Assignment 3

Versions of frameworks

Scala: 2.11.0

Spark: 2.3.1

Running of the program

The jar file produces a output text file using the SON algorithm based on the input parameters passed as arguments to the file. In order to run the file the following commands should be used:

```
spark-submit –class Utkarsh_Gera_SON Utkarsh_Gera_SON.jar <input file path> <support_threshold> <output file path>
```

When the program is about to exit it will also print the time taken for its execution.

As mentioned in the problem description the output file contains the item sets in sorted order with each different item set combinations in terms of their size are separated by a blank line. Ex:

```
(also), (back), (best), (dont), (even), (food), (friendly), (get)

(food, good), (food, place), (food, service)
```

There will be a space after each "," as seen the announcement posted by Anriudh. Also the texts are treated as string so even the numbers like 1 or 10 are treated as string (reference to post – 78 by Niharika).

If the program finds any issue in the arguments or you don't provide these 3 arguments/provide more than 3 arguments the program will exit with a proper message like:

```
Need at three arguments - Input file, support threshold and output file
```

Explanation of the Implementation

SON algorithm is based on the fact that frequent item sets are calculated on chunks which fit the memory size by applying algorithms like sampling, apriori. After this first pass is made the second pass is just to cross check how many of the candidates are actually frequent item sets. The whole point of optimization lies around the first pass in order to achieve small execution time.

By using the mapPartitioner function we apply the frequent item sets calculation algorithm in my case apriori on the chunks of the RDD. But before passing these chunks to mapPartitioner function make sure it is in the format of bucket -> Set of items in that bucket and not the expanded or flattened version as given in the input.

First calculate the frequent singleton items in the form of Map where each item is the key and holds a set of buckets where it appeared as its value for future reuse. But make sure to filter out the infrequent items for this chunk based on modified threshold: (given threshold / number of partitions). Rest is easy, keep a track of all the items used in the item set creation (pairs, triples etc) for previous round and use it to generate further subsets of a size like 2, 3, 4.. till the moment when no item set combination gets produced which makes the item set, used for previous round item combination set creation, zero. Then collect them in a list and pass it for round two which makes a pass on the complete dataset and calculates these potential combination sets actual count across the whole data set and then filters them out based on the actual given threshold. Then just convert item sets to string by sorting them amongst the set as well as amongst the whole line containing the same number of item set combinations.

Performance of the program

1. Small file

Support Threshold	Execution Time
500	11 sec
1000	9 sec

2. Large file

Support Threshold	Execution Time
100000	285 sec
120000	264 sec

Bonus for Problem 3

Ques. Describe why did we need to use such a large support threshold and where do you think there could be a bottleneck that could result in a slow execution for your implementation, if any.

Ans. If we increase the size of a dataset such as this then the probability of occurrence of a frequently word will nearly remain same however its absolute number will increase to counter the bigger number in the denominator of the probability.

Since the dataset is for the yelp reviews one can easily expect that even by increasing the size of data set the occurrence of new words will have less probability than the occurrence of frequently used words since these are user reviews or views. Had the threshold been kept to a lower value then we would have found out humongous amount of frequent and hence the size of the threshold and the size of the data set go hand in hand, after all we need frequent items in a large data set.

I implemented the algorithm in 3 different ways and each way had its own advantages, The bottleneck in the SON algorithm is the pass 1 when you are calculating the frequent item sets in that chunk assuming they have same distribution in the other chunks. In this implementation we make a hash map for item to corresponding baskets it appears in so that it becomes easy to find the baskets in which the combinations produced using that item set occurred rather than iterating over the chunk again and again.

Let’s take a case of an item set that has a frequency like 80 to 90% of the given threshold so when the SON algorithm takes and distributes the dataset approximately 90% of the chunks will report it frequent as the threshold for each chunk will be dropped to given (threshold/ number of partitions). In worst case nearly all the chunks will report it frequent(assuming it gets distributed among partitions the most optimum way) and will use it to find more combinations using this item. But when the results are combined and those will be dropped in the second pass, which will make all that time used in first pass by this item useless. Now let’s assume a case there are more of this kind of item in the dataset, so unless we don’t have a good threshold or all the items in the dataset have quite the same frequency/densely distributed around the threshold, we will end up in unnecessary calculation which will cause a bottleneck in the my implementations performance.

Second such case would be that there are items that are frequent in particular chunk but not it other chunks and this kind of case occurs with majority of the partitions/chunks. I mean the partitions don’t have a good distribution throughout the database or it by chance happens that the majority of chunks that get formed have too many local maximum item sets and their combinations which are not globally frequent. This is a rare case but this would also cause bottleneck as they will also lead to formation of frequent item set combinations in that chunk which late on second pass would get removed.