CSC4008 Data Mining

117010279 Ziren WANG

March 2020

3 Mining Frequent Patterns, Associations, and Correlations

3.1 Basic Concepts

Items: the product in one specific basket;
Transaction: each of the basket;
Rule: $X \Rightarrow Y$: (1) Support = $\frac{frq(X,Y)}{N}$ (2) confidence = $\frac{frq(X,Y)}{frq(X)}$ (3) Lift = $\frac{support}{Supp(X)\times Supp(Y)}$;
Frequent pattern: a pattern (a set of items, sub-sequences, substructures, etc.) that occurs frequently in a data set;
Itemset: A set of one or more items;
(Absolute) support of X: Frequency or occurrence of an itemset X;
(Relative) support s: the fraction of transactions that contains X (i.e., the probability that a transaction contains X). An itemset X is frequent if X's support is no less than a minimum support threshold;
An itemset X is closed and frequent if X is frequent and there exists no proper super-itemset $Y(X \subset Y)$, with the same support as X;
An itemset X is a maximal and frequent if X is frequent and there exists no frequent super-itemset $Y(X \subset Y)$;
Downward closure property: Any subset of a frequent itemset must be frequent.

3.2 Apriori algorithm: A Candidate Generation-and-Test Approach

3.2.1 Algorithm Overview

Apriori property: All nonempty subsets of a frequent itemset must also be frequent.
Apriori pruning principle: If there is any itemset which is infrequent, its superset should not be generated/tested!
Algorithm:

1. Scan DB once to get frequent 1 itemset, set k = 1 ;
2. Generate length (k+1) candidate itemsets from length k frequent itemsets;
3. Test whether the candidates is frequent based on DB, remove those are not;
4. Check whether no candidate set can be generated. If yes, terminate; otherwise k++, goes to 2.

Example is shown as Figure 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). For efficient implementation, 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 $l_i[1] < l_i[2] < \cdots < l_i[k-1]$. 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. The resulting itemset formed by joining l_1 and l_2 is $\{l_1[1], l_1[2], \ldots, l_1[\text{k-2}], l_1[\text{k-1}], l_2[\text{k-1}]\}$.

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, which can be huge. To reduce C_k's size, 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.

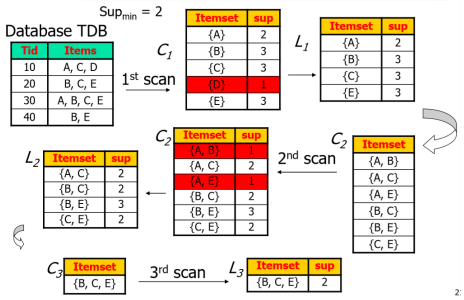


Figure 1: Apriori Algorithm Example

(a) Join: $C_3 = L_2 \bowtie L_2 = \{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\}$
$\bowtie \{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\}$
$= \{\{I1, I2, I3\}, \{I1, I2, I5\}, \{I1, I3, I5\}, \{I2, I3, I4\}, \{I2, I3, I5\}, \{I2, I4, I5\}\}$.

(b) Prune using the Apriori property: All nonempty subsets of a frequent itemset must also be frequent. Do any of the candidates have a subset that is not frequent?

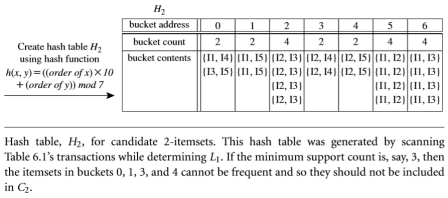
- The 2-item subsets of {I1, I2, I3} are {I1, I2}, {I1, I3}, and {I2, I3}. All 2-item subsets of {I1, I2, I3} are members of L_2. Therefore, keep {I1, I2, I3} in C_3.
- The 2-item subsets of {I1, I2, I5} are {I1, I2}, {I1, I5}, and {I2, I5}. All 2-item subsets of {I1, I2, I5} are members of L_2. Therefore, keep {I1, I2, I5} in C_3.
- The 2-item subsets of {I1, I3, I5} are {I1, I3}, {I1, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I1, I3, I5} from C_3.
- The 2-item subsets of {I2, I3, I4} are {I2, I3}, {I2, I4}, and {I3, I4}. {I3, I4} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I4} from C_3.
- The 2-item subsets of {I2, I3, I5} are {I2, I3}, {I2, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I5} from C_3.
- The 2-item subsets of {I2, I4, I5} are {I2, I4}, {I2, I5}, and {I4, I5}. {I4, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I4, I5} from C_3.

(c) Therefore, $C_3 = \{\{I1, I2, I3\}, \{I1, I2, I5\}\}$ after pruning.

Figure 2: Join and Prune

3.2.2 Improvement of Algorithm

- **Hash table**: can reduce the size of candidate k-itemset(K¿1). Especially useful when k = 2.
  Scan the database, get frequent 1-itemset and generate all 2-itemset for each transaction, hash (i.e., map) them into the different buckets of a hash table, increase the bucket counts.Then, Remove 2-itemsets, which are not frequent.



Hash table, H_2, for candidate 2-itemsets. This hash table was generated by scanning Table 6.1's transactions while determining L_1. If the minimum support count is, say, 3, then the itemsets in buckets 0, 1, 3, and 4 cannot be frequent and so they should not be included in C_2.

Figure 3: 3 Example of Hash Table of 2-itemsets

Notice: A k-itemset whose corresponding hashing bucket count is below the threshold cannot be frequent. hen we reduce the number of candidates.

- **Partitioning the Data**: any itemset that is potentially frequent in DB must be frequent in at least one of the partitions of DB.
  Thus, we can perform multiple scans in the partitioned data then merge sub-results together. In the end, find global frequent itemsets among all candidates.

- **Other improvement**: related with (1)Reducing passes of transaction database scans; (2)Shrinking number of candidates; (3)Facilitating support counting of candidates etc.

3.3 Frequent Pattern Growth Approach

Motivations: Apriori algorithm Generates a huge number of candidate sets. It repeatedly scan the whole database and check a large set of candidates by pattern matching. We want to find a less costly searching

process without candidate generation.
Apriori is a BFS algorithm while FP-Growth is a DFS algorithm, which grows long patterns from short ones using local frequent items only.

3.3.1 Algorithm Overview

Algorithm: FP_growth. Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

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

Output: The complete set of frequent patterns.

Method:

1. The FP-tree is constructed in the following steps:
   (a) Scan the transaction database D once. Collect F, the set of frequent items, and their support counts. Sort F in support count descending order as L, the *list* of frequent items.
   (b) Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in D do the following.
   Select and sort the frequent items in *Trans* according to the order of L. Let the sorted frequent item list in *Trans* be $[p|P]$, where p is the first element and P is the remaining list. Call insert_tree($[p|P]$, T), which is performed as follows. If T has a child N such that $N.item\text{-}name = p.item\text{-}name$, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call insert_tree(P, N) recursively.
2. The FP-tree is mined by calling FP_growth(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth(Tree, α)
(1)  if Tree contains a single path P then
(2)      for each combination (denoted as β) of the nodes in the path P
(3)          generate pattern β ∪ α with support_count = minimum support count of nodes in β;
(4)  else for each a_i in the header of Tree {
(5)      generate pattern β = a_i ∪ α with support_count = a_i.support_count;
(6)      construct β's conditional pattern base and then β's conditional FP_tree Tree_β;
(7)      if Tree_β ≠ ∅ then
(8)          call FP_growth(Tree_β, β); }
```

Figure 4: FP-Growth Algorithm

3.3.2 Property of Algorithm

(1) Completeness: FP-Growth Preserves complete information for frequent pattern mining. It never break a long pattern of any transactions;
(2) Compactness: FP-Growth reduce irrelevant info—infrequent items are gone. Items in frequency descending order: the more frequently occurring, the more likely to be shared. Thus, it never be larger than the original database (not count node-links and the count field).

3.3.3 Advantages of the Pattern Growth Approach

- Divide-and-conque: Decompose both the mining task and DB according to the frequent patterns obtained so far, which Leads to focused search of smaller databases;
- No candidate generation, no candidate test;
- Compressed database: FP-tree structure;
- No repeated scan of entire database;
- Basic ops: counting local freq items and building sub FP-tree, no pattern search and matching;
- A good open-source implementation and refinement of FP-Growth.

3.4 ECLAT: Frequent Pattern Mining with Vertical Data Format

Vertical format: t(AB) = $\{T_{11}, T_{25}...\}$
Deriving frequent patterns based on vertical intersections: t(X) $\Rightarrow$ t(Y): transactions having X always have Y. Using diffset to accelerate mining and reduce the cost of memory usages.

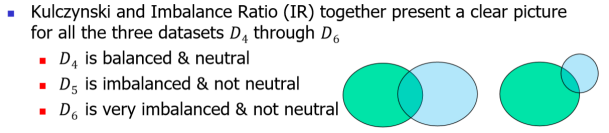
3.5 Measures the Interestingness of Patterns

Note: strong rules are not necessarily interesting!
We have several measurements as following:

- $lift = \frac{P(A \cup B)}{P(A)P(B)}$
- $\chi^2 = \sum \frac{(Observed - Expected)^2}{Expected}$
- all_conf(A,B) $= \frac{sup(A)}{max\{sup(A), sup(B)\}} = min\{P(A|B), P(B|A)\}$
- max_conf(A,B) $= max\{P(A|B), P(B|A)\}$
- Kulc(A,B) $= \frac{1}{2}(P(A|B) + P(B|A))$
- cosine(A,B) $= \frac{P(A \cup B)}{\sqrt{P(A) \times P(B)}} = \frac{sup(A \cup B)}{\sqrt{sup(A) \times sup(B)}} = \sqrt{P(A|B) \times P(B|A)}$

Null-transaction: a transaction that does not contain any of the itemsets being examined.
null-invariant: a measure is Null-invariant if its value is free from the influence of null-transactions. E.g.:all_conf(A,B); max_conf(A,B); Kulc(A,B); cosine(A,B) are all null-invariant
Null-invariance is an important property for measuring association patterns in large transaction databases. Because in many real-world scenarios, $\bar{m}\bar{c}$ is usually huge and unstable. We may define IR (Imbalance Ratio) to measure the imbalance of two itemsets A and B in rule implications:

- IR(A,B) $= \frac{|sup(A) - sup(B)|}{sup(A) + sup(B) - sup(A \cup B)}$



Data	*mc*	$\overline{m}c$	$m\bar{c}$	$\overline{m}\bar{c}$	*all_conf.*	*max_conf.*	*Kulc.*	*cosine*	IR
D_1	10,000	1,000	1,000	100,000	0.91	0.91	0.91	0.91	0.0
D_2	10,000	1,000	1,000	100	0.91	0.91	0.91	0.91	0.0
D_3	100	1,000	1,000	100,000	0.09	0.09	0.09	0.09	0.0
D_4	1,000	1,000	1,000	100,000	0.5	0.5	0.5	0.5	0.0
D_5	1,000	100	10,000	100,000	0.09	0.91	0.5	0.29	0.89
D_6	1,000	10	100,000	100,000	0.01	0.99	0.5	0.10	0.99

Figure 5: Imbalance Ratio Demonstration