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 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 variants have gone through the quality control and filtering step for multiple tests 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 genotype item pattern is obtained by using the information for each SNP variant marker where genotype\_id, the position\_id and allele base is used for generating apriori item patterns (i.e Chr5\_Pos26963862\_A).

The implementation plan is as follows;

Design of Apriori Algorithm Methods:

For the design of apriori algorithm methods large\_1\_itemset() generation, apriori\_gen(), 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. As well as the pruning step to generate the Large-k-itemset for the next generation.

**Algorithm Psuedocode**

**Algorithm load\_transaction\_database()**

Input:

transaction\_database\_infile – The transaction database input file.

Output:

transaction\_database – The transaction database data structure.

**Function load\_transaction\_database(transaction\_database\_infile)**

transaction\_database = dictionary()

Open transaction\_database\_infile

# Iterate over each row of the transaction database input file.

Foreach row in the transaction\_database\_infile:

(transaction\_id, transaction\_value) = row

transaction\_database[transaction\_id] = transaction\_value

return transaction\_database

**Algorithm large\_1\_itemset()**

Large 1-itemset generation

Input:

transaction\_database - The transaction database data structure.

Output:

large\_1\_itemset – The large 1 itemset data structure.

**Function large\_1\_itemset(transaction\_database)**

large\_1\_itemset = dictionary()

Foreach transaction\_id in transaction\_database:

transaction\_value = transaction\_database[transaction\_id]

# Split transaction value by ‘,’ delimiter.

items = transaction\_value.split(“,”)

foreach item in items:

# Get the items in the large 1-itemset.

large\_1\_itemset[item] = item

# Count the number of unique items in transaction database.

large-1-itemset\_counter[item] += 1

return large\_1\_itemset

**Algorithm apriori\_gen()**

Candidate generation and join step

Input:

large\_k\_itemset – The large k itemset data structure.

Output:

candidate\_k\_itemset - A superset of the set of all large k-itemsets.

Function apriori\_gen(large\_k\_itemset):

Insert into candidate\_k\_itemset

Select p.item1, p.item2, …. , p.itemk-1, q.itemk-1

From large\_k\_itemset p, large\_k\_itemset q

Where p.item1 = q.item1, … , p.itemk-2 = q.itemk-2

p.itemk-1 < q.itemk-1

return candidate\_k\_itemset

**Algorithm pruning\_step()**

The pruning step.

Input:

candidate\_k\_itemset - A superset of the set of all large k-itemsets.

Output:

large\_k\_itemset – The large k itemset data structure.

Function pruning\_step(candidate\_k\_itemset):

foreach itemset in candidate\_k\_itemset:

foreach (k – 1)-subsets s of itemset:

if (s not in large\_k\_itemset)

delete itemset from candidate\_k\_itemset

large\_k\_itemset = candidate\_k\_itemset

return large\_k\_itemset

**Algorithm Apriori()**

The apriori algorithm.

Input:

transaction\_database\_infile – The transaction database input file.

minimum\_support – The minimum support count.

Output:

frequent\_k\_itemsets – The frequent k-itemset data structure.

**Function Apriori(transaction\_database\_infile, minimum\_support)**

# Get the transaction database data structure.

transaction\_database = load\_transaction\_database(transaction\_database\_infile)

# Get the large 1-itemset from the transaction database data structure.

large\_1\_itemset = large\_1\_itemset(transaction\_database)

k = 2

while(large\_k\_itemset != null):

# Get the candidate k-itemset from the apriori generation function.

candidate\_k\_itemset = apriori\_gen(large\_k\_itemset)

# Iterate over each transaction in the transaction database.

foreach transaction\_id in transaction\_database:

# The transaction value entry.

transaction\_value = transaction\_database[transaction\_id]

# Obtain the subset of the candidate\_k\_itemset in the transaction\_value entry.

transaction\_candidates = subset(candidate\_k\_itemset, transaction\_value)

foreach transaction\_candidate in transaction\_candidates:

# Get support count of transaction\_candidate.

transaction\_candidate.count++

end

end

foreach candidate in candidate\_k\_itemset:

if(candidate.count >= minimum\_support)

large\_k\_itemset[candidate] = candidate

large\_k\_itemset[candidate] += 1

# Increment k by 1.

k++

end

frequent\_k\_itemsets[k] = large\_k\_itemset

return frequent\_k\_itemsets

**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/>

**Algorithm support**

support(itemX) = Number of transactions that itemX appears

function support(itemX, total\_transactions)

return support

**Algorithm confidence**

function confidence(itemX, itemY)

confidence(itemX 🡪 itemY) =

return confidence

**Algorithm lift**

function lift(itemX, itemY)

lift(itemX 🡪 itemY) =

return lift

**Algorithm leverage**

function leverage(itemX, itemY)

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

return leverage

**Algorithm conviction**

function conviction(itemX, itemY)

conviction(itemX 🡪 itemY) =

return leverage

**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 Data Structure**

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 Data Structure**

Transaction Database will be scanned for unique items.

**Data Structure**: Dictionary

**Key**: Each item in the itemset as the key string deliminated by comma ‘,’ characters.

**Value**: Each item in the itemset as the key string deliminated by comma ‘,’ characters.

**Large 1-Itemset Support Counter Data Structure**

The large 1-itemset support counter where each unique item is counted.

**Data Structure**: Dictionary

**Key**: Each item in the itemset as the key string deliminated by comma ‘,’ characters.

**Value**: The support count for that item.

**Candidate k Support Counter Data Structure**

**Data Structure**: Dictionary

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

**Key**: Each item in the itemset as the key string deliminated by comma ‘,’ characters.

**Value**: The support count for that item.

**Large k-Itemset Data Structure**

Transaction Database will be scanned for items.

**Data Structure**: Dictionary

**Key**: Each item in the itemset as the key string deliminated by comma ‘,’ characters.

**Value**: Each item in the itemset as the key string deliminated by comma ‘,’ characters.

**Large k-Itemset Support Counter Data Structure**

**Data Structure**: Dictionary

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

**Key**: Each item in the itemset as the key string deliminated by comma ‘,’ characters.

**Value**: The support count for that item.

All candidate data structures will have ordered Itemsets

**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 from the ML\_Project\_1. 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, lift, leverage and conviction 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 function

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.

**For the apriori\_gen and pruning step**

I needed to preset all itemset\_counters to 0 before counting number of itemsets in transaction.

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.

I needed to keep a dictionary data structure of itemset lengths.

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

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

The 1\_large\_itemset, apriori\_gen and pruning step were changed to the following;

generate\_candidate\_set1(self)

apriori\_gen(self, prev\_candidate\_itemset, k) = apriori\_gen

prune\_k\_itemset(self, candidate\_support\_count, candidate\_transaction\_ids)

All methods (functions) and code were well-documented to accommodate all changes and to make it easier for the next person to read the code and use the program.

**Apriori function**

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

I 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. Wherever it counts support (item frequency) then add transaction ids for the purpose of tracking which transactions have frequent itemsets.

Please see all methods (functions) in the source code for all changes as there were quite a few compared to the algorithm pseudocode and data structure design. This was due to implementation issues as they arose during development. The source code is well commented and documents these changes.

**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. I documented this in the metadata for the test datasets and are found in the test\_datasets directory.

As seen in the MDSC\_679/ML\_Project\_2/test\_datasets/test\_dataset1 folder. The transaction\_database1.tsv file was used as the transaction database used for implementation and development. The results were compared to the contents of the transaction\_database1\_metadata.txt file. The results were identical so I went on to test the other transaction databases in the MDSC\_679/ML\_Project\_2/test\_datasets/test\_dataset1 folder.

**Test Dataset 1**

After implementation was finished, I used the transaction\_database2.tsv and transaction\_database3.tsv files in the MDSC\_679/ML\_Project\_2/test\_datasets/test\_dataset1 folder to test the time and memory usage of the program. The results were compared to the contents of the transaction\_database2\_metadata.txt and transaction\_database3\_metadata.txt files. The results were identical for each transaction database.

As shown in the following table the time and memory usage of the program is as follows;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Database Name | Number of Transactions | Number of unique items | Number of Frequent Itemsets | Real Time | Memory Usage |
| transaction\_database1.tsv | 9 | 5 | 7 | 0m0.151s | Under 1GB |
| transaction\_database2.tsv | 5 | 4 | 11 | 0m0.140s | Under 1GB |
| transaction\_database3.tsv | 4 | 4 | 5 | 0m0.151s | Under 1GB |

Memory usage of Under 1GB was proposed because the slurm scheduler system records in GBs and there was 0 GBs of memory used. The “Under 1GB” for memory usage indicates that the seff slurm command read 0.00 GB for memory. So, I believe that this means that the RAM usage was under 1GB.

**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 making a script to do this. As I started to run out of time, I reserved this for the real genomic dataset. Instead, I just ran the script on the datasets in the test dataset 2 directory. The following table shows the results of this experiment.

**Test Dataset 2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Database Name | Number of Transactions | Number of unique items | Number of Frequent Itemsets | Real Time | Memory Usage |
| data.tsv | 5 | 29 | 0 | 1s | Under 1GB |
| data2.tsv | 274 | 9 | 488 | 1s | Under 1GB |
| data3.tsv | 502 | 9 | 484 | 1s | Under 1GB |
| data4.tsv | 9903 | 89 | no results | 1D:15h:33m:11s | 29.51 GB (Out of memory ?) |
| data5.tsv | 95286 | 2261 | no results | 14h:38m:12s | 30.73 GB (Out of memory ?) |
| data6.tsv | 83335 | 1787 | no results | 7h:48m:43s | 29.82 GB (Out of memory ?) |
| data7.tsv | 522661 | 154 | no results | 2d:00h:00m:22s | 5.75 GB (Out of memory ?) |
| kaggle.tsv | 23418 | 14 | 2491 | 00h:02m:12s | 23.77 MB |
| tesco.tsv | 5 | 17 | 46 | 1s | Under 1GB |
| tesco2.tsv | 8 | 6 | 8 | 1s | Under 1GB |

The “Under 1GB” for memory usage indicates that the seff slurm command read 0.00 GB for memory. Transaction databases data4.tsv, data5.tsv, data6.tsv and data7.tsv returned with MEMORY LIMIT EXCEEDED indicating that those runs had no results. This indicated to me that these datasets needed to be partitioned as directed in the instructions for using a “real” genomic dataset.

**Time Complexity of algorithm**

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

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

Time analysis based on input size using the github.com repository dataset. Since I ran out of time, I just estimated that the time complexity of my algorithm to be similar. It could be that my algorithm actually runs in a worse time complexity than stated.

**Genomic Data Example**

Use the dataset from the first project ML\_Project\_1.

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 is 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 32-40 SNP variants per SNP set. The genotypes will be formatted into SNP-Sets using the code in the quality\_control.py script from ML\_Project\_1. I need to use a Then having the possibility of more than 32 per gene I thought a better approach would be to choose 24. 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. The job array script I executed on arc.ucalgary.ca ran and terminated indicated that I ran out of memory. So, I thought that I would have to decrease the number of items in the dataset. Having the possibility of more than 24 per gene I thought a better approach would be to choose 10. The reason being that 210= 1024 is more of an obtainable goal.

Then the number 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 chromosomes 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 10 SNP variants per set. We can use modulus operator (%) to figure out the remainder. Calculation 3000/10 = 300. We then have 300 files partitioned for the job array and we can schedule individual jobs using the job array syntax in job submission shell scripts.

**Real Dataset 1**

Genotypes were filtered using MAF >= 0.01, p-value alpha < 0.00065 and the major allele (reference) was encoded as 0 and the minor allele (alternative) was encoded as 1 and phenotypes were filtered by missing values and filtered genotype ids in the quality control step of ML\_Project\_1. There were 230 genotyped individuals and 223 SNP variants used as input for the apriori algorithm. There were 23 files in this dataset. This dataset was used to show that I know how to partition files and submit them to the HPC cluster. Unfortunately, since this wasn’t a dataset to use for obtaining novel insights there were no novel findings. Please see the following table for test and resource usage statistics.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Database Name | Number of Transactions | Number of unique items | Number of Frequent Itemsets | Real Time | Memory Usage |
| apriori\_genotypes\_part1.tsv | 230 | 10 | 6743 | 1m26.049s | Under 1GB |
| apriori\_genotypes\_part10.tsv | 230 | 10 | 14937 | 1m22.346s | Under 1GB |
| apriori\_genotypes\_part11.tsv | 230 | 10 | 14438 | 1m22.257s | Under 1GB |
| apriori\_genotypes\_part12.tsv | 230 | 10 | 11141 | 1m29.302s | Under 1GB |
| apriori\_genotypes\_part13.tsv | 230 | 10 | 12108 | 1m22.383s | Under 1GB |
| apriori\_genotypes\_part14.tsv | 230 | 10 | 7305 | 1m24.263s | Under 1GB |
| apriori\_genotypes\_part15.tsv | 230 | 10 | 14052 | 1m26.435s | Under 1GB |
| apriori\_genotypes\_part16.tsv | 230 | 10 | 14056 | 1m17.587s | Under 1GB |
| apriori\_genotypes\_part17.tsv | 230 | 10 | 12027 | 1m25.496s | Under 1GB |
| apriori\_genotypes\_part18.tsv | 230 | 10 | 3533 | 1m25.985s | Under 1GB |
| apriori\_genotypes\_part19.tsv | 230 | 10 | 3030 | 1m21.504s | Under 1GB |
| apriori\_genotypes\_part2.tsv | 230 | 10 | 9980 | 1m20.779s | Under 1GB |
| apriori\_genotypes\_part20.tsv | 230 | 10 | 6170 | 1m22.480s | Under 1GB |
| apriori\_genotypes\_part21.tsv | 230 | 10 | 10739 | 1m24.884s | Under 1GB |
| apriori\_genotypes\_part22.tsv | 230 | 10 | 11141 | 1m26.248s | Under 1GB |
| apriori\_genotypes\_part23.tsv | 230 | 10 | 54 | 1m17.468s | Under 1GB |
| apriori\_genotypes\_part3.tsv | 230 | 10 | 3625 | 1m25.154s | Under 1GB |
| apriori\_genotypes\_part4.tsv | 230 | 10 | 4184 | 0m0.265s | Under 1GB |
| apriori\_genotypes\_part5.tsv | 230 | 10 | 12568 | 1m15.733s | Under 1GB |
| apriori\_genotypes\_part6.tsv | 230 | 10 | 14451 | 1m24.960s | Under 1GB |
| apriori\_genotypes\_part7.tsv | 230 | 10 | 9497 | 1m28.886s | Under 1GB |
| apriori\_genotypes\_part8.tsv | 230 | 10 | 13232 | 0m22.646s | Under 1GB |
| apriori\_genotypes\_part9.tsv | 230 | 10 | 4558 | 1m26.433s | Under 1GB |

**Real Dataset 2**

Genotypes were filtered using MAF >= 0.01, Bonferroni corrected adjusted p-value of alpha < 0.05 for filtering significant pvalues and the major allele (reference) was encoded as 0 and the minor allele (alternative) was encoded as 1 and phenotypes were filtered by missing values and filtered genotype ids in the quality control step. There were 230 genotyped individuals and 9 SNP variants used as input for the apriori algorithm. There were no novel findings. Please see the following table for test and resource usage statistics.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Database Name | Number of Transactions | Number of unique items | Number of Frequent Itemsets | Real Time | Memory Usage |
| apriori\_genotypes\_part9.tsv | 230 | 9 | 70499 | 00h:00m:23s | Under 1GB |

**Computation Resources**

Program implementation, development and analyses were performed on a MacBook Pro running MacOSX Big Sur with Quad-Core Intel Core i5 2 GHz (4 cores) and 32 GB RAM memory.

The Aprori Algorithm was performed on the test\_dataset1, test\_dataset2 and real genomic dataset from ML\_Project\_1 on a HPC cluster running Slurm as the job scheduler (arc.ucalgary.ca) in interactive mode using the cpu2019 partition with 38 GB RAM on one CPU per node for each dataset.

**Data and Source code Availability.**

All python scripts (commented), Installation documentation, genotypes and phenotype data are available at the following github repositiory <https://github.com/kevmu/MDSC_679>. Follow the README.md for github repository download and installation instructions for the ML\_Project\_2 pipeline.

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