ASSOCIATION RULE MINING

USING:

1. **APRIORI ALGORITHM : {HASH BASED AND TRANSACTION REDUCTION}**
2. **FP GROWTH {TOP- DOWN}**

ABOUT THE DATASETS USED:

We tried various datasets, but boiled down to two datasets and compared the performance on both of them. The two datasets we used are:

1. BMSWebView2 (Gazelle) (KDD CUP 2000) : This dataset was used in KDD CUP 2000. It contains clickstream data from an e-commerce company. It has a total of 77,512 transactions, with 3340 unique items. The average transaction length is 4.62. The items in the transactions were present in numeric format.
   Example of what the transaction looks like is shown below:

```
55283 -1 55287 -1 56181 -1 89453 -1 -2
81955 -1 81959 -1 81975 -1 -2
55595 -1 -2
84731 -1 84795 -1 228995 -1 -2
55779 -1 -2
228795 -1 -2
82835 -1 82875 -1 82879 -1 88683 -1 88953 -1 -2
```

   Each transaction ends with a '-2' at the end, and within each transaction the individual items separated by a '-1' string, and the corresponding numbers in between are the unique items numbers.

2. retail_utility : It contains customer transactions from an anonymous Belgian retail store. (source FIMI: http://fimi.ua.ac.be/data/). There are 88,162 transactions present in this data, with a total unique item count of 16,470 and the average number of items per transactions is 10.3.

```
49 71 72 73
40 74 75 76 77 78 79 80
37 39 40 42 49 80 81 82
83 84 85
42 86 87 88 89
40 49 90 91 92 93 94 95 96 97 98 99 100 101 102
37 39 40 49 90
40 42 103 104 105 106 107 108 109
```

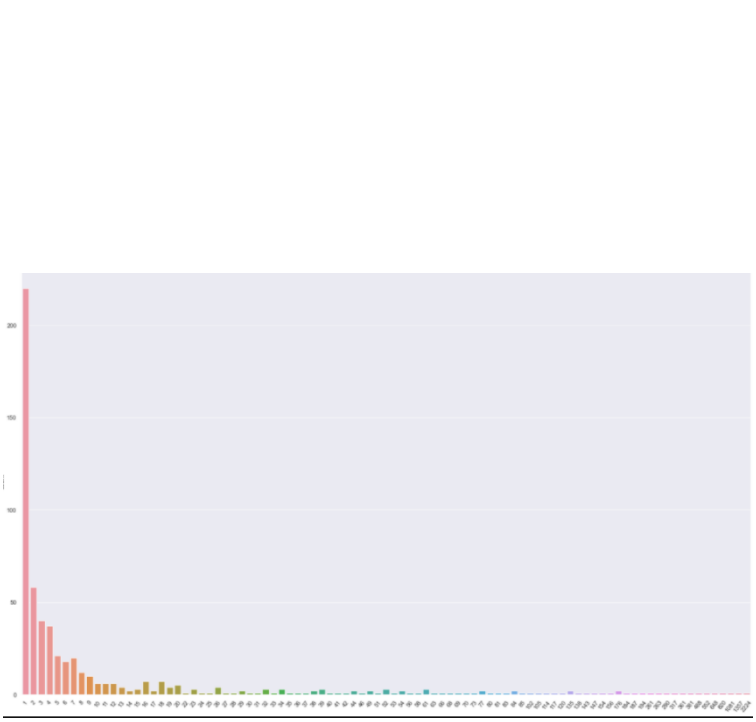
   An example of the retail utility data can be seen above, we can see the transactions, each item is space separated and the individual transactions are line separated.

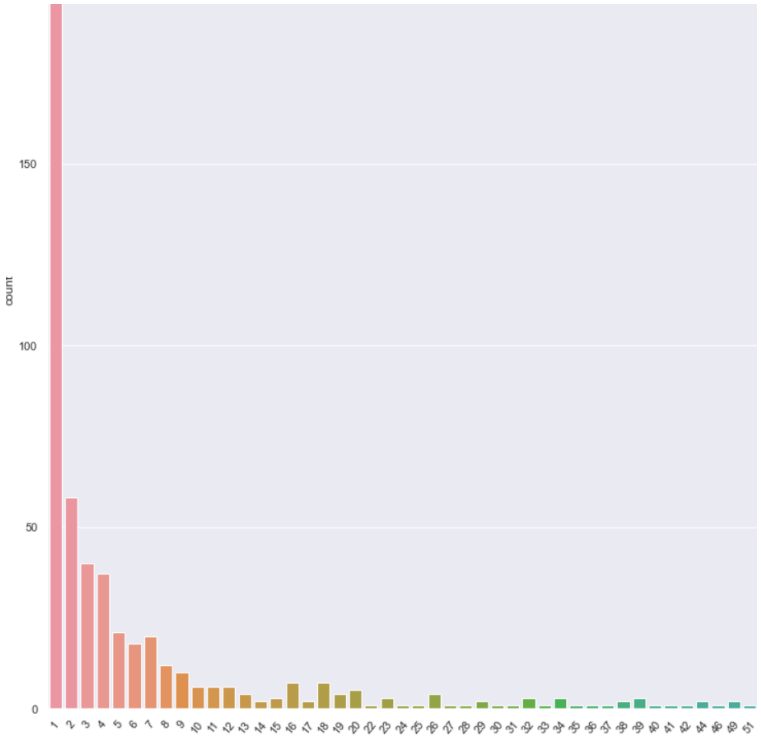
STATS ABOUT THE DATA:

Both of the dataset we used are quite suitable for association rule mining. We can see the distributions of frequencies of items for both of them.

1. Retail_Utility data: We know that there are very few items who dominate across all the transactions, we can confirm this by seeing the number of occurrences of each frequency for different items follows.

   Below we 2 images, second image is the zoomed in version of the first.





From both of the above images, we see that most of the items occur only once across all the transactions, so occurrence of single occurrence items is highest, and the frequency distribution of is severely right skewed. In the right image, we can see the above facts more clearly. There are 583 unique frequencies of frequency items.

The most frequently occurring along with the items are shown below:

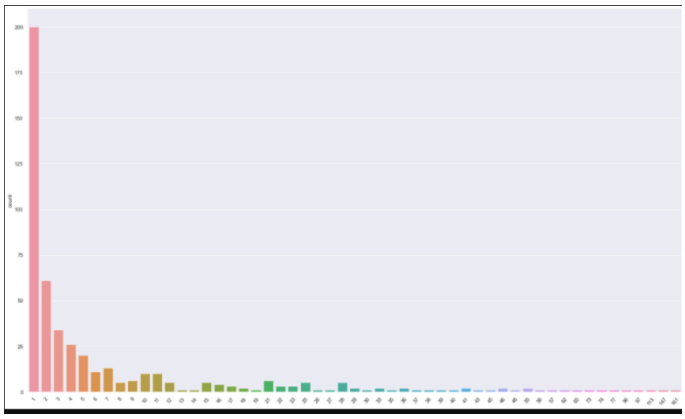
```
top_items
```

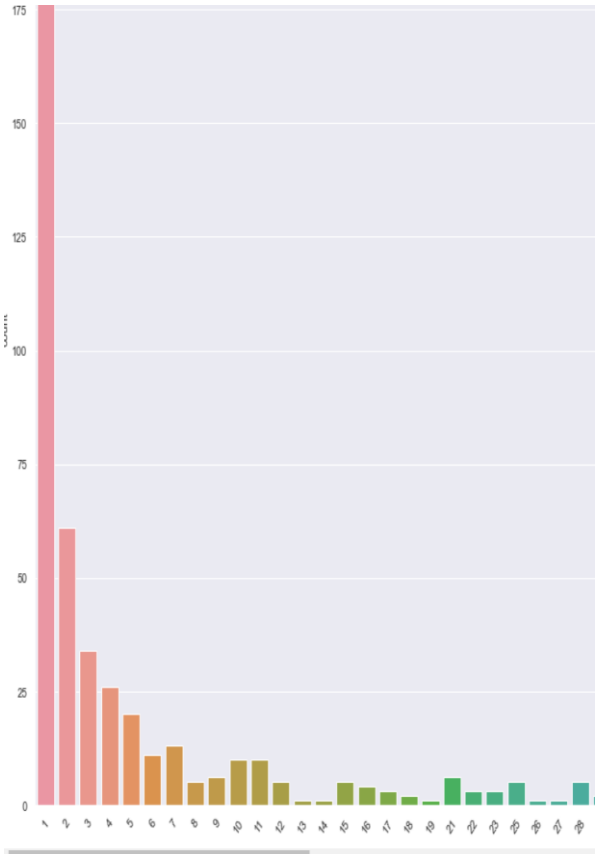
```
[(40, 50675),
 (49, 42135),
 (39, 15596),
 (33, 15167),
 (42, 14945),
 (66, 4472),
 (90, 3837),
 (226, 3257),
 (171, 3099),
 (238, 3032),
 (37, 2936),
 (111, 2794),
 (311, 2594),
 (102, 2237),
 (476, 2167),
 (272, 2094),
 (414, 1880),
 (439, 1863),
 (1328, 1786),
 (148, 1779),
 (271, 1734),
 (2239, 1715),
 (80, 1600),
 (61, 1489),
 (534, 1487),
 (256, 1474),
 (12926, 1467),
 (1147, 1426),
 (186, 1376),
 (10, 1372),
 (16011, 1316),
 (124, 1302),
```

From the above list we can see that item number 40 occurs the greatest number of times. Out of 88 thousand transactions, it alone occurs in more than 50 thousand transactions, followed by 49, 39,33,42,66. Upon our implementation of association rule mining algorithms we'll find out the combinations of these items.

2. BMSWebView2 (Gazelle) (KDD CUP 2000) : This dataset also follows similar trend as the above retail utility data.

The distribution followed same pattern like above and most real world datasets.





From both of the above images, we see that most of the items occur only once across all the transactions, so occurrence of single occurrence items is highest, and the frequency distribution of is severely right skewed. In the right image, we can see the above facts more clearly. There are 475 unique frequencies of frequency of items.

The most frequently occurring items are shown as follows:

```
(56769, 1602),
(55287, 1477),
(55543, 1434),
(55551, 1423),
(55875, 1331),
(55367, 1296),
(55343, 1293),
(55831, 1291),
(82719, 1255),
(55335, 1227),
(84731, 1226),
(55887, 1125),
(56057, 1111),
(55275, 1105),
(203729, 1088),
(55307, 1056),
(55463, 1046),
(55339, 1046),
(55843, 1032),
(55871, 1031),
```

On comparing it with the previous set of data, we can see that the frequency of the top few elements is more uniformly occurring. Out of 77.5 thousand transactions, they comparatively occur a smaller number of times roughly around ~1.94 percent of transactions.

THE MINING ALGORITHMS

1. APRIORI ALGORITHM

Apriori is an algorithm for frequent item set mining and association rule learning over relational databases. It proceeds by identifying the frequent individual items in the database and extending them to larger and larger item sets as long as those item sets appear sufficiently often in the database.

The crux of working of the Apriori algorithm looks as follows:

For each transaction T in the dataset:

For each candidate itemset, C:

Check to see if C is a subset of T

If so, increment the count of C

For each candidate itemset:

If the support meets the minimum, keep this item

Return list of frequent item sets.

There are 2 implementations which we tried for implementing the Apriori algorithm:

i.) Transaction Reduction
ii.) Hash Based Technique

1. Transaction Reduction

A transaction that does not contain any frequent k-itemsets cannot contain any frequent (k+1)-item sets. Therefore, such a transaction can be marked or removed from further consideration because subsequent database scans for j-itemsets, where j > k, will not need to consider such a transaction.

It required first to find the frequency of all the items, and then proceed ahead with all the transactions to find the association rules, at each step (k) there only remained candidates which have higher support than min_support ,we removed the transactions containing them separately in a list of discarded transactions, and for every iteration updated these lists (candidate itemsets (k+1) and transactions containing these itemsets discarded transactions(k)). We continued this till a convergence criterion is reached, for our implementation the convergence criterion is till the length of current frequent itemset size reaches 0. It means no k+1 size of itemsets will be frequent, so no need to check them.

The snippets from our implementation is shown below:

```
convergence=False

while convergence==False:
    self.Candidates.update({k : self.join_itemsets(self.Itemsets[k-1])})

    if(verbose==1):
        print(f'Table Candidates {k} \n ')
        self.print_table(self.Candidates[k],[self.count_itemset_occurance(it,self.transactions) for it in self.Candidates[k]])

    frequency,support_count,new_discarded=self.get_freuqent_itemsets(self.Candidates[k],
                                                                     self.transactions,self.min_support,self.Discarded_itemsets)

    self.update_tables(k,frequency,support_count,new_discarded)

    if len(self.Itemsets[k])==0:
        convergence=True
    else:
        print(f'Table Itemsets {k} \n ')
        self.print_table(self.Itemsets[k],self.Itemset_support_counts[k])

    k+=1
```

Here, we start with convergence as False, and continue till the length of current frequent item set size reaches 0 which ends the loop by marking convergence=True.

In the third part of the code we see a function '*get_frequent_itemsets*' is called, this finds the frequently occurring items, and marks the itemsets which are not frequent. It also checks if the item has been discarded before.

An excerpt from our implementations shows this idea.

```
discarded_before=False

if(k > 0):
    for item in previously_discarded[k]:
        if(set(item).issubset(set(itemsets[s]))):
            discarded_before=True
            break

if not discarded_before:
    count=self.count_itemset_occurance(itemsets[s],transactions)
    if count/num_transactions >=minimum_support:
        L.append(itemsets[s])
        supp_count.append(count)
    else:
        new_discarded.append(itemsets[s])
```

If the current item is a part of set of previously discarded transaction of size[k], then by following the apriori property and superset of this item cannot be a frequent item, and we come out of the loop immediately, else if none of the items in the current itemset have been discarded before, then we continue, and find the frequencies of the individual items, which we already found while beginning this algorithm, it required only one pass, if the support of current itemset is greater than the minimum support count, then we insert it into the current set of frequent item sets, else we add it to the set of discarded transaction.

This process is fast, because the length of itemsets don't increase more than 4-5 , and is hence computationally very cheap, almost constant, the only case in which it can take time is when the size of discarded transaction grows in every call to this function, this can happen if none of the items are frequent, which can happen *when support count is very low.* In worst case, if the support count is less than 1/|size of transactions| ,then all the itemsets will be infrequent, but that is just senseless and is not applicable in real-world, so in amortized cost, it is almost constant, or takes as much time as searching in the set of discarded transactions , which can also be improved by using a hash map to store the discarded transactions etc, or a set which is internally implemented as a Red Black tree, and will take log(number of items in the discarded set).

After finding the k frequent item sets for different k, we can now start to find the association rules.

For every nonempty subset S of frequent item sets,

output the rule X ⇒ (X−S), if

Support_count(X) /support_count(S) ≥ min conf, where min conf is the minimum confidence threshold.

```
for i in range(1,len(self.Itemsets)):
    for j in range(len(self.Itemsets[i])):
        s = self.powerset(set(self.Itemsets[i][j]))

        s.pop()#remove subset with all the items
        for z in s:
            S = set(z)
            X=set(self.Itemsets[i][j])
            X_S=set(X-S)

            support_x=self.count_itemset_occurance(X,self.transactions)
            support_x_s=self.count_itemset_occurance(X_S,self.transactions)

            confidence=support_x / support_x_s

            if support_x >= self.min_support:
                self.create_rule_set(X,X_S,S,confidence,support_x)
```

Since, we are using only those transactions which contain the frequent item sets, this process is fast as well. The only time taking step is finding the power set, and counting the itemset occurance in the frequent transactions.

2. Hash Based Technique

A hash-based technique can be used to reduce the size of the candidate k-itemsets , Ck , for k > 1.When scanning each transaction in the database to generate the frequent 1-itemsets, L1, we can generate all the 2-itemsets for each transaction, hash (i.e., map) them into the different buckets of a hash table structure, and increase the corresponding bucket counts. A 2-itemset with a corresponding bucket count in the hash table that is below the support threshold cannot be frequent and thus should be removed from the candidate set.

We implemented the above idea, using python's inbuilt *defaultdict()*.

The calls to the function which implement the above implementation is shown below:

```
def fit_transform(self,transaction,min_support=0.1):
    self.transactions=transactions
    self.min_support=min_support
    self.min_support_frequency=self.min_support*len(self.transactions)

    self.find_one_frequent_candidates()
    self.find_one_frequent_itemsets()

    self.find_two_frequent_candidates()
    self.find_two_frequent_itemsets()

    self.find_three_frequent_candidates()
    self.find_three_frequent_itemsets()

    final_results=self.find_final_frequent_itemsets()
```

First we find the transaction database for finding the 1- frequent candidates followed by 1-frequent item sets, and similarly for 2-freqeunt and 3 frequent, each time reducing the size in k-frequent for k+1 frequent.

The way these functions work is, for example when k=2, i.e. we have to generate 2-frequent item sets,

For that , we'll need to generate the candidates for 2-frequent item sets, while generation itself, if we find an item which is not in 1-frequent item sets which we found in the previous iteration, we simply ignore it and continue with next iteration, even though it requires database scan for checking the item is eligible for being a candidate item set or not, it is very fast when generating frequent itemset from the candidate item sets, because now the algorithm only looks at the item sets which have a higher chance of being k+1 frequent.

Lets, see how the above explanation looks like when implemented.

Finding the candidates:

```
def find_two_frequent_candidates(self):
    for transaction in tqdm(self.transactions):
        for index1 in range(len(transaction)-1):
            if(transaction[index1] not in self.frequent_one_itemsets):
                continue
            for index2 in range(index1+1,len(transaction)):
                if(transaction[index2] not in self.frequent_one_itemsets):
                    continue
                candidate_pair=self.merge_itemsets(transaction[index1],transaction[index2])
                self.candidate_itemset_pairs[candidate_pair]+=1
```

Finding the frequent item sets from the candidates:

```
def find_two_frequent_itemsets(self):
    for item_pair in self.candidate_itemset_pairs.keys():
        if(self.candidate_itemset_pairs[item_pair] > self.min_support_frequency):
            self.frequent_itemset_pairs[item_pair]=self.candidate_itemset_pairs[item_pair]
```

As we can see, every thing is present in hash tables, the computations are extremely fast for a reasonable min_support count, it only proceeds to find k+1 item set candidates if and only if the current set of candidates were present in k frequent item sets. This makes the calculation very fast.

We do similar technique for finding the 3- frequent item sets.

```
for transaction in tqdm(self.transactions):
    for index1 in range(len(transaction)-2):
        if(transaction[index1] not in self.frequent_one_itemsets):
            continue
        for index2 in range(index1+1,len(transaction)-1):
            if(transaction[index2] not in self.frequent_one_itemsets):
                continue
            pair1=self.merge_itemsets(transaction[index1],transaction[index2])
            if pair1 not in self.frequent_itemset_pairs:
                continue

            for index3 in range(index2+1,len(transaction)):
                if(transaction[index3] not in self.frequent_one_itemsets):
                    continue

                all_pairs=self.create_pairsets(transaction[index1],
                                               transaction[index2],
                                               transaction[index3])
                for pair in all_pairs:
                    if(pair not in self.frequent_itemset_pairs):
                        continue

                itemset_triple=self.merge_itemsets(transaction[index1],
                                               transaction[index2],
                                               transaction[index3])

                self.candidate_itemset_triples[itemset_triple]+=1
```

```
def find_three_frequent_itemsets(self):
    for itemset in self.candidate_itemset_triples.keys():
        if(self.candidate_itemset_triples[itemset] > self.min_support_frequency):
            self.frequent_itemset_triples[itemset]=self.candidate_itemset_triples[itemset]
```

First we see in one frequent item sets, if not present move on to next item without executing anything below it, if it is frequent, check in 2- frequent item sets and apply the same above criteria, if and only if both the conditions are satisfied then proceed to generate 3- item set candidates, for finding the 3-frequent item sets. Now, from the 3 item set candidates, keep only those who have support greater than required min support and output the results.

2. FP GROWTH ALGORITHM

The apriori algorithm which we saw in the previous section requires scan of the entire database or parts of it(in optimized cases) every time, multiple times whenever we need to find k+1 size frequent item sets , the repeated database scans cause the algorithm to be slow.

We need an algorithm which doesn't scan the database every time.

We now see an algorithm which does the above and performs faster than Apriori on most real life datasets.

The first scan of the database is the same as Apriori, which derives the set of frequent items (1-itemsets) and their support counts (frequencies).

An FP-tree is then constructed as follows. First, create the root of the tree, labelled with "null." Scan database D a second time. The items in each transaction are processed in L order (i.e., sorted according to descending support count), and a branch is created for each transaction.

The FP-tree is mined as follows. Start from each frequent length-1 pattern (as an initial suffix pattern), construct its conditional pattern base (a "sub-database," which consists of the set of prefix paths in the FP-tree co-occurring with the suffix pattern), then construct its (conditional) FP-tree, and perform mining recursively on the tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

We implemented the FP tree algorithm in Python.

The implementation begins with specifying the number of transactions as the min support the item sets should occur in, the algorithm internally takes this as min_support count and proceeds further taking it into account.

```
fp_tree_object=build_fp_tree(transactions,4170)
```

The above snippet shows this implementation, it calls the *build_fp_tree* function with the set of transactions and tells it out of the total number of transactions, it should occur in 4170 of them.

Then we call this object to build the fp tree which returns a header table and the tree root, the header table contains the addresses of the beginning.

The algorithm proceeds ahead with calling a *create_tree* function, which is implemented as follows:

```
self.remove_infrequent_items()

frequent_items=self.get_frequent_itemsets()
if(len(frequent_items)==0):
    return None,None

for keys in self.header_table:
    self.header_table[keys]= [self.header_table[keys],None]

self.fp_root=fp_tree_node('0',1,None)

for transaction,transaction_occurances in self.transactions.items():
    temp_transactions=defaultdict()

    for item in transaction:
        if item in frequent_items:
            temp_transactions[item]=self.header_table[item][0]

    if len(temp_transactions)>0:
        ordered_items=self.get_ordered_itemsets(temp_transactions)
        self.update_fp_tree(self.fp_root,ordered_items,transaction_occurances)
return self.fp_root,self.header_table
```

The algorithm begins with removing infrequent items, which removes the items which do not qualify the min support criterion. Then we initialize the header table and build the tree as stated above by recursively updating the header table and creating new node whenever a transaction with frequent item sets is found.

```
if ordered_items[0] in tree_node.child_node:
    tree_node.child_node[ordered_items[0]].update_item_frequency(occurance_cnt)
else:

    tree_node.child_node[ordered_items[0]]=fp_tree_node(ordered_items[0],occurance_cnt,tree_node)

    if not self.header_table[ordered_items[0]][1]:
        self.header_table[ordered_items[0]][1]=tree_node.child_node[ordered_items[0]]
    else:
        #update headertable
        target_node=tree_node.child_node[ordered_items[0]]
        start_node=self.header_table[ordered_items[0]][1]

        while(start_node.next_node != None):
            start_node=start_node.next_node
        start_node.next_node=target_node
```

After creating the FP tree, we need to mine it to find the frequent item sets.

The itemset mining begins by using the header table sorted in non increasing order.

```
sorted_items=self.get_ordered_itemsets(header_tab)
for base in sorted_items:
    temp_frequent_set=suffix_set.copy()
    temp_frequent_set.add(base)

    self.frequent_itemset_list.append(temp_frequent_set)

    conditional_base=self.find_suffix_path(base,header_tab[base][1])
```

The get_ordered_itemsets returns the reverse sorted list of header table items, i.e. we start from the largest and move in non increasing order down the table.

```
def get_ordered_itemsets(self,itemsets):
    sorted_items=sorted(itemsets,key=lambda key : itemsets[key][0],reverse=True)
    return sorted_items
```

We then find the suffix path by moving down the tree. An example of what the tree looks like is shown below.

```
Item : Φ  Frequency: 1  Parent : None
  Item : 33  Frequency: 2784  Parent : <__main__.fp_tree_node object at 0x00000271C3E7FC88>
    Item : 66  Frequency: 96  Parent : <__main__.fp_tree_node object at 0x00000271C130E388>
  Item : 40  Frequency: 47860  Parent : <__main__.fp_tree_node object at 0x00000271C3E7FC88>
    Item : 39  Frequency: 3977  Parent : <__main__.fp_tree_node object at 0x00000271C130E7C8>
      Item : 42  Frequency: 1020  Parent : <__main__.fp_tree_node object at 0x00000271C130E648>
        Item : 33  Frequency: 170  Parent : <__main__.fp_tree_node object at 0x00000271C130E048>
          Item : 66  Frequency: 10  Parent : <__main__.fp_tree_node object at 0x00000271C130EBC8>
        Item : 66  Frequency: 51  Parent : <__main__.fp_tree_node object at 0x00000271C130E048>
      Item : 33  Frequency: 415  Parent : <__main__.fp_tree_node object at 0x00000271C130E648>
        Item : 66  Frequency: 10  Parent : <__main__.fp_tree_node object at 0x00000271C1313E48>
      Item : 66  Frequency: 92  Parent : <__main__.fp_tree_node object at 0x00000271C130E648>
    Item : 49  Frequency: 28184  Parent : <__main__.fp_tree_node object at 0x00000271C130E7C8>
      Item : 39  Frequency: 5976  Parent : <__main__.fp_tree_node object at 0x00000271C130E108>
        Item : 42  Frequency: 1971  Parent : <__main__.fp_tree_node object at 0x00000271C130EA88>
          Item : 33  Frequency: 447  Parent : <__main__.fp_tree_node object at 0x00000271C130E3C8>
            Item : 66  Frequency: 27  Parent : <__main__.fp_tree_node object at 0x00000271C130E9C8>
          Item : 66  Frequency: 103  Parent : <__main__.fp_tree_node object at 0x00000271C130E3C8>
        Item : 33  Frequency: 783  Parent : <__main__.fp_tree_node object at 0x00000271C130EA88>
          Item : 66  Frequency: 46  Parent : <__main__.fp_tree_node object at 0x00000271C130EB88>
        Item : 66  Frequency: 149  Parent : <__main__.fp_tree_node object at 0x00000271C130EA88>
      Item : 42  Frequency: 5278  Parent : <__main__.fp_tree_node object at 0x00000271C130E108>
        Item : 33  Frequency: 1190  Parent : <__main__.fp_tree_node object at 0x00000271C130EC88>
          Item : 66  Frequency: 95  Parent : <__main__.fp_tree_node object at 0x00000271C130E8C8>
        Item : 66  Frequency: 321  Parent : <__main__.fp_tree_node object at 0x00000271C130EC88>
      Item : 66  Frequency: 860  Parent : <__main__.fp_tree_node object at 0x00000271C130E108>
```

We can see that the tree begins with the root as null (phi) and has no parent, the subsequent nodes go down the path as they appear, each node stores the address of its parent as well.

We then generate the conditional pattern bases by finding the suffixes, and create the tree based on those suffixes and recursively find the frequent item sets.

```
conditional_patterns={}

while(node is not None):
    suffix_path=list()
    self.find_root(node,suffix_path)

    if(len(suffix_path)>1):
        conditional_patterns[str(suffix_path
    node=node.next_node
return conditional_patterns
```

We keep traversing the next node and find the suffixes for the frequent item sets.

The fp tree node structure looks like the following :

```
class fp_tree_node:
    def __init__(self,item_name,frequency,parent):
        self.parent=parent
        self.item_name=item_name
        self.frequency=frequency
        self.child_node=defaultdict()
        self.next_node=None

    def update_item_frequency(self,frequency):
        self.frequency+=frequency

    def display_fp_tree(self,index=1):
        print('  '*index + f' Item : {self.item_name}  Frequency: {sel
```

TASK 3: PERFORMANCE AND RESULTS

1. Retail Utility Dataset:

We tried this dataset and compared performance and results on the above two implementations and mlxtend's library implementation of Apriori.

We compared the performance on different min_support values, which are
[0.001, 0.01, 0.05, 0.1, 0.5, 0.95, 0.1]

For threshold=0.05,
The results are as follows:
a.) Transaction Reduction

```
Itemset | Frequency
[33]  :  15167
[39]  :  15596
[40]  :  50675
[42]  :  14945
[49]  :  42135
[66]  :  4472
```

```
Table Itemsets 2

Itemset | Frequency
[33, 40]  :  8455
[33, 49]  :  8034
[39, 40]  :  10345
[39, 49]  :  7944
[40, 42]  :  11414
[40, 49]  :  29142
[42, 49]  :  9018
```

```
Table Candidates 3

Itemset | Frequency
[33, 40, 49]  :  5402
[39, 40, 49]  :  6102
[40, 42, 49]  :  7366
```

Final results of Transaction Reduction.

	Itemset	Support
0	{40, 33}	0.095903
2	{33, 49}	0.091128
4	{40, 39}	0.117341
6	{49, 39}	0.090107
8	{40, 42}	0.129466
10	{40, 49}	0.330551
12	{49, 42}	0.102289
14	{40, 33, 49}	0.061274
20	{40, 49, 39}	0.069213
26	{40, 49, 42}	0.083551

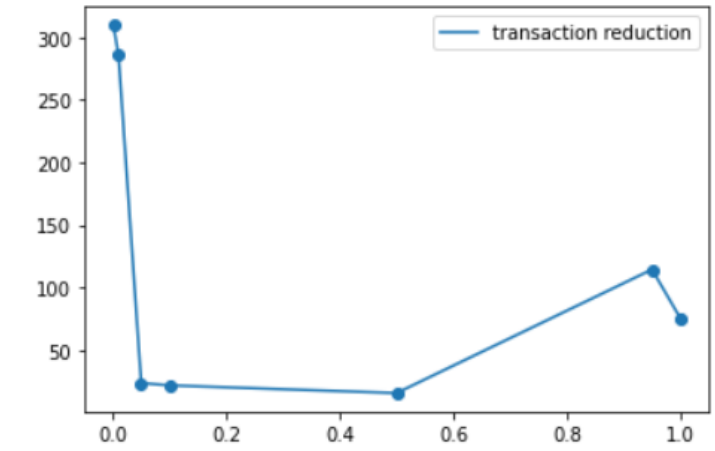
b.) Results from Hashing

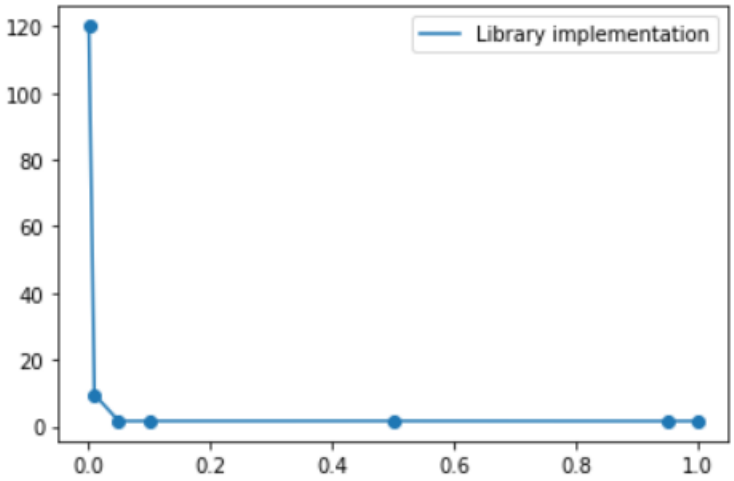
	support	itemsets
0	0.172036	(33)
1	0.176902	(39)
2	0.574794	(40)
3	0.169517	(42)
4	0.477927	(49)
5	0.050725	(66)
6	0.095903	(40, 33)
7	0.091128	(33, 49)
8	0.117341	(40, 39)
9	0.090107	(49, 39)
10	0.129466	(40, 42)
11	0.330551	(40, 49)
12	0.102289	(49, 42)
13	0.061274	(40, 33, 49)
14	0.069213	(40, 49, 39)
15	0.083551	(40, 49, 42)

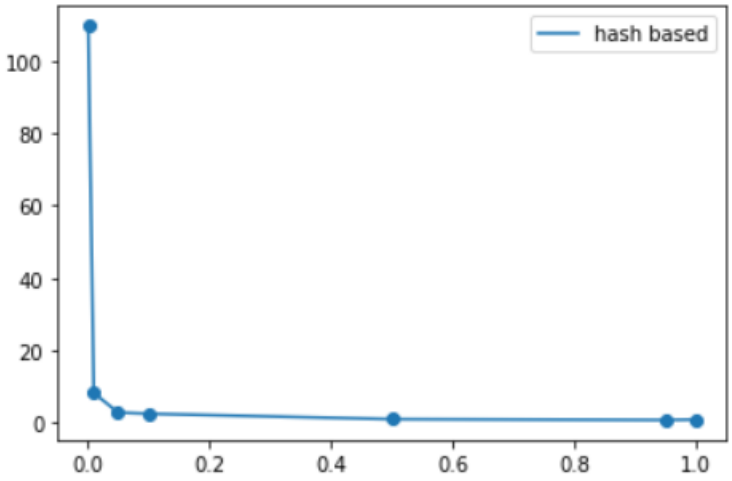
c.) Results from mlxtend's llbrary implementation

	Itemset	Frequency
0	66	4472
1	[33, 40, 49]	5402
2	[39, 40, 49]	6102
3	[40, 42, 49]	7366
4	[39, 49]	7944
5	[33, 49]	8034
6	[33, 40]	8455
7	[42, 49]	9018
8	[39, 40]	10345
9	[40, 42]	11414
10	42	14945
11	33	15167
12	39	15596
13	[40, 49]	29142
14	49	42135
15	40	50675

We see that our implementation results exactly match with the results of the library implementation of Apriori, it means our algorithm works correctly. Now let's see their performance.
A comparison of times taken by all the three implementations can be seen for the above specified values of thresholds.



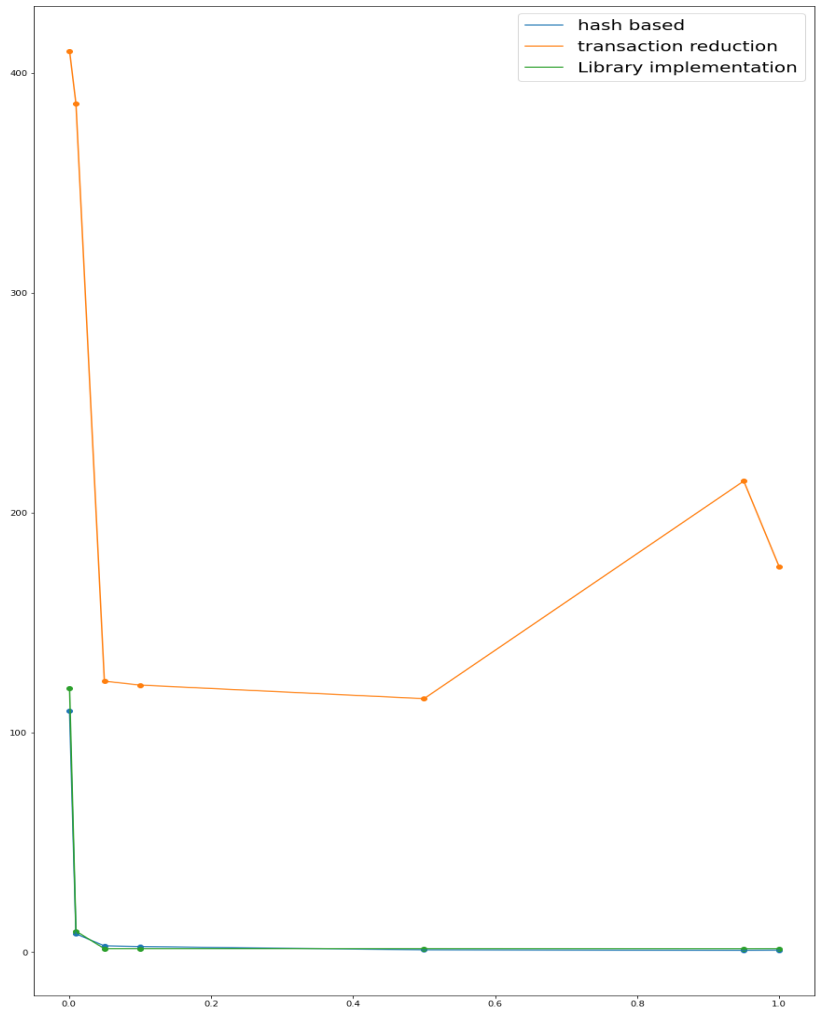




We can see that scale of time taken by transaction reduction is almost 3 times than that of hash based and library implementations.

A plot of all three together can give a better idea of performance difference in all the three implementations.

On the x axis we see minimum support which we mentioned earlier and on the y axis is the time taken in seconds.



We see that the transaction reduction takes a lot of time as compared to the other implementations and the library implementation and hash-based implementation take almost identical time to execute. This indicates the library implementations also use some kind of hash table structure in their code.

Results by fp-growth

We tried this dataset and compared performance and results on the above two implementations and mlxtend's library implementation of FP-growth.

We compared the performance on different min_support values, which are
[0.001, 0.01, 0.05, 0.1, 0.5, 0.95, 0.1]
For comparison of results on threshold=0.05, same as apriori we see them below:

a.) Library Implementation

	support	itemsets
0	0.172036	(33)
1	0.574794	(40)
2	0.176902	(39)
3	0.169517	(42)
4	0.477927	(49)
5	0.050725	(66)
6	0.095903	(40, 33)
7	0.091128	(33, 49)
8	0.061274	(40, 33, 49)
9	0.117341	(40, 39)
10	0.090107	(49, 39)
11	0.069213	(40, 49, 39)
12	0.129466	(40, 42)
13	0.102289	(49, 42)
14	0.083551	(40, 49, 42)
15	0.330551	(40, 49)

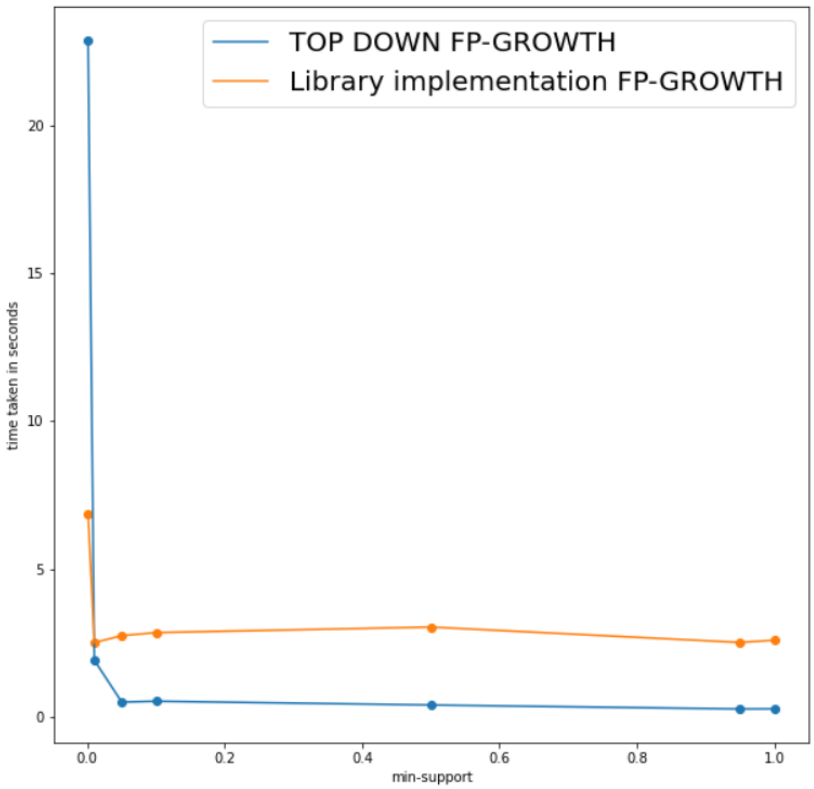
We used mlxtend's library implementation of FP Growth, the results are consistent with our apriori results and mlxtend's apriori .
Also, the time taken by fp growth is lower than apriori for all cases, this is because apriori spends a lot of its time scanning the database multiple times, but this is not the case with the fp growth algorithm.

b.) Top Down Implementation

```
[{40},
 {49},
 {40, 49},
 {39},
 {39, 40},
 {39, 49},
 {39, 40, 49},
 {42},
 {40, 42},
 {42, 49},
 {40, 42, 49},
 {33},
 {33, 40},
 {33, 49},
 {33, 40, 49},
 {66}]
```

The results are same as all the above results of apriori and library fp growth, this means our implementation is correct.

The performance comparison of top down and library implementation can be seen below.



We see that for very low threshold our implementation takes higher time than the library implementation but after sometime the time taken by it is lesser than the one of mlxtend's.

2. BMSWebView2 (Gazelle) (KDD CUP 2000)

We tried this dataset and compared performance and results on the above two implementations and mlxtend's library implementation of Apriori.

The item sets size was very high to be included in this report.

So for, min threshold = 0.04
Sample results are as follows :

a.) Transaction reduction

```
[{55267}, {55323}]
```

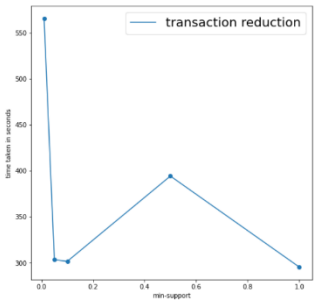
b.) Hash Based

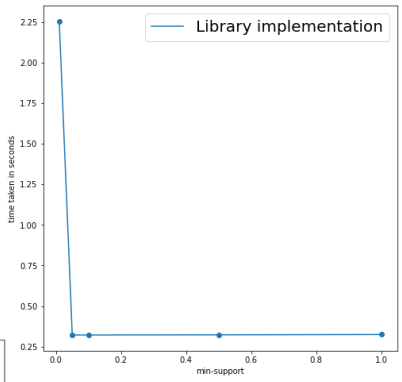
```
[[55323,
 [55267,
```

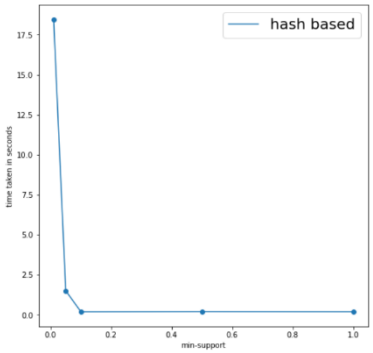
c.) Library implementation

	support	itemsets
0	0.048586	(55267)
1	0.044083	(55323)

We compared the performance on different min_support values, which are
[0.001, 0.01, 0.05, 0.1, 0.5, 1]

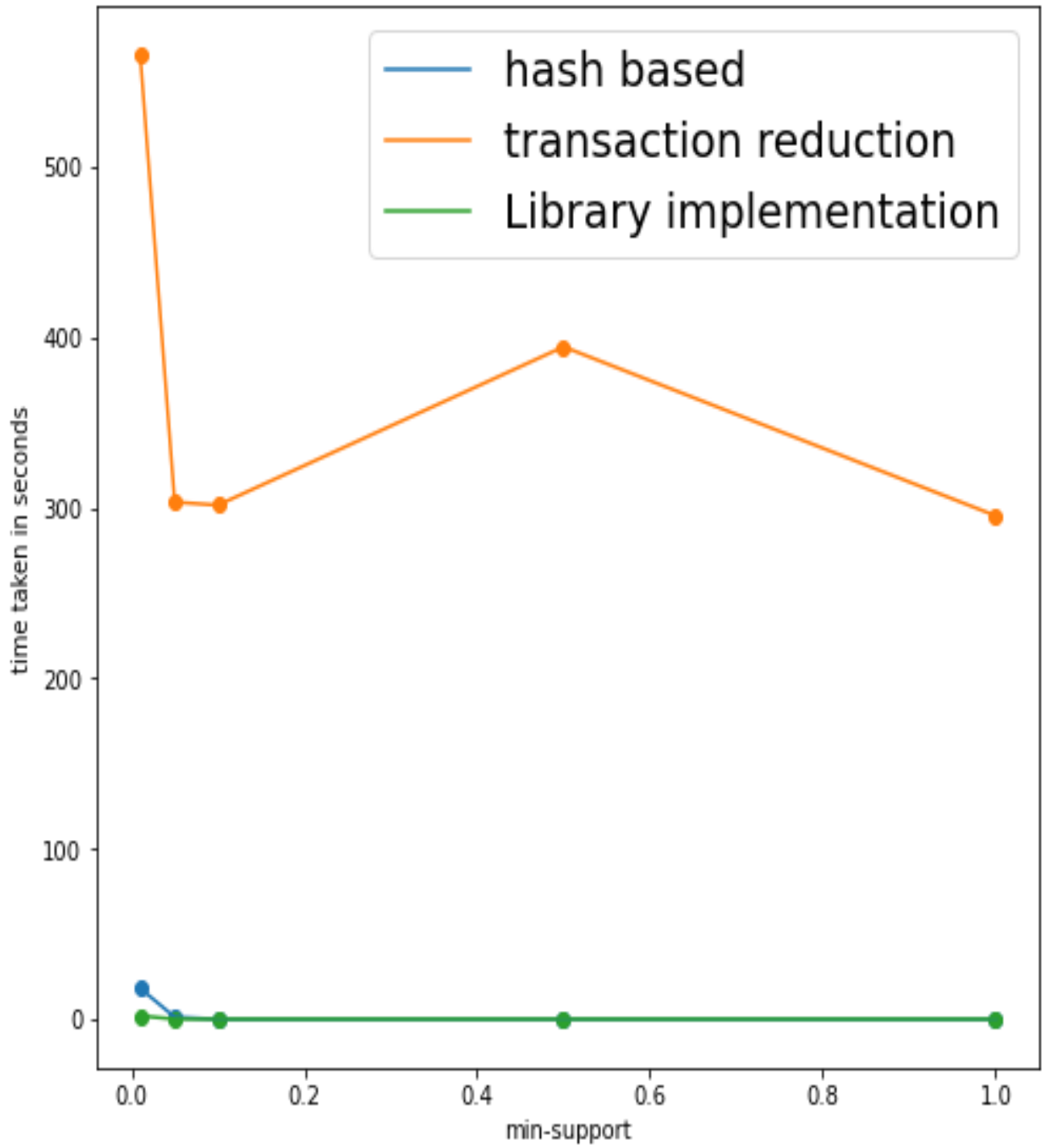






On this dataset, the library implementation performed overall better than our implementations, the transaction reduction took a lot of time as compared to library and hash based technique.

Let's see the performance comparison of all the three implementations of Apriori.



RESULTS FROM FP GROWTH

For threshold= 0.04

a.) Top Down

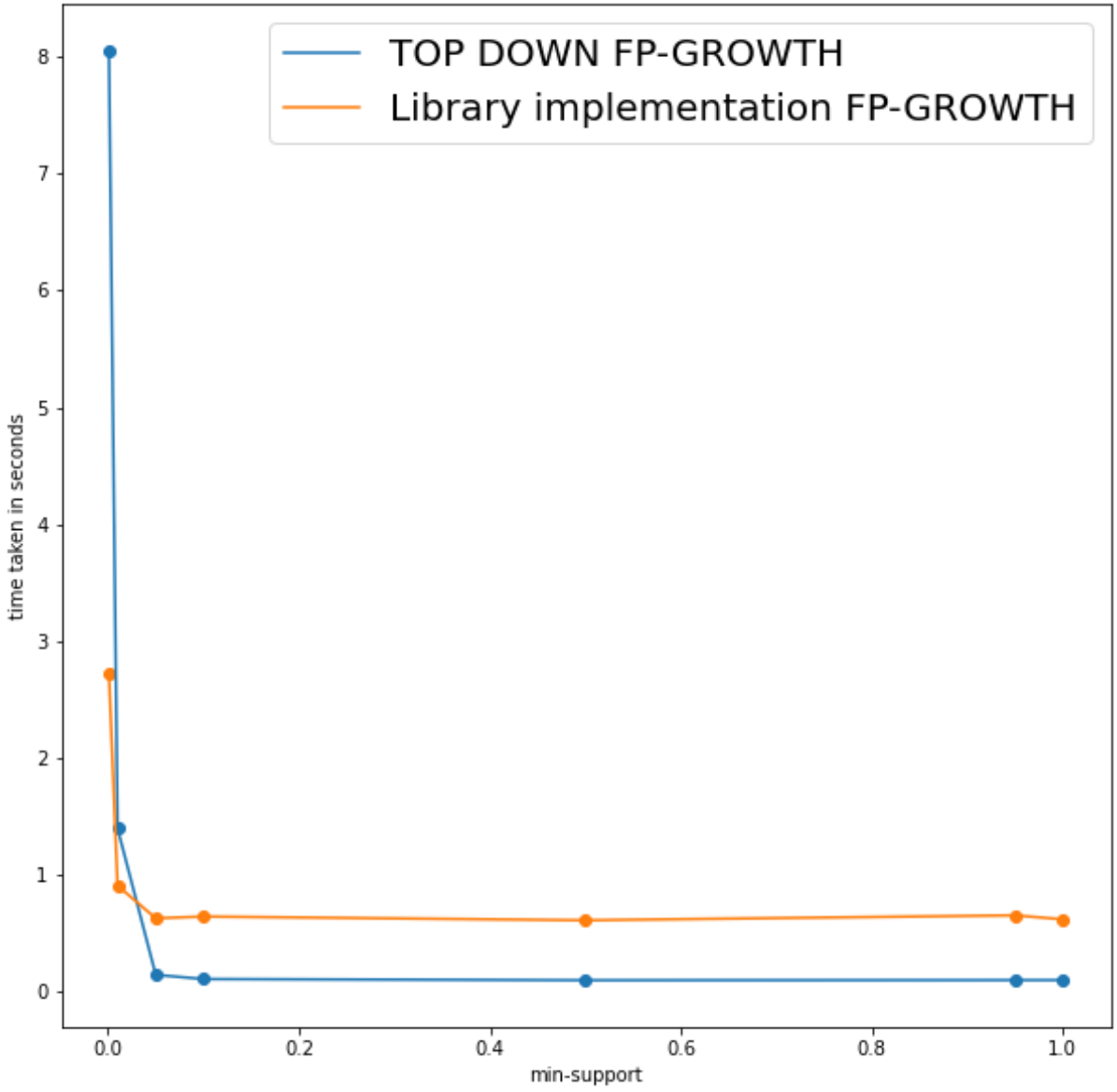
```
{55267}, {55323}
```

b.) Library Implementation

	support	itemsets
0	0.044083	(55323)
1	0.048586	(55267)

Comparison of performance of both the implementations for threshold values:

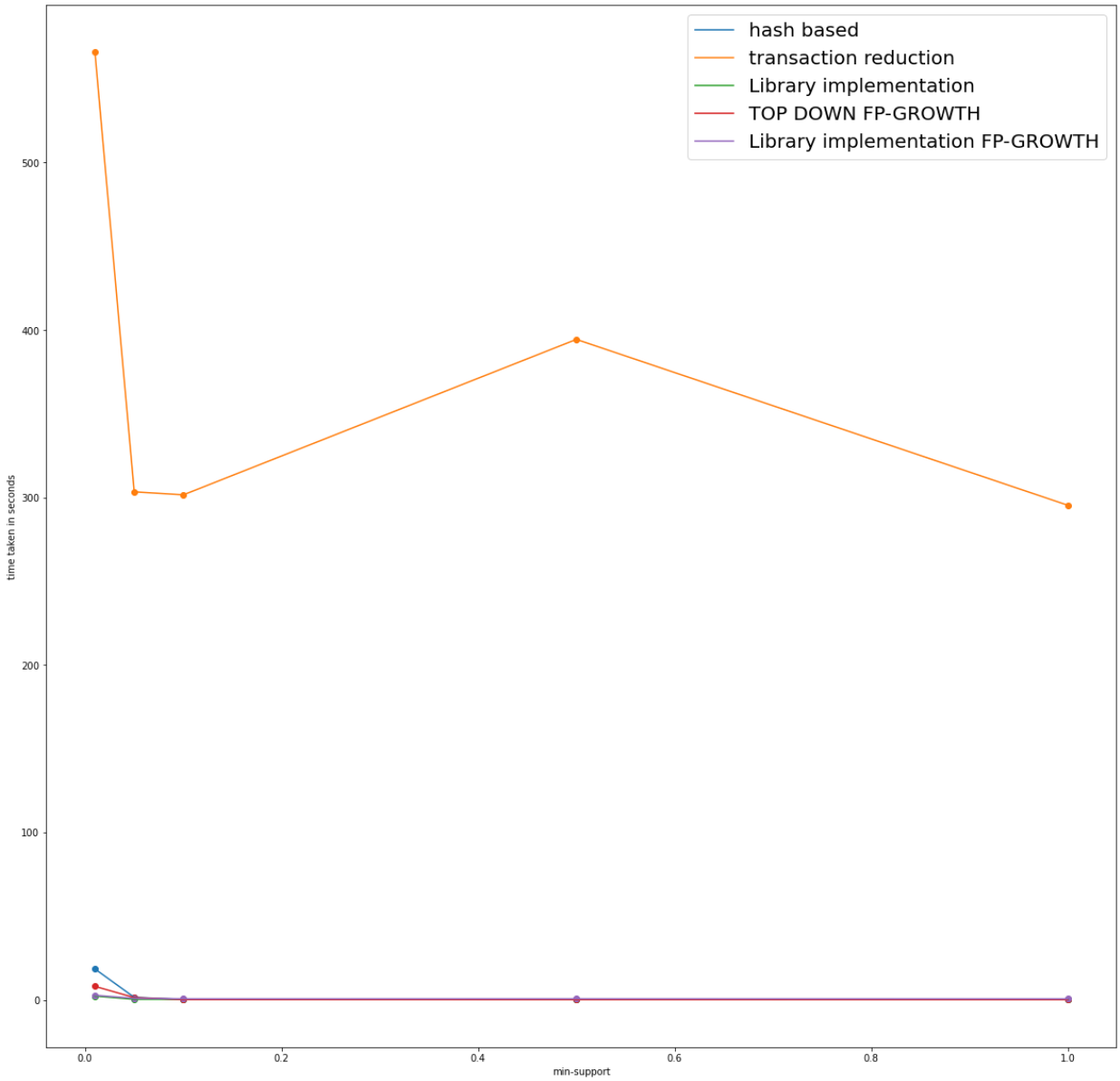
```
min_thresh=[0.001,0.01,0.05,0.1,0.5,0.95,1]
```



Again, we see that like the retail dataset, for the real world BMS Web View dataset also our top down fp-growth performed little better than the library implementation, even though we can see that for very low threshold values, the library implementation performed better than our top down approach.

OBSERVATIONS

Below we can see the time taken by all the algorithms on the real world BMSWebView2 (Gazelle) (KDD CUP 2000) dataset.



Because this dataset is small, it is difficult to see the performance of other methods in comparison with each other, but we can observe that the transaction reduction performs poor when compared to others, even on small dataset. Similar observations was also seen in the retail_utility dataset.

A comparison between both the algorithms can be seen below:

Parameter	Apriori	Fp growth
Technique	Uses apriori property with join and prune approach.	Constructs FP Tree and conditional pattern bases from the database from items which satisfy the min support.
Memory Utilization	Due to large number of candidates generated, it requires large memory space.	Due to compact structure and no candidate generation, it requires quite less memory/
Number of scans	Multiple scans are requires for generating candidate sets, though through optimizations, the size and number of scans can be significantly reduced, like by hashing (reduces number and size of scans) and transaction reduction(reduces size of scan).	Scans the database only twice, One for creating the fp tree and other for mining it. So fixed for 2 scans only.
Time	Execution time is more as time is wasted in producing candidates for every size.	Smaller execution time.
Dominant Data structure	Array/ Dictionary Based	Tree + Dictionary Based

From the above comparisons, and our practical implementations, we can conclude that the apriori algorithm is slower than FP Growth, and may not perform great when scaled to larger real-life datasets, where as FP Growth is quite fast and scalable as it requires less memory and uses optimal data structures for mining and item set generation.

The optimizations we implemented tend to work better, but the transaction reduction based implementation still couldn't perform as well as the hash based implementation of apriori and other algorithms. Even though it performs significantly better than normal implementation of Apriori.

So, according to us, the best way to mine datasets based on this project will be to go with the **BOTTOM UP FP-GROWTH** algorithm, as it is fast and generates good results.

REFERENCES:

[1] DATASET 1: BMSWebView2 (Gazelle) (KDD CUP 2000) http://www.philippe-fournier-viger.com/spmf/datasets/BMS2.txt

[2] DATASET 2: Retail_utility *retail* , http://www.philippe-fournier-viger.com/spmf/datasets/retail.txt

[3] Section 6.2.4 , Han, Kimber, Pei, The-Morgan-Kaufmann-Series-in-Data-Management-SystemsJiawei-Han-Micheline-Kamber-Jian-Pei-Data-Mining.-Concepts-and-Techniques-3rd-Edition-MorganKaufmann-2011