**Association Rules & Frequent Itemsets**

All you ever wanted to know about diapers, beers and their correlation!

---

**The Market-Basket Problem**

Given a database of transactions, find rules that will predict the occurrence of an item based on the occurrences of other items in the transaction.

### Market-Basket transactions

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bread, Milk</td>
</tr>
<tr>
<td>2</td>
<td>Bread, Diaper, Beer, Eggs</td>
</tr>
<tr>
<td>3</td>
<td>Milk, Diaper, Beer, Coke</td>
</tr>
<tr>
<td>4</td>
<td>Bread, Milk, Diaper, Beer</td>
</tr>
<tr>
<td>5</td>
<td>Bread, Milk, Diaper, Coke</td>
</tr>
</tbody>
</table>

### Example of Association Rules

- (Diaper) → (Beer)
- (Milk, Bread) → (Eggs, Coke)
- (Beer, Bread) → (Milk)

Implication here means co-occurrence, not causality!

---

**Applications of Market-Basket Analysis**

- Supermarkets
  - Placement
  - Advertising
  - Sales
  - Coupons
- Many applications outside market basket data analysis
  - Prediction (telecom switch failure)
  - Web usage mining
- Many different types of association rules
  - Temporal
  - Spatial
  - Causal

---

**Definition: Frequent Itemset**

- **Itemset**
  - A collection of one or more items
  - Example: (Milk, Bread, Diaper)
- k-itemset
  - An itemset that contains k items
- **Support count (σ)**
  - Frequency of occurrence of an itemset
  - E.g. σ(σ(Milk, Bread, Diaper)) = 2
- **Support**
  - Fraction of transactions that contain an itemset
  - E.g. σ(Milk, Bread, Diaper) = 2/5
- **Frequent Itemset**
  - An itemset whose support is greater than or equal to a minsup threshold

---

**Definition: Association Rule**

- **Association Rule**
  - An implication expression of the form X → Y, where X and Y are itemsets
  - Example: (Milk, Diaper) → (Beer)
- **Rule Evaluation Metrics**
  - **Support (s)**
    - Fraction of transactions that contain both X and Y
    - E.g. s(Milk, Diaper, Beer) = 2/5
  - **Confidence (c)**
    - Measures how often items in Y appear in transactions that contain X
    - E.g. c(Milk, Diaper, Beer) = 2/5

Example:

\[ \sigma(Milk, Diaper) = 2/5 \]
\[ \sigma(Milk, Diaper, Beer) = 2/5 \]
\[ c(Milk, Diaper, Beer) = 0.4 \]
\[ c(Milk, Diaper) = 0.67 \]
Aspects of Association Rule Mining

• How do we generate rules fast?
  - Performance measured in:
    • Number of database scans
    • Number of itemsets that must be counted
• Which are the interesting rules?

Association Rule Mining Task

• Given a set of transactions T, the goal of association rule mining is to find all rules having:
  - support ≥ minsup threshold
  - confidence ≥ minconf threshold
• Brute-force approach:
  - List all possible association rules
  - Compute the support and confidence for each rule
  - Prune rules that fail the minsup and minconf thresholds

⇒ Computationally prohibitive!

Mining Association Rules

Example of Rules:

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
<th>Rule</th>
<th>Support</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bread, Milk</td>
<td>(Milk,Diaper) → (Beer)</td>
<td>0.4</td>
<td>0.67</td>
</tr>
<tr>
<td>2</td>
<td>Bread, Diaper, Beer, Eggs</td>
<td>(Diaper,Beer) → (Diaper)</td>
<td>0.4</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>Milk, Diaper, Beer, Coke</td>
<td>(Diaper,Beer) → (Milk)</td>
<td>0.4</td>
<td>0.67</td>
</tr>
<tr>
<td>4</td>
<td>Bread, Milk, Diaper, Beer</td>
<td>(Beer) → (Milk)</td>
<td>0.4</td>
<td>0.67</td>
</tr>
<tr>
<td>5</td>
<td>Bread, Milk, Diaper, Coke</td>
<td>(Diaper) → (Milk)</td>
<td>0.4</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Milk</td>
<td>(Milk) → (Diaper,Beer)</td>
<td>0.4</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Observations:
• All the above rules are binary partitions of the same itemset:
  (Milk, Diaper, Beer)
• Rules originating from the same itemset have identical support but can have different confidence
• Thus, we may decouple the support and confidence requirements

Finding Association Rules

Two-step approach:
1. Frequent Itemset Generation
   • Generate all itemsets whose support ≥ minsup
2. Rule Generation
   • Generate high confidence rules from each frequent itemset, where each rule is a binary partitioning of a frequent itemset
• Frequent itemset generation is still computationally expensive

Frequent Itemset Generation

• Brute-force approach:
  - Each itemset in the lattice is a candidate frequent itemset
  - Count the support of each candidate by scanning the database

Given d items, there are $2^d$ possible candidate itemsets

Complexity $\approx O(NMw)$ ⇒ Expensive since $M = 2^d$
**Computational Complexity**

- Given $d$ unique items:
  - Total number of itemsets: $2^d$
  - Total number of possible association rules:
    
    $$R = \sum \binom{d}{k} \cdot \left( \sum \binom{d-k}{j} \right)$$
    
    $$= 3^d - 2^d + 1$$

If $d=6$, $R = 602$ rules

**Frequent Itemset Generation Strategies**

- Reduce the number of candidates ($M$)
  - Complete search: $M = 2^d$
  - Use pruning techniques to reduce $M$

- Reduce the number of transactions ($N$)
  - Reduce size of $N$ as the size of itemset increases
  - Used by DHP and vertical-based mining algorithms

- Reduce the number of comparisons ($NM$)
  - Use efficient data structures to store the candidates or transactions
  - No need to match every candidate against every transaction

**Reducing Number of Candidates**

- Apriori principle:
  - If an itemset is frequent, then all of its subsets must also be frequent
  - Apriori principle holds due to the following property of the support measure:
    
    $$\forall X, Y : (X \subseteq Y) \Rightarrow s(X) \geq s(Y)$$

    - Support of an itemset never exceeds the support of its subsets
    - This is known as the anti-monotone property of support

**Illustrating Apriori Principle**

- Items (1-itemsets)
  - Bread: 4
  - Coke: 2
  - Milk: 4
  - Beer: 3
  - Diaper: 1

  Minimum Support = 3

- Pairs (2-itemsets)
  - (Bread, Milk): 3
  - (Bread, Beer): 2
  - (Milk, Beer): 2
  - (Milk, Diaper): 3
  - (Beer, Diaper): 3

  (No need to generate candidates involving Coke or Eggs)

- Triples (3-itemsets)
  - (Bread, Milk, Diaper): 3

**The Idea of the Apriori Algorithm**

- start with all 1-itemsets
- go through data and count their support and find all "large" 1-itemsets
- combine them to form "candidate" 2-itemsets
- go through data and count their support and find all "large" 2-itemsets
- combine them to form "candidate" 3-itemsets
  ...

large itemset: itemset with support > $s$
candidate itemset: itemset that may have support > $s$
The Apriori Algorithm

- **Join Step**: \( C_k \) is generated by joining \( L_{k-1} \) with itself
- **Prune Step**: Any \((k-1)\)-itemset that is not frequent cannot be a subset of a frequent \(k\)-itemset

**Pseudo-code**

\[
\begin{align*}
C_k & : \text{Candidate itemset of size } k \\
L_k & : \text{Frequent itemset of size } k \\
L_1 & = \{\text{frequent items}\}; \\
& \text{for } (k = 1; L_k \neq \emptyset; k++) \text{ do begin} \\
& \quad C_{k+1} = \text{candidates generated from } L_k; \\
& \quad \text{for each transaction } t \text{ in database do} \\
& \quad \quad \text{increment the count of all candidates in } C_{k+1} \\
& \quad \quad \text{that are contained in } t \\
& \quad L_{k+1} = \text{candidates in } C_{k+1} \text{ with min_support} \\
& \quad \text{end} \\
& \text{return } \bigcup_k L_k;
\end{align*}
\]

Apriori Algorithm from Agrawal et al. (1993)

1) \( L_1 = \{\text{large 1-itemsets}\} \)
2) for \( k = 2; L_{k-1} \neq \emptyset; k++ \) do begin
3) \( C_k = \text{apriori-gen}(L_{k-1}); // \text{New candidates} \\
4) \text{for each transaction } t \in D \text{ do} \\
5) \quad C_t = \text{subset}(C_k, t); // \text{Candidates contained in } t \\
6) \quad \text{forall candidates } c \in C_t \text{ do} \\
7) \quad c.\text{count}++; \\
8) \text{end} \\
9) \text{foreach } L_k \text{ do} \\
10) \quad L_k = \{c \in C_k | c.\text{count} \geq \text{minsup}\} \\
11) \text{Answer} = \bigcup_k L_k;
\]

Apriori Algorithm Example (s = 50%)

<table>
<thead>
<tr>
<th>Database D</th>
<th>( L_1 )</th>
<th>( L_2 )</th>
<th>( C_1 )</th>
<th>( C_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction 1</td>
<td>( {1})</td>
<td>( {2})</td>
<td>( {1})</td>
<td>( {1})</td>
</tr>
<tr>
<td>Transaction 2</td>
<td>( {2})</td>
<td>( {3})</td>
<td>( {3})</td>
<td>( {2})</td>
</tr>
<tr>
<td>Transaction 3</td>
<td>( {1,2})</td>
<td>( {1,3})</td>
<td>( {3})</td>
<td>( {2})</td>
</tr>
<tr>
<td>Transaction 4</td>
<td>( {3})</td>
<td>( {3})</td>
<td>( {1,3})</td>
<td>( {3})</td>
</tr>
</tbody>
</table>

Algorithm to Guess Itemsets

- **Naïve way**:
  - Extend all itemsets with all possible items
- **More sophisticated**:
  - **Join** \( L_{k-1} \) with itself, adding only a single, final item e.g.: \( (1 2 3), (1 2 4), (1 3 4), (1 3 5), (2 3 4) \) produces \( (1 2 3 4) \) and \( (1 3 4 5) \)
  - **Remove itemsets with an unsupported subset**
    - e.g.: \( (1 3 4 5) \) has an unsupported subset: \( (1 4 5) \)
    - if minsup = 50%
  - Use the database to further refine \( C_k \)

Apriori: How to Generate Candidates?

**STEP 1: Self-join operation**

insert into \( C_k \)
select \( p.\text{item}_1, p.\text{item}_2, \ldots, p.\text{item}_{k-1}, q.\text{item}_{k-1} \)
from \( L_{k-1}: p, L_{k-1} \) q
where \( p.\text{item}_1 = q.\text{item}_1, \ldots, p.\text{item}_{k-2} = q.\text{item}_{k-2}, \)
\( p.\text{item}_{k-1} < q.\text{item}_{k-1} \);

**STEP 2: Subset filtering**

forall itemsets \( c \in C_k \) do
forall \((k-1)\)-subsets \( s \) of \( c \) do
if \( s \not\in L_{k-1} \) then
  delete \( c \) from \( C_k \);

How to Count Supports of Candidates?

- **Why counting supports of candidates is 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 Candidate Itemsets

• \( L_3 = \{ abc, abd, acd, ace, bcd \} \)

• Self-joining: \( L_3 \times L_3 \)
  - \( abcd \) from \( abc \) and \( abd \)
  - \( acde \) from \( acd \) and \( ace \)

• Pruning based on the Apriori principle:
  - \( acde \) is removed because \( ade \) is not in \( L_2 \)

• \( C_4 = \{ abcd \} \)

Run Time of Apriori

• \( k \) passes over data where \( k \) is the size of the largest candidate itemset

• Memory chunking algorithm \( \Rightarrow 2 \) passes over data on disk but multiple in memory

Toivonen 1996 gives a statistical technique which requires \( 1 + e \) passes (but more memory)

Brin 1997 - Dynamic Itemset Counting \( \Rightarrow 1 + e \) passes (less memory)

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

Is Apriori Fast Enough? — Performance Bottlenecks

• 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, e.g., \( \{ a_1, a_2, ..., a_{100} \} \), one needs to generate \( 2^{100} \approx 10^{30} \) candidates.
  - Multiple scans of database:
    - Needs \((n+1)\) scans, where \( n \) is the length of the longest pattern