Big Data in NYC for Taxi rides management

with Apache Spark

Gabriele Favia matr. 579166

January 2020

Contents

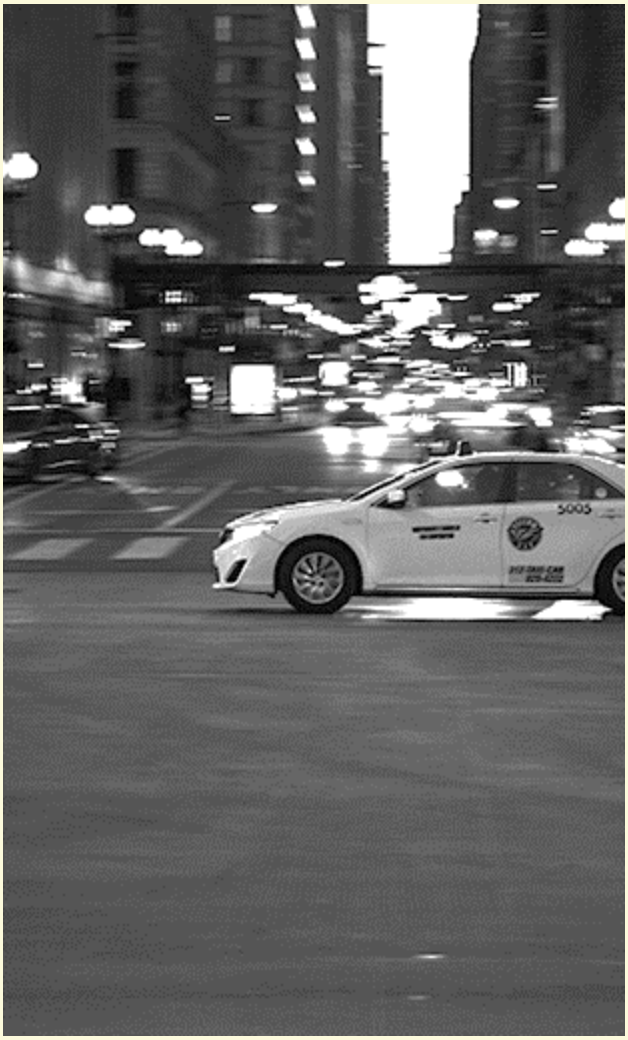
The NYC Taxi and Limousine Commission

The New York City Taxi and Limousine Commission (NYC TLC) is an agency of the New York City government that licenses and regulates the medallion taxis and for-hire vehicle industries, including app-based companies.
The TLC's regulatory landscape includes medallion (yellow) taxicabs, green or Boro taxicabs, black cars (including both traditional and app-based services), community-based livery cars, commuter vans, paratransit vehicles (ambulettes), and some luxury limousines [1].

But what is the difference between yellow and green taxi (cab in American English)?

The famous NYC yellow taxis provide transportation exclusively through street-hails.
The number of taxicabs is limited by a finite number of medallions issued by the TLC.
It's possible to access this mode of transportation by standing in the street and hailing an available taxi with your hand. The pickups are not pre-arranged.

The green taxis know as well as "Boro" cab or Street Hail Livery (SHL) are permitted to accept street-hails above 110th Street in Manhattan and in the outer-boroughs of New York City. The SHL program allows livery vehicle owners to license and outfit their vehicles with green borough taxi branding, meters, credit card machines, and ultimately the right to accept street hails in addition to pre-arranged rides [2].



Requests

Design and implementation in Spark (Pyspark) of an application answering the following questions:

1. The total number of passengers in yellow and green taxi in 2018.
2. For each rate code, the total amount charged to passengers of yellow and green taxi in 2018.
3. For each PULocationID, the list of related DOLocationID for yellow taxi in 2018, ordered by increasing average trip distance.

Data dictionary

The two kind of file of interest, yellow and green have a different header, which means that the software has the need to distinguish and operate the two cases differently.

Moreover, the green and yellow taxis dataset produced from January 2016 to June 2016, follow a different data dictionary that requires some variation to the parsing procedure for the points 4 and 5, while for the point 6, the dataset cannot be use because of the absence of these fields.

The fields of interest to be extracted from both the yellow and green cabs files are:

Field	Description
lpep_pickup_datetime or tpep_pickup_datetime	The pickup time, useful to select the year.
Passenger_count	The number of passengers in the vehicle.
RatecodeID	The final rate code in effect at the end of the trip. 1 = Standard rate 2 = JFK 3 = Newark 4 = Nassau or Westchester 5 = Negotiated fare 6 = Group ride
total_amount	The total amount charged to passengers. Does not include cash tips.

While the fields of interest exclusively extracted from yellow files are:

Field	Description
PULocationID	TLC Taxi Zone in which the taximeter was engaged.
DOLocationID	TLC Taxi Zone in which the taximeter was disengaged.
trip_distance	The elapsed trip distance in miles reported by the taximeter.

A comprehensive description of all the fields is available at nyc.gov website both for yellow [3] and the green [4] cabs.

Explaining the algorithms

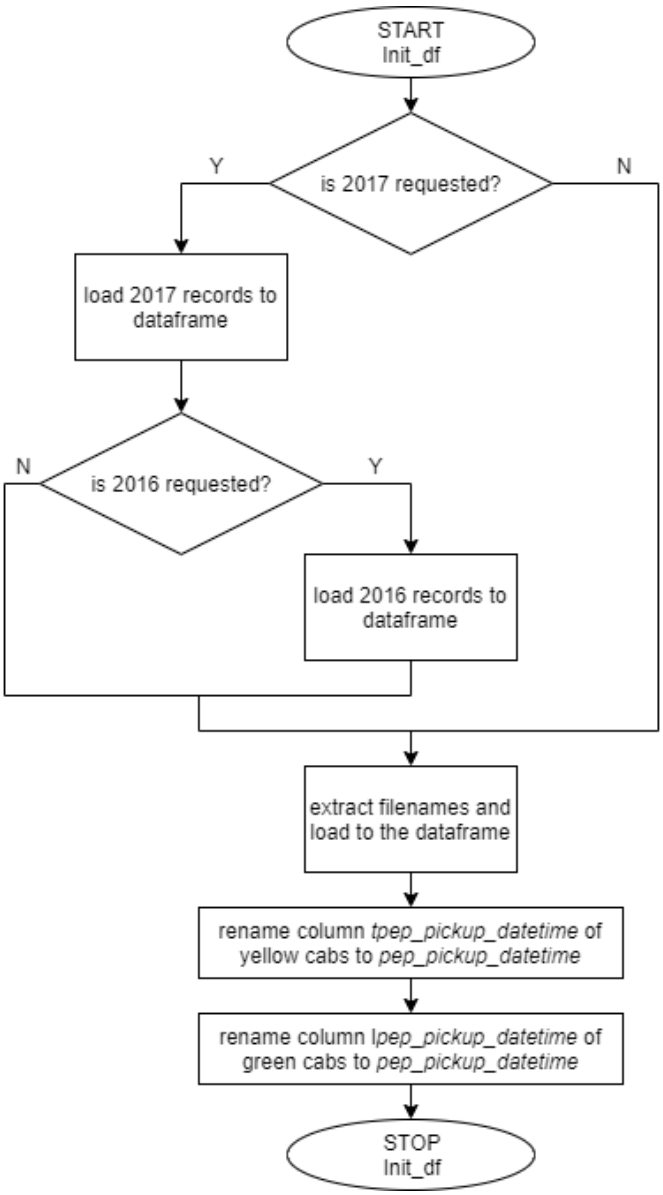
All the three tasks' algorithms share the same "initialization procedure", called *Init_df*.

This consists of parsing the parameters given by the user with the Spark command line and prepare the dataset for the following operations.

In fact, the command parameter "—years" specify whether the software has to load other years, beside 2018 files.

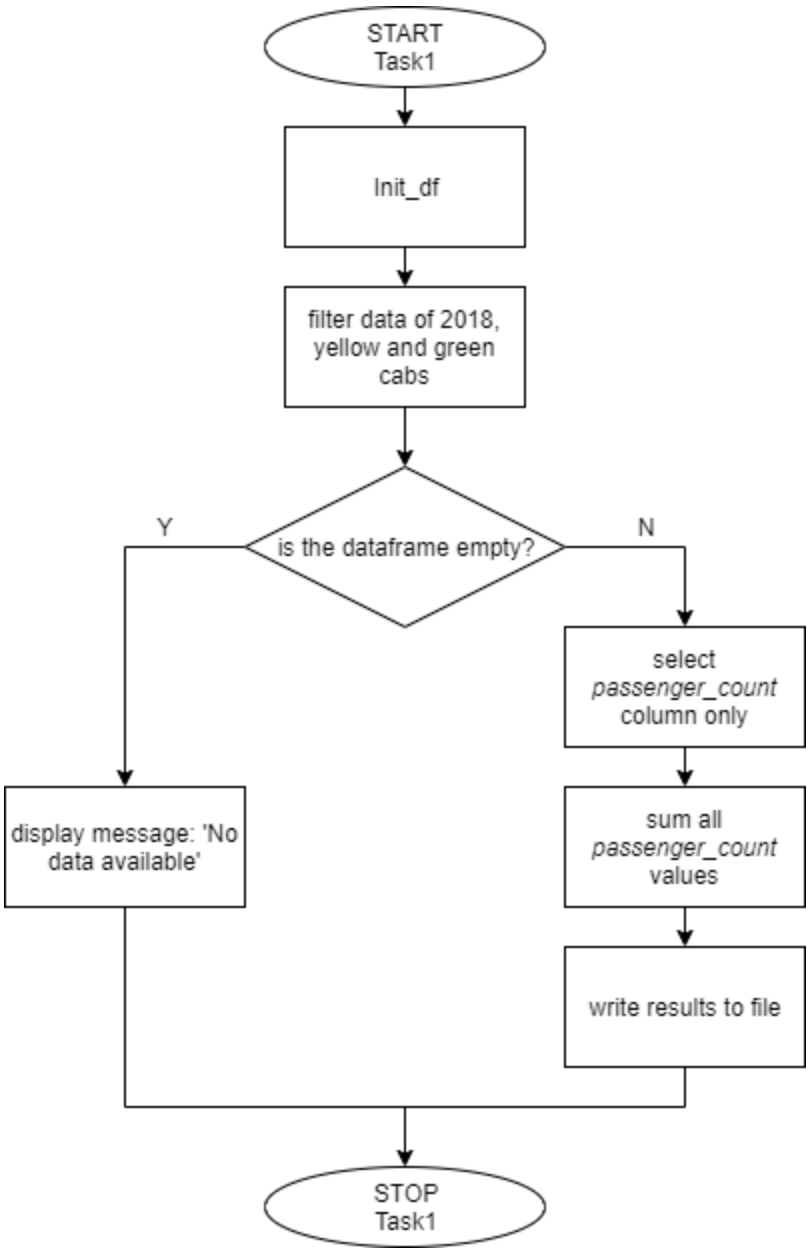
The second step is extracting the name of the source file from each record to a new column in order to be able to distinguish from data related to yellow or green taxi.

The third step renames the column name of the departure time to *pep_pick_datedatime* because of the different original column name between green and yellow cab (check Data dictionary for further information); in this way will be easier to check the year is a uniform way.



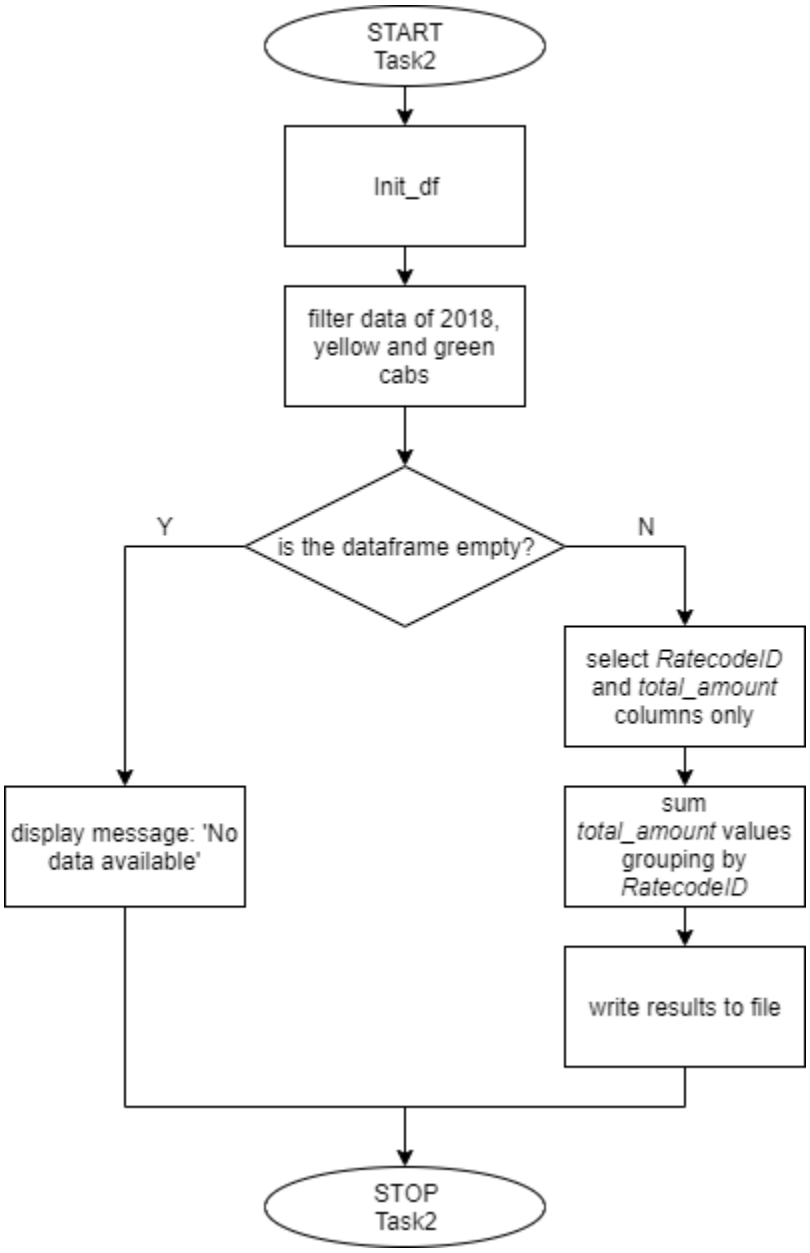
Task 1

The total number of passengers in yellow and green taxi in 2018.



Task 2

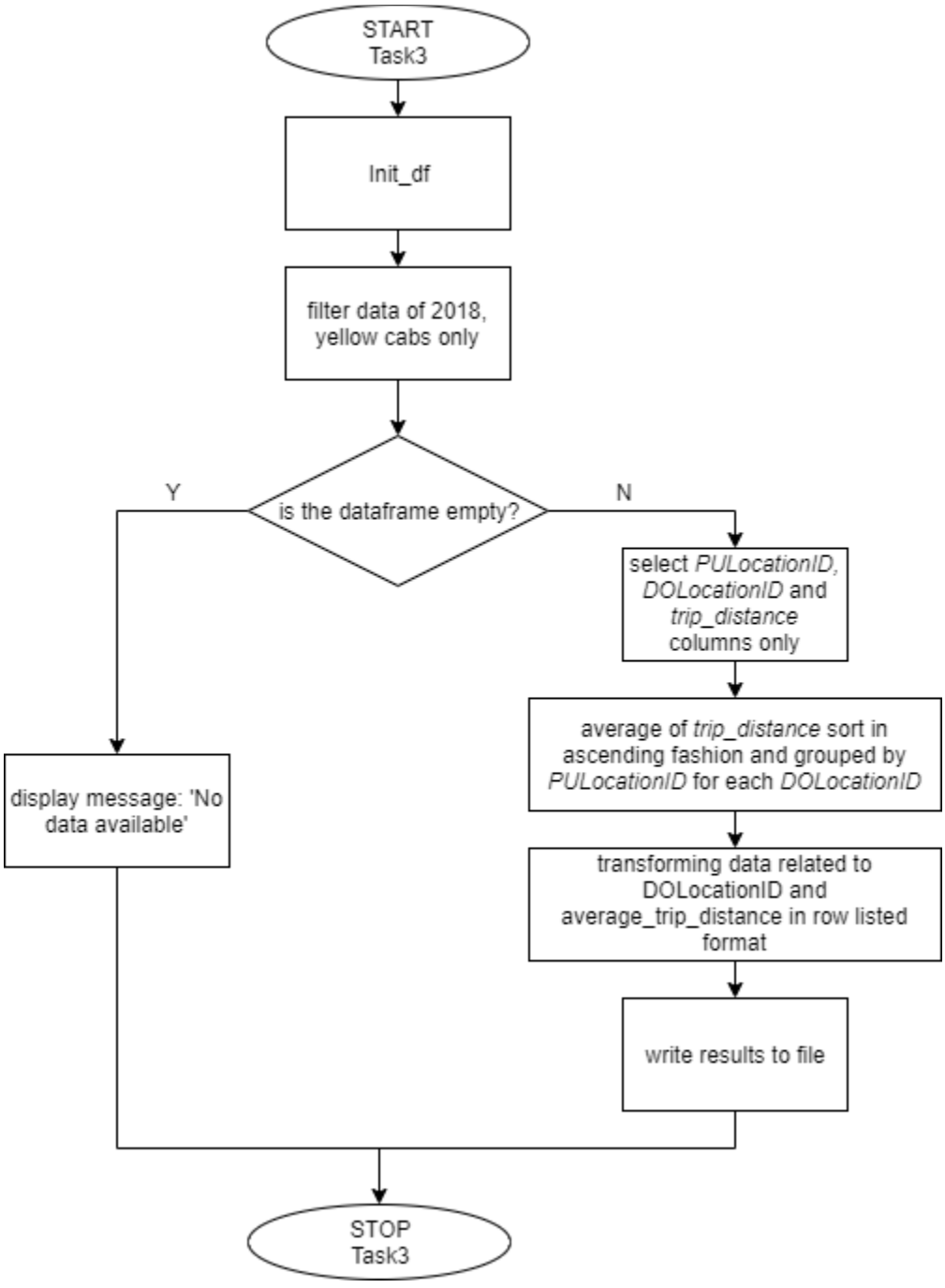
For each rate code, the total amount charged to passengers of yellow and green taxi in 2018.



Task 3

For each *PULocationID*, the list of related *DOLocationID* for yellow taxi in 2018, ordered by increasing average trip distance.

The core step is showing the results in a single row fashion: to achieve this, the main dataset grouped by both *PULocationID* and *DOLocationID* with the column of the sorted averages of *trip_distance* has been split up to two dataframes to make the linearizing possible and then unified again with a join function on the field *PULocationID*.



Usage

The Spark command accepts the following parameters:

Parameter	Value	Description
--master	*local*	Runs Spark in local mode.
	yarn	Runs Spark using Hadoop HDFS.
*--input**	*<path>*	The path containing the files to load into the dataframe.
*--output**	*<path>*	The target path to save the file with the result.
--years	*<string>*	Explicit which years must be included in the file loading (2018 is always loaded by default). Not specifying years will include all the 2016,2017,2018 files.

* When Spark is running on HDFS, it gets for granted the first part of the absolute path (`hdfs://namenodehost/user/<username>`), so it just needs the rest of the path as string. When running in local mode it needs the full path, starting with `file://<path>` in the local filesystem).

Since these parameters are specifically build for the applications, they must be set after the name of the python script to run.

The parameter *–-master* is part of Spark and must be the first one in the command line, instead.

Calling the task1 script in local mode with 2018 files only

```
$SPARK_HOME/bin/spark-submit --master local task1.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/ --years 2018
```

Calling the task1 script in local mode, including also 2017 files

```
$SPARK_HOME/bin/spark-submit --master local task1.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/ --years 2017
```

Calling the task1 script in local mode, including also 2017 and 2016 files

```
$SPARK_HOME/bin/spark-submit --master local task1.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/ --years 2017,2016
```

```
$SPARK_HOME/bin/spark-submit --master local task1.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/
```

Calling the task1 script in yarn mode with 2018 files only

```
$SPARK_HOME/bin/spark-submit --master yarn task1.py --input
<more_path>/input/ --output <more_path>/output/ --years 2018
```

Calling the task1 script in yarn mode, including also 2017 files

```
$SPARK_HOME/bin/spark-submit --master yarn task1.py --input
<more_path>/input/ --output <more_path>/output/ --years 2017
```

Calling the task1 script in yarn mode, including also 2017 and 2016 files

```
$SPARK_HOME/bin/spark-submit --master yarn task1.py --input
<more_path>/input/ --output <more_path>/output/ --years 2017,2016
```

```
$SPARK_HOME/bin/spark-submit --master yarn task1.py --input
<more_path>/input/ --output <more_path>/output/
```

Calling the task2 script in local mode with 2018 files only

```
$SPARK_HOME/bin/spark-submit --master local task2.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/ --years 2018
```

Calling the task2 script in local mode, including also 2017 files

```
$SPARK_HOME/bin/spark-submit --master local task2.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/ --years 2017
```

Calling the task2 script in local mode, including also 2017 and 2016 files

```
$SPARK_HOME/bin/spark-submit --master local task2.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/ --years 2017,2016
```

or

```
$SPARK_HOME/bin/spark-submit --master local task2.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/
```

Calling the task2 script in yarn mode with 2018 files only

```
$SPARK_HOME/bin/spark-submit --master yarn task2.py --input
<more_path>/input/ --output <more_path>/output/ --years 2018
```

Calling the task2 script in yarn mode, including also 2017 files

```
$SPARK_HOME/bin/spark-submit --master yarn task2.py --input
<more_path>/input/ --output <more_path>/output/ --years 2017
```

Calling the task2 script in yarn mode, including also 2017 and 2016 files

```
$SPARK_HOME/bin/spark-submit --master yarn task2.py --input
<more_path>/input/ --output <more_path>/output/ --years 2017,2016
```

```
$SPARK_HOME/bin/spark-submit --master yarn task2.py --input
<more_path>/input/ --output <more_path>/output/
```

Calling the task3 script in local mode with 2018 files only

```
$SPARK_HOME/bin/spark-submit --master local task3.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/ --years 2018
```

Calling the task3 script in local mode, including also 2017 files

```
$SPARK_HOME/bin/spark-submit --master local task3.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/ --years 2017
```

Calling the task3 script in local mode, including also 2017 and 2016 files

```
$SPARK_HOME/bin/spark-submit --master local task3.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/ --years 2017,2016
```

or

```
$SPARK_HOME/bin/spark-submit --master local task3.py --input
file:///home/<user>/<more_path>/input/ --output
file:///home/<user>/<more_path>/output/
```

Calling the task3 script in yarn mode with 2018 files only

```
$SPARK_HOME/bin/spark-submit --master yarn task3.py --input
<more_path>/input/ --output <more_path>/output/ --years 2018
```

Calling the task3 script in yarn mode, including also 2017 files

```
$SPARK_HOME/bin/spark-submit --master yarn task3.py --input
<more_path>/input/ --output <more_path>/output/ --years 2017
```

Calling the task3 script in yarn mode, including also 2017 and 2016 files

```
$SPARK_HOME/bin/spark-submit --master yarn task3.py --input
<more_path>/input/ --output <more_path>/output/ --years 2017,2016
```

or

```
$SPARK_HOME/bin/spark-submit --master yarn task3.py --input
<more_path>/input/ --output <more_path>/output/
```

Results

Task 1

Year	SUM(passenger_count)
2018	163968005

Task 2

RatecodeID	SUM(total_amount)
125	146931
7	484100224
51	50871599
124	8489877
169	26531718
205	34382728
54	7752295
15	4745360
234	360127
232	82041
155	34881902
132	6293622
154	1422473
200	14080181
11	10165664
101	7849635
138	4543505
69	53575184
29	25104240
42	379169010
112	124717695
73	4343419
87	247756
64	3552269
3	1923079999
113	184793
30	60951
34	6031254
133	15199839
59	1145463
162	703620
146	59085664
139	12093020
250	24134516
160	7625337

8	285769
258	14361899
203	10548378
22	28392997
28	19930116
184	1923215
85	27828798
52	113860684
35	61849122
16	11525637
251	309124
183	11088482
171	10910115
187	316740
71	41964294
98	6156025
188	79612701
99	52268577
195	33186496
223	217128467
47	22745803
107	283820
214	293385
179	82981786
248	15770624
202	2544949
96	833174
221	1197738
43	116975755
5	2700863064
31	9387459
163	565230
100	777871
18	28959267
70	15366203
174	45365641
206	1596099
168	48362557
224	18980
61	153714483
218	19521153
27	540701
75	667604220
166	468092518
219	11786559
17	101879711
126	16571583

131	15622068
140	283002
26	42527402
227	18450297
120	1827341
46	3689897
130	233826550
147	15325361
207	784017
164	578471
78	30796025
208	18078946
77	27295231
89	74577712
228	38860178
136	29904434
198	13631272
6	3900632
257	15073884
118	382703
185	23057466
256	126133894
230	1072089
201	2657552
177	39467139
244	389520136
60	11911376
229	275273
68	388664
90	235530
246	193664
194	2676950
19	8733340
41	584650775
128	2875468
23	1092480
102	9603476
55	89604498
238	182315
263	9098381
111	590435
197	43595654
220	28156744
167	26699852
95	339999071
93	75248285
40	80828678

38	11887075
25	308415117
189	45514385
233	246678
135	16977006
190	16397049
156	1385620
44	134637
144	130622
176	87186
82	478787284
241	19520911
115	1899948
193	43034194
53	5473880
245	406718
92	124017792
231	435048
122	10109877
247	68507386
108	23280313
117	13701651
86	11634391
58	1254713
261	181961
204	51454
81	22654881
33	377790537
242	53097350
114	223688
213	47343292
150	14940850
170	587741
178	8973865
259	18401388
153	4067681
48	851822
217	12587524
243	93631260
173	32000610
180	8513443
240	3351036
148	174623
141	222986
159	45087152
97	362831838
239	470673

209	61222
106	39933297
67	9888128
158	120201
236	29999292
84	116621
143	121505
79	333016
24	37439272
9	7124312
212	27070165
116	184928386
32	22747749
152	90549854
186	683031
134	101236608
88	234974
1	147833749243
149	21965958
237	305335
20	16182698
142	413243
56	35418210
127	48676130
36	50324805
211	205066
10	21422723
37	50489449
165	34226363
49	146260808
255	343552508
222	29208647
253	4419015
172	265842
181	341726208
63	14753722
65	331119967
225	76834213
235	35716099
265	11270888
4	537070614
121	24597473
39	69884501
252	5230414
210	67200304
62	40055379
12	3946

83	61459966
215	26541003
123	29995499
109	72689
249	252144
13	252779
260	238632733
157	9582000
191	26344462
14	59236446
182	40307887
21	22558802
66	307628345
264	12485515
94	10949565
175	5909850
91	52252449
74	739692052
72	46316034
137	700447
161	583982
151	81367
262	673108
129	302747285
76	105558824
2	15694159500
196	118170909
254	39856341
192	14952211
80	97794436
226	138056732
145	183581322
50	153389
57	2002672
45	58911
216	29985460
119	24753841

Task 3 (first row only)

PULocationID	COLLECT_SET(DOLocationID) AS group_doloc	COLLECT_LIST(avg_trip) AS group_trip
1	264, 1, 243, 43, 233, 229, 244, 42, 4, 79, 265, 163, 23, 144, 156, 166, 236, 13, 68, 107, 211, 164, 48, 145, 125, 238, 251, 158, 231, 65, 249, 90, 148, 186, 239, 33, 113, 45, 114, 246, 234, 214, 261, 161, 25, 230, 141, 50, 237, 100, 142, 225, 170, 137, 206, 14, 162, 112, 24, 151, 41, 82, 228, 255, 140, 169, 83, 80, 132, 263, 74, 197, 75, 138, 152, 181, 93, 223, 121, 67	0.20, 1.22, 6.19, 6.45, 7.24, 7.57, 7.82, 7.79, 9.12, 9.60, 9.88, 10.64, 10.84, 11.82, 12.70, 13.37, 13.31, 13.33, 13.34, 13.56, 13.73, 14.06, 13.97, 14.01, 14.16, 14.36, 14.59, 14.64, 14.82, 15.12, 15.03, 15.28, 15.32, 15.32, 15.41, 15.58, 15.49, 15.49, 15.79, 16.00, 16.29, 16.30, 16.33, 16.34, 16.54, 16.61, 16.95, 17.10, 17.12, 17.42, 17.61, 17.67, 17.87, 18.10, 18.30, 18.49, 19.33, 19.54, 19.85, 19.85, 21.02, 21.60, 22.12, 22.20, 22.79, 22.81, 22.52, 23.62, 23.35, 24.70, 24.90, 25.20, 25.40, 25.67, 25.79, 26.16, 27.10, 29.91, 31.27, 39.27

Performance comparison

The program instances are run on the following virtual machine:

The operative system is Ubuntu 18.4.4 hosted by VirtualBox vers. 6.0.14 which are given:

2 cores;
4096 MB RAM;
60GB disk space;

Equipped with Spark vers. 3.0.0, Hadoop vers. 3.1.2, Java Open JDK vers. 1.8.0

The host machine with:

Intel© Core™ i7 6500U and SSD.

Time for completion

years: 2018

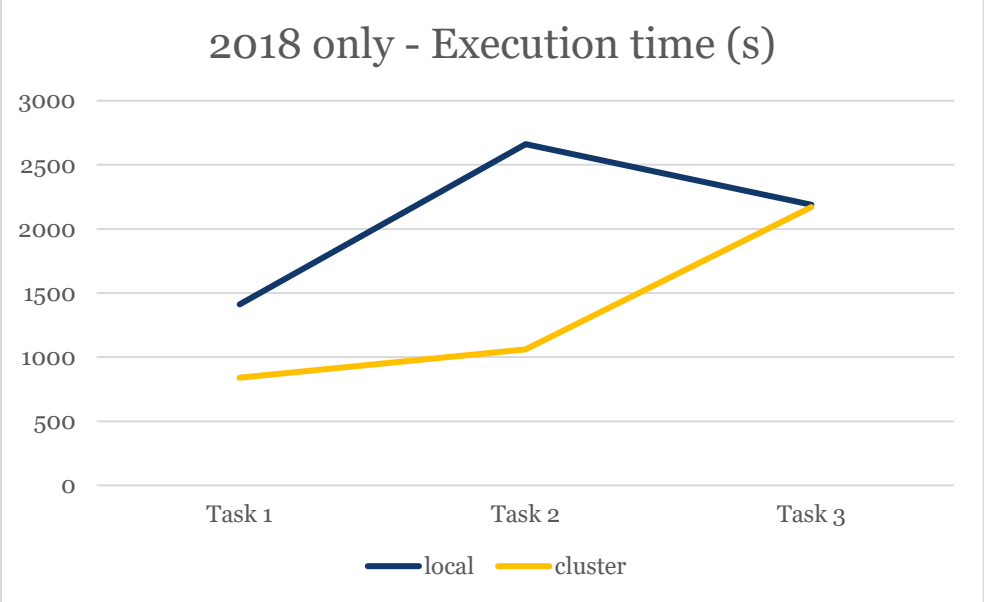
Task	Execution time in local mode (s)	Execution time in cluster mode (s)
Task 1	1412	840
Task 2	2662	1060
Task 3	2189	2171

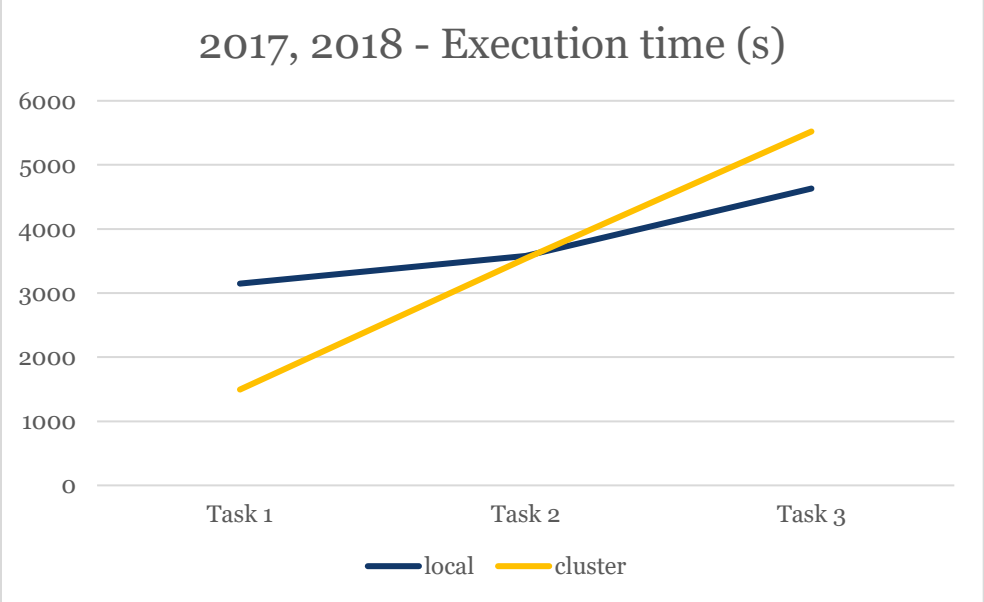
years: 2017, 2018

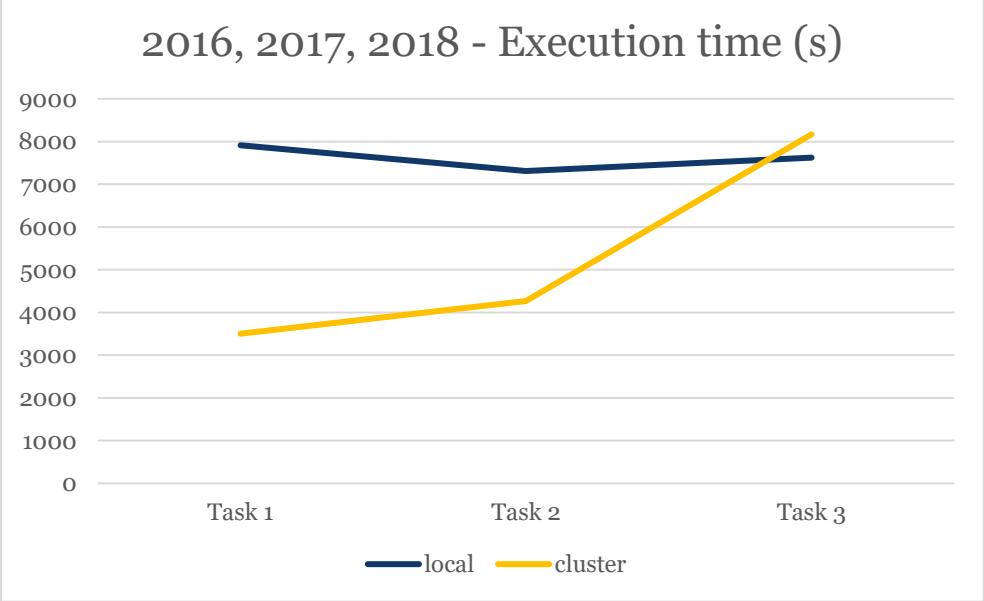
Task	Execution time in local mode (s)	Execution time in cluster mode (s)
Task 1	3148	1495
Task 2	3580	3540
Task 3	4631	5521

years: 2016, 2017, 2018

Task	Execution time in local mode (s)	Execution time in cluster mode (s)
Task 1	7913	3503
Task 2	7310	4268
Task 3	7623	8169


2018 only - Execution time (s)
3000
2500
2000
1500
1000
500
0
Task 1
Task 2
Task 3
local
cluster


2017, 2018 - Execution time (s)
6000
5000
4000
3000
2000
1000
0
Task 1
Task 2
Task 3
local
cluster


2016, 2017, 2018 - Execution time (s)
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Task 1
Task 2
Task 3
local
cluster

The cluster mode can perform better or worse than the local mode, depending mainly the amount of data: the biggest differences are noticeable in the case of Task 1 (with all 2018, 2017 and 2016 data ~ 30M of rows) where the cluster mode is +55.73 % faster than local mode in terms of completion time. Overall, the cluster mode behaves better than the local mode, except when managing all the data and 2017+2018 in the Tasks 2 and Task 3, reaching a maximum of +19.21 % more needed time in comparison to local mode.

Limitations

The results concerning the execution times are biased by the state of the host machine which was loaded with different tasks in background, causing a not constant access time and bandwidth availability on the SSD on the host operating system, so these must be taken as an overall idea of the average execution time for the machine used.

References

[1] https://en.wikipedia.org/wiki/New_York_City_Taxi_and_Limousine_Commission
[2] https://www.quora.com/What-is-the-difference-between-Green-Cabs-and-Yellow-Cabs
[3] https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf
[4] https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_green.pdf

Code

Task 1

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
from pyspark.sql import functions as F
from pyspark.sql.functions import input_file_name
import argparse

#istantiate session
spark = SparkSession.builder.appName("SumPassengers").getOrCreate()

#definining a function that extracts the file name from the path of the
file-related row
get_only_file_name = spark.udf.register('get_only_file_name', lambda x:
x.rsplit('/', 1)[-1])

#initializing input and output folder
input_folder = ""
output_folder = ""

#reading parameters given by the user
parser = argparse.ArgumentParser()
parser.add_argument("--years", help="Years to consider divided by a comma.
2018 is always considered.")
parser.add_argument("--input", help="The input path folder.")
parser.add_argument("--output", help="The output path folder.")

args = parser.parse_args()

if args.input: #making sure the path ends with a slash
     input_folder = args.input
     if input_folder[-1:] != "/":
          input_folder = input_folder + "/"
if args.output:
     output_folder = args.output
     if output_folder[-1:] != "/":
          output_folder = output_folder + "/"
if args.years:
     years = args.years
     #reads the 2018 files in any case
     fileSource_jan_sept = input_folder + '*2018-0[1-9]*.csv';  #dividing
te perios because of the different beheavior of the regex
     fileSource_oct_dec = input_folder + '*2018-1[0-2]*.csv';
     temp_df = spark.read.csv(fileSource_jan_sept, header=True,
mode="DROPMALFORMED");
     dataframe = spark.read.csv(fileSource_oct_dec, header=True,
mode="DROPMALFORMED")
```

```
     dataframe = dataframe.union(temp_df)

     if '2016' in years:
          fileSource_jul_sept = input_folder + '*2016-0[7-9]*.csv';
          fileSource_oct_dec = input_folder + '*2016-1[0-2]*.csv';
          temp_df = spark.read.csv(fileSource_jul_sept, header=True,
mode="DROPMALFORMED");
          temp_df2 = spark.read.csv(fileSource_oct_dec, header=True,
mode="DROPMALFORMED")
          dataframe = dataframe.union(temp_df)
          dataframe = dataframe.union(temp_df2)
     if '2017' in years:
          fileSource_jan_sept = input_folder + '*2017-0[1-9]*.csv';
          fileSource_oct_dec = input_folder + '*2017-1[0-2]*.csv';
          temp_df = spark.read.csv(fileSource_jan_sept, header=True,
mode="DROPMALFORMED");
          temp_df2 = spark.read.csv(fileSource_oct_dec, header=True,
mode="DROPMALFORMED")
          dataframe = dataframe.union(temp_df)
          dataframe = dataframe.union(temp_df2)
else:
     fileSource = input_folder + '*.csv'
     #loads the datadrame considering the presence of a header and
skipping the malformed rows (like the empty space rows after the header)
     dataframe = spark.read.csv(fileSource, header=True,
mode="DROPMALFORMED")

#appliying the user defined function to get the file name from the path,
in order to be able to distinguish between yellow and green dataset files
dataframe = dataframe.withColumn("filename",
get_only_file_name(input_file_name()))

#normalizing the column related to the departure datetime to have the same
name for both yellow and green cab datasets
dataframe = dataframe.withColumnRenamed('tpep_pickup_datetime',
'pep_pickup_datetime')
dataframe = dataframe.withColumnRenamed('lpep_pickup_datetime',
'pep_pickup_datetime')

#filtering data related to year 2018 only for yellow and greeen cabs
dataframe =
dataframe.filter(dataframe['pep_pickup_datetime'].like('2018%')).filter(da
taframe['filename'].like('yellow_tripdata_%') |
dataframe['filename'].like('green_tripdata_%'))

#skips further processing if the dataset is empty
if len(dataframe.head(1)) > 0:
     #selection of only the required fields for the computation
     dataframe = dataframe.select('passenger_count')

     #caching the dataframe to speedup the next processing
```

```
    dataframe.cache()
    # Change 'passenger_count' column type to interger
    dataframe = dataframe.withColumn('passenger_count',
dataframe['passenger_count'].cast(IntegerType()))

    #using lambda functions with map and reduce tasks to sum up the
passenger amount
    value = dataframe.select(F.sum('passenger_count')).collect()[0][0]

    print("The result is " + str(value))

    df_final = spark.createDataFrame([('2018', value)], ['year',
'passenger_count'])

    df_final.show()

    #write the result in a csv file, where
    #    >result is reported in a single file
    #    >headers are preserved
    #    >the previous resulting csv is overwritten
    df_final.coalesce(1).write.option("header","true").mode('overwrite').
csv(output_folder + "results_task1.csv")
else:
    print("Data not available")

spark.stop()
```

Task 2

```
from pyspark.sql import SparkSession
from pyspark.sql.types import DecimalType
from pyspark.sql.functions import input_file_name
import argparse

#istantiate session
spark = SparkSession.builder.appName("RatecodeAmount").getOrCreate()

#definining a function that extracts the file name from the path of the
file-related row
get_only_file_name = spark.udf.register('get_only_file_name', lambda x:
x.rsplit('/', 1)[-1])

#initializing input and output folder
input_folder = ""
output_folder = ""

#reading parameters given by the user
parser = argparse.ArgumentParser()
parser.add_argument("--years", help="Years to consider divided by a comma.
2018 is always considered.")
parser.add_argument("--input", help="The input path folder.")
parser.add_argument("--output", help="The output path folder.")

args = parser.parse_args()

if args.input: #making sure the path ends with a slash
     input_folder = args.input
     if input_folder[-1:] != "/":
          input_folder = input_folder + "/"
if args.output:
     output_folder = args.output
     if output_folder[-1:] != "/":
          output_folder = output_folder + "/"
if args.years:
     years = args.years
     #reads the 2018 files in any case
     fileSource_jan_sept = input_folder + '*2018-0[1-9]*.csv'; #dividing
te perios because of the different beheavior of the regex
     fileSource_oct_dec = input_folder + '*2018-1[0-2]*.csv';
     temp_df = spark.read.csv(fileSource_jan_sept, header=True,
mode="DROPMALFORMED");
     dataframe = spark.read.csv(fileSource_oct_dec, header=True,
mode="DROPMALFORMED")
     dataframe = dataframe.union(temp_df)

     if '2016' in years:
```

```
          fileSource_jul_sept = input_folder + '*2016-0[7-9]*.csv';
          fileSource_oct_dec = input_folder + '*2016-1[0-2]*.csv';
          temp_df = spark.read.csv(fileSource_jul_sept, header=True,
mode="DROPMALFORMED");
          temp_df2 = spark.read.csv(fileSource_oct_dec, header=True,
mode="DROPMALFORMED")
          dataframe = dataframe.union(temp_df)
          dataframe = dataframe.union(temp_df2)
     if '2017' in years:
          fileSource_jan_sept = input_folder + '*2017-0[1-9]*.csv';
          fileSource_oct_dec = input_folder + '*2017-1[0-2]*.csv';
          temp_df = spark.read.csv(fileSource_jan_sept, header=True,
mode="DROPMALFORMED");
          temp_df2 = spark.read.csv(fileSource_oct_dec, header=True,
mode="DROPMALFORMED")
          dataframe = dataframe.union(temp_df)
          dataframe = dataframe.union(temp_df2)
else:
     fileSource = input_folder + '*.csv'
     #loads the datadrame considering the presence of a header and
skipping the malformed rows (like the empty space rows after the header)
     dataframe = spark.read.csv(fileSource, header=True,
mode="DROPMALFORMED")

#appliying the user defined function to get the file name from the path,
in order to be able to distinguish between yellow and green dataset files
dataframe = dataframe.withColumn("filename",
get_only_file_name(input_file_name()))

#normalizing the column related to the departure datetime to have the same
name for both yellow and green cab datasets
dataframe = dataframe.withColumnRenamed('tpep_pickup_datetime',
'pep_pickup_datetime')
dataframe = dataframe.withColumnRenamed('lpep_pickup_datetime',
'pep_pickup_datetime')

#filtering data related to year 2018 only for yellow and greeen cabs
dataframe =
dataframe.filter(dataframe['pep_pickup_datetime'].like('2018%')).filter(da
taframe['filename'].like('yellow_tripdata_%') |
dataframe['filename'].like('green_tripdata_%'))

#skips further processing if the dataset is empty
if len(dataframe.head(1)) > 0:
     #selection of only the required fields for the computation
     dataframe = dataframe.select('RatecodeID', 'total_amount')

     #caching the dataframe to speedup the next processing
     dataframe.cache()
```

```
     #change 'total_amount' column type to decimal type with two digits of
precision
     dataframe = dataframe.withColumn('total_amount',
dataframe['total_amount'].cast(DecimalType(32,2)))

     #grouping PULocationID making the sum of total_amount
     df_final = dataframe.groupBy('RatecodeID').agg({"total_amount":
"sum"})

     df_final.show()

     #write the result in a csv file, where
     #    >result is reported in a single file
     #    >headers are preserved
     #    >the previous resulting csv is overwritten
     df_final.coalesce(1).write.option("header","true").mode('overwrite').
csv(output_folder + "results_task2.csv")
else:
     print("Data not available")

spark.stop()
```

Task 3

```
from pyspark.sql import SparkSession
from pyspark.sql.types import DecimalType, StructType
from pyspark.sql import functions as F
from pyspark.sql.functions import input_file_name
from pyspark.sql.functions import concat_ws
import argparse

#istantiate session
spark = SparkSession.builder.appName("PUDO_amount").getOrCreate()

#definining a function that extracts the file name from the path of the
file-related row
get_only_file_name = spark.udf.register('get_only_file_name', lambda x:
x.rsplit('/', 1)[-1])


#initializing input and output folder
input_folder = ""
output_folder = ""

#reading parameters given by the user
parser = argparse.ArgumentParser()
parser.add_argument("--years", help="Years to consider divided by a comma.
2018 is always considered.")
parser.add_argument("--input", help="The input path folder.")
parser.add_argument("--output", help="The output path folder.")

args = parser.parse_args()

if args.input: #making sure the path ends with a slash
     input_folder = args.input
     if input_folder[-1:] != "/":
          input_folder = input_folder + "/"
if args.output:
     output_folder = args.output
     if output_folder[-1:] != "/":
          output_folder = output_folder + "/"
if args.years:
     years = args.years
     #reads the 2018 files in any case
     fileSource_jan_sept = input_folder + '*2018-0[1-9]*.csv'; #dividing
te perios because of the different beheavior of the regex
     fileSource_oct_dec = input_folder + '*2018-1[0-2]*.csv';
     temp_df = spark.read.csv(fileSource_jan_sept, header=True,
mode="DROPMALFORMED");
     dataframe = spark.read.csv(fileSource_oct_dec, header=True,
mode="DROPMALFORMED")
```

```
     dataframe = dataframe.union(temp_df)

     if '2016' in years:
          fileSource_jul_sept = input_folder + '*2016-0[7-9]*.csv';
          fileSource_oct_dec = input_folder + '*2016-1[0-2]*.csv';
          temp_df = spark.read.csv(fileSource_jul_sept, header=True,
mode="DROPMALFORMED");
          temp_df2 = spark.read.csv(fileSource_oct_dec, header=True,
mode="DROPMALFORMED")
          dataframe = dataframe.union(temp_df)
          dataframe = dataframe.union(temp_df2)
     if '2017' in years:
          fileSource_jan_sept = input_folder + '*2017-0[1-9]*.csv';
          fileSource_oct_dec = input_folder + '*2017-1[0-2]*.csv';
          temp_df = spark.read.csv(fileSource_jan_sept, header=True,
mode="DROPMALFORMED");
          temp_df2 = spark.read.csv(fileSource_oct_dec, header=True,
mode="DROPMALFORMED")
          dataframe = dataframe.union(temp_df)
          dataframe = dataframe.union(temp_df2)
else:
     fileSource = input_folder + '*.csv'
     #loads the datadrame considering the presence of a header and
skipping the malformed rows (like the empty space rows after the header)
     dataframe = spark.read.csv(fileSource, header=True,
mode="DROPMALFORMED")

#appliying the user defined function to get the file name from the path,
in order to be able to distinguish between yellow and green dataset files
dataframe = dataframe.withColumn("filename",
get_only_file_name(input_file_name()))

#normalizing the column related to the departure datetime to have the same
name for both yellow and green cab datasets
dataframe = dataframe.withColumnRenamed('tpep_pickup_datetime',
'pep_pickup_datetime')
dataframe = dataframe.withColumnRenamed('lpep_pickup_datetime',
'pep_pickup_datetime')

#filtering data related to yellow cabs and year 2018 only
dataframe =
dataframe.filter(dataframe['pep_pickup_datetime'].like('2018%')).filter(da
taframe['filename'].like('yellow_tripdata_%'))

#skips further processing if the dataset is empty
if len(dataframe.head(1)) > 0:
     #selection of only the required fields for the computation
     dataframe = dataframe.select('PULocationID', 'DOLocationID',
'trip_distance')

     #caching the dataframe to speedup the next processing
```

```
    dataframe.cache()

    #grouping PULocationID and DOLocationID making the average of
total_amount and renaming the result in avg_trip_distances
    res_df = dataframe.groupBy('PULocationID',
'DOLocationID').agg({'trip_distance' :
'avg'}).withColumnRenamed('avg(trip_distance)',
'avg_trip_distances').orderBy('avg_trip_distances')

    #rounding the result of the average to two decimals
    res_df = res_df.withColumn('avg_trip_distances',
res_df['avg_trip_distances'].cast(DecimalType(32,2)))

    #transforming the row repeated data to one key (PULocationID) with
the list of DOLocationID in a single row format
    df1 =
res_df.groupBy('PULocationID').agg(F.collect_list('DOLocationID'))
    df1 = df1.withColumn('collect_list(DOLocationID)', concat_ws(", ",
'collect_list(DOLocationID)'))

    #transforming the row repeated data to one key (PULocationID) with
the list of avg_trip_distances in a single row format
    df2 =
res_df.groupBy('PULocationID').agg(F.collect_list('avg_trip_distances'))
    df2 = df2.withColumn('collect_list(avg_trip_distances)', concat_ws(",
", 'collect_list(avg_trip_distances)'))

    #merging the two dataset on PULocationID
    df_final = df1.join(df2, 'PULocationID', 'inner')

    #renaming the columns for better clarity
    df_final = df_final.withColumnRenamed('collect_list(DOLocationID)',
'DOLocationIDs')
    df_final =
df_final.withColumnRenamed('collect_list(avg_trip_distances)',
'avg_trip_distances')

    #shows the output to the terminal
    df_final.show()

    #write the result in a csv file, where
    #    >result is reported in a single file
    #    >headers are preserved
    #    >the previous resulting csv is overwritten
    df_final.coalesce(1).write.option("header","true").mode('overwrite').
csv(output_folder + "results_task3.csv")
else:
    print("Data not available")


#stops the session
```

```
spark.stop()
```