Homework 2: Apriori Algorithm

Yohan Jhaveri

February 20st 2020

1 Collaboration Statement

For this homework, I collaborated with Katherine Walton and we discussed various ideas on optimization techniques for this algorithm.

2 Results

2.1 Computer Specifications:

- Model: MacBook Pro (Retina, 13-inch, Early 2015)
- Processor: 2.7 GHz Intel Core i5
- Memory: 8 GB 1867 MHz DDR3

2.2 Runtimes:

Dataset: T10I4D100K.txt

- Minimum Support Threshold: Time
- 500: 25.27s
- 1000: 16.88s
- 1500: 16.84s
- 2000: 17.55s
- 2500: 18.21s
- 3000: 17.53s
- 3500: 17.96s
- 4000: 17.74s
- 4500: 17.68s
- 5000: 17.81s

3 Optimizations

3.1 Data Encoding:

I encoded the transaction information into a 2 dimensional array with each column representing a unique item and each row representing a unique transaction. The binary values in the array represented whether the specific item was present in the specific transaction.

3.2 Counting of two-item combinations:

In my original algorithm, I used the 2D array to find the frequencies of n-item combinations where n $>$ 1. I noticed however, that for n $=$ 2, the items took about 95% of the time (Overall time on my laptop was 10 minutes for a minimum support threshold of 500 on T10I4D100K.txt). Therefore, to optimize this, I iterate over each of the two-item combinations for each transaction using two nested for loops, keeping a track of how many times they appear using a dictionary. This ended up being significantly more efficient, improving my runtime to about 25 seconds for the same parameters.

3.3 Combination Frequency:

I took advantage of numpy's parallel operations and to find whether a combination of n items existed in each of the m transactions, I selected the columns corresponding to those n items, resulting in a $(m \times n)$ matrix. I then performed row-wise multiplication (for each transaction), where I multiplied all the binary values together in each row to yield a resulting binary value corresponding to that row. The result is a (m * *1*) vector, where each of the m rows have one corresponding binary value that indicated whether that transaction contained the n items in the selected combination. I then took the sum of this vector to find the number of transactions this combination was present in. This was an easy way to find the support of the item combination.

4 Experiences

I have learned about packages like defaultdict and the power of parallelized operations such as those in numpy.