CSE4502/5717: Big Data Analytics

Prof. Sanguthevar Rajasekaran
Notes by Zigeng Wang (TA)

April 16th, 2018

1. Association Rules Mining

- Let D be a database (DB) of transactions.
- A transaction is a set of items.
- Let I be the set of all possible items with $|I| = d$.
- Any transaction $t \in DB$ is a subset of I.

We are interested in finding *Rules* of the form:

$$X \rightarrow Y \; where \; X \neq \emptyset, Y \neq \emptyset, X \cap Y = \emptyset, X \subseteq I, Y \subseteq I$$

- An itemset is a subset of I.
- A k-itemset is an itemset with k items.

For any itemset X, let $\sigma(X)$ denote the number of transactions in D that contain X.

- $Support$ for the $Rule\ X \rightarrow Y$ is $\frac{\sigma(X \cup Y)}{n}$ where $n = |D|$.
- $Confidence$ for the $Rule\ X \rightarrow Y$ is $\frac{\sigma(X \cup Y)}{\sigma(X)}$.

Problem

Given $minSupport, minConfidence$ and a database DB of transactions, identify all the $Rules\ X \rightarrow Y$ for which the $support$ is $\geq minSupport$ and the $confidence$ is $\geq minConfidence.$

- An itemset X is $frequent$ if itemset $\sigma(X) \geq \mathrm{n} \cdot minSupport$

A generic approach

1. Identify all the *frequent* itemsets.
2. For each frequent itemset generate rules with a $confidence \geq minConfidence$.

For example, if X' is a *frequent* itemset and $X' = X \cup Y$, check if $X \rightarrow Y$ has enough confidence. Note that, here, $X \neq \emptyset,\ Y \neq \emptyset\ and\ X \cap Y = \emptyset$.

1.1 Identifying Frequent Itemsets

1.1.1 A Brute Force Algorithm

In order to find the frequent k-itemsets, we can use a naïve 2-step approach like the following:

1. Generates all the k-itemsets.
2. For each itemset, it scans the database and checks if the itemset is frequent.

Assume that we store every transaction t as a bit array of size d where $t[i]$ is 1 if item i is in the transaction and 0 otherwise. The following shows an example for $t = (2, 4, 5)$.

0	**1**	**0**	**1**	**1**	**0**	**...**	**0**
1	2	3	4	5	6	...	d

Therefore, it takes $O(k)$ time to check if an itemset of size k can be found in a transaction and takes $O(nk)$ time in n transactions. Thus, for all possible k-itemsets, the running time of the entire algorithm is $O\left(\binom{d}{k} nk\right)$.

1.1.2 Apriori Principle

- If itemset *X* is not frequent then no superset of *X* is frequent.
- If itemset *X* is frequent then every subset of *X* is also frequent.

Example

Given database D as the following and $minSupport = ¼$, we want to find the all frequent itemsets.

#	Transactions
t1	Bread, Milk, Salt
t2	Salt, Pepper, IceCream
t3	Milk, Salt
t4	Sugar, IceCream, Salt
t5	Milk, Coffee, Sugar
t6	IceCream, Salt
t7	IceCream
t8	IceCream, Sugar, Salt

From database D, we have the following observations:

- $n = |D| = 8$
- $I = \{Bread, Coffee, IceCream, Milk, Pepper, Salt, Sugar\}$
- $d = |I| = 7$
- A itemset X is *frequent,* iff $\sigma(X) \geq \mathrm{n} \cdot minSupport = 8 \times \frac{1}{4} = 2.$

Let F_k stand for the set of frequent k-itemsets, for any k. Then we have:

F_1 = {(IceCream), (Milk), (Salt), (Sugar)}
F_2 = {(Milk, Salt), (Salt, IceCream), (Salt, Sugar), (IceCream, Sugar)}
F_3 = {(Salt, Sugar, IceCream)}
F_4 = ∅

If we are using the brute force algorithm, it will generate 7 1-itemsets, $\binom{7}{2}$ 2-itemsets, $\binom{7}{3}$ 3-itemsets and $\binom{7}{4}$ 4-itemsets. In total,

$$No.\,of\ itemsets\ processed\ in\ the\ Brute\ Force\ Algo. = \binom{7}{1} + \binom{7}{2} + \binom{7}{3} + \binom{7}{4} = 98$$

On the other hand, the Apriori algorithm will generate 7 1-itemsets, $\binom{4}{2}$ 2-itemsets, $4 \times 2 = 8$ 3-itemsets and 1 4-itemset. In total,

$$No.\,of\ itemsets\ processed\ in\ the\ Brute\ Force\ Algo. = \binom{7}{1} + \binom{4}{2} + 8 + 1 = 22$$

- Let C_k denote the candidates of frequent k-itemsets.

The Apriori Algorithm[1] can be defined as the following:

```
k := 1;
Compute F_1 = {i ∈ I | σ(i) ≥ n · minSupport};
while F_k ≠ ∅ do
    k := k + 1;
    Generate candidates C_k from F_{k−1};
    for T ∈ DB do
        for C ∈ C_k do
            if C ⊆ T then
                σ(C) := σ(C) + 1;
            end
        end
    end
    F_k := ∅;
    for C ∈ C_k do
        if σ(C) ≥ n · minSupport then
            F_k := F_k ∪ {C};
        end
    end
end
```

<u>Generation of Candidates</u>

- $F_{\mathrm{k}-1} \times F_1$ method:
  To every frequent (k-1)-itemset, add every frequent item (1-itemset), to generate candidates.
  The time for candidate generation using this method is $O(|F_{k-1}||F_1|k)$.
- $F_{k-1} \times F_{k-1}$ method:
  Let: $a_1, a_2, \cdots, a_{k-2}, a_{k-1}$ and $b_1, b_2, \cdots, b_{k-2}, b_{k-1} \in F_{k-1}$.

[1] Figure from Marius Nicolae's note, March 2014

If $a_i = b_i, \forall i = 1, \cdots, k-2$ then generate candidate $a_1, a_2, \cdots, a_{k-2}, a_{k-1}, b_{k-1}$

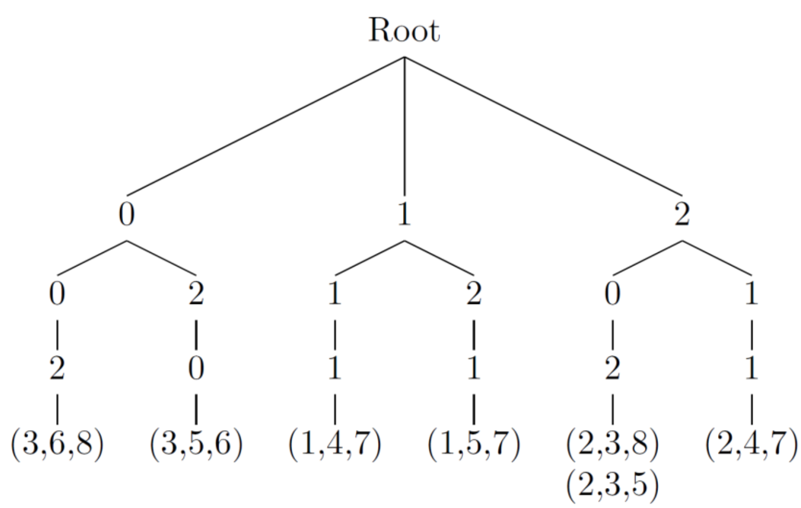
The time for candidate generation using this method is $O(|F_{k-1}|^2 k)$.

Candidate Pruning[2]

- Based on Apriori principle, if C is a candidate in C_k, check if every $k-1$ subset of C is *frequent* ($\in F_{k-1}$). If not, discard the candidate.
- To check if a $k-1$ itemset belongs to F_{k-1} we can use a $k-1$ leveled **Hash Tree**.
    - A Hash Tree is a tree where every node contains a hash table. Itemsets are inserted in the tree based on the hash values of their items. Specically, at the root, hashing is done on the first item of the itemset hashed. In the next level of the tree, hashing is done on the second item of the itemset, etc. Thus, if the itemsets are of size k, then there will be k levels in the tree.

Example

Consider the following itemsets: (2,3,8), (3,5,6), (1,4,7), (2,3,5), (3,6,8), (1,5,7), (2,4,7) and the hash function $h(x) = x \; mod \; 3$. Then the hash tree looks as follows (empty subtrees omitted because of space limitations).



- If we build a Hash Tree for F_{k-1} then we can check if an itemset is in F_{k-1} in $O(k)$ time.
- An itemset of size k has k different subsets of size $k-1$. We can search each subset in the hash tree in time $O(k)$. Therefore the time for pruning is $O(|C_k|k^2)$

Other Techniques

To avoid generating a candidate many times, we can keep any itemset in increasing order of the items in it. When we generate a new itemset from an existing one, we will only add elements larger than the largest element in the existing itemset.

---

[2] Referenced from Marius Nicolae's note, March 2014