Wyndham Hudson
CS6220 - HW 4

Problem 1.1: Implementing and Evaluating a Frequent Pattern Mining Algorithm

Implementation:

Find the frequent itemsets based on the priori algorithm.

I wrote my implementation in Python based on the example given in the slides :

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

Example set of transactions to draw frequent itemsets from given *minSupport* = 2

The apriori algorithm was separated into 3 steps: Join, Scan, and Prune

Join Step:

I implemented my join step taking two inputs: *joinStep(L, k)*.

where *L* represents either the pruned list of transactions from the previous iteration of the algorithm if $k \geq 2$ or the full list of transactions if $k = 1$. In this function, *k* represents the size of the subsets that are being joined and created.

In the Join step, initially I separated each transaction into individual items to create a dictionary of each item and how many times it occurred. In the case of our example, the first join step returned a dictionary C1 = {'A':2, 'B':3, 'C':3, 'D':1, 'E':3}. However, the Join step operates differently for subsets of 2 or more items. In that case, our *k* term dictates the number of elements in each subset. In the apriori algorithm, it is typical that our *k* term increases by one each iteration. In my implementation, I join each list from our *L* list of transactions with each other list in *L* and convert it to a *set*() to remove repeating elements. If the remaining number of elements in the set equals the desired number of elements *k*, then I push this set to the dict *C*.

From our example above, if our input is *L1* = {'A':2, 'B':3, 'C':3, 'E':3} and *k* = 2, my Join step will return: C = {('A','B'):0, ('A','C'):0, ('A','E'):0, ('B','C'):0, ('B','E'):0, ('C','E'):0}

Scan Step:

I implemented my Scan step: *scan(C, Transactions)*.

After each Join step, the Scan step is called to find all the occurrences of the subsets of C in the total Transactions list.

In the case from above, given our input of C = {('A','B'):0, ('A','C'):0, ('A','E'):0, ('B','C'):0, ('B','E'):0, ('C','E'):0} and the example of transactions, the Scan step would return the same dictionary with the updated values: C = {('A','B'):1, ('A','C'):2, ('A','E'):1, ('B','C'):2, ('B','E'):3, ('C','E'):2}. I implemented this Scan set by comparing each transaction to a set addition of the subset and transaction itself. If the subset existed in the transaction, the addition would return the same as the transaction.

Prune Step:
I implemented my Prune step: *pruneStep(C, minSupp)*.

This function iterates through each scanned subset of C (given from the Scan function above) and, if the value of the subset is greater than or equal to the minSupp variable, returns that subset in a new dictionary L.

From our example from the Scan step above, given a dictionary C = {('A','B'):1, ('A','C'):2, ('A','E'):1, ('B','C'):2, ('B','E'):3, ('C','E'):2} and minSupp = 2. The Prune step would return L = { ('A','C'):2, ('B','C'):2, ('B','E'):3, ('C','E'):2}. The subsets all now occur at least minSupp number of times. The process after pruning would repeat with the Join step and continue until we have found every frequent itemset or gone through every possible itemset.

A second implementation was included in this report, which made use of optimizations in the apriori algorithm. This second implementation will be used for comparison to the naive implementation that I created and described above. Using the "apyori" library, another implementation was tested using the same datasets. This implementation is shown in 'apyori_test.py'.

Testing:
Two datasets were tested against two different implementations.

Dataset 1:
The first dataset was the example dataset given in the examples on the slides. This dataset included in the repository as “test1.txt” and displays as

A C D
B C E
A B C E
B E

The output file names from both my implementation and the apyori implementation are provided below:

minSupp	**My implementation**	**Apyori implementation**
0	test1_myImp_out0.txt	test1_apyori_out0.txt
1	test1_myImp_out1.txt	test1_apyori_out1.txt
2	test1_myImp_out2.txt	test1_apyori_out2.txt
3	test1_myImp_out3.txt	test1_apyori_out3.txt

You can run these files yourself (assuming you have Python installed) by running the command:

python hw4.py test1.txt minSupp outputFileName

or

python apyori_test.py test1.txt minSupp outputFileName

where minSupp is replaced by an integer of your choice and outputFileName is the output file name you want to hold the frequent datasets.

The resulting frequent itemsets are the same from both my implementation and the apyori implementation.

Dataset 2:
While the first dataset helped validate that my implementation was working correctly, the dataset was so small that there was no discernable runtime difference between my implementation and the optimized implementation in apyori.

Therefore, a second dataset was downloaded from http://fimi.uantwerpen.be/data/. The dataset, titled “retail.dat”, contains the anonymized retail market basket data from an anonymous Belgian retail store.

Since this dataset was much larger, the amount of time that my implementation took to run was compared to the apyori set.

I arbitrarily picked 4 different minimum support sizes and the resulting file outputs:

minSupp	**My implementation**	**Apyori implementation**
500	retail_myImp_out500.txt	retail_apyori_out500.txt
1200	retail_myImp_out1200.txt	retail_apyori_out1200.txt
1800	retail_myImp_out1800.txt	retail_apyori_out1800.txt
3000	retail_myImp_out3000.txt	retail_apyori_out3000.txt

You can run these files yourself (assuming you have Python installed) by running the command:

python hw4.py retail.dat minSupp outputFileName

or

python apyori_test.py retail.dat minSupp outputFileName

I also tracked how much time it took my implementation to run compared to the apyori:

minSupp	**My implementation (sec)**	**Apyori implementation (sec)**
500	8575.58	4.09
1200	121.57	2.91
1800	29.87	3.71
3000	10.10	2.21

There is clear evidence that the apyori implementation uses efficient code and optimization, based on the speed-up from my base implementation.

Lessons learned:

One place where speed is lost is during the Scan function of my code, as the tuples used as keys in the dictionary must be converted to sets. Using sets consistently could result in significant speed-up. Additionally, it seems that the Join and Scan steps can most likely be performed simultaneously, that is after a set is joined, the transactions are scanned or as the subsets are being joined together, increment and scan the transaction itself. This combination of Scan and Join could result in significant speedup.

Another interesting observation was that as the minimum support decreased, my implemenation's runtime rose exponentially. This differed from apyori which remained linear as minSupp decreased, indicating a potential different approach altogether.