Frequent Itemset Mining & Association Rules

CS246: Mining Massive Datasets
Jure Leskovec, Stanford University
http://cs246.stanford.edu
Supermarket shelf management – Market-basket model:

- **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
- **A classic rule:**
  - If someone buys diaper and milk, then he/she is likely to buy beer
  - Don’t be surprised if you find six-packs next to diapers!
The Market-Basket Model

- A large set of *items*
  - e.g., things sold in a supermarket
- A large set of *baskets*
- Each basket is a small subset of *items*
  - e.g., the things one customer buys on one day

- A general many-to-many mapping (association) between two kinds of things
  - But we ask about connections among “items”, not “baskets”

<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>
Given a set of baskets
Want to discover association rules
- People who bought \{x, y, z\} tend to buy \{v, w\}
  - Amazon!

2 step approach:
- 1) Find frequent itemsets
- 2) Generate association rules

### Input:

<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>

### Rules Discovered:
- \{Milk\} --> \{Coke\}
- \{Diaper, Milk\} --> \{Beer\}
Applications – (1)

- **Items** = products; **Baskets** = sets of products someone bought in one trip to the store
- **Real market baskets:** Chain stores keep TBs of data about what customers buy together
  - Tells how typical customers navigate stores, lets them position tempting items
  - Suggests tie-in “tricks”, e.g., run sale on diapers and raise the price of beer
  - Need the rule to occur frequently, or no $$’s
- **Amazon’s people who bought X also bought Y**
Applications – (2)

- **Baskets** = sentences; **Items** = documents containing those sentences
  - Items that appear together too often could represent plagiarism
  - Notice items do not have to be “in” baskets

- **Baskets** = patients; **Items** = drugs & side-effects
  - Has been used to detect combinations of drugs that result in particular side-effects
  - **But requires extension:** Absence of an item needs to be observed as well as presence
Outline

First: Define
- Frequent itemsets
- Association rules:
  - Confidence, Support, Interestingness

Then: Algorithms for finding frequent itemsets
- Finding frequent pairs
- A-Priori algorithm
- PCY algorithm + 2 refinements
Frequent Itemsets

- **Simplest question:** Find sets of items that appear together “frequently” in baskets
- **Support** for itemset $I$: Number of baskets containing all items in $I$
  - Often expressed as a fraction of the total number of baskets
- Given a **support threshold** $s$, then 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 \{Beer, Bread\} = 2
Example: Frequent Itemsets

- **Items** = \{milk, coke, pepsi, beer, juice\}
- **Support threshold** = 3 baskets

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

- **Frequent itemsets:** \{m\}, \{c\}, \{b\}, \{j\}, \{m,b\}, \{b,c\}, \{c,j\}.
Association Rules:

- If-then rules about the contents of baskets
- \( \{i_1, i_2, \ldots, i_k\} \rightarrow j \) means: “if a basket contains all of \( i_1, \ldots, i_k \) then it is likely to contain \( j \)”
- In practice there are many rules, want to find significant/interesting ones!
- **Confidence** of this association rule is the probability of \( j \) given \( I = \{i_1, \ldots, i_k\} \)

\[
\text{conf}(I \rightarrow j) = \frac{\text{support}(I \cup j)}{\text{support}(I)}
\]
Not all high-confidence rules are interesting

- The rule $X \rightarrow \text{milk}$ may have high confidence for many itemsets $X$, because milk is just purchased very often (independent of $X$) and the confidence will be high.

**Interest** of an association rule $I \rightarrow j$: difference between its confidence and the fraction of baskets that contain $j$

$$\text{Interest}(I \rightarrow j) = \text{conf}(I \rightarrow j) - \Pr[j]$$

- Interesting rules are those with high positive or negative interest values (usually above 0.5)
Example: Confidence and Interest

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

- **Association rule:** \(\{m, b\} \rightarrow c\)
  - **Confidence** = \(\frac{2}{4} = 0.5\)
  - **Interest** = \(|0.5 - \frac{5}{8}| = \frac{1}{8}\)
    - Item \(c\) appears in \(\frac{5}{8}\) of the baskets
    - Rule is not very interesting!
Problem: Find all association rules with support $\geq s$ and confidence $\geq c$

- Note: Support of an association rule is the support of the set of items on the left side

Hard part: Finding the frequent itemsets!

- If $\{i_1, i_2, \ldots, i_k\} \rightarrow j$ has high support and confidence, then both $\{i_1, i_2, \ldots, i_k\}$ and $\{i_1, i_2, \ldots, i_k, j\}$ will be “frequent”

$$conf(I \rightarrow j) = \frac{support(I \cup j)}{support(I)}$$
Step 1: Find all frequent itemsets \( I \)
   - (we will explain this next)
Step 2: Rule generation
   - For every subset \( A \) of \( I \), generate a rule \( A \rightarrow I \setminus A \)
     - Since \( I \) is frequent, \( A \) is also frequent
     - **Variant 1:** Single pass to compute the rule confidence
       - \( \text{confidence}(A,B \rightarrow C,D) = \frac{\text{support}(A,B,C,D)}{\text{support}(A,B)} \)
     - **Variant 2:**
       - **Observation:** If \( A,B,C \rightarrow D \) is below confidence, so is \( A,B \rightarrow C,D \)
       - Can generate “bigger” rules from smaller ones!

Output the rules above the confidence threshold
Example

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

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

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

2) Generate rules:
   - \( b \rightarrow m: c = 4/6 \quad b \rightarrow c: c = 5/6 \quad b, c \rightarrow m: c = 3/5 \)
   - \( m \rightarrow b: c = 4/5 \quad ... \quad b, m \rightarrow c: c = 3/4 \)
   - \( b \rightarrow c, m: c = 3/6 \)
To reduce the number of rules we can post-process them and only output:

- **Maximal frequent itemsets:**
  No immediate superset is frequent
  - Gives more pruning

or

- **Closed itemsets:**
  No immediate superset has the same count (> 0)
  - Stores not only frequent information, but exact counts
### Example: Maximal/Closed

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

- **A** has count 4 and is not maximal, so it is not closed.
- **B** has count 5 and is not maximal, so it is closed.
- **AB** has count 4 and is maximal, so it is closed.
- **BC** has count 3 and is maximal, so it is closed.
- **ABC** has count 2 and is not maximal, so it is not closed.

**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.**
We are releasing HW1 today

- It is due in 2 weeks
- The homework is long
  - Requires proving theorems as well as coding
- Please start early

Recitation session:

- Review of **probability** and **proof techniques**
- Tomorrow 4:15-5:30pm in Gates B03
Finding Frequent Itemsets
Back to finding frequent itemsets

Typically, data is kept in flat files rather than in a database system:

- Stored on disk
- Stored 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 sets of size $k$

Note: We want to find frequent itemsets. To find them, we have to count them. To count them, we have to generate them.
The true cost of mining disk-resident data is usually the **number of disk I/Os**

- In practice, association-rule algorithms read the data in **passes** – all baskets read in turn

- We measure the cost by the **number of passes** an algorithm makes over the data
For many frequent-itemset algorithms, **main-memory** is the critical resource

- As we read baskets, we need to count something, e.g., occurrences of pairs of items
- The number of different things we can count is limited by main memory
- Swapping counts in/out is a disaster (**why?**)
The hardest problem often turns out to be finding the frequent **pairs** of items \( \{i_1, i_2\} \)

- **Why?** Freq. pairs are common, freq. triples are rare
  - **Why?** Probability of being frequent drops exponentially with size; number of sets grows more slowly with size

Let’s first concentrate on pairs, then extend to larger sets

**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
Naïve Algorithm

- Naïve approach to finding frequent pairs
- Read file once, counting in main memory the occurrences of each pair:
  - From each basket of $n$ items, generate its $\frac{n(n-1)}{2}$ pairs by two nested loops
- Fails if $(#\text{items})^2$ exceeds main memory
  - Remember: #items can be 100K (Wal-Mart) or 10B (Web pages)
    - Suppose $10^5$ items, counts are 4-byte integers
    - Number of pairs of items: $10^5(10^5-1)/2 = 5\times10^9$
    - Therefore, $2\times10^{10}$ (20 gigabytes) of memory needed
Two approaches:

- **Approach 1**: Count all pairs using a matrix
- **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 for the hashtable

**Note:**

- **Approach 1** only requires 4 bytes per pair
- **Approach 2** uses 12 bytes per pair (but only for pairs with count > 0)
Comparing the 2 Approaches

Triangular Matrix

4 bytes per pair

Triples

12 per occurring pair
Comparing the two approaches

- **Approach 1: Triangular Matrix**
  - \( n \) = total number items
  - Count pair of items \( \{i, j\} \) only if \( i < j \)
  - 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 \( (i - 1)(n - i/2) + j - 1 \)
  - Total number of pairs \( n(n - 1)/2 \); total bytes = \( 2n^2 \)
  - **Triangular Matrix** requires 4 bytes per pair

- **Approach 2** uses **12 bytes** per occurring pair
  - **(but only for pairs with count > 0)**
  - Beats Approach 1 if less than \( 1/3 \) of possible pairs actually occur
Comparing the two approaches

- **Approach 1: Triangular Matrix**
  - \( n = \) total number items
  - Count pair of items \( \{i, j\} \) only if \( i < j \)
  - 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 \( (i-1)(n-i)/2 + j - 1 \)
  - Total number of pairs \( n(n-1)/2 \); total bytes = 2\( n^2 \)
  - Triangular Matrix requires 4 bytes per pair

- **Approach 2** uses 12 bytes per pair (but only for pairs with count > 0)
  - Beats Approach 1 if less than \( 1/3 \) of possible pairs actually occur

Problem is if we have too many items so the pairs do not fit into memory. Can we do better?
A-Priori Algorithm
A two-pass approach called A-Priori limits the need for main memory.

Key idea: monotonicity

- If a set of items $I$ appears at least $s$ times, so does every subset $J$ of $I$.

Contrapositive for pairs:
If item $i$ does not appear in $s$ baskets, then no pair including $i$ can appear in $s$ baskets.

So, how does A-Priori find freq. pairs?
A-Priori Algorithm – (2)

- **Pass 1:** Read baskets and count in main memory the occurrences of each *individual item*
  - Requires only memory proportional to #items

- **Items that appear \( \geq s \) times are the frequent items**

- **Pass 2:** Read baskets again and count in main memory only those pairs where both elements are frequent (from Pass 1)
  - 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)
Main-Memory: Picture of A-Priori

- **Pass 1**: Item counts
- **Pass 2**: Frequent items
  - Counts of pairs of frequent items (candidate pairs)

Main memory
You can use the triangular matrix method with $n = \text{number of frequent items}$

- May save space compared with storing triples

**Trick**: re-number frequent items 1, 2, ... and keep a table relating new numbers to original item numbers

- **Item counts**
- **Frequent items**
- **Old item #s**
- **Counts of pairs of frequent items**

**Pass 1**

**Pass 2**
For each $k$, we construct two sets of $k$-tuples (sets of size $k$):

- $C_k =$ candidate $k$-tuples = 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
Example

Hypothetical steps of the A-Priori algorithm

- \( C_1 = \{ \{b\} \{c\} \{j\} \{m\} \{n\} \{p\} \} \)
- 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.
One pass for each \(k\) (itemset size)

Needs room in main memory to count each candidate \(k\)-tuple

For typical market-basket data and reasonable support (e.g., 1%), \(k = 2\) requires the most memory

Many possible extensions:

- Association rules with intervals:
  - For example: Men over 65 have 2 cars
- Association rules when items are in a taxonomy
  - Bread, Butter \(\rightarrow\) FruitJam
  - BakedGoods, MilkProduct \(\rightarrow\) PreservedGoods
- Lower the support \(s\) as itemset gets bigger
PCY (Park-Chen-Yu) Algorithm
Observation:
In pass 1 of A-Priori, most memory is idle
- We store only individual item counts
- Can we use the idle memory to reduce memory required in pass 2?

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
  - For each bucket just keep the count, not the actual pairs that hash to the bucket!
PCY Algorithm – First Pass

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;

- Few things to note:
  - Pairs of items need to be generated from the input file; they are not present in the file
  - We are not just interested in the presence of a pair, but we need to see whether it is present at least \( s \) (support) times
Observations about Buckets

- **Observation**: 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 😞
  - So, we cannot use the hash to eliminate any member (pair) of a “frequent” bucket.
- **But, for a bucket with total count less than $s$, 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**: Only count pairs that hash to frequent buckets.
Replace the buckets by a bit-vector:

1 means the bucket count exceeded the support $s$ (call it a frequent bucket); 0 means it did not.

4-byte integer counts 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:
  1. Both \( i \) and \( j \) are frequent items
  2. The pair \( \{i, j\} \) hashes to a bucket whose bit in the bit vector is 1 (i.e., a frequent bucket)

- Both conditions are necessary for the pair to have a chance of being frequent
Main-Memory: Picture of PCY

Pass 1
- Hash table for pairs
- Item counts

Pass 2
- Bitmap
- Frequent items
- Counts of candidate pairs

Main memory
Buckets require a few bytes each:

- **Note:** we do not have to count past $s$
- #buckets is $O(main-memory\ size)$

On second pass, a table of \((item, item, count)\) triples is essential (we cannot use triangular matrix approach, why?)

- Thus, hash table must eliminate approx. 2/3 of the candidate pairs for PCY to beat A-Priori
Refinement: Multistage Algorithm

- **Limit the number of candidates to be counted**
  - **Remember:** Memory is the bottleneck
  - Still need to generate all the 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**

- Requires 3 passes over the data
Main-Memory: Multistage

- **First hash table**
  - Item counts
  - Bitmap 1

- **Second hash table**
  - Freq. items
  - Bitmap 1
  - Bitmap 2
  - Counts of candidate pairs

**Pass 1**
- Count items
- Hash pairs \{i,j\}

**Pass 2**
- Hash pairs \{i,j\} into Hash2 iff:
  - \(i,j\) are frequent,
  - \{i,j\} hashes to freq. bucket in B1

**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
Count only those pairs \( \{i, j\} \) that satisfy these candidate pair conditions:

1. Both \( i \) and \( j \) are frequent items
2. Using the first hash function, the pair hashes to a bucket whose bit in the first bit-vector is 1
3. Using the second hash function, the pair hashes to a bucket whose bit in the second bit-vector is 1
Important Points

1. The two hash functions have to be independent

2. We need to check both hashes on the third pass
   - If not, we would end up counting pairs of frequent items that hashed first to an infrequent bucket but happened to hash second to a frequent bucket
Refinement: Multihash

- **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
Main-Memory: Multihash

Pass 1

Main memory

First hash table

Second hash table

Item counts

Counts of candidate pairs

Pass 2

Freq. items

Bitmap 1

Bitmap 2
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 $\geq s$