Mohammadreza Taremi-96101902

Assignment 3

---

1)

A. If $S(B) = \frac{Support(B)}{N}$ then $1 - S(B)$ is a metric which somewhat shows the probability of the absence of B in the basket, Therefore **Conviction** can be defined as: comparing the probability of A appearing without B if they were independent with the actual frequency of A appearing without B.

B.

$$Conf(A \to B) = \Pr(B|A) = \frac{\Pr(A,B)}{\Pr(A)} \; != \; Conf(B \to A) = \Pr(A|B) = \frac{\Pr(A,B)}{\Pr(B)}$$

$$Lift(A \to B) = \frac{Conf(A \to B)}{S(B)} = \frac{\frac{\Pr(A,B)}{\Pr(A)}}{\frac{P(B)}{N}.N} = \frac{\frac{\Pr(A,B)}{\Pr(B)}}{\frac{P(A)}{N}.N} = \frac{Conf(B \to A)}{S(A)} = Lift(B \to A)$$

$$Conv(A \to B) = \frac{1 - S(B)}{1 - Conf(A \to B)} \; != \; \frac{1 - S(A)}{1 - Conf(B \to A)} = Conv(B \to A)$$

Therefore, only Lift is symmetric.

C.

Based on the definition of Conf:

$$Conf(A \to B) = \frac{\Pr(A,B)}{\Pr(A)}$$

If $\Pr(B)$ is near 1, then $Conf(A \to B)$ is near 1 as well, but this does not necessarily guarantees a strong association rule; thus this imposes a drawback on confidence metric. Lift and Conviction do not possess such a defect, because they include $\Pr(B)$ explicitly not directly.

2)

Item	1	2	3	4	5	6	7	8	9	10
Prob	1	0.5	0.34	0.25	0.2	0.17	0.14	0.12	0.11	0.10

At least one of the objects listed above can be found in each of the baskets. If we assume that the selection of item *i* is independent from the selection of item *j*, the probability of containing both items is: $\Pr(i)\Pr(j)$

Therefore we have:

- All the baskets contain item 1
- 0.5 of the baskets contain item 2
- A third contain item 3 and so on

Frequent itemsets: {1},{2},{3},{4},{5},{6},{7},{8},{9},{10},{1,2},{1,3},{1,4}

{1,5},{1,6},{1,7},{1,8},{1,9},{1,10},{2,3},{2,4},{2,5},{1,2,3},{1,2,4},{1,2,5}

3)

a. First of all, I tried to read the dataset using *sc.textFile* function. As we know, this function joins each of the dataset's columns and returns a string format. Therefore, I wrote a function named *parseLine* which gets a string row of our dataset and returns ORIGINE_CAR_KEY, PASS_DAY_TIME, and DEVICE_CODE respectively.

```
def parseLine(line):
  fields = line.split(',')
  DEVICE_CODE = int(fields[0])
  ORIGINE_CAR_KEY = int(fields[2])
  PASS_DAY_TIME = pd.to_datetime(fields[6]).day
  return (ORIGINE_CAR_KEY,PASS_DAY_TIME,DEVICE_CODE)
```

In the next step, we have to set key/value for our triplet. We take the first two elements as keys and the third one as value, then by using *gropuByKey()* function, we group the DEVICE_CODE parameter**,** which is a sign of car's location, based on a car's plate and traversing date.

```
traffic_rdd = traffic_rdd.map(lambda x:(tuple([x[0],x[1]]),x[2]))
traffic_rdd_gb = sorted(traffic_rdd.groupByKey().mapValues(list).collect())
```

The output of the code snippet above is shown below:

```
[((7631925, 1), [631795]),
 ((7631926, 1),
  [631350, 212501, 900258, 900217, 900243, 900221, 900243, 900258]),
 ((7631929, 1), [803001]),
 ((7631931, 1), [900244, 900207]),
 ((7631935, 1), [175, 900191, 22010072]),
 ((7631942, 1), [900246, 212601, 631351, 212601, 900255]),
```

The first element of the key, is ORIGINAL_CAR_KEY, the second one is the day of week, and the values are the DEVICE_CODEs of the cameras which have taken a picture from cars. For example, there is a car with ORIGINAL_CAR_KEY of *7631925*, in day *1* that has been observed by a camera with *631795* DEVICE_CODE.

b)

In order to implement the A-priori algorithm we must do the following steps:

- Reading data with PySpark
- Parsing data to Spark RDD objects
- Finding support values of items with "MapReduce"
- Deciding minimum support value
- Creating the following support tables with "MapReduce"

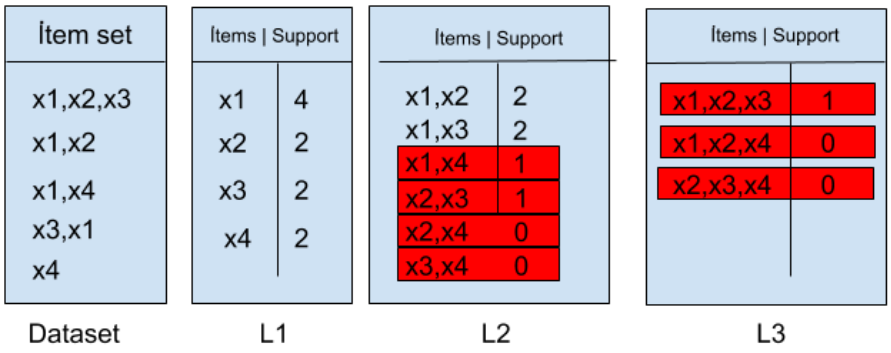


Figure 1

- Calculating confidence values
- Deciding which path is a frequent path(High Confidance)

In part a, we have implemented the first two steps, now we have to go on and do the others.

Now that we have extracted the paths (DEVICE_CODE is representing the paths), we must create a RDD which includes all of them. In other words, wehave a car with its specific ORIGINAL_CAR_KEY value (can be considered as customer) and also the paths it has taken (can be considered as the products which buyer chooses while shopping), and with possessing such a dataset we can solve this problem. To clarify, we need to extract the values from <key,value> pairs of the *traffic_rdd_gb* RDD. Indeed, they consist our baskets!

We do this by the following code snippet:

```
path_baskets = []

#erecting the baskets of paths
for i in range(len(traffic_rdd_gb)):
  path_baskets.append(traffic_rdd_gb[i][1])
```

```
] path_baskets
```

```
[[631795],
 [631350, 212501, 900258, 900217, 900243, 900221, 900243, 900258],
 [803001],
 [900244, 900207],
 [175, 900191, 22010072],
```

Each item in this list represents a basket. For example, the second element includes all of the paths (products) which a car (a customer) has taken.

Now in order to be able to impose MapReduce function on our dataset, we have to parse it to a RDD. We do this by the following single line of code:

```
path_rdd = sc.parallelize(path_baskets)
```

Getting First Support Values of Items for A-priori

We need to combine all of the RDD objects into one long object. After doing so, we can calculate each unique item's frequency. These frequencies will be our first supports values. First we can convert every item to a "tuple" object and add "1" as a second item of "tuple". We can sum these values by using the "reduceByKey" method. By summing tuple's second numbers we can get every unique item's frequency (how many time occurs on customers' transactions or how many times a path has been chosen by a driver). We will also need to list unique items in the feature sections. So, we can also obtain unique items by using the "distinct" method.

```
#parse dataset to rdd
path_rdd = sc.parallelize(baskets)

#unique items in our dataset
uniquePaths = path_rdd.distinct()

#add 1 to each of paths
supportRdd = path_rdd.map(lambda path: (path , 1))

# Sum of values by key
supportRdd = supportRdd.reduceByKey(lambda x,y:x+y)
```

```
supportRdd.collect()
```

```
[(631350, 788),
 (900258, 12731),
 (900244, 67756),
 (22010072, 4879),
 (900246, 53033),
 (100, 9483),
 (900278, 8322),
```

In the output, we can see that path 631350 has been chosen 788 times. Now we can compute each of the paths first support values.

Min Support Value

In order to decide which item-sets will stay in the support tables, we need to define a min support value. **We can choose min support value as the minimum frequency that is in the first support values array** (table).

```
# Define minimum support value
minSupport = supports.min()

#filter first supportRDD with minimum support
supportRdd = supportRdd.filter(lambda item: item[1] >= minSupport)

#Creating base RDD, this rdd will be updated in every iteration
baseRdd = supportRdd.map(lambda item: ([item[0]] , item[1]))

supportRdd = supportRdd.map(lambda item: item[0]) #We will this rdd in the next sections
                                                  #in order to create combinations of items
supportRddCart = supportRdd
```

Building the Figure 1 Table

In this part, we will use a while loop which generates combinations of items. There will be different support tables that will be created in every loop. This while loop will end when there isn't any combination whose support value more than min support value. As a matter of fact, we are creating figure 1 tables in this part.

We already created a "supportRdd" object which only includes the first table's item set without support values (which only have one item in each row). Now, we will use this RDD in a while loop to combine it with unique items in order to create other support tables.

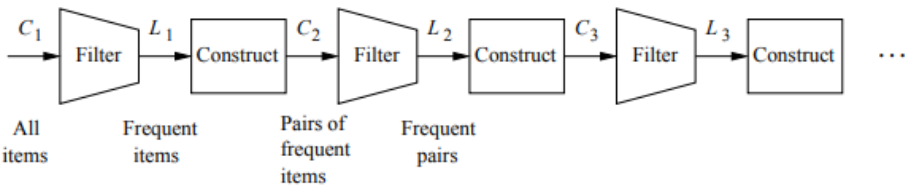
To clarify, we have already created the table L_1. Now we have to erect the others. To do so, we can use *cartesian()* method to generate different combinations of items. After that, we have to remove the duplicated combinations and therefore we need a function such as *removeReplica():*

```
def removeReplica(record):

    if(isinstance(record[0], tuple)):
        x1 = record[0]
        x2 = record[1]
    else:
        x1 = [record[0]]
        x2 = record[1]

    if(any(x == x2 for x in x1) == False):
        a = list(x1)
        a.append(x2)
        a.sort()
        result = tuple(a)
        return result
    else:
        return x1
```

A-priori algorithm will filter every combination table according to min support value. When there is no item set while loop will end. In other words, first we create pairs of itemset, triplets and … and then we filter them based on their support. The following diagram is a visualization of this algorithm:



And the following code snippet, is performing both **Construction** and **Filtering**.

```
#in this cell,we will create the combinations of roads

combination_length = 2

while(supportRdd.isEmpty() == False):
  #to generate various combinations
  combined = supportRdd.cartesian(uniquePaths)
  #remove the duplicated ones
  combined = combined.map(lambda item: removeReplica(item))

  combined = combined.filter(lambda item: len(item) == combination_length)
  combined = combined.distinct()

  combined_2 = combined.cartesian(lblitems)
  combined_2 = combined_2.filter(lambda item: all(x in item[1] for x in item[0]))

  combined_2 = combined_2.map(lambda item: item[0])
  combined_2 = combined_2.map(lambda item: (item , 1))
  combined_2 = combined_2.reduceByKey(lambda x,y: x+y)
  combined_2 = combined_2.filter(lambda item: item[1] >= minSupport)

  baseRdd = baseRdd.union(combined_2)

  combined_2 = combined_2.map(lambda item: item[0])
  supportRdd = combined_2
  combination_length += 1
```

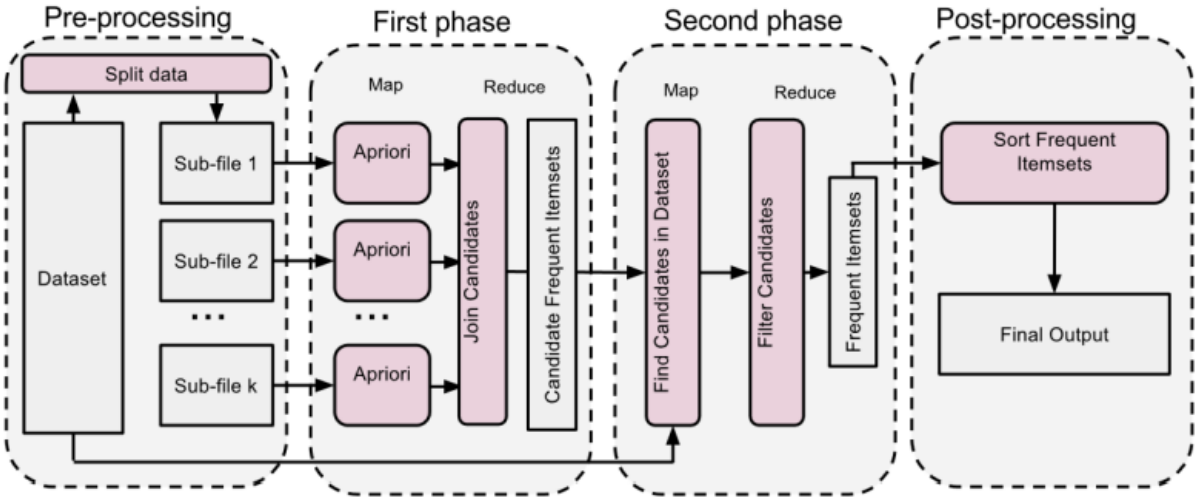
Finally, we can calculate the confidence values. The (pair) results derived from a **sampled** dataset are:

[900244]	[900142]	0.300300
[900142]	[900244]	0.423729
[114]	[631776]	0.645161
[900276]	[900158]	0.909091
[900158]	[900276]	1.041667
...	...	...
[143]	[212802]	3.333333
[22010043]	[22010078]	8.333333
[100701119]	[22010040]	1.369863
[100701119]	[22010080]	1.369863
[22010057]	[22010060]	2.941176

c) The SON algorithm lends itself well to a parallel-computing environment. Each of the chunks can be processed in parallel, and the frequent itemsets from each chunk combined to form the candidates.

We can distribute the candidates to many processors, have each processor count the support for each candidate in a subset of the baskets, and finally sum those supports to get the support for each candidate itemset in the whole dataset.

As a matter of fact, we have to implement the following model:



To sum up, we have to implement the following steps:

- Divide the dataset into 3 subdatasets
- Run the A-priori algorithm on each subdataset based on the new threshold to find the candidates
- Combine the achieved frequent itemsets (candidates)
- Filter the final set based on the original threshold

We do the dividing step by the following codes:

```
#part 3-SON algorithm

#new support threshold
new_minSupport = supports.min()/3

#splitting the original rdd into 3 sub rdd
#path_rdd --> path_rdd_1, path_rdd_2, path_rdd_3

path_rdd_1 = lblitems.sample(False, 0.33, 1)
path_rdd_2 = lblitems.sample(False, 0.33, 2)
path_rdd_3 = lblitems.sample(False, 0.33, 3)
```

And the new support threshold is the original one divided by three.

In order to have my results in lower executing time, I used the codes below to get my step1 candidates:

```
#run the a-priori algorith on each dataset
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori

dataset_1 = path_rdd_1.collect()
te = TransactionEncoder()
te_ary = te.fit(dataset_1).transform(dataset_1)
df = pd.DataFrame(te_ary, columns=te.columns_)
results_1 = apriori(df, min_support=new_minSupport)
```

In the final step, we must combine the results and filter the dataset based on the minSupport value, I used the following codes to do so:

```
final_rdd=sc.union([results_1, results_2, results_3])
final_rdd = final_rdd.filter(lambda item: item[1] >= minSupport)
```