EECS 4415 Project 1: Market Basket Analysis

Mohammed A. Fulwala
York University
Toronto, ON
momo01@my.yorku.ca

Abstract

This project conducts a market basket analysis on two datasets; one from a Belgian retail store and the other from Netflix. The aim of this project was to mine frequent itemsets using various algorithms, namely Apriori, PCY, Random Sampling, SON, and Multihash and analyze them using various parameters like the frequent itemsets generated, the runtime of each algorithm, and so on. The results found were interesting for both the datasets, and I shall delve into them below.

1. Apriori

Apriori is an algorithm that was created to make the frequent itemset mining process more efficient. The Naive Algorithm would have exceeded the main memory since it will require multiple gigabytes of memory to find frequent pairs.

The way we implement Apriori in this project is that we find the candidate set of items by looping through the dataset given to us, and then adding all of that in a dictionary where the key is the item and the value is the count of that item in the entire dataset. After that, we prune the candidate set and find the frequent individual items (let us call this L1) that exist using a support value. This means that any item whose count is below the support value is discarded, and any item whose count is equal to or above the support value is not removed. After that we create pairs of items using L1, and loop through the dataset to count the number of times those pairs occur in each basket of the dataset. This becomes our candidate set dictionary for the second pass. After that we prune it by checking the value of each key and the support value and get the frequent pair items (lets call this L2). After that, we use L2 to create candidate set for three items, and maintain their count, and then prune it to get L3, and we keep on doing it until the frequent n-item set is empty, and there is no way we can make the next candidate set.

For Apriori, the Retail dataset took around 2 minutes for 1% support and 10 seconds for 2% support. For the netflix dataset, the program was running for more than 5 hours, so I had to terminate the program and store whatever I could salvage from the frequent itemset (please look at freq-itemset.txt and Table 1 for more information), and this was the case for both support percentages.

2. PCY - Park Chen Yu

The PCY algorithm is a refinement of Apriori. We noticed that in Apriori, the first pass has a lot of the main memory being idle because all it is doing is creating candidate sets for individual items in the first pass. So, PCY decides to give the main memory other things to do so that mining pairs becomes less expensive, computation-wise since finding candidate sets and frequent item sets (C_k and L_k) when k = 2 requires the most memory.

The way we implement PCY in this project is similar to Apriori, which makes sense since it is an extension of apriori. Finding the candidate set C_1 is the same, but as we go through each basket for C_1, we also go use that for loop to find pairs existing in that basket and adding that to a hashtable using a hash function (more about the hash function will be discussed in the discussion section below). After the first pass is done, ie, the candidate set for individual items have been created and the hash table has been filled, we use a bit vector to represent the hash table to cut down computation costs. After that, we find the frequent item set L_1, and use the bit vector as well as L_1 to find the frequent items for L_2. We do that by checking the bit vector to see if the pair is of a frequent bucket, and if the individual items of a pair exist in L_1. For the other candidate sets for n-items, we use the normal apriori method because the bit vector and hashtable is only used to cut down computation time and save main memory for the process that requires the most memory and computation time, that is, when k = 2.

For PCY, the Retail dataset took around 2 minutes for 1% support and 15 seconds for 2% support. For the netflix dataset, the program was running for more than 5 hours, so I had to terminate the program and store whatever I could salvage from the frequent itemset (please look at freq-itemset.txt and Table 1 for more information), and this was

the case for both support percentages.

3. Random Sampling

Random Sampling is a technique in frequent itemset mining to find the frequent itemsets in less than or equal to 2 passes. The way we implement this algorithm in this project is that we choose a random sample of the dataset (**in this case, 5%**), and then use apriori to find the frequent itemsets for that sample. In this case, we may miss some frequent itemsets, but we can capture the majority and for less computation and memory expense. We also adjust the support accordingly by multiplying the support value by the sample percentage, thus making sure we capture all the frequent items fairly.

For Random Sampling, the Retail dataset took around 15 seconds for 1% support and 1 second for 2% support. For the netflix dataset, the program was running for more than 5 hours, so I had to terminate the program and store whatever I could salvage from the frequent itemset (please look at freq-itemset.txt and Table 1 for more information), and this was the case for both support percentages. I however, was able to extract all frequent itemsets if the random sample was set to 4%. That took around 1000 seconds. It is not shown because I did not save those extracts.

4. SON - Savasere, Omiecinski, and Navathe

SON algorithm is also another technique in frequent itemset mining to find the frequent itemsets in less than or equal to 2 passes. The way we implement this algorithm in this project is that we divide the dataset up in chunks, and process those chunks using an in memory algorithm to cover the entire dataset. In this project, we decided that we will divide the dataset into **10 chunks**, and then process these chunks using apriori algorithm one at a time to cover the entire dataset.

For SON, the Retail dataset took around 3 minutes for 1% support and 20 seconds for 2% support. For the netflix dataset, the program was running for more than 5 hours, so I had to terminate the program and store whatever I could salvage from the frequent itemset (please look at freq-itemset.txt and Table 1 for more information), and this was the case for both support percentages.

5. MultiHash - PCY Refinement

Mutlihash is a refinement of PCY algorithm where instead of one hash table, we use multiple hash tables (more than 2) so that we have accurate results. Some of the frequent items in L_k may have some false positives if we use only one hash table, so multiple hash tables helps in removing any false positives.

So the implementation is the same as PCY, except that we decided to have **3 hashtables, and 3 bit vectors**, and the rest is the same as PCY.

Algorithms	Dataset	Support	Time
Apriori	Retail Store	1%	119.5s
Apriori	Retail Store	2%	11.03s
Apriori	Netflix	1%	>5h
Apriori	Netflix	2%	>5h
PCY	Retail Store	1%	121.64s
PCY	Retail Store	2%	13.74s
PCY	Netflix	1%	>5h
PCY	Netflix	2%	>5h
Random Sampling	Retail Store	1%	14.03s
Random Sampling	Retail Store	2%	1.15s
Random Sampling	Netflix	1%	>5h
Random Sampling	Netflix	2%	>5h
SON	Retail Store	1%	173.79s
SON	Retail Store	2%	16.24s
SON	Netflix	1%	>5h
SON	Netflix	2%	>5h
Multihash	Retail Store	1%	124.07s
Multihash	Retail Store	2%	19.20s
Multihash	Netflix	1%	>5h
Multihash	Netflix	2%	>5h

Table 1. Performance Comparison of Algorithms.

For Multihash, the Retail dataset took around 2 minutes for 1% support and 20 seconds for 2% support. For the netflix dataset, the program was running for more than 5 hours, so I had to terminate the program and store whatever I could salvage from the frequent itemset (please look at freq-itemset.txt and Table 1 for more information), and this was the case for both support percentages.

6. Discussions

Please refer to Table 1 for Performance Comparison.

6.1. Performance Comparison

The performance of major algorithms like Apriori and PCY is generally the same for the Retail dataset (I shall not touch a lot of base on the netflix dataset since it only executed partially for me). PCY should be faster than Apriori, but we can see that in this case, Apriori is a little bit faster. The logical reasoning for this, in my opinion, is that the Retail dataset is not large enough to show the significant difference between PCY and Apriori. For small datasets like the Retail one, PCY and Apriori do not exhibit a noticeable difference. The netflix dataset would have shown something significant if it were to execute completely.

6.2. Support Percentage

When the support was set to 1% of the dataset, we noticed that the frequent items were many. When the support

was set to 2%, we noticed that as compared to 1% support, the frequent items were less. This is because higher support means the item has to be of a higher count to qualify as frequent, and in the results, we can see that is exactly what happens.

6.3. Scalability Trends

I was unable to find any trend between 20%, 40%, 60%, 80% and 100% data because my apriori was only able to extract partial frequent items for the netflix dataset. If i had to make a logical guess, I would say that the time taken to completely analyze the 20% dataset would be the shortest and 100% would take the longest amount of time.

6.4. Hashing Function

The hash functions I chose for both PCY and Multihash were:
bucket1 = (int(pair[0]) + int(pair[1])) % bucket-size
bucket2 = (int(pair[0]) * int(pair[1])) % bucket-size
bucket3 = (int(pair[0]) (bitwise-XOR) int(pair[1])) % bucket-size

where pair[0] and pair[1] are the pairs in a tuple (a, b) respectively, and the bucket-size is the size of the hashtable.

The reason I chose the first two hash functions is because the numbers in the dataset seemed very different from each other on first glance, and it looked like the likelihood of collisions would be less, so I chose an easy hash function like these two. For the last one, I remember in one of my second year classes where the bitwise XOR was used as a primary example of a hash function. I lost access to the eclass and the notes of that class, so this is based off of memory.

6.5. False Positives

For random sampling, the chances of false positives are high because the algorithm can detect an item to be frequent in a sample, when in the entire dataset, it may not seem to be so. This can also be the case for SON if we choose more number of chunks, which in that case, will behave like random sampling.

For random sampling, a good percentage size would be around 20-30%. This would mean more computation and memory expenses, but it would also mean more accurate results because if an item is frequent in at least 30% of the data, then it is very likely for it to be frequent in the entire dataset.

For SON, the smaller the chunks are, the more false positives we can accumulate. The best chunk size for SON would be 10. These chunks are big enough to eliminate false positives from the frequent itemset.

7. Conclusion

In conclusion, the project analyzed the five aforementioned algorithms for the Market Basket Analysis. We noticed that random sampling took the least amount of time because of the size of its computation. Apriori, PCY and Multihash were almost the same because of the size of the retail dataset. SON took the longest time because of the amount of chunks it has to process. We chose 10 chunks, so it had to make 10 passes, which took more computational time. And last but not least, the netflix dataset was too big for my system to handle, which is why every algorithm ran for at least 5 hours when the input was the netflix dataset, irrespective of the support percentages. However, for random sampling, I was able to extract every frequent itemset if the sample was set to 4% of all data.

References

[1] J. Szlichta, Large-Data Market Basket Analysis, Lecture 3 (2023)