CPSC 4310
Data Mining and Deep Learning
April 5, 2023

Cody Hodge, Dustin Ward, Julie Wojtiw-Quo

Introduction

Association mining is the process of discovering interesting associations or patterns among items in a database. These associations are built off of relationships between items in the dataset, which can give us key insights into how items are related to each other. We can say that items have a relationship if the fraction of the transactions holding these items is greater than the confidence. There are many applications of association mining in the real world such as market basket analysis, recommendation systems, and fraud detection to name a few. Market basket analysis is where association mining is most commonly used, it is used to find relationships between items that are commonly bought together. This information can inform retailers to make better marketing strategies, optimize store layouts, and create recommendations based on items that the consumer has bought. The main problem that this project is focusing on is the process of finding these frequent itemsets in a database. Finding these frequent itemsets in the database is a computationally expensive task. To solve this we use the Apriori algorithm to solve this problem more efficiently than doing a brute force search throughout the database.

Algorithm Description

<u>Apriori Algorithm Pseudocode</u>

- T is the transaction database
- min_support is the minimum support threshold
- C_k is the set of all candidate k-itemsets
- L_k is the set of all frequent k-itemsets

- apriori_gen is the function to generate candidate k-itemsets from frequent (k-1)-itemsets
- frequent_k_itemsets is the function to find frequent k-itemsets
- L contains all the frequent itemsets

```
Apriori(T, min_support):
   C_1 = generate_candidate_1_itemsets(T)
   L_1 = frequent_1_itemsets(C_1, T, min_support)
   L = [L_1]
   k = 2

   while L_{k-1} is not empty:
      C_k = apriori_gen(L_{k-1})
      L_k = frequent_k_itemsets(C_k, T, min_support)
      L += L_k
      k += 1

   return L
```

Idea 1 Apriori Algorithm Pseudocode

- T is the transaction database
- min_support is the minimum support threshold
- C_k is the set of all candidate k-itemsets
- L_k is the set of all frequent k-itemsets
- apriori_gen is the function to generate candidate k-itemsets from frequent (k-1)-itemsets
- frequent_k_itemsets is the function to find frequent k-itemsets
- Subsets_k+1_itemsets generates all the k+1 itemsets from all of their k-item subsets in L_k
- L contains all the frequent itemsets

```
Apriori(T, min_support):
   C_1 = generate_candidate_1_itemsets(T)
   L_1 = frequent_1_itemsets(C_1, T, min_support)
   L = [L_1]
   k = 2

   while L_{k-1} is not empty:
      C_k = apriori_gen(L_{k-1})
      L_k = frequent_k_itemsets(C_k, T, min_support)
      L += L_k
      C_{k+1} = subsets_k+1_itemsets(C)
      k += 1

   return L
```

Implementation

There were a couple of problems that we had to solve to implement the apriori algorithm, to name a few which are reading all the data from the database, from the database counting the frequency of itemsets, and finally joining these itemsets into new k+1 itemsets. To solve these problems we used a mixture of sets, maps, and vectors to manipulate the data to our needs. We used a map for having itemsets with their count, the map was made of a set and an int. The set is a useful data structure to use because it sorts itself and also only lets unique values be added to it. Using these data structures we were able to overcome these problems to implement the apriori algorithm.

Discussion

Testing our implementation of the apriori algorithm and the Idea 1 version of it we used five different values, 0.1, 0.07, 0.05, 0.018, and 0.012, of minimum support to get different results. These five values were chosen because they gave a wide range of results while still not being too computing expensive. From the results obtained, it was found that Idea 1 of the apriori algorithm is faster than the original apriori algorithm. It was also found that the Idea 1 version also scanned the database less than the original version. Because Idea 1 scanned the database fewer times it made it less computing expensive and faster than the original. This also helped to limit the amount of memory used during runtime. The original version has to store more values in memory because it needs them while it scanned the database to find the frequency of candidate itemsets in the database. These results demonstrate that the runtime and space complexity of the Idea 1 version is better than the original version of the apriori algorithm.

Apriori Algorithm

FIle size	Minimum Support	Database Scans	Time (seconds)	Number of Frequent Itemsets Found
D1K	0.1	3	2.93	44

D10K	0.1	3	34.42	48
D50K	0.1	3	166.22	45
D100K	0.1	3	402.21	51
D1K	0.07	3	14.57	100
D10K	0.07	3	150.23	100
D50K	0.07	3	746.08	100
D100K	0.07	3	1351.27	100
D1K	0.05	3	15.24	100
D10K	0.05	3	149.18	100
D50K	0.05	3	761.14	100
D100K	0.05	3	1334.89	100
D1K	0.018	4	14.73	167
D10K	0.018	3	153.07	100
D50K	0.018	3	764.93	100
D100K	0.018	3	1345.96	100
D1K	0.012	4	19.09	1560
D10K	0.012	4	151.85	237
D50K	0.012	3	757.41	100
D100K	0.012	3	1345.20	100

Idea 1 Apriori Algorithm

FIle size	Minimum Support	Database Scans	Time (seconds)	Number of Frequent Itemsets Found
D1K	0.1	1	2.87	44
D10K	0.1	2	34.57	48
D50K	0.1	2	159.64	45
D100K	0.1	2	383.47	51
D1K	0.07	1.9	14.09	100
D10K	0.07	1.8	146.41	100
D50K	0.07	1.7	713.12	100
D100K	0.07	1.7	1330.55	100
D1K	0.05	1.7	14.36	100
D10K	0.05	1.5	146.36	100
D50K	0.05	1.5	737.08	100
D100K	0.05	1.5	1311.59	100
D1K	0.018	1,1	14.2	167

D10K	0.018	1.2	147.66	100
D50K	0.018	1.2	738.26	100
D100K	0.018	1.2	1327.06	100
D1K	0.012	1.1	16.91	1560
D10K	0.012	1.1	144.56	237
D50K	0.012	1.1	726.39	100
D100K	0.012	1.1	1325.69	100

Conclusion

While apriori algorithm can generate frequent itemsets it has many limitations, it is computationally heavy, memory intensive, and it generates a ton of candidate itemsets during execution, these are some of the problems with among others. For future works, we could try to find faster ways to generate candidate itemsets because it is one of the major bottlenecks of the apriori algorithm. Another topic for future research would be to investigate paralleling processing such as multi-threading the algorithm and splitting the database among the threads. Our findings show that the idea 1 version of the apriori is faster and generates fewer candidate itemsets compared to the original version of the algorithm. While the Idea 1 version is better it is still limited like the original version and has many bottlenecks that could be improved on.