Lab 1 -5 Report

Muhammad Faizan Ali

100518916

**#APRIORI**

The First thing I did was go through all the items in the data file to figure out the total number of items. By knowing this I was able to allocate space for an array in which the index would represent the item number and the value would be the amount of times the item occurred.

Pass 1

During pass 1 I went through the data file and read each bucket 1 by 1. For each bucket I got all the items that were in it and increased the count for that item by 1 to keep track of how many times that item occurs

In between Pass

Got rid of all non frequent items, and renumbered all numbers of items that were frequent from 1 to n(number of frequent items). This allowed me to create an 1D array that can hold the count for all pairs of items using the triangular matrix method. The original names and new names for each frequent item was saved in a map to convert back later. By using the triangular matrix I was able to save a lot of space as I could just use the index of the array to reference a pair of items instead of having to save all this information into a 2d array or map which would take up more space.

Pass 2

During pass 2 I went through the data file and read each bucket 1 by 1 again. For each bucket I got the pairs of frequent items made up of items that themselves were frequent. I figured out which index a pair of item would map to and stored the count for that pair at that index in an array.

After I had gone through all buckets I went through the pair of items and selected all items that met the support threshold.

Based on this graph we can conclude that as we increase the support% the amount of time needed to run the Apriori algorithm decreases as there are less frequent Items available with higher support which means less pairs to compare.

**PSY**

PSY I did similar to Apriori except that in pass 1 as I found the count for each item in the buckets I also found all pairs of items that were available. I used a hash function to calculate the index corresponding to a pair of items and than incremented the count for that index in the hash table by 1. This allowed us to get the count of the pair of items as we went at the same time as the count of each individual items.

I than converted the hash table to a bit vector containing true or false(1 or 0) which allowed us to save a lot of save.

In pass 2 I went through the file again but this time for each pair of item I first checked to see if the bit vector that pair associated with was frequent or not and if the individual items that made it were frequent or not. Only than did I check to see if that pair was frequent.

The PCY implementation for the same Retain data took a bit longer to run compared to the Apriori implementation for low threshold which is probably due to the fact that when the threshold was low almost all items were frequent so you were not able to illuminate many items. But as the threshold increased the amount of time it took PCY to run was about the same Apriori.

**Random Sampling**

This ran the A-Priori algorithm except for only a sample of the data instead of the entire data. To select a sample of the data I first picked what percentage of the data I wanted to use for the sample data (1%-100%) and than figured out how many buckets were there in total in the data. Based on that I was able to figure out how many buckets to select. I generated unique random numbers between 0 and the total number of buckets to figure out which buckets will be selected for the sample data. This allowed all the buckets to have an equal chance of being selected and allowed me to pick an unbiased sample data from the original data to run A-Priori for.

The A-Priori algorithm was almost the same as my original A-Priori algorithm except that I saved all the sample data into memory, so I didn’t have to read the file each time to read the data which saved on disk I/O time. I also didn’t use the Triangular Matrix method to store the pairs as there was not that many pairs, instead I just stored each pair in a map where the key represented the item pair (ex {3,2}) and value was the count of that item. This took up more space in memory but since I was only working with sample data, I could afford to use the extra space required to store the data in return for less computations which allowed the algorithm to run faster.

I also reduced the support threshold proportionally to the sample data in order to help catch more truly frequent items. To reduce the support threshold, I took the percent of data to be selected and divided by 100 and then multiplied it by the threshold. The threshold provided to the program were the same as what you would provide if you were working with the full dataset and were reduced after in the program because this way no matter what percentage of any data you took it would reduce it accordingly.

Reduced threshold = (percentage of sample data/100) \* threshold.

**Option 1**

Netflix Data

|  |  |  |
| --- | --- | --- |
| Apriori | PCY | Random Sampling |
| 45.119 |  | Time: 34.578  Supp: 30  % of data: 15 |

Netflix data took a lot longer to run at low threshold so I was not able to run but I was eventually able to run it at reasonable times and supp 30%

**Instructions To run:**

**Linux:**

Need g++ installed

Make sure data is in the same folder as the source code

Apriori

In command line type:

g++ Apriori.cpp

./a.out

PCY

In command line type:

g++ PCY.cpp

./a.out

RandomSampling

In command line type:

g++ RandomSampling.cpp

./a.out