Report: Assignment 2

Sarah Binta Alam Shoilee

August 2019

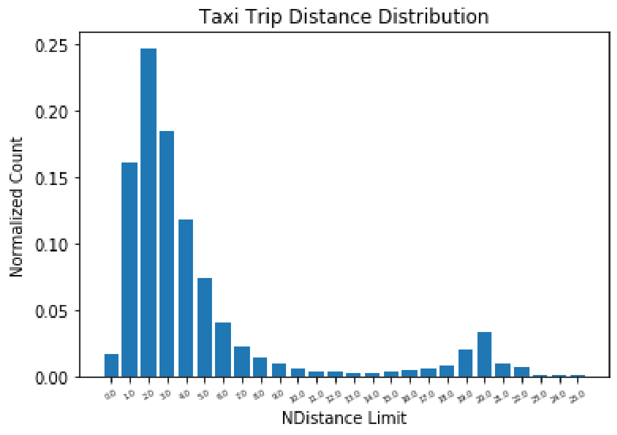
Exercise 1. The first experiment is dedicated to compare the performance between any standalone object-oriented program and Spark frame work. For this experiment, we are given with GPS tracks of taxis in San Francisco which consists trip records of a month from 2010. As a object orientated programming language, Java is chosen to implement the experiment. Due to unreliable data point from the GPS tracker, we need to make some assumption before distance calculation. Here, we only consider those trips that only start from San Francisco city and end in the same city. By doing so, we get rid of any data point that indicates impossible trips such as trip to the earth core. Then, we use the flat surface formula to count the distance of a potential trip. Due to the limitation of flat surface formula, it performs best up to 20km distance. To aid our experiment, we limit our distance to 25km. At the end, we count the bucket size and plot trip distribution. Since our distance is limited to 25 kilometers, we take 25 buckets to map each data to a bucket whose index is closer to the distance value. The algorithm for implementation is described as follow:

```
Result: Plot trip lengths
Read input file;
while InputFile is not empty do
    Parse each data field from input line;
    if Trip start location and end location resides in San Francisco City then
        Calculate distance;
        if Distance <25 then
            Add distance to the list;
        else
            Ignore;
        end
    else
        Ignore;
    end
    Calculate distance count;
    Round distance to closest integer bucket and bucket++;
end
Count each bucket size;
```

Algorithm 1: Calculate Trip Length Distribution

Next, we followed the same algorithmic logic, but using Spark framework. Upon initialization, input file data is distributed over the spark cluster. Each data will go through same location and speed as previous algorithm. We write a map function for location and speed check along with distance calculation. We mark every invalid data in this process. Using the Spark object filter, we get rid of the invalid data points. The map function only returns the distance as key. Then, map each distance to a ¡bucket, distance¿ pair by rounding the distance to it's nearest integer(bucket index). Then, we concatenate and sort all data based on bucket index(key). Next, we substitute the bucket counts with normalized count. Our distribution plot(Figure: 1) also support the fact of having limited bucket, as the bars are near zero(0) after 22km and higher frequencies with 8km range. Both of the above mentioned

Figure 1: San Francisco Taxi Trip Length Distribution over March, 2010



algorithm implements the same logic using two different approaches. In our test data, java implementation tends to execute in less time then spark method. After running few trials, we get the execution time $\approx$ 2.5 seconds for spark implementation and less than 1 second for object-oriented implementation. It is not surprising for the current case because the Java implementation simply read the file and then write operation directly from the file which supposed to execute in O(n) time and in this case n is not too big. On the other hand, distributed file processing uses parallel computation which takes some time in distributing file and also on concatenating the resulted files. DFS is more scalable and suitable for large amount of data.

Exercise 2. In the next part of the experiment, we are interested to count taxi revenue generated from the airport trips. In this phase, we are going to work on the records of segmented GPS data of taxi location in various time, not necessarily, each segment will be a part of a trip. The segments are indicated with a status that shows if it is empty or full. Keeping this in mind, we are ultimately interested to calculate revenue, we can discard the data which is empty in both of it's start and end status. Getting rid of unnecessary data will also clear some spaces during map-reduce operation. To avoid errornous data, we bounded our segment location by San Francisco city coordinate space: 38.2033° to the north; 37.1897° to the south; -122.6445° to the east & -121.5871° to the west. Any data point beyond that will be discarded automatically, we don't have to think about impossible location separately. For example, the following input data contains impossible start location: $633, 2010-03-0214:32:37, 37.77661, 0.06666, M, 2010-03-0214:33:36, 37.76931, -122.40692, M.$We also calculated segment speed where we found out some segment even crosses 4000km/h or even more which is technically not possible. To avoid such circumstances, we keep our segment speed limited to 200km/h. We also check for misformatted data by using a try-catch block. We expect the each field of the input as follows: <Integer> <DateTime> <Double> <Double> <E/M> <DateTime> <Double> <Double> <E/M>

All these segment compatibility check is done within the the first map operation. We get rid of all the unnecessary segment that might not be useful later. In this map operation, we pass the offset byte of each input line as a key and each line data another key value pair. The return key of the mapper is a composite which contains TaxiID and StartTime and anything else other than this two fields are returned as "Text" value. The purpose of having a composite key is to sort data using both the information. In reduce phase, we intend to concatenate segments to form a trip. To do that, we have to find out consecutive segments of a certain taxi which eventually requires a sort operation. Given huge amount of data, sorting data within the reducer might not be a scalable option. By overriding the partitioner and grouping function, we tried to make sure that all the same taxi record resides in same reducer. Depending on the passenger status, we concatenated each sorted consecutive segments and forms a trip record. This trip records is emited as value of reducer and we make the TaxiID to be the key at this phase.

Next we go for another map-reduce operation to calculate the revenue from Taxi trip. To do this, in map phase we only map those trips whose can be considered as airport trip. To find out potential airport trip, we implemented function to determine whether start or end location of a trip resides in 1 kilometer distance of Airport geographical coordinate or not. The trips those satisfies the condition be a airport trip is set as value and the year-month data is sent as a key to aid reducer to group over these value and that helps to calculate revenue in each month period. Finally, after the reduce operation it returns the year-month as key and total revenue for that months as value. Calculating the total revenue requires another map-reduce operation. Considering that our data has been downsized a lot, using another map-reduce will be a waste of resources. For this reason, we calculate the total revenue using another scripting language and present our findings using Figure: 2 and Table:1.

Figure 2: Airport Trip Revenue over each month from San Francisco Taxi Data-set

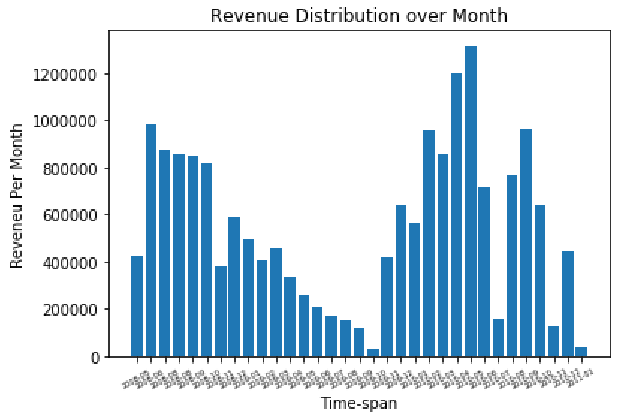


Table 1: Airport Trip Revenue over each month

Year-Month	**Reveneu**
2008-05	427386.0
2008-06	983502.0
2008-07	872947.0
2008-08	858851.0
2008-09	847430.0
2008-10	819473.0
2008-11	377470.0
2008-12	590632.0
2009-01	493747.0
2009-02	403994.0
2009-03	456965.0
2009-04	333549.0
2009-05	258803.0
2009-06	206990.0
2009-07	172997.0
2009-08	153344.0
2009-09	119691.0
2009-10	29360.0
2009-11	421457.0
2009-12	637484.0
2010-01	566103.0
2010-02	958546.0
2010-03	855775.0
2010-04	1202098.0
2010-05	1314542.0
2010-06	714622.0
2010-07	160872.0
2010-08	766811.0
2010-09	967089.0
2010-10	643024.0
2010-11	127436.0
2010-12	443929.0
2011-01	37169.0
Total	18224088.0