DSA5101 Introduction to Big Data for Industry

AY2025/26 Sem1 By Zhao Peiduo

Lecture 1

Frequent Itemsets and Association Rules

Association Rule Discovery Market-basket model:

- **Goal:** Identify items that are bought together by many supermarket customers.
- **Approach:** Process sales data of each customer to find dependencies among items.

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 (what a customer buys).
- Want to discover **association rules** — e.g., people who bought $\{x, y, z\}$ tend to buy $\{v, w\}$.
- A many-to-many mapping between two kinds of things — connections among **items**, not baskets.
- Example:

Input:

TID	Items
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Output: Rules Discovered: {Milk → Coke}, {Diaper, Milk → Beer}

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 as a fraction of total baskets).
- Given a **support threshold** s: sets of items appearing in at least s baskets are **frequent itemsets**.
- Example:Support of {Beer, Bread} = 2 baskets.

Association Rules

- $\{i_1, \ldots, i_k\} \to j$ means: "if a basket contains all i's, it's *likely* to contain j".
- **Confidence:** Probability of j given $I = \{i_1, \ldots, i_k\}$

$$\text{conf}(I \to j) = \frac{\text{support}(I \cup \{j\})}{\text{support}(I)}$$

Interesting Association Rules

- Not all high-confidence rules are interesting — e.g., $X \to$ milk might be high just because milk is common.
- **Interest:** Difference between confidence and the fraction of baskets containing j

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

- Interesting rules have high positive or negative interest values (usually above 0.5).

Finding Association Rules Problem: Find all rules with support $\geq s$ and confidence $\geq c$.

Mining Association Rules

- Find all frequent itemsets I.
- **Rule generation:** For every subset A of I, generate a rule $A \to I \setminus A$ with:

$$\text{conf}(A \to I \setminus A) = \frac{\text{support}(I)}{\text{support}(A)}$$

- **Observation 1:** Single pass over subsets of I to compute confidence.
- **Observation 2:** Monotonicity — if $B \subset A \subset I$, then

$$\text{conf}(B \to I \setminus B) \leq \text{conf}(A \to I \setminus A)$$

- Use monotonicity to prune rules below confidence threshold.

Itemsets: Computation Model

- Data is typically kept in flat files:
  - Stored on disk, too large to fit in main memory.
  - Stored basket-by-basket (e.g., $20, 52, 38, -1, 40, 22, -1, 20, 22, -1, \ldots$)
- **Major cost:** Time taken to read baskets from disk to memory.
- Assume baskets are small — a block of baskets can be expanded in main memory to generate all subsets of size k via k nested loops.
- Large subsets can often be ruled out using **monotonicity**.

Communication Cost is Key

- Running time $\propto$ #passes through data $\times$ data size.
- A pass = reading all baskets sequentially.
- Since data size is fixed, measure speed by number of passes.

Main-Memory Bottleneck

- For many algorithms, main memory is the limiting factor.
- While reading baskets, we need to count occurrences (e.g., pairs).
- The number of distinct items we can count is limited by memory.

Finding Frequent Pairs

- Goal: Find frequent pairs $\{i_1, i_2\}$.
- Frequent pairs are common; frequent triples are rare.
- Probability of being frequent drops exponentially with set size.
- **Approach:**
  - First focus on pairs, then extend to larger sets.
  - Generate all itemsets, but keep only those likely to be frequent.

Counting Pairs in Memory

- **Approach 1:** Use a matrix to count all pairs.
- **Approach 2:** Store triples $[i, j, c]$ meaning count of pair $\{i, j\}$ is c.
- Memory usage:
  - Approach 1: 4 bytes per pair.
  - Approach 2: 12 bytes per pair with count > 0.

Comparing the Two Approaches

- Triangular matrix: Count only if $i < j$, needs $2n^2$ bytes total.
- Approach 2 wins if less than 1/3 of possible pairs occur.

If Memory Fits All Pairs:

1. For each basket, double loop to generate all pairs.
2. Increment count for each generated pair.

Apriori Algorithm

- A two-pass (and beyond) algorithm that limits memory usage using **monotonicity (downward closure)**.
- **Key idea:** If I is frequent, then every subset $J \subset I$ is also frequent.
- **Contrapositive for pairs:** If i is infrequent, then no pair containing i can be frequent.

Steps (all itemset sizes)

- **Pass 1 (for $k = 1$):** Count each item; items with count $\geq s$ form L_1 (frequent items).
- **For $k \geq 2$:**
  - **Candidate generation:** From L_{k-1}, form C_k by joining two $(k-1)$-itemsets that share exactly $k-2$ items; take their union (size k).
  - **Pruning:** Remove any $I \in C_k$ if some $(k-1)$-subset of I is not in L_{k-1} (by monotonicity).
  - **Counting:** Make one pass; count supports of C_k.
  - **Filtering:** $L_k = \{I \in C_k \mid \text{support}(I) \geq s\}$.

Rule Generation (from frequent itemsets)

- For each frequent I and each nonempty $A \subset I$, form the rule $A \to I \setminus A$ with

$$\text{conf}(A \to I \setminus A) = \frac{\text{support}(I)}{\text{support}(A)}.$$

- **Observation (confidence monotonicity):** If $B \subset A \subset I$, then

$$\text{conf}(B \to I \setminus B) \leq \text{conf}(A \to I \setminus A).$$

- Use this to prune rules below the confidence threshold c.

Memory Requirement

- Pass 1: Memory $\propto$ number of (distinct) items.
- Pass k ($k \geq 2$): Memory $\propto$ number of candidates in C_k (for market-basket data and reasonable s, $k = 2$ is often the heaviest).
- **Trick:** Re-number frequent items $1, 2, \ldots$ and keep a map to original item IDs for compact indexing.

Hash Functions

- **Definition:** A hash function h takes a *hash-key value* and produces a *bucket number* $\in [0, B - 1]$.
- Should randomize hash-keys roughly uniformly into buckets.

Indexing using Hash Functions

- Used for indexing to enable fast search/retrieval.
- **Example:** Hash the name to the ordinal position of the first letter, use as bucket index.

PCY (Park–Chen–Yu) Algorithm

- **Observation:** In Apriori's Pass 1, most memory is idle (only item counts stored). Use spare memory to *hash pairs into buckets* and prune candidate pairs before Pass 2.

Pass 1 of PCY

- Maintain a hash table with as many buckets as memory allows.
- For each basket:
  - Count each item's frequency (to get frequent items L_1).
  - For each unordered pair $\{i, j\}$ in the basket: compute bucket $b = h(\{i, j\})$ and increment that bucket's count (cap counts at s if desired).
- After the pass, replace bucket counts with a **bit vector**: bit $b = 1$ iff bucket count $\geq s$, else 0 (bitmap uses about 1/32 the memory of 32-bit integer counts).

Using Hash Buckets to Prune Candidate Pairs

- If a bucket's count $< s$, then no pair hashing to that bucket can be frequent — prune them.
- If a pair is truly frequent, its bucket must be frequent (so it will not be pruned).

PCY – Pass 2

- Count only pairs $\{i, j\}$ that satisfy **both**:
  1. i and j are frequent items (i.e., in L_1), and
  2. The pair hashes to a bucket whose bit is 1 in the bitmap (a "frequent" bucket).
- Note: On this pass, a table of (item, item, count) triples is essential (triangular matrices don't align with hash pruning).
- For PCY to beat Apriori, the hash table should eliminate roughly $\geq 2/3$ of the candidate pairs.

Refinement: Multistage Algorithm (3 passes)

- After Pass 1 of PCY, *rehash only* the pairs that would be considered in PCY's Pass 2 (i.e., both items frequent and first-hash bucket frequent) using an **independent** hash function to a second bucket table.
- Replace the second bucket counts with a second bit vector (slightly smaller, e.g., $\frac{31}{32}$ size).

Multistage – Pass 3

- Count only pairs $\{i, j\}$ that satisfy all:
  1. i and j are frequent,
  2. $\{i, j\}$ hashes to a frequent bucket in the *first* bitmap, and
  3. $\{i, j\}$ hashes to a frequent bucket in the *second* bitmap.
- Effect: Fewer candidate pairs than plain PCY, with combined bitmaps using about $\frac{1}{16}$ of the memory of integer bucket counts.

Important Points

1. The two hash functions must be independent.
2. Both hashes must be checked on the final counting pass.
3. More stages are possible for additional pruning, but each stage needs another bitmap; eventually memory runs out.

Refinement: Multihash (2 passes)

- Use several independent hash tables in *the first pass* and create multiple bitmaps (same total bitmap space as PCY if divided).
- **Pass 2:** Count only pairs whose buckets are frequent in *all* bitmaps (analogous to Pass 3 of multistage).
- *Trade-off:* Halving buckets doubles average bucket count; ensure many buckets still fall below s to retain pruning power.

Main-Memory Details

- We do not need to count a bucket past s.
- On the second pass, a triple table is required (cannot use a triangular matrix).

Adding More Hash Functions

- Either multistage or multihash can use more than two hash functions.
- In multistage, diminishing returns as bit-vectors consume memory.
- In multihash, bit-vectors use same space as one PCY bitmap; too many hash functions cause most buckets to become frequent.

Random Sampling

- If data is too large to fit in main memory, but a random sample fits in memory, then:
  - Run an in-memory frequent-itemset algorithm (e.g., A-Priori) on the sample.
  - Scale down support threshold proportionally.
- Challenge: Itemsets that are frequent in the whole dataset may not appear frequent in the sample.
- Result: May miss some frequent itemsets (false negatives).
- Advantage: Very fast and memory efficient.

SON Algorithm (Savasere, Omiecinski, Navathe)

- Works for distributed or map-reduce environment.
- Divide dataset into k chunks.
- Each chunk fits in memory.
- For each chunk:
  1. Find candidate frequent itemsets in the chunk (local frequent).
  2. Collect all candidates from all chunks.
- Run a second pass over the whole dataset to count the candidates' true support.
- Guarantee: Any itemset that is globally frequent must appear as frequent in at least one chunk.

SON Algorithm – Pass 1

- Each mapper runs A-Priori (or similar) on its chunk.
- Outputs locally frequent itemsets.
- Reducers aggregate all candidates across chunks.
- Result: A superset of globally frequent itemsets.

SON Algorithm – Pass 2

- Each mapper:
  - Counts support of the candidate itemsets from Pass 1 in its chunk.
- Reducers sum counts across mappers.
- Output: Globally frequent itemsets.

Why SON Works

- Suppose itemset I is frequent in the whole dataset.
- Then I must be frequent in at least one chunk.
- Thus I will be found in Pass 1.
- No false negatives.
- May have false positives (candidates that are not globally frequent).

Toivonen's Algorithm

- Another sampling-based algorithm.
- Steps:
  1. Take a random sample of the dataset that fits in memory.
  2. Run A-Priori (or similar) on the sample with a lowered support threshold.
  3. Result: Candidate itemsets (may contain false positives, but hopefully no false negatives).
  4. Run a second pass over the whole dataset to count supports of candidates.
  5. If no "frequent" itemsets are missed, done.
  6. Otherwise, repeat with a larger sample.

Toivonen's Algorithm – Key Idea

- Reduce risk of false negatives by lowering the support threshold in the sample.
- False positives are okay (they will be eliminated in the second pass).
- If a false negative occurs, restart with larger sample.

Toivonen's Algorithm – Example

- True support threshold: 5%.
- Sample size: 10% of dataset.
- Adjusted threshold: 0.5%.
- Find itemsets frequent in sample $\geq 0.5\%$.
- Verify on full dataset.

Toivonen's Algorithm – Advantages and Disadvantages

- Advantage:
  - Typically needs only 2 passes (sample + full dataset).
  - Efficient for large datasets.
- Disadvantage:
  - Risk of false negatives (forces restart).
  - Sample size and threshold adjustment critical.

Comparison: SON vs. Toivonen

- **SON**:
  - Always correct (no false negatives).
  - Requires 2 full passes of dataset.
  - Well-suited for distributed/MapReduce.
- **Toivonen**:
  - May fail and require restart, but usually only 2 passes.
  - Efficient when sample fits in memory.
  - Risk of wasted work if sample is not representative.

Comparison of Frequent Itemset Algorithms

Feature	**Apriori**	**PCY**	**Random Sampling**	**SON**	**Toivonen's**
Number of passes	One for each itemset size	One for each itemset size	< 2	2	< 2
False positives	No	No	No	No	No
False negatives	No	No	Yes	No	No
Memory usage	High	Lower than Apriori for pairs	Lower due to on-the-fly filter	Lower due to on-the-fly filter	Need to store negative border
Scalable to big data	Poor	Slightly better	Very good	Very good	Good
Candidate generation	Explicit, bottoms up	Same as Apriori, except for pairs	Sample-based heuristics	Same as Apriori, per chunk	Sample + negative border