CS 470 Homework 2

Niki Vasan

February 23rd, 2023

Collaboration Statement: I pledge to abide by the Emory honor code. I used the following sources as a reference in the completion of this assignment and did not collaborate with any peers on the implementation of my code.

1. Towards Data Science article on naive implementation
2. Apriori-TID paper
3. Stack overflow resources
4. Python documentation (dictionary, sets)

Approach: I made multiple versions of the algorithm for this assignment.

Attempt 1: After reading the textbook pseudocode and engaging in class discussions, it was apparent that the naive Apriori algorithm would be far too slow for this assignment. This indeed proved to be true after I implemented the algorithm the first time. I had a main apriori function, which takes in the input transcational data, converts it into itemsets and parses said itemsets iteratively for patterns. I also had a function to generate candidate itemsets and a function to map the support of each itemset formed to a global dictionary, which is used in the output.

Determining what data structures I needed to use for this assignment proved to be tricky. For example, I wanted to store the itemsets in a set of sets, but because sets are not hashable, Python does not let you nest them. Instead, I used frozensets, which are immutable sets that are hashable and thus can be nested within regular sets or used as dictionary keys. Sets are also a much more efficient data structure, because using ‘issubset‘ is a faster lookup then using for loops to search.

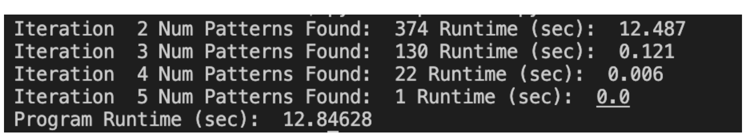
However, this algorithm was still too slow. The runtime exploded on the second iteration (pattern sets length k=2), which likely had to do with my support mapping method. In this method, I iteratively search through each transacation in the database as well as each item in each itemset in a nested for loop to determine the support of each candidate itemset. This is an extremely inefficient process ($O(n^2)$), so I needed to narrow the search space in the transacation data to reduce the time complexity.

Attempt 2: I found a paper on a more efficient implementation of the algorithm by Mohammed Al-Maolegi1 and Bassam Arkok2 (linked above in the collaboration statement). They propose an algorithmic optimization where you keep track of the transacation IDs of each itemset, or the records in the data where the itemset appears, only search among those itemsets when doing the support mapping. This significantly improves the speed of the algorithm, especially over large datasets.

To accomplish this, I use python dictionaries, which are equivalent to hash maps. Originally, I tried using a Node class to store the indices of each itemset, but I found the dictionary approach to be more intuitive for me personally. When generating the L1itemsets, I store both the itemsets and the transaction indices in a dictionary as the keys and values respectively, meaning the algorithm only scans the entire database once to get the support mapping. When I generate new candidates k+1, I only need to look at the stored intersection of the two transacation indices to determine the support count of k+1 itemsets.

Experiences/Lessons Learned: This assignment took a lot of thought and trial and error. I learned the importance of choosing appropriate data structures, even for what may appear to be extremely small operations. My algorithm does not perform completely accurately. This is likely due to the way I am generating the candidates. Currently, I am converting the two itemsets I need to merge into sets and then performing a set union, but this may not be the most accurate or efficient approach.

Results: The algorithm performs relatively fast on the provided dataset (12sec), but is not completely accurate. It currently finds 1096 association rules instead of 1073.



```
Iteration  2 Num Patterns Found:  374 Runtime (sec):  12.487
Iteration  3 Num Patterns Found:  130 Runtime (sec):  0.121
Iteration  4 Num Patterns Found:  22 Runtime (sec):  0.006
Iteration  5 Num Patterns Found:  1 Runtime (sec):  0.0
Program Runtime (sec):  12.84628
```