Apriori Algorithm for Frequent Itemset Mining

CS470 Data Mining

Gerardo Gomez

October 22, 2025

1 Algorithmic Optimizations

Several optimization techniques were implemented to achieve efficient performance:

1.1 Transaction Pruning

After finding frequent 1-itemsets, all infrequent items were removed from transactions. This reduces transaction sizes and speeds up subsequent operations. At each iteration, transactions that were too small to contain k-itemsets were also removed.

1.2 Bitmap Matrix Representation

The transaction database was represented as a binary matrix where each row is a transaction and each column is an item. This enables fast membership testing and vectorized operations using NumPy.

1.3 Vectorized Support Counting

Instead of using slow Python loops, NumPy's bitwise AND operations on matrix columns were used to count support for candidates. This is significantly faster than naive implementations.

1.4 Efficient Candidate Generation

The F(k-1) $\times$ F(k-1) method with lexicographic ordering was used to generate candidates efficiently and avoid duplicates.

1.5 Apriori Property Pruning

Before counting support, all subsets of each candidate were verified to be frequent. This eliminates many candidates and reduces expensive computations.

2 Experimental Results

2.1 T10I4D100K Dataset

The implementation was tested on the T10I4D100K dataset with varying minimum support values:

The implementation successfully completed the required test case (support=500) in 7.83 minutes, well under the 15-minute limit.

Table 1: Results on T10I4D100K dataset

Min Support	Frequent Itemsets	Max Size	Time (minutes)
3000	60	1	0.04
2000	155	1	0.64
1000	385	3	4.24
500	1073	5	7.83

2.2 Chess Dataset

The Chess dataset (3,196 transactions) was tested with minimum support of 2900. The results:

- Frequent itemsets found: 473
- Maximum itemset size: 7 items
- Execution time: 0.08 seconds

2.3 Mushroom Dataset

The Mushroom dataset (8,124 transactions) was tested with minimum support of 2000. The results:

- Frequent itemsets found: 2,367
- Maximum itemset size: 11 items
- Execution time: 0.51 seconds

3 Experiences and Lessons Learned

My initial implementation without optimizations was extremely slow, which taught me that for large datasets, algorithmic optimizations are essential. Using NumPy's vectorized operations instead of Python loops improved performance by approximately 6x, demonstrating how critical performance considerations are when working with large-scale data mining tasks.

Implementing Apriori from scratch gave me a much deeper understanding than just reading about it. I now truly understand why the Apriori property is so powerful for pruning the search space and how candidate generation works. This hands-on experience was invaluable in cementing my understanding of the algorithm's mechanics.

I learned the importance of testing with various support values during development. Starting with high support values allowed me to quickly verify correctness before optimizing for lower support values that take longer to run. Choosing the right data structures also made a huge difference. Using frozensets for itemsets, dictionaries for fast lookups, and NumPy arrays for the bitmap matrix were all crucial for achieving good performance.

I developed the solution iteratively, first getting a correct but slow version working,then optimizing the slowest parts. This approach was much more effective than trying to optimize everything at once. The iterative development process helped me understand which optimizations provided the most benefit and allowed me to make informed decisions about where to focus my efforts.