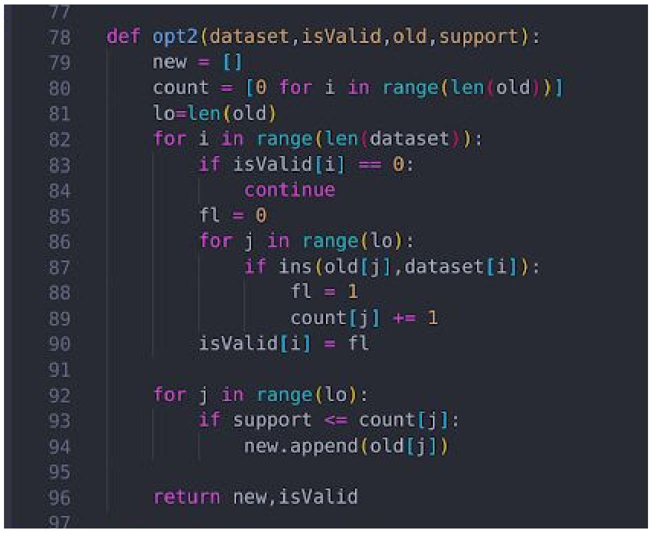
Association Rule Mining Project

Varun Chhangani - 2019121011
Shivaan Sehgal - 2018111026

1. Apriori

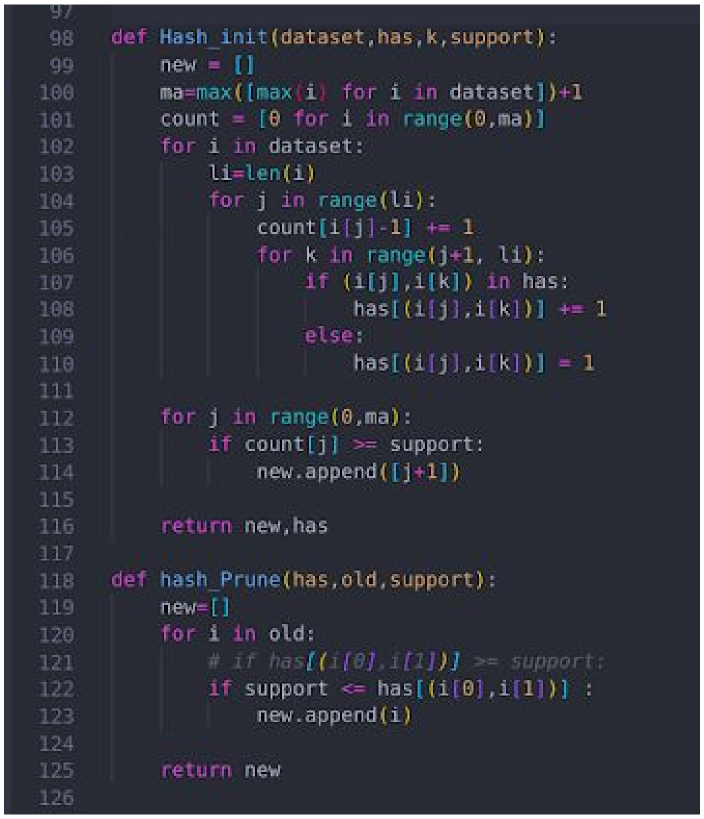
Optimization Strategy

1. Transaction Reduction : We removed the dataset which doesn't contain the previous frequent set.



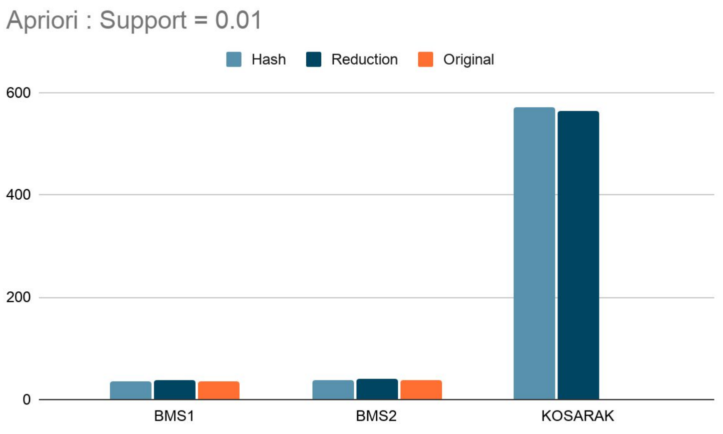
```python
def opt2(dataset,isValid,old,support):
    new = []
    count = [0 for i in range(len(old))]
    lo=len(old)
    for i in range(len(dataset)):
        if isValid[i] == 0:
            continue
        fl = 0
        for j in range(lo):
            if ins(old[j],dataset[i]):
                fl = 1
                count[j] += 1
        isValid[i] = fl

    for j in range(lo):
        if support <= count[j]:
            new.append(old[j])

    return new,isValid
```

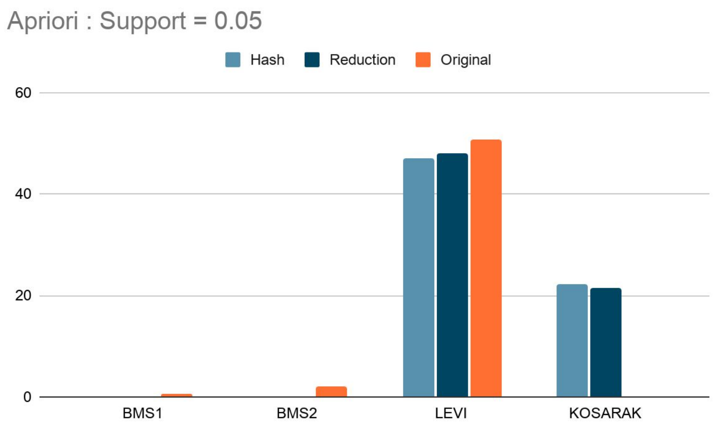
2. Hashing : Our hash function takes count of each pair items in a set of all possible tuples while counting frequent sets of unit size.

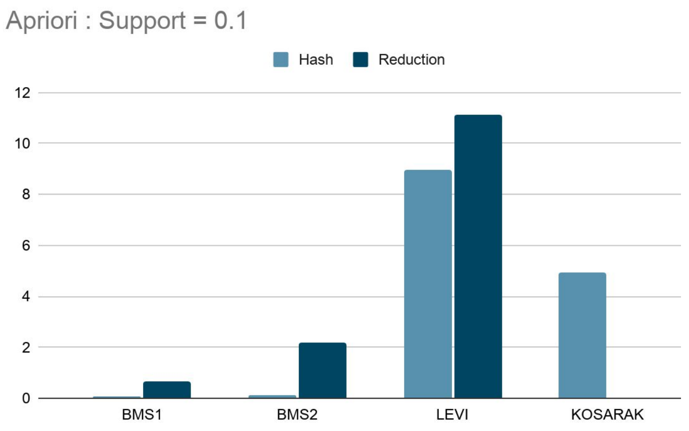


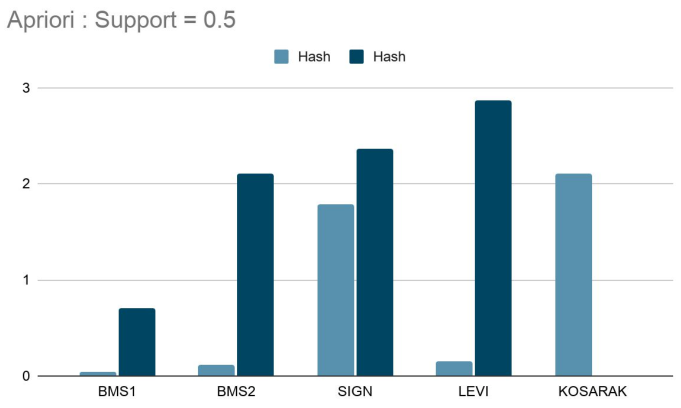
```python
def Hash_init(dataset,has,k,support):
    new = []
    ma=max([max(i) for i in dataset])+1
    count = [0 for i in range(0,ma)]
    for i in dataset:
        li=len(i)
        for j in range(li):
            count[i[j]-1] += 1
            for k in range(j+1, li):
                if (i[j],i[k]) in has:
                    has[(i[j],i[k])] += 1
                else:
                    has[(i[j],i[k])] = 1

    for j in range(0,ma):
        if count[j] >= support:
            new.append([j+1])

    return new,has

def hash_Prune(has,old,support):
    new=[]
    for i in old:
        # if has[(i[0],i[1])] >= support:
        if support <= has[(i[0],i[1])] :
            new.append(i)

    return new
```

Analysis of Apriori



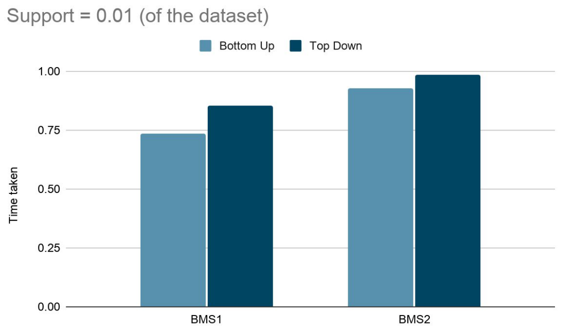


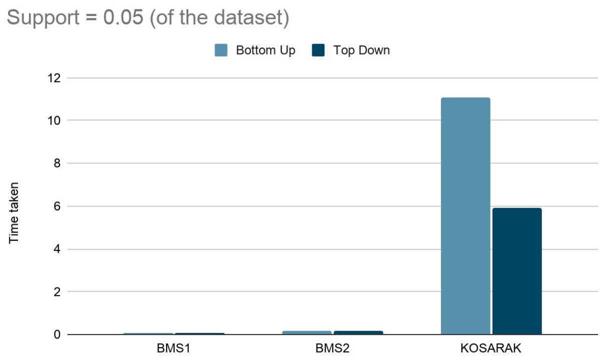


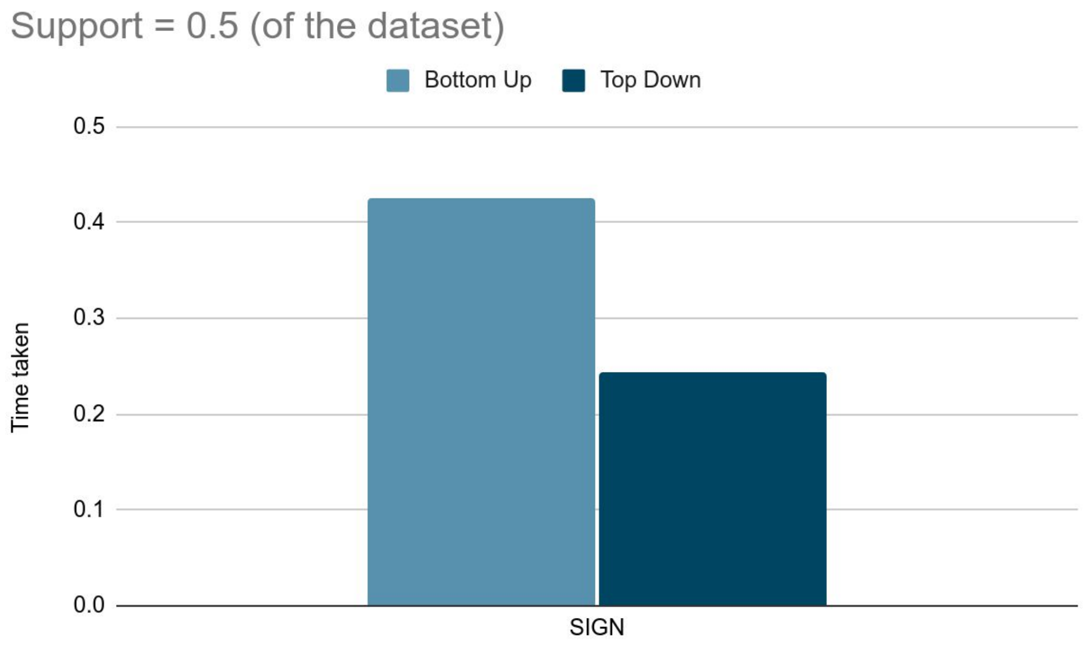


1. For very small support values the time taken by algos is quite large as the number of iterations . For small datasets as support is almost negligible they are taking very large time.
2. In above conditions where sequence item set number is more hashing is performing better than other algo.As the size increases the number of collisions decrease.
3. As the support increases the time decreases. For bigger dataset it is becoming even faster.
4. Levi being a data set with a huge number of items is taking comparative a lot of time according to its size.

2. FP- Growth







Optimization Strategy:

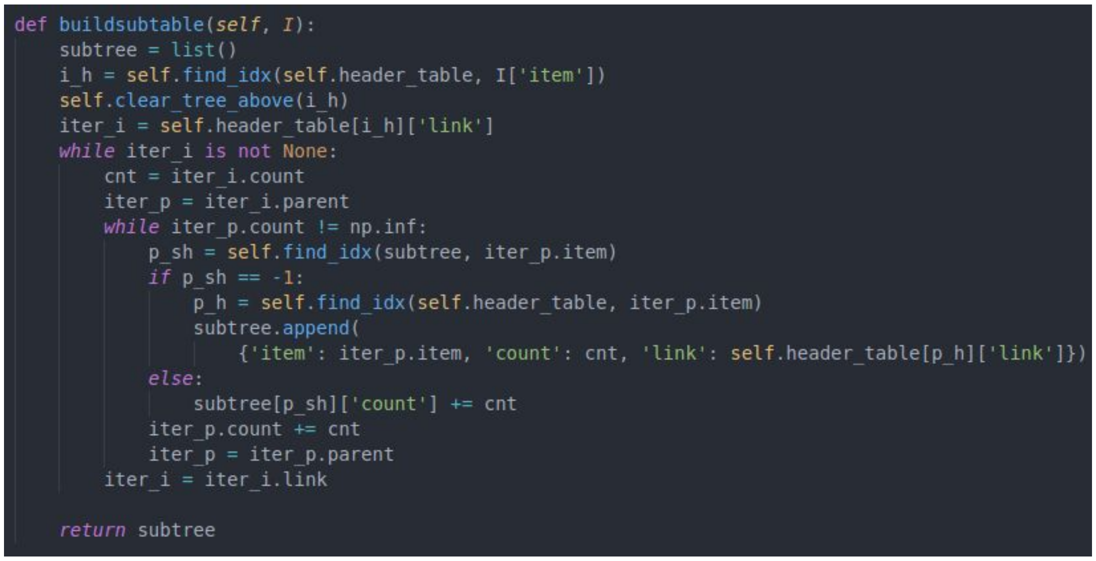
I used the method proposed in https://doi.org/10.1007/3-540-47887-6_34 to make a subheader table for each entry in header table (going in decreasing value of item count).

Furthermore, we also removed the items that had counts less than the support after the first pass; thus reducing the complexity of the second pass.

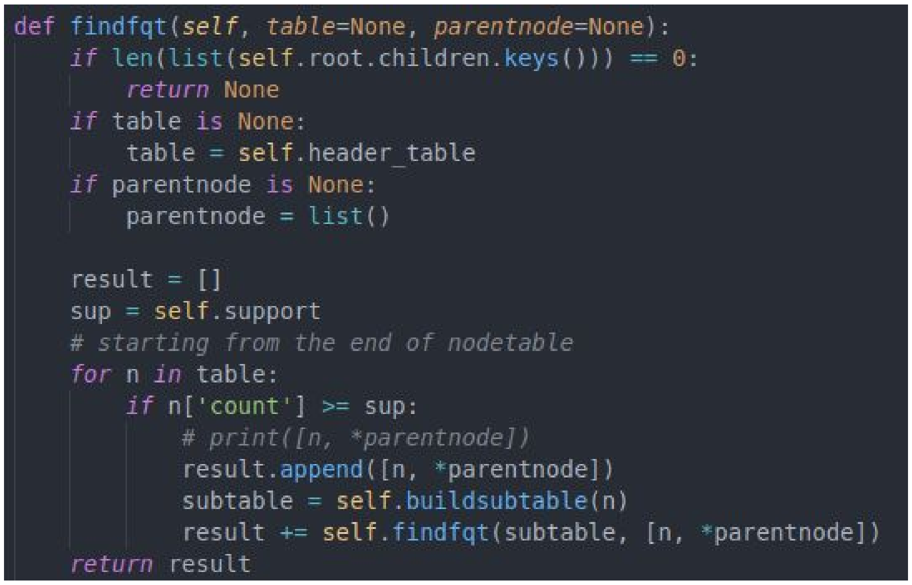
Algorithm 1: TD-FP-Growth

Input: a transaction database, with items in each transaction sorted in the lexicographic order, a minimum support: *minsup*. **Output:** frequent patterns above the minimum support. **Method:** build the FP-tree; then call mine-tree ($\varnothing$, H);

```
Procedure mine-tree(X, H)
(1) for each entry I (top down order) in H do
(2)     if H(I) >= minsup, then
(3)         output IX;
(4)         create a new header table H_I by call buildsubtable(I);
(5)         mine-tree(IX, H_I);
Procedure buildsubtable(I)
(1) for each node u on the side-link of I do
(2)     walk up the path from u once do if encounter a J_node v then
(3)         link v into the side-link of J in H_I;
(4)         count(v) = count(v) + count(u);
(5)         H_I (J) = H_I (J) + count(u);
```



```python
def buildsubtable(self, I):
    subtree = list()
    i_h = self.find_idx(self.header_table, I['item'])
    self.clear_tree_above(i_h)
    iter_i = self.header_table[i_h]['link']
    while iter_i is not None:
        cnt = iter_i.count
        iter_p = iter_i.parent
        while iter_p.count != np.inf:
            p_sh = self.find_idx(subtree, iter_p.item)
            if p_sh == -1:
                p_h = self.find_idx(self.header_table, iter_p.item)
                subtree.append(
                    {'item': iter_p.item, 'count': cnt, 'link': self.header_table[p_h]['link']})
            else:
                subtree[p_sh]['count'] += cnt
            iter_p.count += cnt
            iter_p = iter_p.parent
        iter_i = iter_i.link

    return subtree
```



```python
def findfqt(self, table=None, parentnode=None):
    if len(list(self.root.children.keys())) == 0:
        return None
    if table is None:
        table = self.header_table
    if parentnode is None:
        parentnode = list()

    result = []
    sup = self.support
    # starting from the end of nodetable
    for n in table:
        if n['count'] >= sup:
            # print([n, *parentnode])
            result.append([n, *parentnode])
            subtable = self.buildsubtable(n)
            result += self.findfqt(subtable, [n, *parentnode])
    return result
```

3. Comparative Case Study

Dataset Analysis :

	Sequence count	Item count	Avg. sq. length
BMS1	59,601	497	2.42
BMS2	77,512	3,340	4.62
SIGN	800	267	51.99
LEVI	5,834	9,025	33.8
KOSARAK	990,000	41,270	8.1

1. SIGN and LEVI are dense datasets
2. BMS1 and BMS2 are relatively sparse
3. KOSARAK dataset is large dataset

In Apriori:

1. In the dataset as SIGN and LEVI with large average sq. length hashing is producing good results as compared to reduction as a single pass in reduction at low support is significant.
2. Time is inversely proportional to support (fraction of support length to the total dataset length)
3. Reduction is doing better in sparse dataset at low support with less number of sets, because in other cases major time is consumed by branching if statements.
4. The 2-itemset 2-candidate generation becomes very slow since it is O(n^2) with respect to the number of unique elements, when number of unique elements are large then the dataset.

In FP-Growth:

1. The top down approach works better in dense datasets, whereas for sparse datasets bottom up is a better strategy. This is because we need to traverse less in the tree as compared to bottom up as we don't need to go from all the bottom nodes to the root. Instead in dense datasets, we only need to see only a few top nodes as the association rules are more likely to lie in only few frequent items in a dense dataset (we can employ pigeon hole principle to verify this).

2. Top down works better when there are fewer frequent item sets. Thus, for a high value of support top down is a good choice. However if the number of frequent itemsets is high, bottom up is a better choice.
3. The complexity of top down and bottom up approach is the same. Thus, the selection of a particular algorithm depends on what is the data distribution and dataset size.
4. The top down strategy works better for smaller size datasets. This is because we don't need to go from all bottom nodes to the root. Instead we will have to go to only few nodes from the top when the item counts is already less. However, for datasets with larger item counts, the dataset will be a bit more sparse and more nodes will be required to be traversed.
5. As the support increases, the runtime of both, top down and bottom up, decreases. This is because we have less dataset to deal with after the first pass in case the support is high.