**APRIORI ALGORITHM**

Rubén Ventura

9305420195

**CONTENT**

[SUMMARY OF THE ALGORITHM 3](#_Toc36981632)

[DESCRIPTION OF THE CODE 4](#_Toc36981633)

[INPUT/OUTPUT 4](#_Toc36981634)

[**read\_file():** 4](#_Toc36981635)

[**write\_file( rules, support)** 4](#_Toc36981636)

[APRIORI ALGORITHM 5](#_Toc36981637)

[**apriori( transactions, minSup)** 5](#_Toc36981638)

[**generate\_1\_itemset\_candidates( transactions)** 6](#_Toc36981639)

[**generate\_k\_itemset\_candidates( frequent\_sets, k)** 6](#_Toc36981640)

[**prune( transactions, candidates, minSup)** 7](#_Toc36981641)

[ASSOCIATION RULES 8](#_Toc36981642)

[**get\_association\_rules( frequent\_itemsets, itemsets\_support)** 8](#_Toc36981643)

[**generate\_more\_rules( itemset, associative\_itemsets, itemsets\_support, rules)** 8](#_Toc36981644)

[**calculate\_confidence( itemset, associative\_item\_sets, itemsets\_support, rules)** 9](#_Toc36981645)

[INSTRUCTIONS FOR COMPILING 9](#_Toc36981646)

[ADDITIONAL SPECIFICATIONS 9](#_Toc36981647)

# **SUMMARY OF THE ALGORITHM**

I decided to develop the program using Python, since it’s a language I’m not too familiar with yet and I really want to learn it (since it’s one of the most useful and popular programming languages at the moment).

I tried to split my code in 4 different parts. The first one is the little bit of arguments testing before starting the algorithm.

Next one is the Apriori algorithm. Firstly, the program reads the input file and formats it in the way I considered would work the best given my limited knowledge (using the set data type mostly). After that, it checks all the transactions trying to find different items and counts how many times they appear. Once this is done, it prunes the list, removing those whose support is below the threshold. Once we have the first generation of frequent items, we set a loop that keeps generating k+1 candidates (starting at 2), pruning them and adding the new frequent sets and support to the dictionaries, and will end when the generated list is empty.

When the loop ends, we have two dictionaries with all the frequent itemsets and their support. We then proceed to find the association rules for each itemset, where we basically fetch the frequent itemsets recursively for all the candidates we can find. When no more candidates can be found, we calculate the confidence of each association and store the itemset, its associations and their confidences in a list called rules.

Finally, when we have gathered all the data we were asked for, we proceed to write it in the output file specified in the arguments.

# **DESCRIPTION OF THE CODE**

## ***· INPUT/OUTPUT***

### **read\_file():**

#Read transaction file and return the transactions as collections of ints

def read\_file():

    #Each line is a transaction

    #Each item is separated by a tab

    transactions = []

    with open(filePath + sys.argv[inputFilePosition], 'r') as f:

        for transaction in f.read().split('\n'):

            #a) If we want the items to be stored as strings (slower)

            #transactions.append(transaction.split('\t'))

            #b) If we want the items to be stored as integers and not strings (faster):

            transactions.append( [int(item) for item  in transaction.split('\t')])

    return transactions

We open the file specified by the user (with read permission only), read each line and store it as a transaction and then get every item in those lines, separated by a \t and save it as an integer in a list.

### **write\_file( rules, support)**

#Writing the solution in the output file

def write\_file(rules, support\_dic):

    with open(filePath + sys.argv[outputFilePosition], 'w') as f:

        #Write the data

        for i in range(len(rules)):

            pattern = set(rules[i][0])

            associations = rules[i][1]

            support = support\_dic[rules[i][0]]

            confidence = rules[i][2]

            f.write(str(pattern) + "\t" + str(associations) + "\t" + str(support) + "\t" + str(confidence) + "\n")

We open the file specified by the user (gets created if it didn’t exist) and proceed to write every piece of information we need. The frequent itemsets, associative itemsets and confidence are all stored in the rules list in positions 0, 1, 2, respectively, while support\_dic has all the pairs of frequent itemsets and their support. We iterate through the rules list and get the information we need and write it in the file. Rerunning the program multiple times will overwrite the file.

## ***· APRIORI ALGORITHM***

### **apriori( transactions, minSup)**

#APRIORI

def apriori(transactions, minSup):

    #1 - Get frequent 1-itemsets

    candidates\_1 = generate\_1\_itemset\_candidates(transactions)

    #Reformat transactions from a list of lists of integers to a list of sets

    transactions\_sets = list(map(set, transactions))

    #Get the first list of frequent 1-itemsets and their support

    frequent\_1\_itemsets, itemsets\_support = prune(transactions\_sets, candidates\_1, minSup)

    #Initialize list of solutions with the first patterns

    frequent\_itemsets = [frequent\_1\_itemsets]

    #Set k=2 (k+1) to start looping until we run out of patterns

    k = 2

    while (len(frequent\_itemsets[k-2]) > 0):

        #Generate k-itemset candidates

        k\_itemset\_candidates = generate\_k\_itemset\_candidates(frequent\_itemsets[k-2], k)

        #Prune

        frequent\_k\_itemsets, k\_itemsets\_support = prune(transactions\_sets, k\_itemset\_candidates, minSup)

        #Update the support dictionary with the new itemsets

        itemsets\_support.update(k\_itemsets\_support)

        #If the new generated itemset list is empty, we exit the loop

        if frequent\_k\_itemsets == []:

            break

        frequent\_itemsets.append(frequent\_k\_itemsets)

        k += 1

    return frequent\_itemsets, itemsets\_support

We execute the Apriori algorithm. We generate 1-itemsets from the transactions we receive as argument. We then reformat the transaction list so that it’s easier to work with it and proceed to get the frequent 1-itemsets and their support. After this first frequent itemsets are identified, we prepare a loop which will iterate starting at 2 and will generate candidate itemsets of length 2, prune the list of candidates and add the remaining itemsets (frequent itemsets) and their support to the frequent itemsets list and the support dictionary. The loop will keep on going until no more candidates are generated.

### **generate\_1\_itemset\_candidates( transactions)**

#Gets each different item in the database and returns a list of sets

# (needs to be frozenset in order to be used as keys in other parts of the code)

# we need to add the individual items as lists of items

def generate\_1\_itemset\_candidates(transactions):

    candidates = []

    for transaction in transactions:

        for item in transaction:

            if [item] not in candidates:

                candidates.append([item])

    candidates.sort()

    return list(map(frozenset, candidates))

Iterates through the list of transactions, looking for new items and stores them in the candidates list. It’s important to note we add the candidates to the list as a list of only one item instead of reading them as an integer, since we need it to be iterable. Also, we make sure to reformat the list and set the candidates as frozenset, since we need them to be inmutable in order to act as keys for our support dictionay.

### **generate\_k\_itemset\_candidates( frequent\_sets, k)**

#Gets different combinations of k+1 items, given frequent k-itemsets

def generate\_k\_itemset\_candidates(frequent\_sets, k):

    candidate\_k\_itemsets = []

    lenFrequent\_sets = len(frequent\_sets)

    #There's probably a more efficient method for doing this

    for i in range(lenFrequent\_sets):

        for j in range(i+1, lenFrequent\_sets):

            combinations1 = list(frequent\_sets[i])[:k-2]

            combinations2 = list(frequent\_sets[j])[:k-2]

            combinations1.sort()

            combinations2.sort()

            if combinations1 == combinations2:

                candidate\_k\_itemsets.append(frequent\_sets[i] | frequent\_sets[j])

    return candidate\_k\_itemsets

Similarly to the last function, we generate new candidates, but this time we look for all the possible combinations of the given items.

### **prune( transactions, candidates, minSup)**

#We compare each candidate with the database and return a list of the frequent ones

#We also take the chance to get the support of each new itemset

def prune(transactions, candidates, minSup):

    support\_count = {}

    for transaction in transactions:

        for candidate in candidates:

            if candidate.issubset(transaction):

                if candidate not in support\_count:

                    support\_count[candidate] = 1

                else:

                    support\_count[candidate] += 1

    lenTransactions = float(len(transactions))

    frequent\_sets = []

    itemsets\_support = {}

    for itemset in support\_count:

        support = float("{:.2f}".format(support\_count[itemset] / lenTransactions))

        if support >= minSup:

            frequent\_sets.insert(0,itemset)

        itemsets\_support[itemset] = support

    return frequent\_sets, itemsets\_support

We take each of the candidates we’re given and start counting how many times it appears in the original transaction database (we are able to use the .issubset(superSet) method since we reformatted the transactions) and store the value in a temporary dictionary. Once we’re finished counting, we proceed to calculate the support of each itemsets, select those whose support is above the threshold, update the support value in the dictionay (even if not selected) and add them to frequent\_sets.

## ***· ASSOCIATION RULES***

### **get\_association\_rules( frequent\_itemsets, itemsets\_support)**

#For all the itemsets in the frequent\_itemsets list we search for possible association rules and calculate their confidence

def get\_association\_rules(frequent\_itemsets, itemsets\_support):

    rules = []

    for i in range(1, len(frequent\_itemsets)):

        #Get sets with 2 or more items

        for itemset in frequent\_itemsets[i]:

            #change to frozenset if it needs to be used as index

            associative\_item\_sets = [set([item]) for item in itemset]

            #Try to generate more association rules

            if (i > 1):

                generate\_more\_rules(itemset, associative\_item\_sets, itemsets\_support, rules)

            else:

                calculate\_confidence(itemset, associative\_item\_sets, itemsets\_support, rules)

    return rules

We select each frequent itemset (consisting of at least 2 items, less than that would not make sense) and try to get as many associative\_items as possible for each item. If only one associative item is identified, we proceed to calculate the confidence of the association. If we identify more than one, we try to get more associations. The rules list is updated by calculate\_confidence( ) and it’s returned at the end with all the frequent items, their association rules and the confidence for each rule.

### **generate\_more\_rules( itemset, associative\_itemsets, itemsets\_support, rules)**

#We try to generate more associations recursively

def generate\_more\_rules(itemset, associative\_item\_sets, itemsets\_support, rules):

    lais = len(associative\_item\_sets[0])

    candidates = []

    #Try to merge sets

    if (len(itemset) > (lais + 1)):

        candidates = generate\_k\_itemset\_candidates(associative\_item\_sets, lais + 1)

        candidates = calculate\_confidence(itemset, candidates, itemsets\_support, rules)

        #Merge sets

        if (len(candidates) > 1):

            generate\_more\_rules(itemset, candidates, itemsets\_support, rules)

For each associative\_itemset we are given, we try to generate the largest itemset possible (by using the generate\_k\_itemsets). If we find a large associative itemset, we calculate the confidence (and add it to the rules list). If also there are candidates left, we redo the whole operation with those candidates.

### **calculate\_confidence( itemset, associative\_item\_sets, itemsets\_support, rules)**

#Calculate confidence for each association, append to the rules and return the new associations list

#(in order to use it to generate more rules)

def calculate\_confidence(itemset, associative\_item\_sets, itemsets\_support, rules):

    associations = []

    for association in associative\_item\_sets:

        confidence = float("{:.2f}".format(itemsets\_support[itemset] / itemsets\_support[itemset - association]))

        rules.append((itemset-association, association, confidence))

        associations.append(association)

    return associations

We calculate the confidence for each association we send as a parameter. We format it as we are told to do and store it in the rules list. We return the associations list, since it’s used as the candidates for further iterations of the generate\_more\_rules( ) function.

# **INSTRUCTIONS FOR COMPILING**

The whole program is written python 3.x, in a single .py file. It expects 3 arguments as input, in this order: minimum support, input file, output file. So, going to the directory where file is, the command for compiling would be:

python apriori.py [minimum support] [input file name] [output file name]

# **ADDITIONAL SPECIFICATIONS**

Minimum support is expected to be an integer, ranging from 1 to 100 (both included).

Both input and output files will share the same path, which can be changed easily. I decided to set by default as “../”, (this way the code stays in the project folder and the txt files are kept outside) so the input file is expected to be in the directory above the “project\_ apriori”folder.