Data Mining Assignment2 - Frequent Itemsets

Georgios Leventis

November 2019

1 Introduction - Frequent Itemset Problem

One of the most frequent problems in the data mining field is the discovery and extraction of frequent items in sets of items(baskets). The problem can also be worded as finding itemsets that appear frequently in many of the same baskets. This is especially important for retail organizations that want to design and implement a customized marketing program[Fast Algorithms].

This problem is found in the "market-based" model of data. There is a large set of items I = $\{i_1, i_2, ..., i_m\}$ and a large set of baskets T = $\{t_1, t_2, ..., t_n\}$. Each basket is a subset of $T \subset I$.

The solution of the aforementioned problem aims to discover and create association rules, that are of the form $X \rightarrow Y$(if X then Y, where X and Y are itemsets). Someone that bought {x, y, z} usually buys {v, w}. Two more terms need to explained:

- Support: Support for itemset I is the number of baskets containing all items in I. Given a threshold s, sets of items that appear in at least s baskets are frequent.
- Confidence: Confidence of the rule I $\rightarrow$ j is the ratio of the support for $I \cup \{j\}$ to the support for I, so: conf(I$\rightarrow$j) = $\frac{support(I \cup j)}{support(I)}$

2 Solving the Frequent Itemset Problem

There are some naive algorithms for solving the problem. However, they do not scale well if the dataset of the baskets is too large. This is where the Apriori algorithm comes in.

2.1 Apriori Algorithm

The Apriori Algorithm limits the memory demands by imposing a two-pass approach. During the 1st pass it reads the baskets and counts in the main memory the occurrences of each item. It requires O(n) memory, with n = number of items. In the second pass it reads again the baskets and counts only

the pairs where both elements are frequent. It requires memory proportional to the square of frequent items; 2^m instead of 2^n.

3 Implementation

The implementation consists of the following functions:

- ReadBaskets(): This function reads the data and returns a list that contains all the baskets and the support, which is set to be 1% of the size of the list.

```
"""
Returns baskets --> a list of all the baskets
"""
def ReadBaskets():

    infile = ('T10I4D100K.dat')
    baskets = []

    with open(infile, 'r') as file:
        for transaction in file:
            j = transaction.rstrip().split(" ")
            basket = []
            for item in j:
                item = int(item)
                basket.append(item)
            baskets.append(basket)

    support = int(0.01 * len(baskets))

    return baskets, support

baskets, support = ReadBaskets()
```

Figure 1: ReadBasket() function

- CreateItemWithSupport(baskets, support): Calculates the frequency that each items appears in the big list and returns a dictionary of singletons, with the item as the key and the number of appearances of this item in the list as the value.

```
"""
Returns items_with_support --> dictionary with key = item and value = # of occurrences, i.e. {25: 522}
"""
def CreateItemsWithSupport(baskets, support):
    items_with_support = Counter(x for xs in baskets for x in set(xs))

    for key in list(items_with_support):
        if items_with_support[key] < support:
            del items_with_support[key]
    return items_with_support

items_with_support = CreateItemsWithSupport(baskets, support)
```

Figure 2: CreateItemsWithSupport() function

- CreateL1(items_with_support): Creates and returns a list that contains all the frequent singletons.

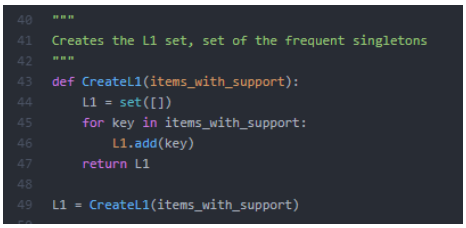

Figure 3: createL1() function

- getLk(Lk_1, baskets, support, k): This functions at first creates the C_k set of all the possible combinations, using the L_{k-1} set of frequent itemsets. Then it creates the L_k set of frequent items, after utilizing the checkToAdd function, to check if it can add a specific itemset to the set. Lastly, it deletes any itemset with support less than the threshold.

Figure 4: getLk() function

- checkToAdd(tuples, Lk_1, k): Checks if the itemsets in C_k actually exist in L_{k-1}. If they do it returns True, enabling the getLk function to add them to the set L_k .

Figure 5: checkToAdd() function

- Apriori(L1, baskets, support): This function is initialized with $L_k = L_1$ and adds the frequent itemsets, regardless of the size, in a list. It continues to be executed, until the size of L_k is zero.

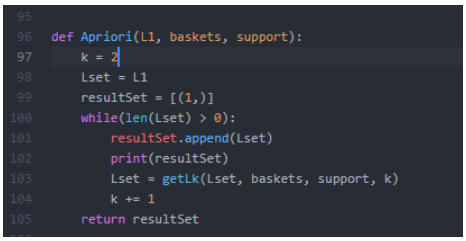

Figure 6: apriori() function

4 Results

The results of the algorithm can vary, depending on the support threshold used. If the threshold is too large, then the program will produce only singletons and take little time to complete. As the threshold becomes smaller, then the algorithm will be able to produce itemsets with length more than one. The following figure depicts the results of the algorithm having as support 1% of the length of the dataset;1000: Lastly, Figure 8 shows the correlation between the

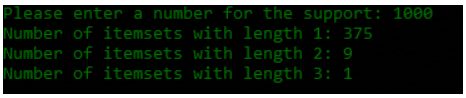

Figure 7: Results with 1% support threshold

support threshold and the execution time of the algorithm. As seen, the larger the support, the smaller the execution time.

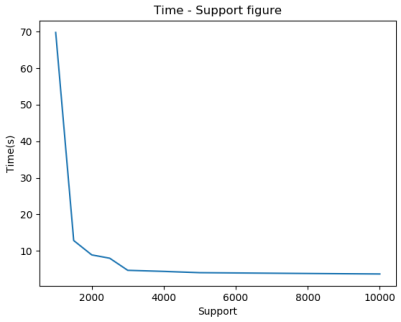

Figure 8: Support - Time correlation