Bashar Alhafni **INF-553**
USC ID: 9065856238 **Fall 2018 Assignment 3**

Versions used for this assignment:

Spark: 2.3.1
Scala: 2.11.0

Detailed description of my algorithm:

Initially, I read the data into an RDD of (review_index, word) and then I group these tuples by review_index. So I end up with an RDD of the following format:
(review_index, (word1, word2, word3…))

I pass this RDD as partitions to the apriori function using mappartitions(). The apriori function will take the data partition and create a map of sets, where the key is a set of items and the value is a set of baskets containing these items. Initially the map is empty, so I have to iterate through the partition to initialize it with singletons. To get the frequent singletons set for this particular partition, I scale the threshold by multiplying it by the size of the partition and dividing it by the total number of elements in all partitions. After that, the main part of the algorithm comes. I iterate through the singletons map to find all the combinations of pairs, triples, etc. until I can't find anymore combinations. Then, I save all of these potential frequent items into an iterable (i.e. iterable of maps) and I emit (frequent items, 1) as part of the mapper of the first phase of the SON algorithm. After that, I reduce the results by key, to combine same frequent items from multiple partitions. Now, the output of this reduce is passed to the mapper of the second phase of the algorithm.
The mapper of the second phase will filter out the actual frequent items among the entire data set because the first phase found the frequent items among the **partitions** but not among the entire data set. After that, I reduce the output of the mapper by key and sum up the number of baskets for this item. At the end, I filter out all the items that aren't frequent and I write the results to a file.

Problem 1:

My code will execute by running the following **command**:

For a support threshold of 30:

```
spark-submit --class Bashar_Alhafni_SON Bashar_Alhafni_SON.jar
/Users/alhafni/Desktop/repos/inf553_data_mining/hw3/inf553_assignment3
/Data/yelp_reviews_test.txt 30
Bashar_Alhafni_SON_yelp_reviews_test30.txt
```

For a support threshold of 40:

```
spark-submit --class Bashar_Alhafni_SON Bashar_Alhafni_SON.jar
/Users/alhafni/Desktop/repos/inf553_data_mining/hw3/inf553_assignment3
/Data/yelp_reviews_test.txt 40
Bashar_Alhafni_SON_yelp_reviews_test40.txt
```

Problem 2:

My code will execute by running the following **command:**

For a support threshold of 500:

```
spark-submit --class Bashar_Alhafni_SON Bashar_Alhafni_SON.jar
/Users/alhafni/Desktop/repos/inf553_data_mining/hw3/inf553_assignment3
/Data/yelp_reviews_small.txt 500
Bashar_Alhafni_SON_yelp_reviews_small500.txt
```

Time taken for this part of the problem:

```
time taken: 14 sec(s)
```

For a support threshold of 1000:

```
spark-submit --class Bashar_Alhafni_SON Bashar_Alhafni_SON.jar
/Users/alhafni/Desktop/repos/inf553_data_mining/hw3/inf553_assignment3
/Data/yelp_reviews_small.txt 1000
Bashar_Alhafni_SON_yelp_reviews_small1000.txt
```

Time taken for this part of the problem:

```
time taken: 7 sec(s)
```

Problem 3:

NOTE: for this problem, I had to increase the memory that is being used by java as part of the spark-submit to avoid getting an OutOfMemoryError.

Bashar Alhafni **INF-553**
USC ID: 9065856238 **Fall 2018 Assignment 3**

My code will execute by running the following **command:**

For a support threshold of 100000:

```
spark-submit --driver-memory 4g --class Bashar_Alhafni_SON
Bashar_Alhafni_SON.jar
/Users/alhafni/Desktop/repos/inf553_data_mining/hw3/inf553_assignment3
/Data/yelp_reviews_large.txt 100000
Bashar_Alhafni_SON_yelp_reviews_large100000.txt
```

Time taken for this part of the problem:

```
time taken: 315 sec(s)
```

For a support threshold of 120000:

```
spark-submit --driver-memory 4g --class Bashar_Alhafni_SON
Bashar_Alhafni_SON.jar
/Users/alhafni/Desktop/repos/inf553_data_mining/hw3/inf553_assignment3
/Data/yelp_reviews_large.txt 120000
Bashar_Alhafni_SON_yelp_reviews_large120000.txt
```

Time taken for this part of the problem:

```
time taken: 264 sec(s)
```

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.

The reason why the threshold is large, is because of the size of the input data. If the threshold was small, then we might not be able to fit the data in memory even after partitioning. Thus, increasing the threshold will decrease the number of frequent items.

The main bottleneck of my implementation is the part where I am finding the combinations of the potential frequent items as part of the Apriori algorithm. It's a huge bottleneck because I am iterating through the potential items twice, so I can generate pairs, triples, etc.