Association Analysis

Mohammed Abdul Kamran (m37), Areebuddin Aatif Mohammed Khaja (areebudd)
School of Engineering and Applied Sciences
University at Buffalo

Abstract

Given a dataset on gene expressions and various diseases, we implement Apriori algorithm to find all frequent itemsets for different support values. From the frequent itemsets we generate association rules satisfying given support and confidence requirements. Finally, we filter the generated rules based on template queries.

1 Introduction

Association Analysis is a two-step process, where first step is to find all itemsets whose support is greater than or equal to the user defined minimum threshold (also called frequent itemsets). We use Apriori algorithm to get all frequent itemsets.

In the second step we generate association rules of the form $X \rightarrow Y$ where X and Y are two distinct set of items. We prune rules which does not satisfy the minimum support and minimum confidence.

1.1 Dataset

For below implementation we use gene expressions data. Each row in data corresponds to a patient sample and each column corresponds to a gene with a binary value (Up or Down). Last column consists of disease names for each sample. Each gene and disease is considered as an item.

2 Implementation

There are three main tasks involved in association rule generation. Task 1 describes how each frequent itemsets are generated using Apriori algorithm. Task 2 describes how different association rules are generated from frequent itemsets. In task 3 we query these rules based on given templates and store all these results in a file.

Figure 1 and Figure 2 shows the flow of the program execution.

Figure 1: Association Rules generation

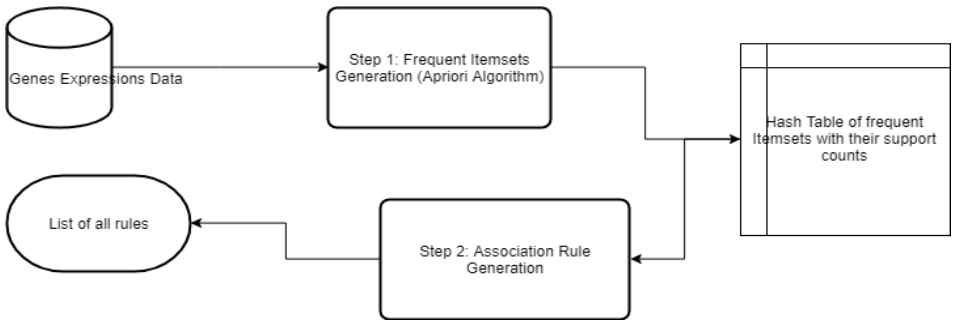
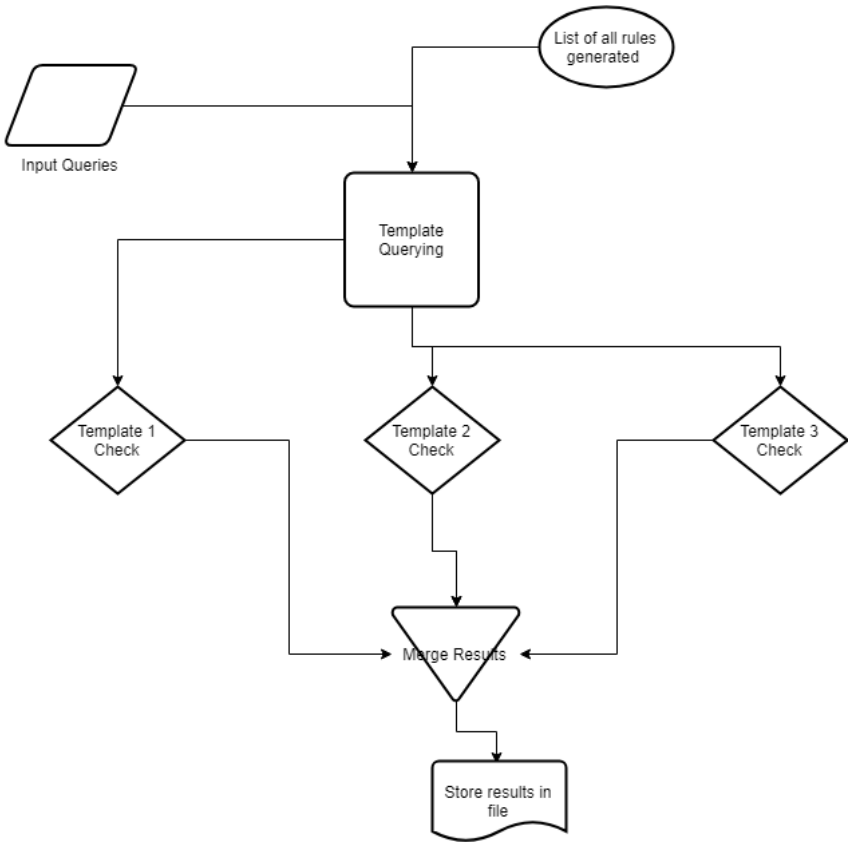


Figure 2: Querying Rules



2.1 Task 1: Frequent Itemsets Generation

We read data from the given dataset and have to find all the possible itemsets starting from length 1 itemsets. The brute force approach is to find all possible combinations of itemsests (2^d $where\ d\ is\ numbers\ of\ items$) and then check if this itemset is frequent or not. An itemset is frequent if the support ≥ minimum support. This is computationally expensive.

2.1.1 Apriori Algorithm

Apriori algorithm is an efficient method of getting frequent itemsests. This algorithm is based on Apriori Principle.

Apriori Principle: All subsets of a frequent itemset must also be frequent. This principle holds true because of the anti-monotone property of support.
Anti-Monotone Property of Support: This property states that the support of an itemset never exceeds the support of its subsets.

1. Generate all frequent itemsets of length 1.
2. Generate length k+1 candidate itemsets from length k itemsets.
3. Prune itemsets containing subsets of length k which are infrequent.
4. Count the support of each candidate itemset.
5. Eliminate the candidates that are infrequent.
6. Repeat above steps until there are no frequent itemsets.

Support of $\{X, Y\}$ = Support Count of $\{X, Y\}$ / Total Number of Transactions. A high level of support implies that the rule is frequent enough.

2.1.2 Code Snippets

Figure 3: Generate length 1 itemsets

```
def generate1ItemSet(data, min_sup):
    C = {}
    size = len(data)
    L = []
    dictionary = dict()
    for row in range(len(data)):
        for col in range(len(data[row])):
            if data[row][col] in C:
                C[data[row][col]] += 1
            else:
                C[data[row][col]] = 1
```

Above figure shows the code to create all length 1 itemsets. Then from length 1 itemsets we prune the itemsets not satisfying minimum support. Figure 4 shows the implementation to check if an itemset satisfies minimum support.

Figure 4: Minimum support check

```
count = 0
for item in C:
    if C[item]/size >= min_sup:
        tmp = set()
        tmp.add(item)
        dictionary[frozenset(tmp)] = C[item]
        L.append(tmp)
        count+=1
```

Figure 5 code snippet finds all candidate itemsets with length k+1 from length k frequent itemsets and removes infrequent itemsets. This code loops until there are no frequent itemsets identified.

Figure 5: Generate k+1 itemsets

```
while True:
    K += 1
    l = set()
    for a in L:
        for b in L:
            tmp = a | b
            tmp = sorted(tmp)
            if len(tmp) == K:
                l.add(frozenset(tmp))
    L = []
    for a in l:
        count = 0
        for i in dataSet:
            if a.issubset(i):
                count+=1
        if count/size >= min_sup:
            L.append(a)
            result[a] = count
```

2.2 Task 2: Association Rules Generation

From step 1 we get all the frequent itemsets satisfying minimum support threshold value as a dictionary where key is the frequent itemset and the value is the support count of each itemset. Since we are storing sets as keys in dictionary and to distinguish each distinct key we have to use frozenset which are immutable version of sets in python.

2.2.1 Algorithm

In brute force approach we have to generate all $2^k - 2$ candidate association rules (where k is length of frequent itemset). This approach is computationally expensive.

For each itemset in the dictionary of frequent itemsets, we run association rule generation algorithm.

1. We generate candidate rules using binary partitioning on a itemset.
2. At each level of binary partitioning of itemset we prune candidate rules whose confidence falls below the minimum threshold.
3. According to the anti-monotone property of the confidence, if the confidence of a candidate rule is less than the minimum threshold then we can prune all the rules which are subset of this rule. The anti-monotone property only works if the candidate rules belong to the same itemset.

Confidence of $\{X \rightarrow Y\}$ = Support Count of $\{X, Y\}$ / Support Count of $\{X\}$. A high level of confidence shows that the rule is true enough to justify the decision based on it.

2.2.2 Code Snippets

Figure 6: Confidence check

```
def checkMinConfidence(self, rule):
    union = rule[0] | rule[1]
    supCount = int(self.freqItemsets[union])
    if (supCount / int(self.freqItemsets[rule[0]])) >= self.min_conf:
        return True
    return False
```

Figure 6 shows the code to find the confidence of each rule based on the support count and then we check if this rule has confidence ≥ minimum confidence.

Figure 7: Binary partitioning of rules

```
while size > 2:
    temp = []
    combinations = itertools.combinations(prev, 2)
    for combination in combinations:
        intersection = combination[0][0] & combination[1][0]
        union = combination[0][1] | combination[1][1]
        if len(intersection) == size - 2:
            tempList = [frozenset(set(intersection)), frozenset(set(union))]
            if self.checkMinConfidence(tempList):
                temp.append(tempList)
```

Figure 7 shows the code where we iteratively join the rules until we get only one length itemset in the head of the rule. We used itertools package to get all the combinations of a rule set and simultaneously we prune rules based on anti-monotone property.

3 Results

3.1 Frequent Itemsets

Support: 30%

1. Number of length 1 frequent itemsets: 196
2. Number of length 2 frequent itemsets: 5340
3. Number of length 3 frequent itemsets: 5287
4. Number of length 4 frequent itemsets: 1518
5. Number of length 5 frequent itemsets: 438
6. Number of length 6 frequent itemsets: 88
7. Number of length 7 frequent itemsets: 11
8. Number of length 8 frequent itemsets: 1

Support: 40%

1. Number of length 1 frequent itemsets: 167
2. Number of length 2 frequent itemsets: 753
3. Number of length 3 frequent itemsets: 149
4. Number of length 4 frequent itemsets: 7
5. Number of length 5 frequent itemsets: 1

Support: 50%

1. Number of length 1 frequent itemsets: 109
2. Number of length 2 frequent itemsets: 63
3. Number of length 3 frequent itemsets: 2

Support: 60%

1. Number of length 1 frequent itemsets: 34
2. Number of length 2 frequent itemsets: 2

Support: 70%

6. Number of length 1 frequent itemsets: 7

3.2 Association Rules

3.2.1 Template 1 Queries

1. “RULE”, “ANY”, [‘G59_Up’] : 26
2. “RULE”, “NONE”, ['G59_Up'] : 91
3. “RULE”, 1, [‘G59_Up’, ‘G10_Down’] : 39
4. “HEAD”, “ANY”, [‘G59_Up’] : 9
5. “HEAD”, “NONE”, [‘G59_Up’] : 108
6. “HEAD”, 1, [‘G59_Up’, ‘G10_Down’] : 17
7. “BODY”, “ANY”, [‘G59_Up’] : 17
8. “BODY”, “NONE”, ['G59_Up'] : 100

9. “BODY”, 1, [‘G59_Up’, ‘G10_Down’] : 7

3.2.2 Template 2 Queries

1. “RULE”, 3 : 9
2. “HEAD”, 2 : 6
3. “BODY”, 1 : 117

3.2.3 Template 3 Queries

1. “1or1”, “HEAD”, “ANY”, [‘G10_Down’], “BODY”, 1, [‘G59_Up’]
   Number of rules generated: 24
2. “1and1”, “HEAD”, “ANY”, [‘G10_Down’], “BODY”, 1, [‘G59_Up’]
   Number of rules generated: 1
3. “1or2”, “HEAD”, “ANY”, [‘G10_Down’], “BODY”, 2
   Number of rules generated: 11
4. “1and2”, “HEAD”, “ANY”, [‘G10_Down’], “BODY”, 2
   Number of rules generated: 0
5. “2or2”, “HEAD”, 1, “BODY”, 2
   Number of rules generated: 117
6. “2and2”, “HEAD”, 1, “BODY”, 2
   Number of rules generated: 3