Name: Kevin Muirhead

UCID#: 00502756

MDSC 679: ML Project #2: Implementing the *Apriori* algorithm

**Apriori Algorithm Design, Implementation and Testing**

The programming language chosen for implementing the *Apriori* algorithm for this project that I am going to utilize is the python scripting language. The specific variant of the original Apriori algorithm is the AprioriTID algorithm from the original Apriori paper (R. Agrawal et. al. 1994). Testing will be done during implementation to ensure that the algorithm is working correctly before going on to more advanced implementation. Advanced testing will be done using the *Arabidopsis thaliana* genotypes to phenotypes data set from the MDSC 679: ML Project 1. The genotypes will be formatted into SNP-Sets using the annotation GFF file for figuring out which SNP variants will be added to each gene with upstream and downstream flanking regions of 2000 bps will be considered the SNP-set. The SNP variants will be obtained from after the filtering step of MDSC 679: ML Project 1. Where the SNP variant have gone through the quality control and filtering step so that they can be used as a smaller sample set of SNP variants to make SNP-Sets out of and we can find the frequent itemsets of each gene for each genotype. Each genotype is a transaction in the database where each individual id is used for the transaction id for the transaction database of genotypes. A program will be designed and implemented in python to generate the SNP-Sets from a file consisting of a list of quality control and filtered SNP variants for each genotype and the annotation GFF file. Each gene will be a separate file where each row is a genotype id in the first column and the second column a list of SNP variants for that gene for that individual. SNP variant encoding notation was adapted from AprioriGWAS (Zhang et al. 2014) with a few modifications.

The implementation plan is as follows;

Design of data structures:

Will design which data structures to use for the various itemsets, support counts, and the transaction database. The data structures are going to be a dictionary if it is a support count data structure and a list of lists for the itemsets data structures.

Data Structure Design:

**Transactions Database**

Parsed from a file. File is formatted so that the first column is the transaction id with column name transaction\_id and the second column is the list of transactions with column name transaction\_values.

**Data Structure**: Dictionary

**Key**: transaction id

**Value**: transaction list of items

**Large 1-Itemset**

Transaction Database will be scanned for unique items and appended to an array.

**Data Structure**: Array

**Index**: At each index there will be an item

**Large 1-Itemset Support Counter**

**Data Structure**: Dictionary

**Key**: itemset as key string deliminated by comma ‘,’ characters and the support count for that itemset.

**Value**:

**Data Structure**: Candidate k Support Counter

Dictionary with itemset as key string deliminated by comma ‘,’ characters and the support count for that itemset.

**Key**:

**Value**:

Large k-Itemset Support Counter

Dictionary data structure with itemset as key string deliminated by comma ‘,’ characters and the support count for that itemset.

**Key**:

**Value**:

All candidate data structures will have ordered Itemsets

Design of Apriori Algorithm Methods:

For the design of apriori algorithm methods Large 1-Itemset() generation, apriori candidate generation(), and pruning\_step () within the apriori algorithm.

Design of Apriori Algorithm Method:

The apriori algorithm will be an object that stores the transaction database, minimum support count, and minimum confidence values.

Apriori Algorithm Methods Design:

To design the algorithm, we need to design functions that calculate the support for each iteration of the Apriori algorithm. We also need a function for candidate generation for the 1-Itemset as well as a function for k-Itemset candidate combinations to start the next iteration after the first iteration.

Psuedocode

Apriori-gen

Candidate generation

Input:

Implementation of Association Rule Metrics

Implement the following association rule metrics from the following article;

<http://rasbt.github.io/mlxtend/user_guide/frequent_patterns/association_rules/>

Support

support(itemX) = Number of transactions that itemX appears

function support(itemX, total\_transactions)

return support

Confidence

function confidence(itemX, itemY)

confidence(itemX 🡪 itemY) =

return confidence

lift

function lift(itemX, itemY)

lift(itemX 🡪 itemY) =

return lift

leverage

function leverage(itemX, itemY)

leverage(itemX 🡪 itemY) = support(itemX U itemY) – (support(X) \* support(Y))

return leverage

conviction

function conviction(itemX, itemY)

conviction(itemX 🡪 itemY) =

return leverage

Basic Implementation

Implementation of the Apriori algorithm is going to be based on the AprioriTID algorithm. I will use smaller test datasets to implement the algorithm so that we know it is correct before we expand on it for finding frequent itemsets in genotypes of several individuals. An expansion plan is covered in the next Implementation sections. Smaller test datasets are easier to evaluate correct answers given a transaction database. The support, confidence, and lift values for each candidate itemset are simple and quick. Once I am confident that the algorithm is working correctly the idea is to iron out potential bugs that I did not anticipate when working with smaller datasets. Intermediate sized datasets will be used and the results will be checked at each implementation step. Once the problem is worked out I can evaluate the results and see if the expected value matches our test cases and results.

**Basic Implementation Issues**

While programming the apriori algorithm I ran into some unforeseen implementation issues due to how I was originally thinking about the problem based on the pseudocode. Some of it was due to how I code in python versus other languages. Yet other issues lead me to go back to the drawing board to rethink how some components of the apriori algorithm could be more efficient, robust, or an alternate approach to solving the problem. This was a good thought and working process as I believe having a plan on how to implement an algorithm versus just programming and seeing where it goes is more efficient for time and resources as well as code reusability, repurposing, maintainability.

Large 1-Itemset Candidate Generation

For the Large 1-Itemset step in the generate

Started with L1 generation

Realized that it would probably be better to make this step a hybrid before the k = 2 step in the apriori function. This is because I thought it would be better to prune items with a support count < the min\_support\_count. This way I could start with less candidates for the k = 2 candidate generation step saving minimal time but could be significant time in the best case scenario with a transaction database with a large number of transactions N.

### Apriori-gen step

### pruning step

### Needed to preset all itemset\_counters to 0 before counting number of itemsets in transaction.

### Put first that I just kept counters for min\_support pruning purposes only and then after realized that I needed to add another datastructure.

## I am testing functions and steps as I go on a test dataset so that I know what the answer is and so that I can debug my program. At each step I am checking the expected results. Will make this dataset into a test at the end. Since I used it for testing I will fine more datasets to test the finished product by finding examples online at the following websites x,y,z.

### Remember to add that I needed to keep a dictionary datastructure of itemset lengths

Itemset\_counter\_set[‘1’] = large\_1\_itemset\_counter

Itemset\_counter\_set[‘k’] = large\_k\_itemset\_counter

#### Apriori function

Placed all functions above into order of that shown in figure 1 of the apriori ibm paper. Added a frequent\_itemset dictionary data structure so that I can capture all the k-itemset and k-itemset counters.

### Decided that I needed to capture the transaction ids in each apriori-gen step as well as the large-1-itemset. Preferably in the step that adds the large-k-itemset and large-k-support-counter. Where ever it counts support (item frequency) then add transaction ids for the purpose of tracking which transactions have frequent itemsets.

Basic Test Example

I obtained a tutorial on the Apriori Algorithm from geeksforgeeks.com (https://www.geeksforgeeks.org/apriori-algorithm/). This is a good resource for figuring out how to find examples on how the algorithm is supposed to behave. I used the example for testing purposes and as a conceptual aid in order to implement the basic AprioriTID algorithm. I also used a few other resources on youtube (Apriori Algorithm (Associated Learning) – Fun and Easy Machine Learning) and the internet to go over how the algorithm is supposed to work and to encourage thought provoking design ideas for what data structures and functions to use for implementing the algorithm.

Advanced Testing

The expansion plan is to perform a partitioning strategy for larger datasets to speed up the time the algorithm takes to execute and terminate. In order to do this I plan on

Test example

Advanced Test Example

Time Complexity of algorithm

Publication of AprioriTID algorithm: O(n) = 2n

Implentation of AprioriTID algorithm:

Time analysis based on input size using the github.com repository dataset

Genomic Data Example

Use the dataset from the first project

Partition the SNP variants into SNP-Sets

During the course of approaching this task, I figured out that gene SNP-sets in the way that I previously described are not going to work due to the fact that the time complexity of this algorithm is a least O(n) = 2n. This makes it unfeasible to really have more than 30-40 SNP variants per SNP set. Since there are roughly 15000 SNP variants after the filtering step for the ML Project 1 dataset. Then having the possibility of more than 30-40 per gene I thought a better approach would be to choose 32. The reason being that 232 = 4 294 967 296 seems to be somehow like an obtainable goal. At least to test it first to see if it works. I would then just have to ensure that the SNP-set generator script I made can change the number of SNPs per set.

Then the amount of SNPs per gene would just be chunks. A better idea than just using genes and boundaries is to just make an algorithm based on chromsomes as a set of SNPs. That way we can figure out an optimal number of SNPs per set that can be used to make the individual SNP sets of comparable size to not have a set considerably less than the others if there are like less than 50 % of the entire number of SNP variants per chromosome. This way we can eliminate the complexity of the original plan. The original plan would have made it harder to code overall.

Example:

Let’s make up some rough numbers.

Say there are 15000 SNP variants/5 Chromosomes = 3000 SNP variants/chromosome

We want 32 SNP variants per set. We can use modulus operator (%) to figure out the remainder. #### Continue with math about SNP sets for each chromosome. 3000%32 = 24 SNPs remaining per set. If there were less than 16 SNPs remaining per set than I could tally to the end 24 SNPs per chromosome.

References:

1. “Fast Algorithms for Mining Association Rules”, Agrawal, R., Ramakrishnan, S. (1994), Proc. 20th int. conf. very large data bases, VLDB 1215(pp. 487-499). doi: 10.1.1.40.7506
2. “***AprioriGWAS*, a New Pattern Mining Strategy for Detecting Genetic Variants Associated with Disease through Interaction Effects**”,  
   Zhang, Q., Long, Q., Ott, J. (2014), PLOS Computational Biology 10(6): e1003627. https://doi.org/10.1371/journal.pcbi.1003627