**Big Data Lab #5 Report**

Ryan Goulding - 100556101

# Code

For this lab’s I chose to code the algorithms in python. To run them install python to your machine and type “python <filename>” in your command line, filename being one of the following: apriori.py, pcy.py, and son.py, rs.py. The program will then print it’s results to a file named “<filename>\_results.dat” and output to the console how long it took.

You can find my files in the submitted zip folder, it’s contains the following:

* apriori.py - my apriori program
* pcy.py – my pcy program
* son\_rs.py – my son program with random sampling
* retail.dat – the set of market baskets used for testing
* lab05\_report.docx – this document
* AlgorithmRuntimes.xlsx – a spreadsheet containing runtime test results

# Apriori

## Code Breakdown

The apriori algorithm is an algorithm used for finding frequent item sets from transactional based data like market baskets (online stores like amazon). The algorithm works in two passes; the first pass counts the appearance of every item within each basket. The second pass finds every possible pair of items from a basket, and checks if those items appear more than our support threshold. If an item passes these checks’ we say it’s “supported”.

Pass 1

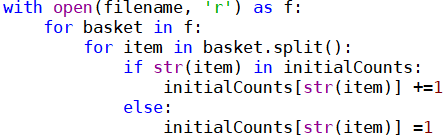


Figure 1

The first pass of my apriori program starts by opening the file we’re reading our data from. It then begins looping for every basket (line) of items and then starts a nested loop for every item in the basket. We have to split the basket before the nested loop, otherwise the program will start counting spaces. Finally, the code checks if the item is in the list of item counts. If it is then increment it’s count by 1, otherwise add it to the list with a value of 1.

Pass 2

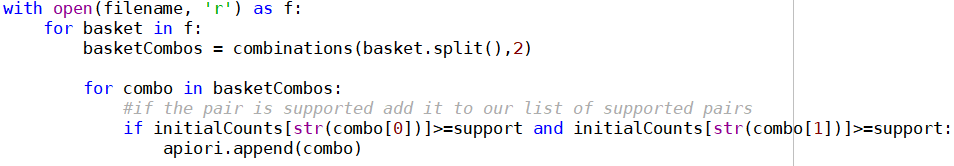


Figure 2

The second pass of my apriori program starts in a similar way to the first pass, it opens the file and starts looping through it basket by basket. This time however, I calculate every possible pairing of that basket and store it to a list. Then I proceed to loop through that list for every combination and check if it’s supported by our initial counts that we calculated in the first pass. If it is, I add it to the list of supported pairs. An unfortunate result of this code is that you can get duplicate combinations in the supported pairs list. This is due to the fact that I don’t check if the element is in the list already. By adding the code that does check we slow down the average completion time from 16 seconds to about 3 minutes.

## Apriori Analysis

Support % vs. Time

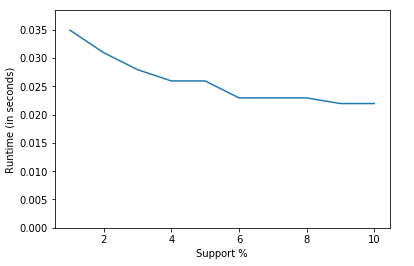


Figure 3

Figure 3 is graph of runtime in seconds vs support threshold percentage for my apriori program. As you can see when the support threshold gets higher the amount of time for completion goes lower. This is due to the fact that the first pass is always the same speed if you're using the same dataset. The second pass does much less work due to more candidate numbers being culled from the higher support threshold. In short, since your support threshold is higher, you’ll be appending less candidate pairs, which means less data being loaded into memory.

# PCY

## Code Breakdown

The Park-Cheng-Yung algorithm or PCY algorithm is similar to the apriori algorithm, but we add a few steps to both passes. On the first pass as well as counting the appearances of each item in the basket, we also calculate every possible pair from said basket and hash it to a hash table. If the pair exists in the hash table, increment its count by 1, if not initialize its count to 1. The second pass is almost the exact same as apriori, except now we hash the possible pairs to a hash table as well as check if its supported. If we can hash it to our map and both items in the pair are supported, then add it to the supported pairs list. The hash function I chose for this algorithm is the default hash function that’s built into the standard python library. The reason I chose this one instead of making my own function was that I found many of them had numerous collisions due to their mathematic functions not being injective.

Pass 1

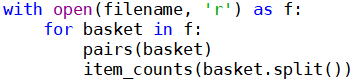


Figure 4

The first pass of my PCY program begins looping through every basket in the file, for each basket it hashes and counts the possible pairs to a hash table and counts each individual item.

Between passes

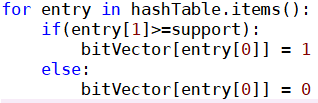


Figure 5

In between the first and second pass I loop through the hash table and convert it to a bit vector using the same hash as a key. Its value will be 1 or 0 depending on if it meets our support threshold.

Pass 2

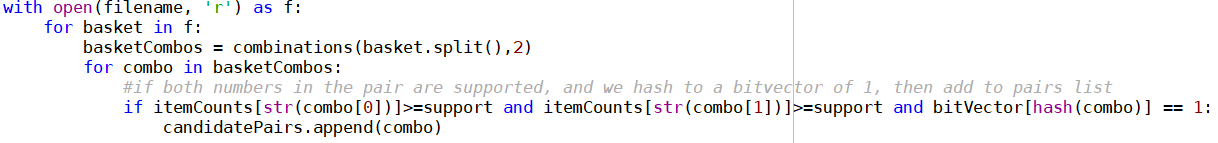


Figure 6

The second pass loops through the file again and calculate all the possible pairs of that basket. Then it loops through those pairs to check if both elements are supported and that the pair hashes to our bit vector with a value of 1.

## PCY Analysis

Support % vs. Time

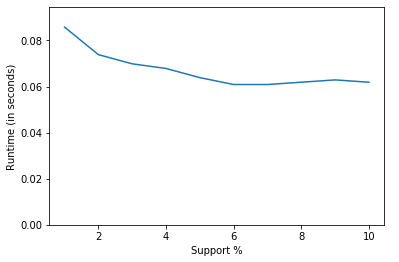


Figure 7

Figure 7 is a graph of runtime in seconds vs support threshold percentage for my PCY program. Similar to the last graph (figure 4) as the support threshold rises, the runtime decreases, this is due to the same reasons my apriori program sped up. The biggest difference between apriori and PCY is that PCY takes about double the time to complete. This is due to PCY creating a hash table as well as performing item counts in the first pass. As more numbers are pruned out because of the high support threshold, it creates less work for finding candidate pairs in the second pass. This result was expected because of apriori and PCY’s similarity in their function.

Buckets Number vs Time

As I use bigger and bigger files for my PCY program the bucket number increases as the possible pairs of items that could be grows. This in turn causes the amount of time for completion to increase as we have to loop through even more entries in the hash table and bit vector.

Optimal Buckets

Since I used the built in hash function for my hash table, there are no optimal buckets. This is because the hash function creates no collisions. The hash is “perfect” in the sense that it will always hash a unique pair to a unique value, in other words its one-to-one/injective. Figure 8 is my attempt to see if different support thresholds yield different bucket amounts, they do not.

# 

Figure 8

# Random Sampling & SON

## Apriori Random Sampling

The random sampling algorithm I developed for my previous apriori program works by taking a small sample of the input file at random and processing it as if it were the full file. This is very different from the apriori program I created. Apriori always reads the file in sequential order from start to finish, but with random sampling I shuffle the line’s so every run is different. I determined the best sample percentage to be about 5%, this gave a sample size of about 4408 baskets. A downside of random sampling is the false positives; the random selection of lines leads to possible outliers becoming more prevalent and common pairs being missed completely. The key idea of random sampling is monotonicity, if a candidate number is supported in a random sample, then it must be supported in the full file.

## Apriori SON

Building off of my random sampling algorithm, I redesigned it for chunking instead of sampling. Now Instead of processing 5% of the file’s lines randomly, I altered the code to chunk the input file into 255 Byte pieces to process. I found 255 Byte’s to be the most appropriate as it loads about 10-15 lines depending on how long they are. The major difference between my SON and apriori program is in the first pass, since chunking uses the idea of monotonicity. If an item appears frequently in our chunk, it must be frequent through the rest of the data. Therefore, I had to change my program to find the candidate numbers before moving on to the next chunk.

# Apriori vs. PCY vs. RS vs. SON

Figure 9 (raw data can be found in AlgorithmRuntimes.xlsx)

To compare the algorithms I programmed for this report I created a test script that fed them samples of the retail.dat file in 10,000 line increments. In figure 9 you can see the runtime’s vs the amount of data processed for each of the algorithms. PCY and apriori were the fastest and almost identical in code, but PCY took about 5 time’s apriori’s runtime on average. I concluded this was due to PCY’s unsuccessfully pruning of 2/3rd’s of the data. The outlier on the graph is random sampling, which took an incredibly long time compared to PCY or apriori. I believe this is due to my random sampling program not being as optimized as possible, but even un-optimized it shouldn’t take much longer than apriori which only takes on average 2 seconds to run. My SON program takes a reasonable amount of time considering it’s processing single chunks of 255 bytes. The key point I took away from the testing, is that the simplest algorithms run the fastest, while the algorithms with more verification (false positive pruning) take much longer. Therefore, the algorithms with higher computational complexity should take longer, ignoring the outlier (random sampling) figure 9 agrees with my point.

# Netflix Dataset

## Experiment

To process through the Netflix dataset, chose my apriori program I developed in lab 1. The reason being that my apriori is the fastest algorithm by about 5 seconds. Unfortunately, with a support of 2 and 5, I couldn’t process through the full file within a reasonable time. About 2 hours into the first run with a support of 2 I stopped the program, I ran it again with a support of 5. I only waited about 30 minutes before stopping the program this time.

## Findings

From my experiment I found that my apriori program was insufficient to process through the full Netflix dataset. With a support of 2% or 5%, I estimate it would’ve taken my program upwards of 2 hours to complete. I believe the reason my apriori couldn’t fully process the file is due to my naive way of implementing it, specifically the nested for loops and combination processing.

# Conclusion

In Conclusion, besides random sampling, the results yielded from the various experiments are consistent with our lectures. Apriori is the fastest because it’s the simplest and PCY takes a bit longer since it’s just apriori with a hash table generated between passes. SON takes a big longer, but by chunking the data we can be confident in our choices for candidate pairs. This is because of monotonicity, which states that if an item is frequent in a chunk of the data, it must be frequent in the full data.