['meanspeed'] = x['total_distance'].div(x['total_time']).mul(3600)
    x=x.
 ↪drop(columns=['driver','latitude','longitude','distances','delay','timestamp','speeds']).
 ↪rename(columns={'zone':'dzone'})
    return x
```

```
Wall time: 0 ns
```

We can apply this function to the grouped data, and group again these results for each zone, taking the average on every separate driver:

[ ]:
```
gps_speeds=gpsclean.groupby(['driver','zone']).apply(speed_compute).
 ↪groupby('dzone').mean().sort_values('meanspeed',ascending=False)
```

```
C:\Users\Bilal\AppData\Local\Temp\ipykernel_23492\2877436997.py:1: UserWarning:
`meta` is not specified, inferred from partial data. Please provide `meta` if
the result is unexpected.
  Before: .apply(func)
  After:  .apply(func, meta={'x': 'f8', 'y': 'f8'}) for dataframe result
  or:     .apply(func, meta=('x', 'f8'))            for series result
  gps_speeds=gpsclean.groupby(['driver','zone']).apply(speed_compute).groupby('d
zone').mean().sort_values('meanspeed',ascending=False)
```

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:

[ ]:
```
%%time
def nb_zone(x):
    x['count'] = len(x)
    return x
gps_meanvisited = gpsclean[['driver','zone']].drop_duplicates().
 ↪groupby('driver').apply(nb_zone).groupby('zone').mean().sort_values('count',␣
 ↪ascending=False)
```

```
Wall time: 40.9 ms
```

```
<timed exec>:4: UserWarning: `meta` is not specified, inferred from partial
data. Please provide `meta` if the result is unexpected.
```

```
  Before: .apply(func)
  After:  .apply(func, meta={'x': 'f8', 'y': 'f8'}) for dataframe result
  or:     .apply(func, meta=('x', 'f8'))            for series result
```

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 diff() as for the speeds, in order to spot the rows in which there is a change of zone.

[ ]:
```
%%time
def nb_zone2(x):
    x = x.sort_values('timestamp', ascending=False)
    x['count'] = x.zone.diff().ne(0).cumsum().max()
    return x
gps_meanexchange = gpsclean[['driver','zone','timestamp']].groupby('driver').
 ↪apply(nb_zone2)
gps_meanexchange = gps_meanexchange.drop_duplicates(subset=['driver','zone']).
 ↪groupby('zone').mean().sort_values('count', ascending=False)
```

```
Wall time: 36.9 ms
```

```
<timed exec>:5: UserWarning: `meta` is not specified, inferred from partial
data. Please provide `meta` if the result is unexpected.
  Before: .apply(func)
  After:  .apply(func, meta={'x': 'f8', 'y': 'f8'}) for dataframe result
  or:     .apply(func, meta=('x', 'f8'))            for series result
```

1.6 Computation of the delayed quantities

[ ]:
```
%%time
(gps_mostvisited_c) = dask.compute(gps_mostvisited)
```

```
Wall time: 9min 17s
```

[ ]:
```
%%time
(gps_speeds_c) = dask.compute(gps_speeds)
```

```
Wall time: 9min 22s
```

[ ]:
```
%%time
(gps_meanvisited_c) = dask.compute(gps_meanvisited)
```

```
Wall time: 7min 38s
```

[ ]:
```
%%time
(gps_meanexchange_c) = dask.compute(gps_meanexchange)
```

```
Wall time: 7min 34s
```

[ ]:
```
%%time
display(gps_mostvisited_c, gps_speeds_c, gps_meanvisited_c, gps_meanexchange_c)
```

```
(22    10823
 21     8039
 14     4150
 25     3638
 13     3178
 20     1591
 23     1160
 24     1064
 26      747
 0       547
 15      512
 Name: zone, dtype: int64,)
```

```
(        total_time  total_distance  meanspeed
 dzone
 2        206.000000        2.943416  68.734520
 11      1751.166667        8.410023  59.956685
 8       2268.333333        8.147536  30.439522
 30      1020.500000        3.396694  23.760043
 27      2022.272727        5.830054  23.199141
 29       792.115385        2.798897  20.818950
 20      2146.177876        4.317292  19.731458
 15      1447.841797        3.550905  19.554468
 40       754.352941        3.852713  18.390652
 28      1909.116667        4.832468  17.858329
 16      2346.327869        6.145424  17.818974
 17      1302.421053        1.846554  17.480475
 37      1503.102041        4.715008  17.353181
 19      1743.627049        4.699589  16.305877
 12       810.392857        1.702010  15.398160
 26      2527.880857        5.547971  15.283745
 49       862.666667        2.005988  15.079609
 18      1737.784615        3.758791  15.069416
 24      1829.854323        2.914854  13.716427
 14      3748.071807        7.771217  13.170996
 13      2549.153870        4.797012  13.117010
 23      2148.012931        4.774195  13.107724
 38      1177.000000        3.126494  12.342012
 21      5672.036696       12.555548  10.939228
 35       428.100000        1.566745  10.811023
 22      7234.611106       17.017696   9.976861
 25      3147.868059        7.574746   9.528753
 31      1195.474359        1.284915   8.792135
 7        350.000000        0.093051   0.957094,)
```

```
(         count
 zone
 12    6.250000
 8     6.166667
```

```
 35    5.300000
 49    5.250000
 11    5.000000
 40    4.705882
 30    4.333333
 2     4.333333
 29    4.288462
 16    4.251366
 15    4.082031
 27    3.987013
 28    3.966667
 37    3.959184
 17    3.929825
 24    3.909774
 19    3.795082
 13    3.717432
 23    3.603448
 20    3.539283
 14    3.399277
 25    3.282573
 18    3.236923
 31    3.217949
 21    3.055853
 26    2.903614
 22    2.859004
 38    2.363636
 7     2.000000,)
(         count
 zone
 12    9.750000
 8     9.500000
 11    8.333333
 30    7.666667
 13    7.126809
 15    6.933594
 35    6.800000
 49    6.750000
 17    6.649123
 16    6.546448
 24    6.412594
 23    6.230172
 14    6.185301
 40    6.058824
 29    5.903846
 20    5.717788
 27    5.441558
 19    5.413934
```

```
21    5.380893
18    5.212308
25    5.205058
28    5.066667
37    5.061224
22    4.855216
31    4.525641
2     4.333333
26    4.171352
38    2.727273
7     2.000000,)

Wall time: 15 ms
```

1.7 Cleaning up

[ ]:
```
client.restart()
```

```
2023-01-20 23:08:07,367 - distributed.nanny - WARNING - Restarting worker
2023-01-20 23:08:08,028 - distributed.nanny - WARNING - Restarting worker
2023-01-20 23:08:08,273 - distributed.nanny - WARNING - Restarting worker
2023-01-20 23:08:08,277 - distributed.nanny - WARNING - Restarting worker
```

[ ]:
```
<Client: 'tcp://127.0.0.1:58809' processes=0 threads=0, memory=0 B>
```

1.8 Comment on the warnings obtained

There are a few warnings left in this notebook. Namely, "UserWarning: `meta` is not specified, inferred from partial data. Please provide `meta` if the result is unexpected." regarding the .apply() method on groupby()." complains about the lack of explicit definition of the types used in the columns on which apply() is used. However, this is not an issue in this case as the type of data we have is not misleading, and will be easily inferred.