CPSC 4310 Group Project

Spring 2023

Instructor: Dr. J. Zhang

Group Members: Rocky Au, Juhyoung Park, Landon Constantin, Scott Sonnleitner

Introduction

The Problem

Our problem for this project was to implement and measure the execution time of a basic A Priori algorithm and a modified version (“Idea 1”), to compare the two algorithms and gain insight through completing the learning exercise.

Association Mining

Association Mining is the process of discovering association rules, of the form “A implies B”, where both A and B are subsets of some larger set (i.e. a dataset), and the rule implies that both occur together with some given frequency.

Applications of Association Mining?

Association Mining can be used to discover patterns in any data set that can be modelled as a transactional database. This includes the usual idea of transactions. An example would be retailers like Wal-Mart, which can use the information gained from association mining to better optimize their supply chains, product choices, store layouts, and many other aspects of their business.

Discussion

Algorithms Implemented

High-Level Idea:

A Priori:

Generate frequent k-itemset candidates and check the actual frequency of the list of candidate k-itemsets. The ones that are more frequent than the minimum support value pass and are used to generate new candidates (k+1 itemsets) in the next round of the algorithm. The algorithm is repeated using the k+1 itemsets.

Idea 1:

The main difference from A Priori Algorithm is that candidates are generated as soon as the itemset meets the minimum support value. This means that as soon as an itemset has met or exceeded the minimum support value while it is being counted, it is put on the list of frequent itemsets and ignored for the rest of the current pass of the algorithm. As soon as the k-itemset is added to the candidate itemset list (for the next round), we check if we can generate the k+1 itemset (check against existing frequent itemsets–lexicographic rule).

Pseudo-Code:

Setup:
1) Generate a list of all 1-itemsets
2) Calculate minimum support count (actual number of transactions)

A Priori:

```
while (Current itemset is frequent) {
 Output round number
Scan each transaction checking for itemsets in the candidate itemset list and increment the candidate count whenever found.
 Output the frequency of the current itemset (store in file)
 Generate candidate itemsets for next round (lexicographic rule)
 Increment round number
}
```

Idea 1:

```
(The setup is the same.)
while (Current itemset is frequent) {
 Output round number
Scan each transaction checking for itemsets in the candidate itemset list and increment the candidate count whenever found
    if (new candidate count is greater than minimum support value)
{
add the candidate itemset to the frequent itemset list for the next round
Generate candidate itemsets for next round (lexicographic rule)
}
 Output the frequency of the current itemset (store in file)
 Increment round number
}
```

Implementation Issues

Data Structures Used:

A Priori:

The databases themselves are vectors of vectors of strings. Frequent itemsets were a map with the itemset as the key and the count as the value pair.

For every itemset we were scanning every transaction, which resulted in far too many scans of the database. We fixed this by correcting the algorithm to scan every frequent itemset for each transaction.

Idea 1:

The pair is a count and a flag that indicates which set it's part of. This tells us when it was found and when it was added to the frequent itemset. This prevents unnecessary repeat scans.

Other Issues:
A Priori:
Comparing two vectors with the lexicographic rule required operator overloading ("<", comparing the last item numbers in both vectors).
Idea 1:
Originally wanted to use a queue to cycle through frequent itemsets, but the itemsets were being erroneously added to the end of the queue when they should not have been. We solved this by adding a flag to each frequent itemset.

Results

Datasets Used:
The datasets were created by randomly generating first the size of each transaction in the range 5 to 15 and assigning items randomly to each transaction (items were labelled i1, i2… i99) to assign the required number of items to each.

Label	Description
D1K	1,000 items
D10K	10,000 items
D50K	50,000 items
D100K	100,000 items

Parameters Used:
The minimum support value was set to 1, 5, 8, or 10%, and each of the databases was scanned for the level of support. The results, including execution time in seconds and number of frequent itemsets found are included below.

Table of Execution Times:

		Dataset Used							
		D1K		D10K		D50K		D100K	
Experiment		Execution Time (seconds)	Number of Frequent Itemsets Found	Execution Time (seconds)	Number of Frequent Itemsets Found	Execution Time (seconds)	Number of Frequent Itemsets Found	Execution Time (seconds)	Number of Frequent Itemsets Found
A Priori									
Minimum Support Value (%)	1	10.8839	729	54.2403	329	184.722	221	508.917	279
	5	2.00659	76	21.4077	78	109.94	80	209.449	80
	8	0.467392	35	4.75898	35	25.1426	36	58.375	39
	10	0.136909	17	0.616327	10	2.5227	8	8.83363	13
	15	0.027389	0	0.288741	0	1.42128	0	2.94602	0
Idea 1									
Minimum Support Value (%)	1	6.07112	729	13.1899	329	38.1737	221	107.221	279
	5	0.759996	76	4.9566	78	24.1764	80	48.402	80
	8	0.144797	35	1.30745	35	6.90114	36	15.9049	39
	10	0.042003	17	0.241912	10	0.974228	8	2.92214	13
	15	0.0157738	0	0.152563	0	0.787065	0	1.56789	0

Example Frequent Itemsets:
From D100K_Apriori_1:
{ i97, i99 } | Count: 1028
{ i98 } | Count: 10007
{ i98, i99 } | Count: 1014
From D10K_Apriori_5:
{ i92 } | Count: 988
{ i93 } | Count: 1003
{ i94 } | Count: 902

From D50K_Apriori_5:
{ i92 } | Count: 988
{ i93 } | Count: 1003
{ i94 } | Count: 902
From D100K_Idea1_10:
{ i94 } | Count: 10079
{ i95 } | Count: 10049
{ i98 } | Count: 10007

Future Work

Some ideas for future work include expanding this project to implement other modified versions of A Priori algorithm described in academic papers (e.g. Xuequn Shang, Kai Uwe Sattler. Depth-First Frequent Itemset Mining in Relational Databases. AC'05, March 13-17, Santa Fe, New Mexico, USA, 2005.).

Also, the project could be expanded to compare the performance different programming languages (e.g. Python vs C++).

Similarly, the source code for generating the datasets could be repurposed for other experiments and adjusted as needed.

Challenges

Some basic challenges that were overcome included dealing with large datasets using C++ is (C++ was an awkward language to use for this purpose). Also, lack of familiarity with some data structures' features was a challenge that we had to overcome (comparison of vectors for the lexicographic rule).

Another implementation issue was preventing repeated scans of the same frequent itemset, which was fixed by implementing a flagging system.

Getting output required running all of the parts of the experiment on the same machine, which took longer for some of the scans. Similarly, changes to the algorithm required repeating the experimental procedure from scratch to get comparable results (from the same machine).

Conclusion

Correctness of Idea 1 was verified (same number of frequent itemsets). Idea 1 was faster on the same dataset compared to the A Priori implementation, and Idea 1 also increased less as the size of the dataset increased, compared to the regular A Priori implementation (Idea 1 increased 17.66 times in D100K vs D1K, whereas A Priori increased 46.76 times in D100K vs D1K).

Similarly, the execution time of higher minimum support values decreased significantly, as was expected.

Overall, C++ was reasonably fast at this task (Anecdotally, some other groups reported significantly longer execution times).