Frequent Item Sets & Association Rules

Vassilis Christophides
christop@csd.uoc.gr
http://www.csd.uoc.gr/~hy562
University of Crete Fall 2018

Some History

- Bar code technology allowed retailers to collect massive volumes of sales data
  - Basket data: transaction date, set of items bought
  - Data is stored in tertiary storage

- Leverage information for marketing
  - How to design coupons?
  - How to organize shelves?

- The birth of data mining!
  - Agrawal et al. (SIGMOD 1993) introduced the problem of mining a large collection of basket data to discover association rules
  - Many papers followed…
Example: Supermarket Shelf-Management

- **Goal:** Identify items that are bought together by sufficiently many customers
- **Approach:** Process the sales data collected with barcode scanners to find dependencies among items
  - Given a set of transactions (market-basket model), find rules that will predict the occurrence of an item based on the occurrences of other items in the transactions
- **A classic rule:**
  - If one buys diaper and milk, then he is likely to buy beer
  - Don't be surprised if you find six-packs next to diapers!

```
Rules Discovered:
{Milk} --> {Coke}
{Diaper, Milk} --> {Beer}
```

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

Application Examples of Association Rules

- **Items** = products; **baskets** = sets of products someone bought in one visit to the store
  - Reveals typical buying behaviour of customers
    - **Marketing and sales promotion** (suggests tie-in “tricks”)
      - a product $p$ appearing as rules’ consequent can be used to determine what should be done to boost $p$ sales
      - a product $p'$ appearing as rules' antecedent can be used to see which other products would be affected if the store discontinues selling $p'$
      - a rule $p' \rightarrow p$ an be used to see what products $p'$ should be sold to promote sale of $p$, e.g., run sale on diapers and raise beer' price
    - **Shelf management:** position certain items strategically
    - **Recommendation,** e.g., Amazon’s people who bought $X$ also bought $Y$
  - **High support** needed, or no €€’s
  - Only useful if many buy diapers & beer
The Market-Basket Model

- A large set of items, e.g., things sold in a supermarket
  \[ I = \{ i_1, i_2, \ldots, i_m \} \]

- A large set of baskets/transactions, e.g., the things one customer buys in one visit to the store
  \[ t \text{ a set of items, and } t \subseteq I \]

- Transaction Database \( T \): a set of transactions \( T = \{ t_1, t_2, \ldots, t_n \} \)

- Our interests: Identify associations among “items”, not “baskets”
  \[ \text{E.g., People who bought Diaper tend to buy Beer} \]

---

Market-Baskets and Associations

- A many-many mapping (association) between two kinds of things
  \[ \text{E.g., 90\% of transactions that purchase diaper&milk also purchase beer} \]

- Given a set of baskets, discover association rules
  \[ \text{The technology focuses on common events, not rare events ("long tail")} \]

- 2-step approach
  \[ \text{Find frequent itemsets} \]
  \[ \text{Generate association rules} \]

Rules Discovered:
\[ \{ \text{Milk} \} \rightarrow \{ \text{Coke} \} \]
\[ \{ \text{Diaper, Milk} \} \rightarrow \{ \text{Beer} \} \]
Causation vs. Association

\[ X \rightarrow Y \]

- In machine learning, \( X \rightarrow Y \) usually implies a causal relationship
  - "a change in \( X \) (seen as cause) forces a change in \( Y \) (seen as effect)"
  - causation is complex and difficult to prove relationship
- In rule mining, \( X \rightarrow Y \) is an association relationship
  - "\( X \) is associated with \( Y \)"
  - Much easier to calculate and prove
    - of less interest for medical research than for market research
- Association rules indicate only the existence of a statistical relationship between \( X \) and \( Y \)
  - They do not specify the nature of the relationship

Frequent Itemsets

- Simplest question: find sets of items, called itemsets, that appear “frequently” in the baskets
  - E.g., \{milk, diaper, bear\} is an itemset

- Support for itemset \( A \) = the number of baskets containing all items in \( A \)
  - Often expressed as a fraction of the total number of baskets

- Given a support threshold \( s \), sets of items that appear in at least \( s \) baskets are called frequent itemsets

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

Support of \{Milk, Diaper\} = 3
Support of \{Milk, Diaper, Beer\} = 2
Example: Frequent Itemsets

- **Items =** \{milk, cereal, diaper, beer, juice\}
- **Support =** 3 baskets
  - \(B_1 = \{m, c, b\}\)
  - \(B_2 = \{m, d, j\}\)
  - \(B_3 = \{m, b\}\)
  - \(B_4 = \{c, j\}\)
  - \(B_5 = \{m, d, b\}\)
  - \(B_6 = \{m, c, b, j\}\)
  - \(B_7 = \{c, b, j\}\)
  - \(B_8 = \{b, c\}\)
- **Frequent itemsets:** \{m\}, \{c\}, \{b\}, \{j\},
  \{m, b\}, \{b, c\}, \{c, j\}

The Market-Basket Model

- A **k-itemset** is an itemset with \(k\) items
  - E.g., \(A = \{\text{milk, diaper}\}\) is a 2-itemset
  - E.g., \(A' = \{\text{milk, bear, diaper}\}\) is a 3-itemset
- A transaction \(t\) contains an itemset \(A\) in \(I\), if \(A \subseteq t\)
  - E.g., basket \(B_8 = \{\text{milk, cereal, bear, diaper}\}\) contains the 3-itemset \(A'\)
- An **association rule** is an implication of the form:
  \(A \rightarrow B\), where \(A, B \subseteq I\), and \(A \cap B = \emptyset\)
Association Rules

- If-then rules about the contents of baskets
  - \{i_1, i_2, ..., i_k\} \rightarrow j means: “if a basket contains all of \(i_1, ..., i_k\)
    then it is *likely* to contain \(j\)

- A general form of an association rule is Body→Head[Support,Confidence]
  - Antecedent, left-hand side (LHS), body
  - Consequent, right-hand side (RHS), head
  - Support, frequency
  - Confidence, strength

- Example: diapers \rightarrow beer [50%, 60%]
  - “ If buys diapers, THEN buys beer in 60% of the cases in 50% of the transactions”

Support and Confidence

- **Support** for the rule \(A \rightarrow B\): denotes the frequency of the rule within all transactions in the database \(T\), i.e., the probability that a transaction contains the union of \(A\) and \(B\)
  - \(\text{support}(A \rightarrow B [s,c]) = p(A \cup B) = \text{support}\{A,B\}\)

- **Confidence** of the rule \(A \rightarrow B\): denotes the percentage of transactions in the database \(T\), containing \(A\) which also contain \(B\), i.e., the conditional probability that a transaction containing \(A\) also contains \(B\)
  - \(\text{confidence}(A \rightarrow B [s,c]) = p(B|A) = p(A \cup B) / p(A) = \text{support}\{A,B\} / \text{support}\{A\}\)
Example: Confidence

\[ B_1 = \{m, c, b\} \quad B_2 = \{m, d, j\} \]
\[ B_3 = \{m, b\} \quad B_4 = \{c, j\} \]
\[ B_5 = \{m, d, b\} \quad B_6 = \{m, c, b, j\} \]
\[ B_7 = \{c, b, j\} \quad B_8 = \{b, c\} \]

- An association rule: \( \{m, b\} \rightarrow c \)
  - Support \( \{m, b\} = 4 \)
  - Support \( \{m, b, c\} = 2 \)
  - Confidence \( \{m, b\} \rightarrow c \) = \( \frac{2}{4} = 50\% \)

\[
\text{conf}(I \rightarrow j) = \frac{\text{support}(I \cup j)}{\text{support}(I)}
\]

Finding Association Rules

- **Goal**: Find all rules that satisfy the user-specified minimum support (\texttt{minsup}) and minimum confidence (\texttt{minconf})
  - \texttt{support} \( \geq s \) and \texttt{confidence} \( \geq c \)
- **Key Features**
  - Completeness: find all rules
  - Mining with data on disk (not in memory)
- **Hard part**: Finding the frequent itemsets
  - If \( A \rightarrow B \) has high support and confidence, then both \( A \) and \( B \) will be frequent
How to Set the Appropriate MinSup?

- Many real data sets have skewed support distribution

- If \textit{minsup} is too high, we could miss itemsets involving interesting rare items (e.g., expensive products)

- If \textit{minsup} is too low, it is computationally expensive and the number of itemsets is very large

- A single \textit{minsup} threshold may not be always effective

## Association Rule Mining Task

- \textbf{Brute-force} approach:
  - List all possible association rules
    - Given \(d\) unique items:
      - Total number of itemsets = \(2^d\)
      - Total number of ARs = \(R\)
      \[
      R = \sum_{k=1}^{d} \left( \frac{d}{k} \times \sum_{j=0}^{d-k} \left( \frac{d-k}{j} \right) \right)
      \]
      \[
      = 3^d - 2^{d-1} + 1
      \]
  - Compute the support and confidence for each rule
    - Prune rules that fail the \textit{minsup} and \textit{minconf} thresholds
  - Computationally prohibitive!
Counting Frequent Itemsets in One pass

- Each itemset is a candidate frequent itemset
- Count the support of each candidate by scanning the database

<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>

- Match each basket against every candidate
- Complexity $\sim O(NMw)$ $\Rightarrow$ Expensive since $M = 2^d$ !!
  - Need a lot of memory space else swapping counts in/out is very “expensive”

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 $N$ as the size of itemset increases

- 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 the Number of Candidates: The Apriori algorithm

- Rules originating from the same itemset have identical support but can have different confidence
  - Thus, we may decouple the support and confidence requirements
- Two steps:
  1. **Frequent Itemsets**: Find all itemsets \( I \) that have minimum support
     - Usually a computationally expensive phase!
  2. **Key idea**: (anti-)monotonicity property of the support measure
     - If an itemset is frequent, then all of its subsets must also be frequent
     - If an itemset is not frequent, then all of its supersets cannot be frequent
     \[ \forall A, B: (A \subseteq B) \Rightarrow s(A) \geq s(B) \]
     - The support of an itemset never exceeds the support of its subsets
     - This is known as the anti-monotone property of support

The Apriori algorithm

- **Rule generation**: Use frequent itemsets \( I \) to generate rules
  - For every subset \( A \) of \( I \), generate rule \( A \rightarrow I \setminus A \)
    - Since \( I \) is frequent, \( A \) is also frequent
  - Variant 1: Perform a single pass to compute the rule confidence
    - \( \text{conf}(A, B \rightarrow C, D) = \frac{\text{supp}(A, B, C, D)}{\text{supp}(A, B)} \)
  - Variant 2: Filter out bigger rules from smaller ones
    - Observation: If \( A, B, C \rightarrow D \) is below confidence, so is \( A, B \rightarrow C, D \)
  - Output rules above confidence threshold
- In general, confidence does not have an anti-monotone property
  - \( \text{conf}(ABC \rightarrow D) \) can be larger or smaller than \( \text{conf}(AB \rightarrow D) \)
- But confidence of rules generated from the same itemset has an anti-monotone property
  - E.g., \( I = \{A, B, C, D\}: \text{conf}(ABC \rightarrow D) \geq \text{conf}(AB \rightarrow CD) \geq \text{conf}(A \rightarrow BCD) \)
- Confidence is anti-monotone w.r.t. number of items on the RHS of the rule
Example

\begin{itemize}
\item \( B_1 = \{m, c, b\} \) \hspace{1cm} \( B_2 = \{m, d, j\} \)
\item \( B_3 = \{m, c, b, n\} \) \hspace{1cm} \( B_4 = \{c, j\} \)
\item \( B_5 = \{m, d, b\} \) \hspace{1cm} \( B_6 = \{m, c, b, j\} \)
\item \( B_7 = \{c, b, j\} \) \hspace{1cm} \( B_8 = \{b, c\} \)
\end{itemize}

- Support threshold \( s = 3 \), confidence \( c = 0.75 \)

1) Frequent itemsets:
- \( \{b, m\} \) \hspace{1cm} \( \{b, c\} \) \hspace{1cm} \( \{c, m\} \) \hspace{1cm} \( \{c, j\} \) \hspace{1cm} \( \{m, c, b\} \)

2) Generate rules:
- \( b \rightarrow m: c = 4/6 \) \hspace{1cm} \( b \rightarrow c: c = 5/6 \) \hspace{1cm} \( b, c \rightarrow m: c = 3/5 \)
- \( m \rightarrow b: c = 4/5 \) \hspace{1cm} \( b \rightarrow m \rightarrow c: c = 3/4 \)
- \( b \rightarrow c, m: c = 3/6 \)

\[
\text{conf}(A \rightarrow B) = \frac{\text{supp}(A,B)}{\text{supp}(A)}
\]

Frequent Itemset Generation

Given \( d \) items, there are \( 2^d \) possible candidate itemsets.
Illustrating the A-Priori Principle

Rule Generation Example
How to Improve A-priori Efficiency?

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

Compacting Output Rules: Classes of Itemsets

- To reduce the number of rules we can post-process and only output:
  - **Maximal Frequent itemsets**: no immediate superset is frequent
    - Can generate all frequent itemsets (without support)
  - **Closed itemsets**: no immediate superset has the same count (>0)
    - Can generate all frequent itemsets and their support
- Alternately:
  - **Free itemset**: no immediate subset has the same count (>0)
### Example: Maximal/Closed

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
<th>Maximal (s=3)</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>AB</td>
<td>4</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AC</td>
<td>2</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>BC</td>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ABC</td>
<td>2</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- Frequent, but superset BC also frequent
- Frequent, and its only superset, ABC, not freq
- Superset BC has same count
- Its only superset, ABC, has smaller count

### Finding Frequent Itemsets
Computing Itemsets

- Back to finding frequent itemsets
- Typically, data is kept in flat files rather than in a database system:
  - Stored on disk, basket-by-basket
  - Baskets are small but we have many baskets and many items
    - Expand baskets into pairs, triples, etc. as you read baskets
    - Use $k$ nested loops to generate all itemsets of size $k$

Note: To find frequent itemsets, we have to **count** them
- To **count** them, we have to **generate** them

Computation Model

- In practice, association-rule algorithms read data in passes
  - We measure the cost by the **number of passes** over the data
  - => Cost of mining is the **number of disk I/Os**

- The approach:
  - We always need to **generate all the itemsets**
  - But we would only like to **count/keep track of those itemsets** that in the end turn out to be frequent

- For many frequent-itemset algorithms **main-memory** is the critical resource
  - The number of different things we can count as we read baskets is limited by main memory
Finding Frequent Pairs

- The hardest turns out to be finding the frequent pairs of items \( \{i_1, i_2\} \)
  - Often, frequent pairs are common, frequent triples are rare
    - The probability of being frequent drops exponentially with the itemset size; number of itemsets grows more slowly with size
- Naïve Algorithm:
  - Read file one, counting in main memory the occurrences of each pair
    - From each basket of \( n \) items, generate its \( n(n-1)/2 \) pairs using two nested loops
- Problem: fails if \( n^2 \) exceeds main memory
  - Suppose \( 10^5 \) items, counts are 4-byte integers
  - Number of pairs of items: \( 10^5(10^5-1)/2 = 5*10^9 \)
  - Therefore, \( 2*10^{10} \) (20 gigabytes) of memory needed

Counting Pairs in Memory

Two approaches:
- **Approach 1**: Count all pairs using a matrix keeping only the counts \( c \)
- **Approach 2**: Keep a table of triples \([i, j, c]\) = “the count of the pair of items \( \{i, j\} \) is \( c \)”
  - If integers and item ids are 4 bytes, we need approximately 12 bytes for pairs with \( count > 0 \)
  - Plus some additional overhead to organize the table for efficient search (“hashtable”)

Note:
- **Approach 1** only requires 4 bytes per pair
- **Approach 2** uses 12 bytes per pair (but only for pairs with \( count > 0 \))
Triangular Matrix

Approach 1: Triangular Matrix

- \( n \) = total number of items
- Count pair of items \( \{i, j\} \) only if \( i < j \)
  - So use only half of the two-dimensional array
- A more space-efficient way is to use a one-dimensional triangular array
- Keep pair counts in lexicographic order:
  - \( \{1, 2\}, \{1, 3\}, \ldots, \{1, n\}, \{2, 3\}, \{2, 4\}, \ldots, \{2, n\}, \{3, 4\}, \ldots \)
- Pair \( \{i, j\} \) is at position:
  \( (i - 1)(n - i) / 2 + j - i \)
- Total number of pairs \( n(n-1)/2 \); total bytes= \( 2n^2 \)
- Triangular Matrix requires 4 bytes per pair

Comparing the two Approaches

- Approach 2 uses 12 bytes per occurring pair (but only pairs with count > 0)
- Total bytes used is about 12p, where \( p \) is the number of pairs that actually occur
- Beats Approach 1 if less than 1/3 of possible pairs actually occur
- May require extra space for retrieval structure, e.g., a hash table

Problem is if we have too many items so the pairs do not fit into memory. Can we do better?
A-Priori Algorithm

Example
Market-Basket transactions

Items (1-itemsets)

<table>
<thead>
<tr>
<th>Item</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bread</td>
<td>4</td>
</tr>
<tr>
<td>Coke</td>
<td>2</td>
</tr>
<tr>
<td>Milk</td>
<td>4</td>
</tr>
<tr>
<td>Beer</td>
<td>3</td>
</tr>
<tr>
<td>Diaper</td>
<td>4</td>
</tr>
<tr>
<td>Eggs</td>
<td>1</td>
</tr>
</tbody>
</table>

Pairs (2-itemsets)

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>{Bread, Milk}</td>
<td>3</td>
</tr>
<tr>
<td>{Bread, Beer}</td>
<td>2</td>
</tr>
<tr>
<td>{Bread, Diaper}</td>
<td>3</td>
</tr>
<tr>
<td>{Milk, Beer}</td>
<td>2</td>
</tr>
<tr>
<td>{Milk, Diaper}</td>
<td>3</td>
</tr>
<tr>
<td>{Beer, Diaper}</td>
<td>3</td>
</tr>
</tbody>
</table>

Triplets (3-itemsets)

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>{Bread, Milk, Diaper}</td>
<td>3</td>
</tr>
</tbody>
</table>

Minimum Support = 3

If every subset is considered, \(6C_1 + 6C_2 + 6C_3 = 41\)
With support-based pruning, \(6 + 6 + 1 = 13\)
Candidate Generation

- **Contrapositive for pairs**: if item $i$ does not appear in $s$ baskets, then no pair including $i$ can appear in $s$ baskets.
- **Basic principle (Apriori)**:
  - An itemset of size $k+1$ is candidate to be frequent only if all of its subsets of size $k$ are known to be frequent.
- **Main idea**:
  - Construct a candidate of size $k+1$ by combining two frequent itemsets of size $k$.
  - Prune the generated $k+1$-itemsets that do not have all $k$-subsets to be frequent.
- So, how does A-Priori find frequent pairs?
  - A two-pass approach limiting the need for main memory counts.

A-Priori Algorithm

- **Pass 1**: Read baskets and count in main memory the occurrences of each item.
  - Requires only memory proportional to #items.
  - Items that appear at least $s$ times (minsup) are the frequent items.
- **Pass 2**: Read baskets again and count in main memory only those pairs where both elements were found in Pass 1 to be frequent.
  - Requires memory proportional to square of frequent items only (for counts).
  - Plus a list of the frequent items (so you know what must be counted).
Details for A-Priori

- You can use the triangular matrix method with $m =$ number of frequent items
  - May save space compared with storing triples

- Trick: re-number frequent items $1, 2, \ldots, m$ and keep a table relating new numbers to original item numbers

Frequent Triples, Etc.

- For each $k$, we construct two sets of $k$-tuples (sets of size $k$):
  - $C_k =$ candidate $k$-sets = those that might be frequent sets (support $\geq s$) based on information from the pass for $k - 1$
  - $L_k =$ the set of truly frequent $k$-tuples
The Apriori algorithm

Level-wise approach

\[ C_k = \text{candidate itemsets of size } k \]
\[ L_k = \text{frequent itemsets of size } k \]

1. \( k = 1 \), \( C_1 = \text{all items} \)
2. While \( C_k \) not empty
3. Scan the database to find which itemsets in \( C_k \) are frequent and put them into \( L_k \)
4. Use \( L_k \) to generate a collection of candidate itemsets \( C_{k+1} \) of size \( k+1 \)
5. \( k = k + 1 \)

Recall: Example from Last time

\[
\begin{align*}
B_1 &= \{m, c, b\} & B_2 &= \{m, d, j\} \\
B_3 &= \{m, c, b, n\} & B_4 &= \{c, j\} \\
B_5 &= \{m, d, b\} & B_6 &= \{m, c, b, j\} \\
B_7 &= \{c, b, j\} & B_8 &= \{b, c\}
\end{align*}
\]

- Frequent itemsets \( s = 3 \):
  - \{b\}, \{c\}, \{j\}, \{m\}
  - \{b,m\} \{b,c\} \{c,m\} \{c,j\}
  - \{m,c,b\}

- How we can compute them with A-Priori?
A-Priori Algorithm Example

Generate $C_1 = \{ \{b\}, \{c\}, \{j\}, \{m\}, \{n\}, \{p\} \}$
Count the support of itemsets in $C_1$
Prune non-frequent: $L_1 = \{b, c, j, m\}$

Generate $C_2 = \{ \{b,c\}, \{b,j\}, \{b,m\}, \{c,j\}, \{c,m\}, \{j,m\} \}$
Count the support of itemsets in $C_2$
Prune non-frequent: $L_2 = \{ \{b,m\}, \{b,c\}, \{c,m\}, \{c,j\} \}$

Generate $C_3 = \{ \{b,c,m\}, \{b,c,j\}, \{b,m,j\}, \{c,m,j\} \}$ **
Count the support of itemsets in $C_3$
Prune non-frequent: $L_3 = \{ \{b,c,m\} \}$

** Note here we generate new candidates by generating $C_k$ from $L_{k-1}$ and $L_1$.
But that one can be more careful with candidate generation. For example, in $C_3$ we know $\{b, m, j\}$ cannot be frequent since $\{m, j\}$ is not frequent.

A-Priori Algorithm: Memory Details

- The first pass of A-Priori
  - Create two tables
  - Translate items (e.g. strings) to numbers
  - Counters of singletons
- Between the passes of A-priory
  - Many singletons won’t be frequent
  - Create new numbering 1..m just for frequent items
  - Create frequent-items table: array of size n
    - i-th element is zero if not frequent or number 1..m
- The second pass of A-Priori
  - Count all the pairs that consist of two frequent items
  - Maintain triangular matrix of $4*m^2/2$ bytes (or triples structure)
Improvements to A-Priori

Observations

- In pass 1 of the Apriori scheme
  - Only individual item counts are stored
  - Remaining memory is unused

- In pass 2 of the Apriori scheme, it is possible that \((i, j)\) is not frequent even though \(i\) and \(j\) are frequent
  - But we still must count them (and hence need to store them in memory)

- Can we use the idle memory (in pass 1) to reduce the memory required in pass 2?
PCY (Park-Chen-Yu) Algorithm

- **Pass 1 of PCY**: In addition to item counts, maintain a hash table with as many buckets as fit in memory
  - Keep a count for each bucket into which pairs of items are hashed (not the actual pairs that hash to the bucket!)
  - Number of buckets can be smaller than number of pairs
    - Collision is possible!
- **Multistage** improves PCY (latter)

**Observations about Buckets**

- We are not just interested in the presence of a pair, but whether its count is at least the support s threshold
- If a bucket contains a frequent pair, then the bucket is surely frequent
- However, even without any frequent pair, a bucket can still be frequent (false positives)
  - So, we cannot use the hash table to eliminate any member (pair) of a “frequent” bucket
- If a bucket is not frequent, no pair that hashes to that bucket could possibly be a frequent pair
  - For a bucket with total count < s, none of its pairs can be frequent
  - Pairs that hash to this bucket can be eliminated as candidates (even if the pair consists of 2 frequent items)
- **Pass 2 of PCY**: we only count pairs that hash to frequent buckets
  - There are still infrequent pairs that slipped through
PCY Algorithm – Pass 1

- Pairs of items need to be generated from the input file
  - they are not present in the file!
- Before Pass 1 Organize Main Memory
  - Space to count each item: One (typically) 4-byte integer per item
  - Use the rest of the space for as many integers, representing buckets, as we can

```plaintext
FOR (each basket) {
    FOR (each item in the basket)
        add 1 to item’s count;
    FOR (each pair of items) {
        hash the pair to a bucket;
        add 1 to the count for that bucket
    }
}
```

PCY Algorithm – Between Passes

- In pass 2, only need to count pairs that hash to frequent buckets
  - We must count again because we did not keep the information on the pairs, and also because of the collision
  - However, we do not need the count information from pass 1 any more
  - What we need is an indication on whether a pair is possibly frequent or not
- Bit vector serves this purpose well (and consumes less space)
  - 1 means bucket count exceeds the support $s$ (i.e., is frequent); 0 means it did not
  - The hash value now corresponds to the bit position
- 4-byte integers are replaced by bits, so the bit-vector requires $1/32$ of memory
- Also, decide which items are frequent and list them for the second pass
PCY Algorithm – Pass 2

- Count all pairs \{i, j\} that meet the conditions for being a candidate pair:
  - Both \(i\) and \(j\) are frequent items
  - The pair \{i, j\}, hashes to a bucket number whose bit in the bit vector is 1
- Both conditions are necessary for the pair to have a chance of being frequent

PCY Scheme (Pass 2): Memory Details

- Buckets require a few bytes each
  - **Note**: we don’t have to count over \(s\)
  - # buckets is \(O(\text{main-memory size})\)
- On second pass, a table of \((\text{item, item, count})\) triples is essential
- Cannot use triangular matrix scheme. Why?
  - Pairs of frequent items that PCY avoid counting are placed randomly within the triangular matrix
  - No known way of compacting the matrix to avoid leaving space for the uncounted pairs
- Thus, the hash table must eliminate 2/3 of the candidate pairs for PCY to beat \(A\)-priori
Refinement: A Multistage Algorithm

- Limit the number of candidates to be counted
  - Remember: memory is the bottleneck
  - Still need to generate all itemsets but we only want to count/keep track of the ones that are frequent

- Key idea: After Pass 1 of PCY, rehash only those pairs that qualify for Pass 2 of PCY
  - \( i \) and \( j \) are frequent, and
  - \( \{i, j\} \) hashes to a frequent bucket from Pass 1

- On middle pass, fewer pairs contribute to buckets, so fewer false positives—frequent buckets with no frequent pair

- Uses several successive hash tables—requires more than two passes

---

**Multistage Picture**

- First hash table: Main memory
  - Item counts
  - Bitmap 1
  - Pass 1: Count items, Hash pairs \( \{i, j\} \)

- Second hash table: Bitmap 1
  - freq. items
  - Pass 2: Hash pairs \( \{i, j\} \) into Hash2 iff: \( i, j \) are frequent, \( \{i, j\} \) hashes to freq. bucket in B1

- Counts of candidate pairs: Bitmap 2
  - freq. items
  - Pass 3: Count pairs \( \{i, j\} \) iff: \( i, j \) are frequent, \( \{i, j\} \) hashes to freq. bucket in B1, \( \{i, j\} \) hashes to freq. bucket in B2
Multistage – Pass 3

- Count only those pairs \( \{i, j\} \) that satisfy these candidate pair conditions:
  - Both \( i \) and \( j \) are frequent items
  - Using the first hash function, the pair hashes to a bucket whose bit in the first bit-vector is 1
  - Using the second hash function, the pair hashes to a bucket whose bit in the second bit-vector is 1

- Important Points
  - The two hash functions have to be independent
  - We need to check both hashes on the third pass
    - If not, we would wind up counting pairs of frequent items that hashed first to an infrequent bucket but happened to hash second to a frequent bucket

Refinement: The Multihash Algorithm

- Key idea: use several independent hash tables on the first pass

- Risk: halving the number of buckets doubles the average count
  - We have to be sure most buckets will still not reach count \( s \)

- If so, we can get a benefit like multistage, but in only 2 passes!
Numerous approaches and refinements have been studied to keep memory consumption low:
- PCY and its refinements (multistage, multihash)

Either multistage or multihash can use more than two hash functions:
- In multistage, there is a point of diminishing returns, since the bit-vectors eventually consume all of main memory.
- For multihash, the bit-vectors occupy exactly what one PCY bitmap does, but too many hash functions makes all counts $\geq s$.

Limited Pass Algorithms
All (Or Most) Frequent Itemsets in \(< 2\) Passes

- A-Priori, PCY, etc., take \(k\) passes to find frequent itemsets of size \(k\)
- Can we use fewer passes?
- Use 2 or fewer passes for ALL sizes, but may miss some frequent itemsets
  - Approximate solution
    - Simple algorithm: Use random sampling
    - Savasere, Omiecinski, and Navathe (SON) algorithm
    - Toivonen

Random Sampling

- Take a random sample of the market baskets
- Run A-priori or one of its improvements (for itemsets of all sizes, not just pairs),
  - load the sample into the main memory
    - so you don't pay for disk I/O each time you increase the size of itemsets
  - reduce support threshold proportionally to match the sample size
  - be sure you leave enough space for counts
- Use as your support threshold a suitable, scaled-back number
  - E.g., if your sample is 1/100 of the baskets, use \(s/100\) as your support threshold instead of \(s\)
Random Sampling:– Option

- **False positives** will result
  - Itemset may be frequent in the sample but not in the entire dataset (because of the reduced minsup threshold)
  - Run a second pass through the entire dataset to verify that the candidate pairs are truly frequent
    - Can remove false positives totally

- **False negatives** will also result
  - Itemset is frequent in the original dataset but not picked out from the sample
  - Scanning the whole dataset a second time does not help
  - Using smaller threshold helps catch more truly frequent itemsets, but requires more space

SON Algorithm

- Repeatedly read small subsets (chunks) of the baskets into main memory and perform the first pass of the previous algorithm on each subset
  - This is not sampling but processing the entire file in memory-sized chunks

- An itemset becomes candidate if it is found to be frequent in at least one subset of the baskets using a scaled-back support threshold

- On a second pass, count all the candidate itemsets and determine which are frequent in the entire set

- Key “monotonicity” idea: an itemset cannot be frequent in the entire set of baskets unless it is frequent in at least one subset
  - A subset contains a fraction $p$ of whole file (number of subsets is $1/p$)
  - If itemset is not frequent in any subset, then the support in each subset is less than $p \ast s$
  - Hence, the support in whole file is less than $s$: not frequent!
    - $(1/p) \ast s = s$
SON Distributed Version

- SON lends itself to *distributed data mining*
  - Map Reduce

- Baskets distributed among many nodes
  - Subsets of the data may correspond to one or more chunks in distributed file system
  - Compute frequent itemsets at each node
    - Phase 1: Find candidate Itemsets
  - Distribute candidates to all nodes
  - Accumulate the counts of all candidates
    - Phase 2: Find true frequent Itemsets

---

SON MapReduce: Phase 1

- Map
  - Input is a chunk/subset of all baskets; fraction $p$ of total input file
  - Find itemsets frequent in that subset:
    - Use support threshold $= s \cdot p$
  - Output is set of key-value pairs $(\text{FrequentItemset}, 1)$ where $\text{FrequentItemset}$ is found from the chunk

- Reduce
  - Each reduce task is assigned a set of keys, which are itemsets
  - Produce keys that appear one or more times
  - Frequent in some subset; these are candidate itemsets
SON MapReduce: Phase 2

- **Map**
  - Each Map task takes a chunk of the total input data file as well as the output of Reduce tasks from phase 1
  - All candidate itemsets go to every Map task
  - Output is set of key-value pairs (CandidateItemset, support) where the support of the CandidateItemset is computed among the baskets of the input chunk

- **Reduce**
  - Each Reduce task is assigned a set of keys, which are candidate itemsets
  - Sums associated values for each key: total support for CandidateItemset
  - If total support of itemset $\geq s$, emit itemset and its count

---

SON Map Reduce (2 in 1)

www.hadooptpoint.com/finding-frequent-itemsets-using-hadoop-mapreduce-model/
Toivonen’s Algorithm

- Toivonen’s algorithm is a *heuristic* algorithm for finding frequent itemsets from a given set of data

- Given *sufficient* main memory, uses one pass over a small sample and one full pass over data
  - Gives no false positives (always check against the whole)

- BUT, there is a *small but finite probability* it will fail to produce an answer
  - Will not identify frequent itemsets (false negatives)

- Then must be repeated with a different sample until it gives an answer
  - Need only a small number of iterations

---

Toivonen’s Algorithm

- Start as in the random sampling algorithm, but *lower the threshold slightly* for the sample
  - For fraction $p$ of baskets in sample, use $0.8ps$ ($0.9ps$) as support threshold
  - *Example*: if the sample is 1% of the baskets, use $s/125$ as the support threshold rather than $s/100$

- Goal: *avoid missing any itemset* that is frequent in the full set of baskets
  - The smaller the threshold the more memory is needed to count all candidate itemsets and the less likely the algorithm will not find an answer

- Add to the itemsets that are frequent in the sample their *negative border*
  - An itemset is in the negative border if it is not deemed frequent in the sample, but *all* its immediate subsets are (subset by deleting a single item)
Example: Negative Border

- $ABCD$ is in the negative border if and only if:
  1. It is not frequent in the sample, but
  2. All of $ABC$, $BCD$, $ACD$, and $ABD$ are
- $A$ is in the negative border if and only if it is not frequent in the sample
  - Because the empty set is always frequent
  - Unless there are fewer baskets than the support threshold (silly case)

Toivonen’s Algorithm

- In a second pass, count all candidate frequent itemsets from the first pass, and also count their negative border
- If no itemset from the negative border turns out to be frequent, then the candidates found to be frequent in the whole data are exactly the frequent itemsets
- What if we find that something in the negative border is actually frequent? We must start over again!
- Try to choose the support threshold so the probability of failure is low, while the number of itemsets checked on the second pass fits in main-memory
If Something in the Negative Border is Frequent . . .

We broke through the negative border. How far does the problem go?

... tripletons
doubletons
singletons

Frequent Itemsets from Sample

Theorem 1

- Given a data set $D$ and a sample set $S \mid S \leq D$, if there is an itemset $T$ that is frequent in $D$ but not frequent in $S$, then there is an itemset $T'$ that is frequent in $D$ and is in the negative border of $S$
  - False negatives appear in the negative border
- *Proof:* Suppose not; i.e.,
  1. There is an itemset $T$ frequent in $D$ but not frequent in $S$, and
  2. Nothing in the negative border is frequent in the whole
- Let $T'$ be a smallest subset of $T$ that is not frequent in $S$
- All subsets of $T$ are also frequent in the whole ($T$ is frequent + monotonicity)
  - $T'$ is frequent in the whole
- *Thus*, $T$ is in the negative border (else not “smallest”)
Theorem 2

- Given a data set $D$ and a sample set $S \mid S \subseteq D$, if there is an itemset $T$ that is frequent in $D$ and the negative border of $S$, then there is an itemset that is frequent in $D$ and not frequent in $S$
  - By definition, any itemset in the negative border of $S$ is not frequent in $S$. Hence $T$ is frequent in $D$ but not frequent in $S$
- During the second pass of the algorithm, if we found an itemset $T$ of the negative border to be frequent in $D$, then we can assume by this theorem that there is an itemset that is frequent in $D$ but not frequent in $S$;
  - in such a case, we are forced to restart the algorithm as we have already failed to discover at least one itemset that is frequent in $D$
- If we found no itemset of the negative border to be frequent in $D$, then by the previous theorem we are permitted to terminate the algorithm as we have discovered all the frequent itemsets of $D$

Toivonen’s Algorithm

- Toivonen's algorithm is a powerful and flexible algorithm that provides a simplistic framework for discovering frequent itemsets in large data sets while also providing enough flexibility to enable performance optimizations directed towards particular data sets.
- Its deceptive simplicity allows us to discover all frequent itemsets through a sampling process
- Numerous optimizations and approximations can be made to improve the algorithm's performance on data sets with particular properties
  - For instance, using a slightly lowered threshold will minimize the omission of itemsets that are frequent in the entire dataset as such omissions result in additional passes through the algorithm
  - However, the support threshold should also be kept reasonably high so that the counts for the itemsets in the second pass in main memory
Summary

- Market-Basket Data and Frequent Itemsets
  - Many-to-Many relationship
- Associating rules
  - Confidence and Support
- The A-Priori Algorithm
  - Combine only frequent subsets
- The PCY algorithm
  - Hash pairs to reduce candidates
- Multistage and Multihash algorithm
  - Multiple hashes
- Randomized and SON algorithm
  - Sample, Divide into Chunks and treat as samples by MapReduce
- Toivonen's Algorithm
  - Negative Border

References

- CS246: Mining Massive Datasets Jure Leskovec, Anand Rajaraman, Jeff Ullman, Stanford University, 2014
- CS5344: Big Data Analytics Technology, TAN Kian-Lee, National University of Singapore 2014
- CS059: Data Mining, Panayiotis Tsaparas University of Ioannina, Fall 2012
Research on Pattern Mining: A Road Map