Feb 17 2018

Tyler Angert
CS378
HW1

Summary:

This is an implementation of the Apriori algorithm written in Python. Given a database of transactions, it finds the item sets/combinations which occur frequently together.

Overview:

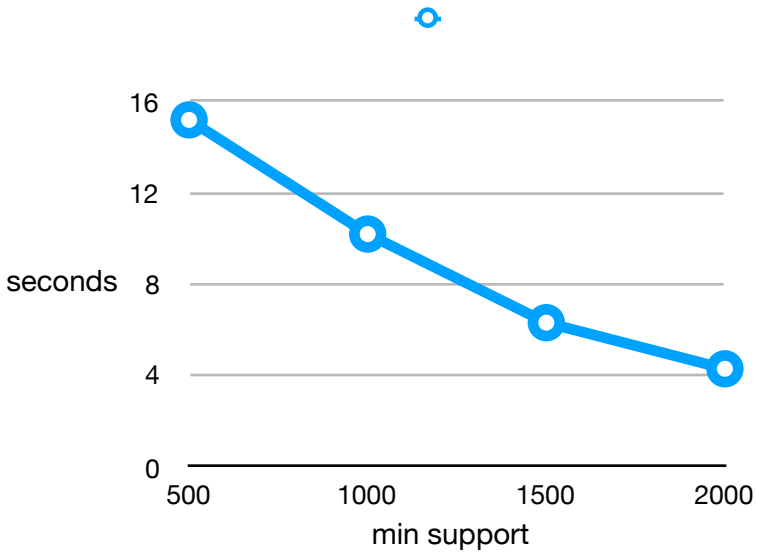
1. Count single frequent item sets
2. Count double frequent item sets after filtering the transactions from the singles
3. Self joining/pruning on repeat for each subsequent iteration to generate new candidate sets

Optimization (attempts):

1. Pruning the database after self joining on each iteration
    1. checked if subsets of current item were present in previous candidate set
2. Instead of forming every double combination from the single frequent itemset, going through and directly forming combinations from the pruned database from each transaction. This guaranteed that I did not have to check non existent combinations of single frequent items.
3. Removing transactions from the database if all items have been discarded
    1. This technique was pretty standard and was checked at the beginning of each candidate set generation
4. Removing actual items from the transactions if those are infrequent
    1. this shortened the actual lists needed to be tested
5. Skipping transactions to be tested if the length of the item set was longer than the transaction itself
    1. this prevented item sets of length 2, for example, for being compared against transactions of length 1

Results:

Unfortunately, my support count method has some bugs and was unable to deliver complete results for min support 500 because of some optimization problems I wasn't able to workout.



However, for support counts above 1000, the algorithm is pretty fast and (seems) completely accurate.

Average times:
1. Min sup 2000: 4.3s (155 elements)
2. Min sup 1500: 6.32s (237 elements)
3. Min sup 1000: 10.2s (385 elements)
4. Min sup 500: 15.02s (incomplete set, only 922 elements)

Experiences and lessons:

Holy crap, this was hard to optimize and really get right. In the end I ended up not being able to make it completely correct and find all of the complete frequent items, but this was surely a lesson in code organization, efficiency, and data structures. Relatively fundamental understanding of set theory was necessary to really get how to optimize the most time consuming parts of the algorithm. Even with such a simple algorithm, it was remarkably difficult to see through bugs and move past previous solutions that didn't work.

Also, I learned a ton of Python and realized I spent way, way too much debugging stupid issues dealing with the API for the set and Counter data structures.