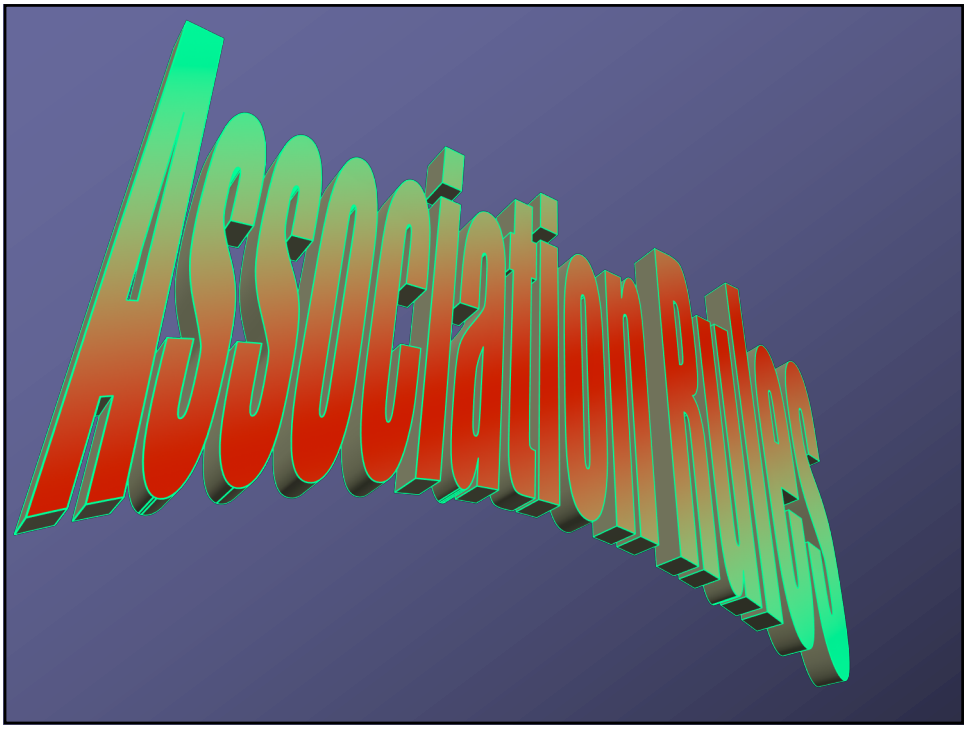


Learning Outcomes

- Understand the concept of association rules
- Define and use support and confidence measures
- Introduce the Apriori algorithm

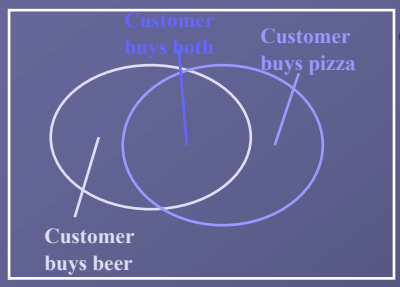
What is Association Mining?

- Association rule mining
  - Find frequent patterns, associations, correlations, or causal structures among sets of items or objects in transaction databases, relational databases, and other information repositories
- Applications
  - Basket data analysis, cross-marketing, catalog design, loss-leader analysis, clustering, classification, etc.
- Examples
  - Rule form: "Body → Head [support, confidence]"
  - buys(x, "Pizza") →buys(x, "beers") [0.5%, 60%]
  - major(x, "CS") ^ takes(x,"DB") → grade(x,"A")[1%, 75%]

Association Rule: Basic Concepts

- Given: (1) a set of transactions, (2) each transaction is a list of items (purchased by a customer in a visit)
- Find: all rules that correlate the presence of one set of items with that of another set of items
  - E.g., *98% of people who purchase tires and auto accessories also get automotive services done*
- Applications
  - *Maintenance Agreement:* (What the store should do to boost Maintenance Agreement sales)
  - *Home Electronics:* (What other products should the store stocks up?)
  - Attached mailing in direct marketing

Rule Measures: Support and Confidence



- Find all the rules $Y \rightarrow Z$ with minimum confidence and support
  - support, *s*, probability that a transaction contains { Y ➔ Z }
  - confidence, *c*, conditional probability that a transaction having Y also contains *Z*

Transaction ID	Items Bought
2000	A,B,C
1000	A,C
4000	A,D
5000	B,E,F

Let minimum support be 50%, and minimum confidence 50%, we have

- *A* ➔ *C* (50%, 66.6%)
- *C* ➔ *A* (50%, 100%)

AR Mining: A Road Map

- Boolean vs. quantitative associations
  - Based on the types of values handled)
  - buys(x, "SQLServer") ^ buys(x, "DMBook") ➔ buys(x, "DBMiner") [0.2%, 60%]
  - age(x, "30..39") ^ income(x, "42..48K") ➔ buys(x, "PC") [1%, 75%]
- Single dimension vs. multiple dimensional associations
  - See examples above
- Single level vs. multiple-level analysis
  - What brands of beers are associated with what brands of pizzas?
- Various extensions
  - Correlation, causality analysis
    - Association does not necessarily imply correlation or causality
  - Max patterns and closed itemsets
  - Constraints enforced
    - E.g., small sales (sum < 100) trigger big buys (sum > 1,000)?

Mining Association Rules—An Example

Transaction ID	Items Bought
2000	A,B,C
1000	A,C
4000	A,D
5000	B,E,F

Min. support 50%
Min. confidence 50%

Frequent Itemset	Support
{A}	75%
{B}	50%
{C}	50%
{A,C}	50%

For rule *A* ➔ *C*:

support = support({*A* U *C*}) = 50%

confidence = support({*A* U *C*})/support({*A*}) = 66.6%

The Apriori principle:

Any subset of a frequent itemset must be frequent

Mining Frequent Itemsets: the Key Step

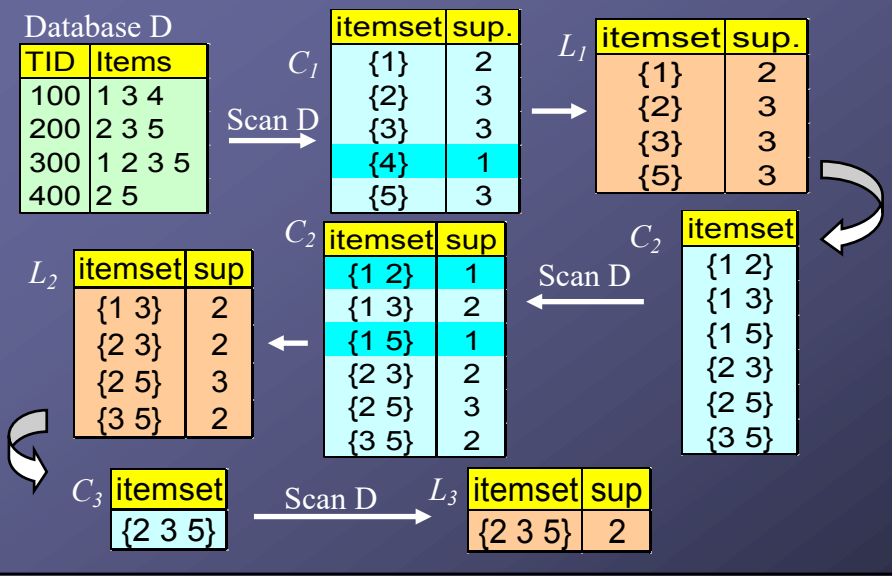
- Find the *frequent itemsets*: the sets of items that have minimum support
  - A subset of a frequent itemset must also be a frequent itemset
    - i.e., if {*AB*} is a frequent itemset, both {*A*} and {*B*} should be frequent itemsets
  - Iteratively find frequent itemsets with cardinality from 1 to *k* (*k*-itemset)
- Use the frequent itemsets to generate association rules

The Apriori Algorithm

- **Join Step**
  - C_k, a set of candidate itemsets of size k, is generated by joining $L_{k-1}$with itself, where
  - L_k is a set of frequent itemset of size k
- **Prune Step**
  - Any (k-1)-itemset that is not frequent cannot be a subset of a frequent k-itemset
- **Apriori Pseudo-code**

```
L1 = {frequent items};
for (k = 1; Lk != {}; k++) do begin
    Ck+1 = candidates generated by joining Lk by itself;
    for each transaction t in dataset do
        increment the count of all candidates in Ck+1 that are
        contained in t
    Lk+1 = candidates in Ck+1 with min_support
end
return L = Uk Lk;
```

The Apriori Algorithm — Example



How to Generate Candidates?

- Suppose the items in L_{k-1} are listed in an order
- Step 1: self-joining L_{k-1}

  For k = 3, if ab, ac, cd belong to L_2, we can join ab and ac ➔ abc, but not ab and cd

  For any k, C_k is obtained by joining

  $a_1a_2\ldots a_{k-2}a_{k-1}$ and $b_1b_2\ldots b_{k-2}b_{k-1}$

  Where if $a_1=b_1$, $a_2=b_2$ … $a_{k-2}=b_{k-2}$

  forall ***itemsets c in C_k*** do

  forall ***(k-1)-subsets s of c*** do

  if *(s is not in L_{k-1})* **then delete** *c* **from** C_k

How to Count Supports of Candidates?

- Why counting supports of candidates a problem?
  - The total number of candidates can be very huge
  - One transaction may contain many candidates
- Method
  - Candidate itemsets are stored in a *hash-tree*
  - *Leaf* node of hash-tree contains a list of itemsets and counts
  - *Interior* node contains a hash table
  - *Subset function*: finds all the candidates contained in a transaction

Example of Generating Candidates

- L_3={*abc, abd, acd, ace, bcd*}
- Self-joining: L_3*L_3
  - *abcd* from *abc* and *abd*
  - *acde* from *acd* and *ace*
- Pruning:
  - *acde* is removed because *ade* is not in L_3
- C_4={*abcd*}

Performance Bottlenecks: Is Apriori Fast Enough?

- The core of the Apriori algorithm
  - Use frequent (k – 1)-itemsets to generate candidate frequent k-itemsets
  - Use database scan and pattern matching to collect counts for the candidate itemsets
- The bottleneck of *Apriori*: candidate generation
  - Huge candidate sets:
    - 10^4 frequent 1-itemset will generate 10^7 candidate 2-itemsets
    - To discover a frequent pattern of size 100, {a_1, a_2, …, a_{100}}, one needs to generate 2^{100} (about 10^{30}) candidates
  - Multiple scans of database:
    - Needs (n +1) scans, n is the length of the longest pattern

Methods to Improve Apriori's Efficiency

- Hash-based itemset counting: A *k*-itemset whose corresponding hashing bucket count is below the threshold cannot be frequent
- Transaction reduction: A transaction that does not contain any frequent k-itemset is useless in subsequent scans
- Partitioning: Any itemset that is potentially frequent in DB must be frequent in at least one of the partitions of DB
- Sampling: mining on a subset of given data, lower support threshold + a method to determine the completeness
- Dynamic itemset counting: add new candidate itemsets only when all of their subsets are estimated to be frequent

Mining FPs Without Candidate Generation

- Compress a large database into a compact, Frequent-Pattern tree (FP-tree) structure
  - highly condensed, but complete for frequent pattern mining
  - avoid costly database scans
- Develop an efficient, FP-tree-based frequent pattern mining method
  - A divide-and-conquer methodology: decompose mining tasks into smaller ones
  - Avoid candidate generation: sub-database test only!

Construct FP-tree from a Transaction DB

- Scan Dataset D to find frequent 1-itemsets (single item patterns)
- Order frequent items in frequency descending order
- Scan Dataset D again to construct FP-tree

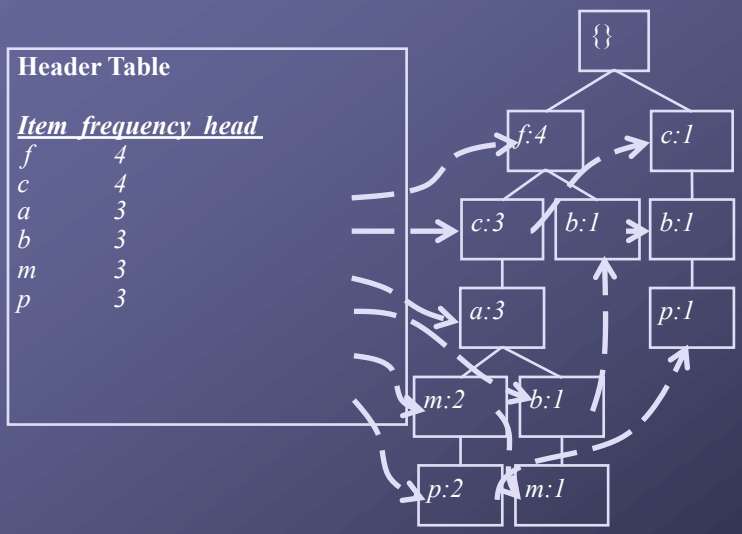
Construct FP-tree from a Transaction DB

min_support = 3

TID	Items bought								Frequent items (ordered)				
10	f	a	c	d	g	i	m	p	f	c	a	m	p
20	a	b	c	f	l	m	o		f	c	a	b	m
30	b	f	h	j	o				f	b			
40	b	c	k	s	p				c	b	p		
50	a	f	c	e	l	p	m	n	f	c	a	m	p

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	r	s
3	3	4	1	1	4	1	1	1	1	1	2	3	1	2	3	0	1

Construct FP-tree from a Transaction DB



Benefits of the FP-tree Structure

- Completeness
  - never breaks a long pattern of any transaction
  - preserves complete information for frequent pattern mining
- Compactness
  - reduce irrelevant information—infrequent items are gone
  - frequency descending ordering: more frequent items are more likely to be shared
  - never be larger than the original database (if not count node-links and counts)