Big Data Analytics Programming : Assignment 2

Feiyang Tang (r0728168)

February 2020

1 Trip Data Exploration: Spark

Firstly, we need to figure out how to calculate distance using the coordinates we obtained from datasets. In this case, we use the following method:

Let λ_1, ϕ_1 and λ_2, ϕ_2 be the geographical longitude and latitude in radians of two points 1 and 2, and $\Delta\lambda$, $\Delta\phi$ be their absolute differences; then $\Delta\sigma$, the central angle between them, is given by the spherical law of cosines if one of the poles is used as an auxiliary third point on the sphere:

$$\Delta\sigma = \arccos\left(\sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 \cos(\Delta\lambda)\right) \tag{1}$$

The problem is normally expressed in terms of finding the central angle $\Delta\sigma$. Given this angle in radians, the actual arc length d on a sphere of radius r can be trivially computed as:

$$d = r\Delta\sigma \tag{2}$$

In this case, r denotes the radius of the earth and is equal to 6371.009 km. In this formula, the earth is approximated by a perfect sphere (even though the earth is more like an ellipsoid).

`2010_03.trips` has ride distance varies from 0km to 10814.01 km. Here we first answer the six queries before we dig into distribution.

1.1 Queries for 6 Questions

1. There are 563 unique taxis are present in the data.

```
taxi_id_lines = trips_lines.map(lambda x: x['taxi_id'])
taxi_id_lines = taxi_id_lines.distinct()
print("There are %d unique taxis" % taxi_id_lines.count())
```

2. Taxi 46 completes 759 trips.

```
taxi_id_lines = trips_lines.filter(lambda x: x['taxi_id'] == '46')
print("Taxi 46 complete %d trips." % taxi_id_lines.count())
```

3. Taxi 1399 participated in the most trips with 1253 trips.

```
taxi_id_cnt_lines = trips_lines.map(lambda x: (x['taxi_id'], 1))
taxi_id_cnt_lines = taxi_id_cnt_lines.reduceByKey(add)
res = taxi_id_cnt_lines.max(key=lambda x: x[1])
print("The taxi %s participated in the most trips." % res[0])
```

4. The longest trips is 10814.01 km.

```
def get_distance(start_latitude, start_longitude, end_latitude, end_longitude):
    delta_latitude = (start_latitude - end_latitude) * 2 * math.pi / 360
    delta_longitude = (start_longitude - end_longitude) * 2 * math.pi / 360
    mean_latitude = (start_latitude + end_latitude) / 2.0 * 2 * math.pi / 360
    R = 6371.009
    distance = R * math.sqrt(delta_latitude ** 2 + \
    (math.cos(mean_latitude) * delta_longitude) ** 2)
    return distance
dis_lines = trips_lines.map(lambda x: x['distance'])
print("The distance of the longest trip is %.2f" % dis_lines.max())
```

5. The top 3 longest taxi trips on average `(taxi_ID,avg_trip_length)` are ('633', 51.6090075482415), ('559', 26.20585200711091), ('1055', 14.349868915651847).

```
taxi_id_distance_lines = trips_lines.map(lambda x: (x['taxi_id'], x['distance']))
taxi_id_distance_lines = taxi_id_distance_lines.groupByKey()
taxi_id_distance_lines = taxi_id_distance_lines.mapValues(list).mapValues(
    lambda distances: sum(distances) / len(distances))
print(taxi_id_distance_lines.top(3, key=lambda x: x[1]))
```

6. On 20100312 the most trips start.

```
date_cnt_lines = trips_lines.map(lambda x: (x['start_date'], 1))
date_cnt_lines = date_cnt_lines.reduceByKey(add)
res = date_cnt_lines.max(key=lambda x: x[1])
print("On %s date the most trips start." % res[0])
```

1.2 Trip Distance Distribution

Now back to the trip distribution calculation. Through very simple queries I found out that only less than 10 records have more than 500km trips, which are very rare for taxi trips. The coordinates showed that most of the trips happened between SFO and Sacramento, Google Maps shows that the distance in between is around 160km according to Figure 1.

In this case, I am going to filter all trips longer than 160km out from our distribution plot and only include those who look valid (<160km). My script told me there are 11 records which have been filtered out from the plotting. The PySpark script (`Exercise1_python/main.py`) produces the following distribution plot as shown in Figure 2.

I also wrote a simple Python script in IPython to calculate the distribution as well as a Java script using Spark (`Exercise1.jar`). Table 1 below

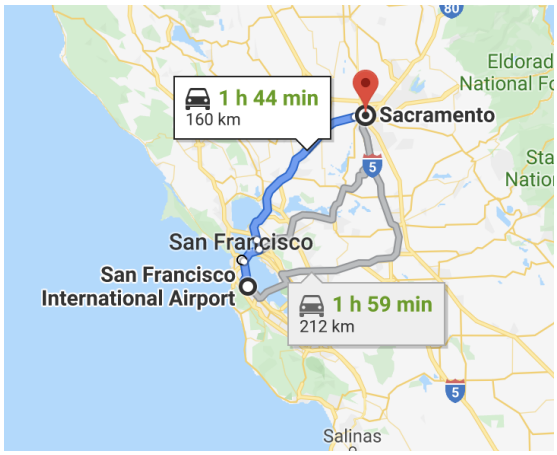

Figure 1: Trip trace from SFO to Sacramento (Google Maps)

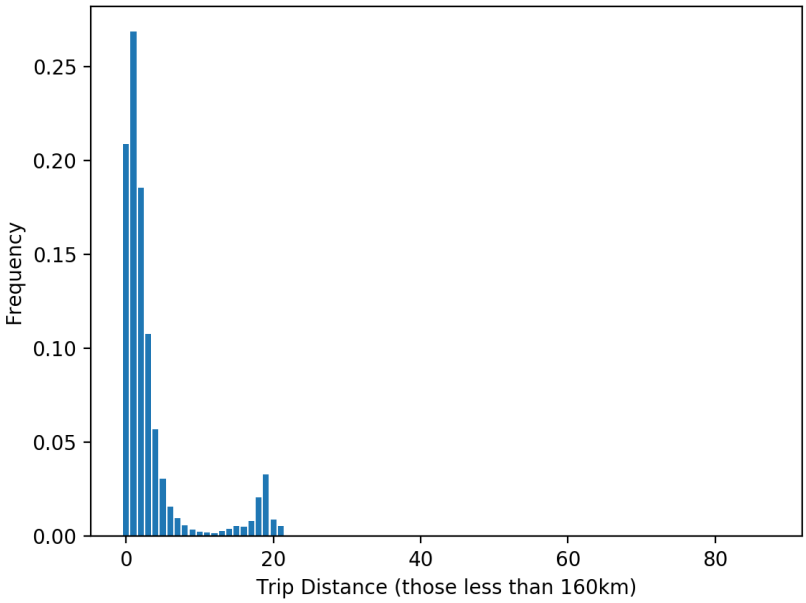

Figure 2: Distribution of the trip distance less than 160km

illustrates the speeds of using different programming languages on my local machine (`macOS 10.15.2 2.3 GHz Quad-Core Intel Core i5`). We have known that Spark is particularly good for iterative computations on large datasets over a cluster of machines. While Hadoop MapReduce can also execute dis-

tributed jobs and take care of machine failures etc., Apache Spark outperforms MapReduce significantly in iterative tasks because Spark does all computations in-memory. It is not surprised to see that Spark makes much faster calculations and Java with spark also exceeds PySpark.

Table 1: Speeds of different methods

Methods	PySpark	Java&Spark	Python
Time	2.29s	1.9s	38.65s

2 Computing Airport Ride Revenue: Hadoop

2.1 Reconstructing Trips

2.1.1 Erroneous Data-points

For trips reconstruction, we first need to identify those erroneous data-points. There are several criteria I consider as erroneous:

1. The trip timestamps from start to end exceed 30mins.
2. The average speed of this trip exceeds 200km/h.
3. Missing information for a record (e.g. `NULL` appears)

Here I give an example of the three erroneous record I identified. First on line 1, this looks like an empty ride, but in fact, as Google Maps shows below in Figure 3, this trip is only 900m and should not take several months.

1. Line 1: 9050,‘2009-12-21 20:43:39’,37.75156,-122.39403,‘E’,‘2010-03-02 12:43:03’,37.75083,-122.38969,‘E’
2. Line 91: 1055,‘2010-02-28 22:33:12’,37.78145,0.55,‘M’,‘2010-03-01 01:03:08’,37.77286,-122.43735,‘E’
3. Line 4470928: 1516,‘2010-03-08 10:38:02’,38.50407,-121.39151,‘E’,NULL,NULL,NULL,NULL

For very high speed trips, on line 91 gives an average speed more than 1000km/h which is unacceptable. And on line 4470928, it has several NULL in the record which makes no sense.

2.1.2 Program Structure

The program receives 4 parameters:

`<data> <trips dir><revenue dir><revenue aggregate type>`

Processing logic consists of two rounds of `MapReduce`: the first round of MapReduce is trip reconstruction.

`Mapper`: Input data format: `<LongWritable, Text>` and output data format is `<Text, TrackInfo>`.

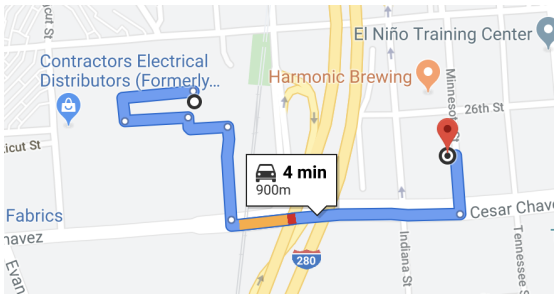

Figure 3: Sample erroneous record (Line 1)

There is no need to use `key`, `value` is a string, and the format of the saved data is `<taxi-id>, <start date>, <start pos(lat)>, <start pos(long)> , <start status>, <end date>, <end pos(lat)>, <end pos(long)>, <end status>`.

Parse the data and save it with the `TrackInfo` object. Here I need to explain the member variables of the object and the role of each variable.

`TrackInfo` implements the `Writable` interface, which can be serialised and de-serialised.

`Reducer`: Input data format: `<Text, TrackInfo>` and output data format is `<Text, Text>`.

The output data `key` is `taxi_id`, and `value` is `<taxi-id>, <month>, <revenue>, <start date>, <start pos(lat)>, <start pos(long)>, <end date>, <end pos(lat)>, <end pos(long)>`.

2.2 Computing Revenue

First things first, we are going to locate all trips that are related to airport, which means at least one of its start or end point locate within 1km distance of (37.62131°N, -122.37896°W). The following function is used to identify airport trips.

```
public boolean isWithinAirport1km() {
    double disStartToAir = getDis(startLatitude, startLongitude,
    37.62131f, -122.37896f);
    double disEndToAir = getDis(endLatitude, endLongitude,
    37.62131f, -122.37896f);
    return disStartToAir < 1.0 || disEndToAir < 1.0;
}
```

This second round of `MapReduce` calculates the revenue.In this case, a `combiner` is used to reduce the data network transmission.

`Mapper`: Input data format is `<Text, Text>`, Output data format is `<Text, Text>`.

According to the parameters passed by the user, choose to use the `month` or `day` as the `key`, and revenue as the `value`.

`Reducer`: Input data format is `<Text, Text>`, Output data format is `<Text, Text>`.

Add the values from the same key, then return the key. The total revenue of `combiner` is the same as `reducer`.

The Java programme spends 184s running the result for `2010_03.segments` on my local machine. When calculating the total revenue came from result files `part-r-00000` to `part-r-00019`, I used the following `hadoop` and `bash` command to first combine 20 records together then compute the total revenue:

```
hadoop dfs -text /user/r0728168/bdap/rev/* | hadoop fs -put - result

$ awk -F',' '{sum+=$2;} END{print "sum =",sum}' result
```

2.2.1 Quantify Efficiency

It is crucial to make sure that `MapReduce` could use the as much as the resource we have on the cluster. We already knew that we have 10 nodes on our cluster, the maximum allocation is `<memory:1536, vCores:4>`.

To maximise our efficiency on performing MapReduce, I use the following settings and list a few remarks based on those (README: `hadoop jar Exercise2.jar Main <data> <trips dir> <revenue dir> <revenue aggregate type>` revenue aggregate type can be `day` or `month`, `day` means to calculate the revenue by day, `month` means to calculate the revenue on a monthly basis):

```
hadoop jar Exercise2.jar Main
-D mapred.reduce.tasks=39
-D mapred.min.split.size=359496729
-D mapreduce.job.reduce.slowstart.completedmaps=0.2
-D mapreduce.map.cpu.vcores=4
-D mapreduce.reduce.cpu.vcores=4
-D mapreduce.map.memory.mb=512
-D mapreduce.reduce.memory.mb=768
/data/all.segments
/user/r0728168/bdap/trip
/user/r0728168/bdap/rev
month
```

- Number of Reducer: The number of reducers is normally 0.95 or 1.75 multiplied by (# of nodes)*(# of maximum containers per node). These guidelines are for big clusters (computation times per mapper/reducer should be around 5 minutes). The default setting for mapper and reducer are 20 tasks each. Also according to my primary experiments, the time spent on each reducer task is not equal i.e. the tasks are skewed. The reason behind this may be as we set `taxi_ID` as key and some taxis just have more records. To adapt to this, I decided to use prime numbers for # of reducer, so I went with 39 in the last.

- Number of Mapper: This is controlled by the split size. In this case I chose to use about 100 mappers for our case but this normally does not affect the result much.
- Numbers of Cores: It would be better to use all the cores available in our cluster so I decided to let all mappers and reducers use all available resources.
- Number of Memories: As we all know that mappers do not require a lot memory to use not like reducers. Reducers normally require a lot of memory to process the data came from mappers (I set memory for mappers as 512MB) which means we need to keep adjusting the number of reducer memory use to adapt to our specific tasks. In this case, I have tried from 256MB, 512MB and 768MB but finally went with 768MB.
- Start time of Reducers: We can also change when reducer task start when processing mapper tasks. This is a tricky one because if we start to process reducers too late or too early, it may cause problems for reducer to run efficiently as it needs to be fed with result from mappers. Typically, I like to keep `mapred.reduce.slowstart.completed.maps` above 0.9 if the system ever has multiple jobs running at once. This way the job does not hog up reducers when they are not doing anything but copying data. In this case, we only have one job running at a time, doing 0.1 would probably be appropriate.

2.2.2 Speed & Revenue Results

The speed comparison table below Table 2 gives illustration on how much time it spent on calculating revenues for two datasets.

Table 2: MapReduce Speed on Local and Cluster Mode

Dataset	`all.segments`	`2010_03.segments`
Local Time Cost	N/A	184.18s
Cluster Time Cost	560.63s = (482.63 + 78)s	90.34s = (53.34 + 37)s

It is obvious that cluster significantly reduced the processing time. As the task consists two parts: trip reconstruction and revenue calculation, the results on cluster are listed as two parts which corresponds to the two separated tasks.

The result of revenue computation is $1,100,870 for 03/2010 the whole month and the entire revenue for `all.segments` is $24,083,400.

2.2.3 Revenue Illustrations

The below figures show the distribution of revenues for `all.segment` dataset and `2010_03.segment` dataset. We can discover the trends within them by analysing the distribution plots.

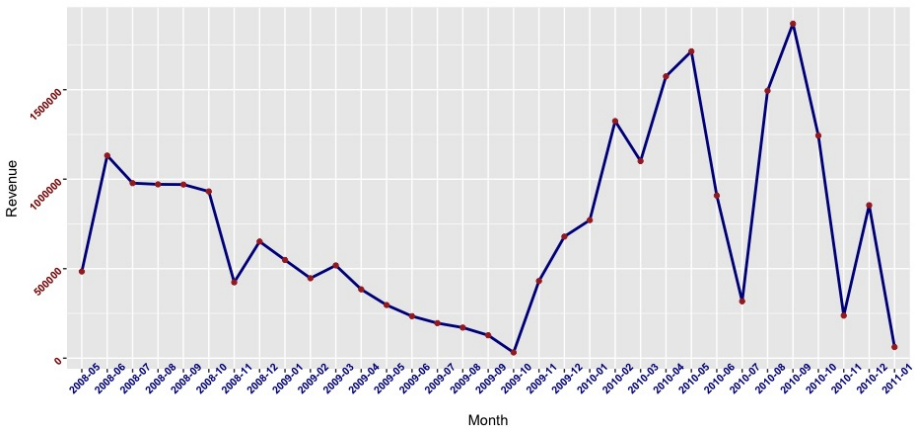

Figure 4: Revenue for all.segment Dataset (by month)

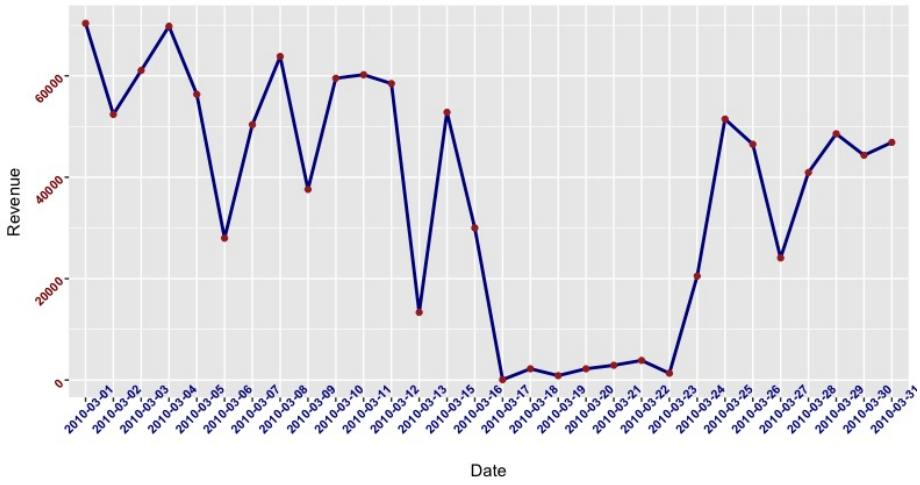

Figure 5: Revenue for 2010_03.segment Dataset (by day)