# EXPERIMENT NO. 9

**Aim:** Implementation of Association rule Mining.(Apriori algorithm)

**Software Used:** Java/C/Python

**Theory:**

Frequent patterns are patterns (such as itemsets, subsequences, or substructures) that appear in a data set frequently. For example, a set of items, such as milk and bread, that appear frequently together in a transaction data set is a frequent itemset. Finding such frequent patterns plays an essential role in mining associations, correlations, and many other interesting relationships among data.

Moreover, it helps in data classification, clustering, and other data mining tasks as well. Thus, frequent pattern mining has become an important data mining task and a focused theme in data mining research.

The Apriori Algorithm: Finding Frequent Itemsets Using Candidate Generation

Apriori is a seminal algorithm proposed by R. Agrawal and R. Srikant in 1994 for mining frequent itemsets for Boolean association rules. The name of the algorithm is based on the fact that the algorithm uses prior knowledge of frequent itemset properties, as we shall see following. Apriori employs an iterative approach known as a level-wise search, where k-itemsets are usedtoexplore (k+1)-itemsets. First, the setof frequent 1-itemsets is found by scanning the database to accumulate the count for each item, and collecting those items that satisfy minimum support. The resulting set is denoted L1.Next, L1 is used to find L2, the set of frequent 2-itemsets, which is used to find L3, and so on, until no more frequent k-itemsets can be found. The finding of each Lk requires one full scan of the database.

To improve the efficiency of the level-wise generation of frequent itemsets, an important property called the Apriori property, presented below, is used to reduce the search space.We will first describe this property, and then show an example illustrating its use.

Apriori property: All nonempty subsets of a frequent itemset must also be frequent A two-step process is followed, consisting of join and prune actions.

**Algorithm: Apriori.** Find frequent itemsets using an iterative level-wise approach based on candidate generation.

**Input:**

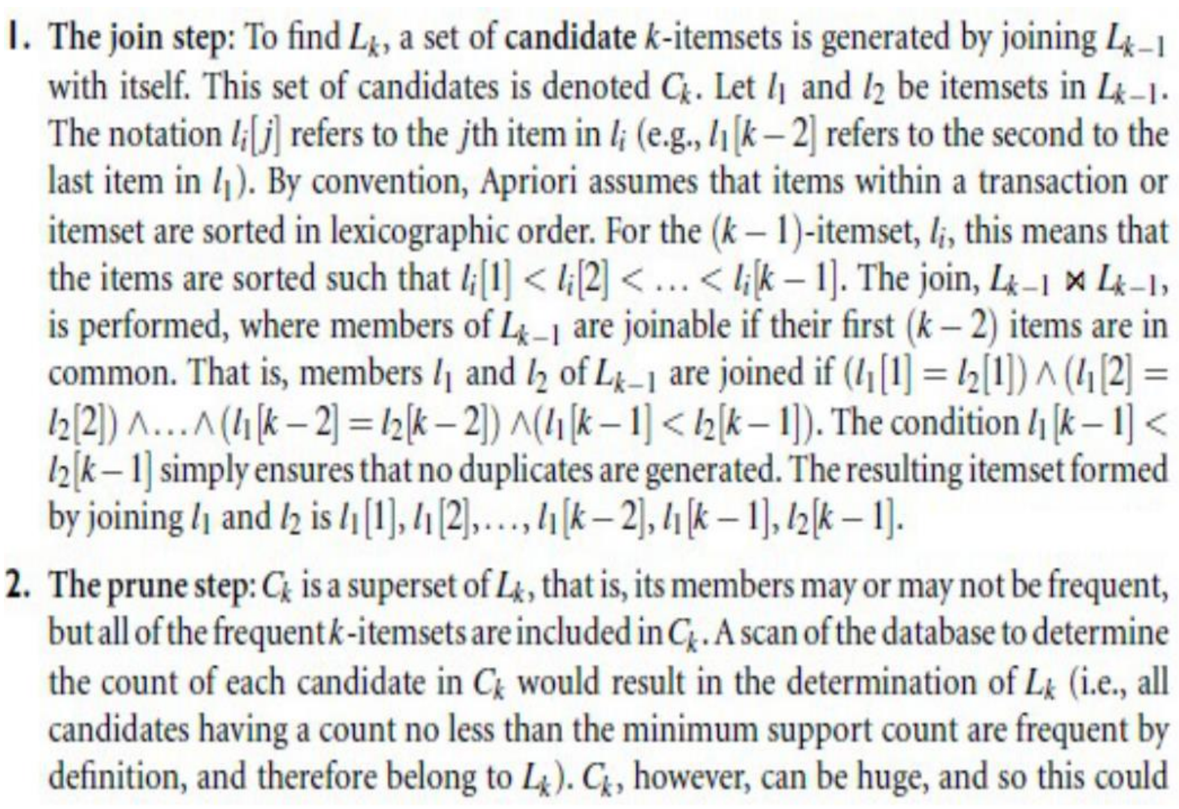
- *D*, a database of transactions;
- *min_sup*, the minimum support count threshold.

**Output:** *L*, frequent itemsets in *D*.

**Method:**

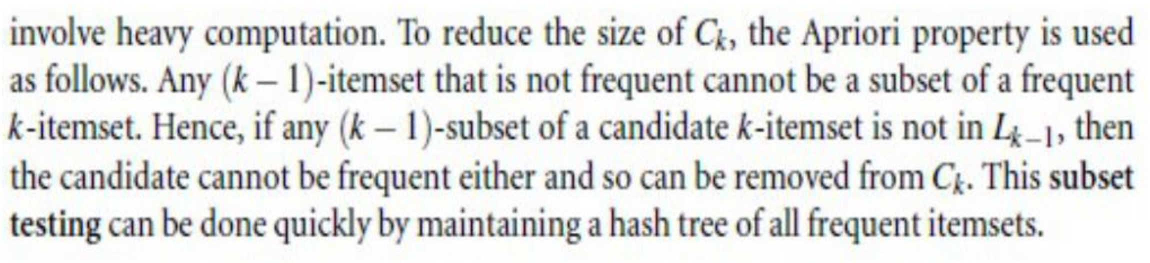
```
(1)   $L_1$ = find_frequent_1-itemsets(D);
(2)   for ($k = 2; L_{k-1} \neq \phi; k++$) {
(3)       $C_k$ = apriori_gen($L_{k-1}$);
(4)       for each transaction $t \in D$ { // scan $D$ for counts
(5)           $C_t$ = subset($C_k, t$); // get the subsets of $t$ that are candidates
(6)           for each candidate $c \in C_t$
(7)               c.count++;
(8)       }
(9)       $L_k = \{c \in C_k | c.count \geq min\_sup\}$
(10)  }
(11)  return $L = \cup_k L_k$;

procedure apriori_gen($L_{k-1}$:frequent $(k-1)$-itemsets)
(1)   for each itemset $l_1 \in L_{k-1}$
(2)       for each itemset $l_2 \in L_{k-1}$
(3)           if $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge ... \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$ then {
(4)               $c = l_1 \bowtie l_2$; // join step: generate candidates
(5)               if has_infrequent_subset($c, L_{k-1}$) then
(6)                   delete $c$; // prune step: remove unfruitful candidate
(7)               else add $c$ to $C_k$;
(8)           }
(9)   return $C_k$;

procedure has_infrequent_subset($c$: candidate $k$-itemset;
            $L_{k-1}$: frequent $(k-1)$-itemsets); // use prior knowledge
(1)   for each $(k-1)$-subset $s$ of $c$
(2)       if $s \notin L_{k-1}$ then
(3)           return TRUE;
(4)   return FALSE;
```

The Apriori algorithm for discovering frequent itemsets for mining Boolean association rules.



1. **The join step**: To find $L_k$, a set of **candidate** $k$-itemsets is generated by joining $L_{k-1}$ with itself. This set of candidates is denoted $C_k$. Let $l_1$ and $l_2$ be itemsets in $L_{k-1}$. The notation $l_i[j]$ refers to the $j$th item in $l_i$ (e.g., $l_1[k-2]$ refers to the second to the last item in $l_1$). By convention, Apriori assumes that items within a transaction or itemset are sorted in lexicographic order. For the $(k-1)$-itemset, $l_i$, this means that the items are sorted such that $l_i[1] < l_i[2] < \ldots < l_i[k-1]$. The join, $L_{k-1} \bowtie L_{k-1}$, is performed, where members of $L_{k-1}$ are joinable if their first $(k-2)$ items are in common. That is, members $l_1$ and $l_2$ of $L_{k-1}$ are joined if $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge \ldots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$. The condition $l_1[k-1] < l_2[k-1]$ simply ensures that no duplicates are generated. The resulting itemset formed by joining $l_1$ and $l_2$ is $l_1[1], l_1[2], \ldots, l_1[k-2], l_1[k-1], l_2[k-1]$.

2. **The prune step**: $C_k$ is a superset of $L_k$, that is, its members may or may not be frequent, but all of the frequent $k$-itemsets are included in $C_k$. A scan of the database to determine the count of each candidate in $C_k$ would result in the determination of $L_k$ (i.e., all candidates having a count no less than the minimum support count are frequent by definition, and therefore belong to $L_k$). $C_k$, however, can be huge, and so this could



involve heavy computation. To reduce the size of $C_k$, the Apriori property is used as follows. Any $(k-1)$-itemset that is not frequent cannot be a subset of a frequent $k$-itemset. Hence, if any $(k-1)$-subset of a candidate $k$-itemset is not in $L_{k-1}$, then the candidate cannot be frequent either and so can be removed from $C_k$. This **subset testing** can be done quickly by maintaining a hash tree of all frequent itemsets.

**Advantages**

- It is an easy-to-implement and easy-to-understand algorithm.
- It can be used on large itemsets.

**Disadvantages**

- Sometimes, it may need to find a large number of candidate rules which can be computationally expensive.
- Calculating support is also expensive because it has to go through the entire database.

Consider the following example. Before beginning the process, let us set the support threshold to 50%, i.e. only those items are significant for which support is more than 50%.

**Example:**

Step 1: Create a frequency table of all the items that occur in all the transactions. For our case:

| Item | Frequency (No. of transactions) |
|---|---|
| Onion(O) | 4 |
| Potato(P) | 5 |
| Burger(B) | 4 |
| Milk(M) | 4 |
| Beer(Be) | 2 |

Step 2: We know that only those elements are significant for which the support is greater than or equal to the threshold support. Here, support threshold is 50%, hence only those items are significant which occur in more than three transactions and such items are Onion(O), Potato(P), Burger(B), and Milk(M). Therefore, we are left with:

| Item | Frequency (No. of transactions) |
|---|---|
| Onion(O) | 4 |
| Potato(P) | 5 |
| Burger(B) | 4 |
| Milk(M) | 4 |

The table above represents the single items that are purchased by the customers frequently.

Step 3: The next step is to make all the possible pairs of the significant items keeping in mind that the order doesn’t matter, i.e., AB is same as BA. To do this, take the first item and pair it with all the others such as OP, OB, OM. Similarly, consider the second item and pair it with preceding items, i.e., PB, PM. We are only considering the preceding items because PO (same as OP) already exists. So, all the pairs in our example are OP, OB, OM, PB, PM, BM.

Step 4: We will now count the occurrences of each pair in all the transactions.

| Itemset | Frequency (No. of transactions) |
|---|---|
| OP | 4 |
| OB | 3 |
| OM | 2 |
| PB | 4 |
| PM | 3 |
| BM | 2 |

Step 5: Again only those itemsets are significant which cross the support threshold, and those are OP, OB, PB, and PM.

Step 6: Now let’s say we would like to look for a set of three items that are purchased together. We will use the itemsets found in step 5 and create a set of 3 items.

To create a set of 3 items another rule, called self-join is required. It says that from the item pairs OP, OB, PB and PM we look for two pairs with the identical first letter and so we get

OP and OB, this gives OPB PB and PM, this gives PBM

Next, we find the frequency for these two itemsets.

| Itemset | Frequency (No. of transactions) |
|---|---|
| OPB | 4 |
| PBM | 3 |

Applying the threshold rule again, we find that OPB is the only significant itemset.

Therefore, the set of 3 items that was purchased most frequently is OPB.

The example that we considered was a fairly simple one and mining the frequent itemsets stopped at 3 items but in practice, there are dozens of items and this process could continue to many items. Suppose we got the significant sets with 3 items as OPQ, OPR, OQR, OQS and PQR and now we want to generate the set of 4 items. For this, we will look at the sets which have first two alphabets common, i.e, OPQ and OPR gives OPQR OQR and OQS gives OQRS

In general, we have to look for sets which only differ in their last letter/item.

Applications:

- Market Basket Analysis
- Network Forensics analysis
- Analysis of diabetic databases
- Adverse drug reaction detection Ecommerce

**Customer analysis**

**PROGRAM:**

```
data = [
        ['T100',['I1','I2','I5']],
        ['T200',['I2','I4']],
        ['T300',['I2','I3']],
        ['T400',['I1','I2','I4']],
        ['T500',['I1','I3']],
        ['T600',['I2','I3']],
        ['T700',['I1','I3']],
        ['T800',['I1','I2','I3','I5']],
        ['T900',['I1','I2','I3']]
        ]
```

```
init = []
for i in data:
    for q in i[1]:
        if(q not in init):
            init.append(q)
init = sorted(init)
print(init)

sp = 0.4
s = int(sp*len(init))
s

from collections import Counter

c = Counter()
for i in init:
    for d in data:
        if(i in d[1]):
            c[i]+=1
print("C1:")
for i in c:
    print(str([i])+": "+str(c[i]))
print()
l = Counter()
for i in c:
    if(c[i] >= s):
        l[frozenset([i])]+=c[i]
print("L1:")
for i in l:
    print(str(list(i))+": "+str(l[i]))
print()
pl = l
pos = 1
for count in range (2,1000):
    nc = set()
    temp = list(l)
    for i in range(0,len(temp)):
        for j in range(i+1,len(temp)):
            t = temp[i].union(temp[j])
            if(len(t) == count):
                nc.add(temp[i].union(temp[j]))
    nc = list(nc)
    c = Counter()
    for i in nc:
        c[i] = 0
        for q in data:
            temp = set(q[1])
            if(i.issubset(temp)):
                c[i]+=1
    print("C"+str(count)+":")
    for i in c:
        print(str(list(i))+": "+str(c[i]))
    print()
    l = Counter()
    for i in c:
        if(c[i] >= s):
            l[i]+=c[i]
    print("L"+str(count)+":")
    for i in l:
        print(str(list(i))+": "+str(l[i]))
    print()
    if(len(l) == 0):
        break
    pl = l
    pos = count
print("Result: ")
print("L"+str(pos)+":")
for i in pl:
    print(str(list(i))+": "+str(pl[i]))
print()

from itertools import combinations
for l in pl:
    c = [frozenset(q) for q in combinations(l,len(l)-1)]
    mmax = 0
    for a in c:
        b = l-a
        ab = l
        sab = 0
        sa = 0
        sb = 0
        for q in data:
            temp = set(q[1])
            if(a.issubset(temp)):
                sa+=1
            if(b.issubset(temp)):
                sb+=1
            if(ab.issubset(temp)):
                sab+=1
        temp = sab/sa*100
        if(temp > mmax):
            mmax = temp
        temp = sab/sb*100
        if(temp > mmax):
            mmax = temp
        print(str(list(a))+" -> "+str(list(b))+" = "+str(sab/sa*100)+"%")
        print(str(list(b))+" -> "+str(list(a))+" = "+str(sab/sb*100)+"%")
    curr = 1
    print("choosing:", end=' ')
    for a in c:
        b = l-a
        ab = l
        sab = 0
        sa = 0
        sb = 0
        for q in data:
            temp = set(q[1])
            if(a.issubset(temp)):
                sa+=1
            if(b.issubset(temp)):
                sb+=1
            if(ab.issubset(temp)):
                sab+=1
        temp = sab/sa*100
        if(temp == mmax):
            print(curr, end = ' ')
        curr += 1
        temp = sab/sb*100
        if(temp == mmax):
            print(curr, end = ' ')
        curr += 1
    print()
    print()
```

**Output :**

```
['I1', 'I2', 'I3', 'I4', 'I5']
C1:
['I1']: 6
['I2']: 7
['I3']: 6
['I4']: 2
['I5']: 2

L1:
['I1']: 6
['I2']: 7
['I3']: 6
['I4']: 2
['I5']: 2

C2:
['I5', 'I2']: 2
['I3', 'I5']: 1
['I4', 'I1']: 1
['I3', 'I2']: 4
['I3', 'I1']: 4
['I2', 'I1']: 4
['I5', 'I1']: 2
['I4', 'I2']: 2
['I3', 'I4']: 0
['I4', 'I5']: 0

L2:
['I5', 'I2']: 2
['I3', 'I2']: 4
['I3', 'I1']: 4
['I2', 'I1']: 4
['I5', 'I1']: 2
['I4', 'I2']: 2

C3:
['I4', 'I5', 'I2']: 0
['I3', 'I5', 'I1']: 1
['I3', 'I1', 'I2']: 2
['I1', 'I5', 'I2']: 2
['I3', 'I4', 'I2']: 0
['I2', 'I4', 'I1']: 1
['I3', 'I5', 'I2']: 1

L3:
['I3', 'I1', 'I2']: 2
['I1', 'I5', 'I2']: 2

C4:
['I5', 'I3', 'I2', 'I1']: 1

L4:
```

```
Result:
L3:
['I3', 'I1', 'I2']: 2
['I1', 'I5', 'I2']: 2

['I3', 'I1'] -> ['I2'] = 50.0%
['I2'] -> ['I3', 'I1'] = 28.57142857142857%
['I3', 'I2'] -> ['I1'] = 50.0%
['I1'] -> ['I3', 'I2'] = 33.33333333333333%
['I2', 'I1'] -> ['I3'] = 50.0%
['I3'] -> ['I2', 'I1'] = 33.33333333333333%
choosing: 1 3 5

['I5', 'I1'] -> ['I2'] = 100.0%
['I2'] -> ['I5', 'I1'] = 28.57142857142857%
['I2', 'I1'] -> ['I5'] = 50.0%
['I5'] -> ['I2', 'I1'] = 100.0%
['I5', 'I2'] -> ['I1'] = 100.0%
['I1'] -> ['I5', 'I2'] = 33.33333333333333%
choosing: 1 4 5
```

**CONCLUSION:** The different association mining algorithms of data mining were studied and one among them named Apriority association mining algorithm was implemented using Python. The need for association mining algorithm was recognized and understood

**SIGN AND REMARK**

**DATE**

| R1 | R2 | R3 | R4 | R5 | Total | Sign |
|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |