


Mining Large Scale Datasets

2024 / 2025

Frequent Itemsets and Association Rules

(Adapted from CS246@Stanford.edu; <http://www.mmds.org>)

Calendar updates

- The missed class on 27 February will be substituted by the following workshop or April, 17h-19h:

<https://indico.hpt/event/134/>
- The workshop was moved on 28th of April – registration is open
 - Participation is encouraged, not mandatory; final exam will not cover Julia
- The missed class on 27th of February will be substituted with an on-line/hybrid class on 30th of April/7th of May – 2nd assignment Q&A
- 2nd Assignment – introduced today; deadline: 7th of May 23h59

Association Rule Discovery

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

Discover association rules:

People who bought $\{x,y,z\}$ tend to buy $\{v,w\}$

- Example application: Amazon

Input:

<i>Basket</i>	<i>Items</i>
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Output:

Rules Discovered:

$\{\text{Milk}\} \rightarrow \{\text{Coke}\}$

$\{\text{Diaper, Milk}\} \rightarrow \{\text{Beer}\}$

Frequent itemset

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

Sets of items that appear in at least s baskets are called **frequent itemsets**

Association rules: Confidence

$I \rightarrow j$

if all items in I appear in a basket then it is likely that j appears in the same basket

Confidence of a rule $I \rightarrow j$ is the probability of j given I , calculated as the ratio of the support of $I \cup \{j\}$ to the support of I

$$\text{confidence } I \rightarrow j = \text{support } I \cup \{j\} / \text{support}(I)$$

i.e. fraction of the baskets with all of I that also contain j

Association rules: Lift

$I \rightarrow j$

if all items in I appear in a basket then it is likely that j appears in the same basket

$$\text{Lift } I \rightarrow j = \frac{\text{confidence}(I \rightarrow j)}{P(j)} = \frac{P(I | j)}{P(I) P(j)}$$

Lift (also known as the observed/expected ratio) is a measure of the degree of dependence between I and j .

A lift of 1 indicates that I and j are independent.

Mining Association Rules

- ▶ Finds combinations of items that occur frequently.
- ▶ Tries to turn those into “If...then...” rules.
- ▶ Measures how confident we are in those rules.
- ▶ Uses optimization tricks to skip bad rules and speed things up.

Short recap

- ▶ **Market-Basket Data:** This model of data assumes there are two kinds of entities: **items** and **baskets**. There is a many–many relationship between items and baskets. Typically, baskets are related to small sets of items, while items may be related to many baskets.
- ▶ **Frequent Itemsets:** The support for a set of items is the number of baskets containing all those items. Itemsets with support that is at least some threshold are called frequent itemsets.
- ▶ **Association Rules:** These are implications that if a basket contains a certain set of items I , then it is likely to contain another particular item j as well. The probability that j is also in a basket containing I is called the **confidence** of the rule. The **interest** of the rule is the amount by which the confidence deviates from the fraction of all baskets that contain j .

Class Exercise

Suppose there are 100 items, numbered 1 to 100, and also 100 baskets, also numbered 1 to 100. Item i is in basket b if and only if i divides b with no remainder. Thus, item 1 is in all the baskets, item 2 is in all fifty of the even-numbered baskets, and so on.

Basket 12 consists of items {1,2,3,4,6,12}, since these are all the integers that divide 12.

Answer the following questions:

- (a) If the support threshold is 5, which items are frequent?
- (b) If the support threshold is 5, which pairs of items are frequent?
- (c) What is the sum of the sizes of all the baskets?

What is the confidence of the following association rules?

- (a) $\{5,7\} \rightarrow 2$.
- (b) $\{2,3,4\} \rightarrow 5$.
- (c) What is the lift of the previous association rules?

Notes:

$$\text{Confidence}(A \rightarrow B) = \text{Support}(A \cup B) / \text{Support}(A)$$

$$\begin{aligned} \text{Lift}(A \rightarrow B) &= \text{Confidence}(A \rightarrow B) / (\text{Support}(B) / N) \\ &= [\text{Support}(A \cup B) \times N] / [\text{Support}(A) \times \text{Support}(B)] \end{aligned}$$

Where:

Support(X) = number of baskets containing itemset X

N = total number of baskets

Itemsets: Computation Model

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 enumerate them.

Item
Item
Item
Item
Item
Item
Item
Item
Item
Item
Item
Etc.

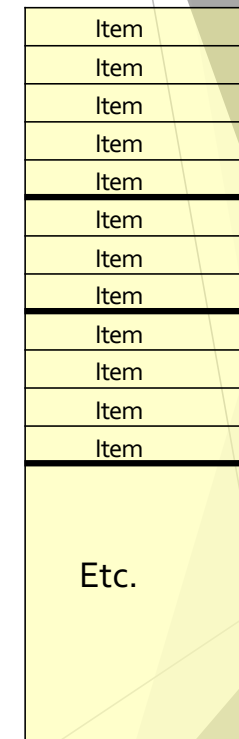
Items are positive integers, and boundaries between baskets are -1.

Computation Model

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

A vertical rectangular box representing a data basket. It is divided into 12 horizontal sections. The top 11 sections are thin and each contains the word 'Item'. The bottom section is significantly larger than the others and contains the text 'Etc.'. The entire box is light yellow with a thin black border.

Item
Item
Item
Item
Item
Item
Item
Item
Item
Item
Item
Etc.

Items are positive integers,
and boundaries between
baskets are -1.

Main-Memory Bottleneck

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 of main-memory is a bad idea

Finding Frequent Pairs

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

Finding Frequent Pairs

The approach:

- We always need to “generate” all the itemsets
- But we would only like to count (keep track of) only those itemsets that in the end turn out to be frequent

Scenario:

- Imagine we aim to identify frequent pairs
- We will need to enumerate all pairs of items
 - For every basket, enumerate all pairs of items in that basket
- **But**, rather than keeping a count for every pair, we hope to discard a lot of pairs and only keep track of the ones that will 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 b of n_b items, generate its $n_b(n_b-1)/2$ pairs by two nested loops
- A data structure then keeps count of every pair

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: $10^5(10^5-1)/2 \approx 5 \cdot 10^9$
 - Therefore, $2 \cdot 10^{10}$ (20 gigabytes) of memory is needed

Counting Pairs in Memory

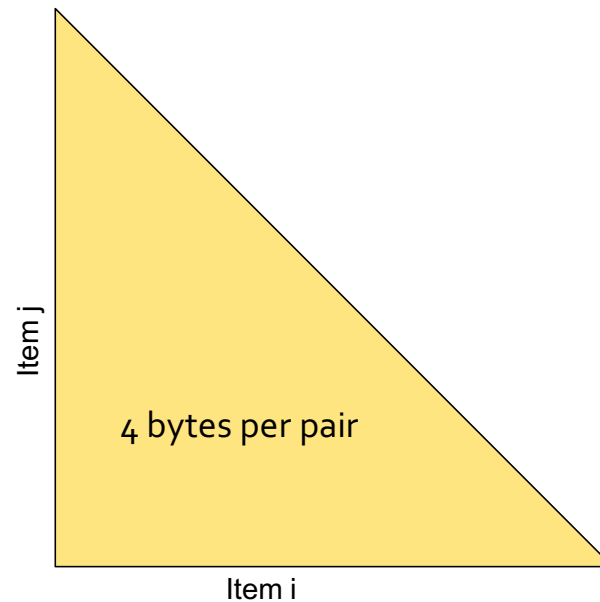
Goal: Count the number of occurrences of each pair of items (i,j) :

Approach 1: Count all pairs using a matrix

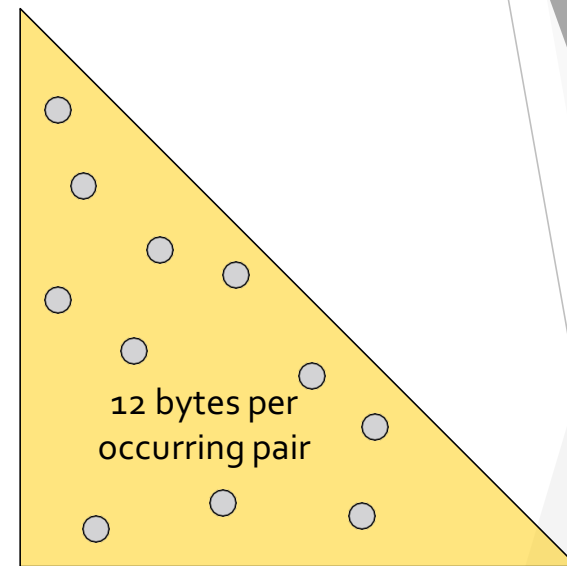
Approach 2: Keep a table of triples $[i, j, c]$ =

- ▶ “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/dictionary

Comparing the two approaches



Triangular Matrix



Triples (item i, item j, count)

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:
 - $\{1,2\}, \{1,3\}, \dots, \{1,n\}, \{2,3\}, \{2,4\}, \dots, \{2,n\}, \{3,4\}, \dots$
- Total number of pairs $n(n-1)/2$; total bytes = $O(n^2)$
- **Triangular Matrix** requires 4 bytes per pair

Approach 2 uses **12 bytes** per occurring pair
(*but only for pairs with count > 0*)

Approach 2 beats Approach 1 if less than **1/3** of possible pairs actually occur

Comparing the two approaches

- ▶ While one could use a two-dimensional array to count pairs, doing so wastes half the space, because there is no need to count pair $\{i,j\}$ in both the i - j and j - i array elements. By arranging the pairs (i,j) for which $i < j$ in lexicographic order, we can store only the needed counts in a one-dimensional array with no wasted space, and yet be able to access the count for any pair efficiently.
- ▶ If fewer than $1/3$ of the possible pairs actually occur in baskets, then it is more space-efficient to store counts of pairs as triples (i,j,c) , where c is the count of the pair $\{i,j\}$, and $i < j$. An index structure such as a hash table allows us to find the triple for (i,j) efficiently.

Finding frequent itemsets

Monotonicity of Itemsets

If a set I of items is frequent, then so is every subset of I

– or –

An itemset cannot be frequent unless all of its subsets are

A-priori algorithm

A **two-pass** approach called **A-Priori** limits the need for main memory

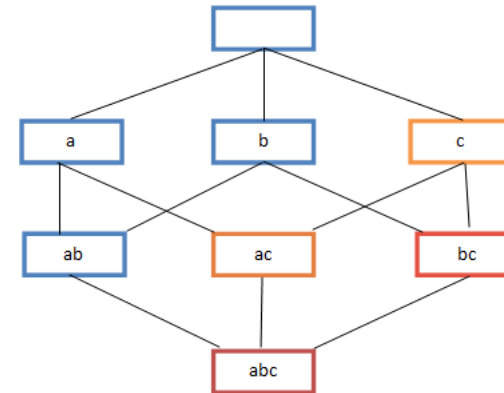
Key idea: *monotonicity*

- 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

Pass 1: Read baskets and count in main memory the # of occurrences of each **individual item**

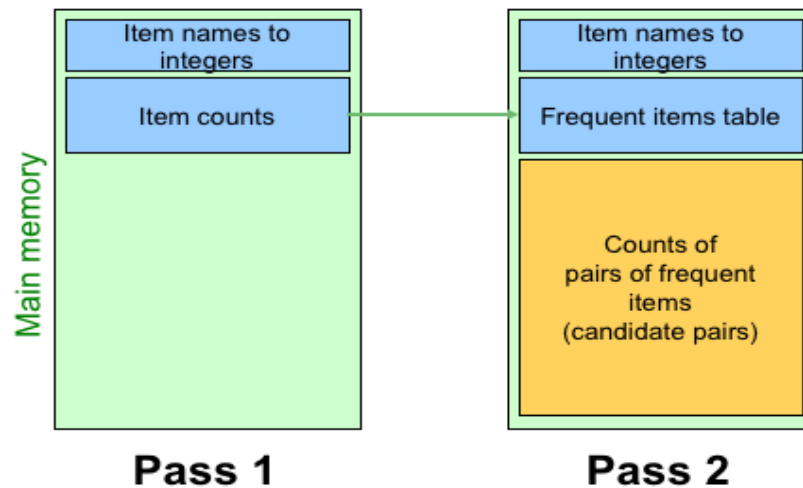
- Requires only memory proportional to #items

Items that appear $\geq s$ times are the frequent items

Pass 2: Read baskets again and keep track of the count of only those pairs where both elements are frequent (from Pass 1)

- Requires memory (for counts) proportional to square of the number of **frequent** items (not the square of total # of items)
- Plus a list of the frequent items (so you know what must be counted)

A-priori algorithm – main memory



Green box represents the amount of available main memory. Smaller boxes represent how the memory is used.

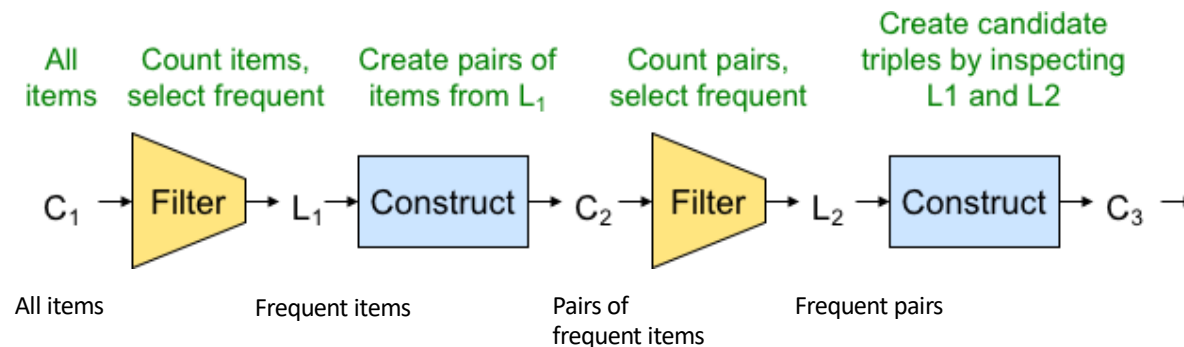
A-Priori algorithm

```
%first pass
for (each basket)
    for (each item i in basket)
        item_counts[i] += 1

%create frequent items table
frequent_items = frequent_items_table(item_counts)

%second pass
for (each item i in basket)
    if i not in frequent_items: continue
    for (each item j in basket)      %with j > i
        if j in frequent_items
            pair_counts[i, j] += 1
```

A-Priori algorithm: $k > 2$



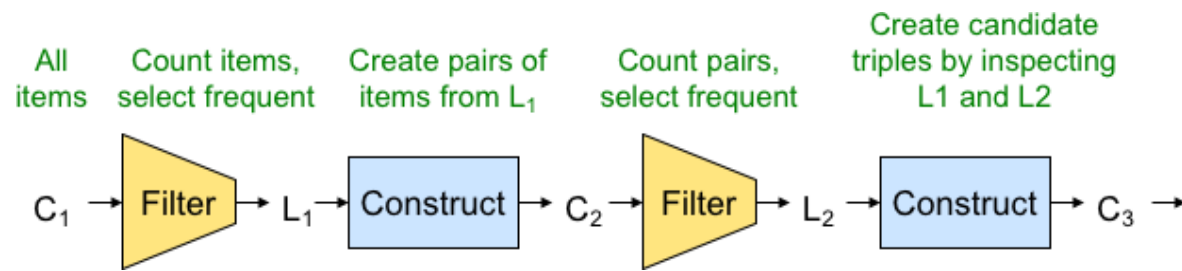
For each k , construct two sets of k -sets (sets of size k):

C_k = candidate k -sets (those sets that might be frequent)

L_k = the set of truly frequent k -sets

C_2, C_3, \dots are 'constructed' implicitly from L_1, L_2, \dots

A-Priori algorithm: $k > 2$



For C_{k+1} , all subsets of size k need to be in L_k

Construction: take a set from L_k , add a new frequent item (from L_1), then check if all subsets are in L_k

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. We get: $L_1 = \{ b, c, j, m \}$
- 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 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 for All Frequent Itemsets

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

Exercise ●

You are given a list of transactions from a small grocery store.

```
transactions = [  
    ['milk', 'bread', 'nuts', 'apple'],  
    ['milk', 'bread', 'nuts'],  
    ['milk', 'bread'],  
    ['milk', 'apple'],  
    ['bread', 'apple'],  
]
```

Write a Python program that implements the Apriori algorithm, limited to finding frequent 1-itemsets and 2-itemsets from the list of transactions.

Use only basic Python data types (such as lists, tuples, and sets). You can use `itertools.combinations()` to help with 2-itemset generation

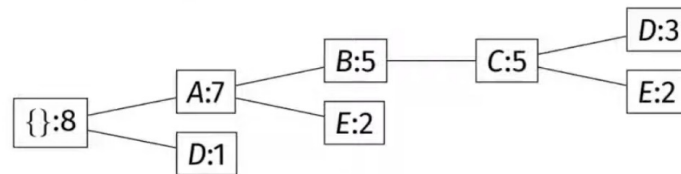
Recap

- ▶ **Monotonicity of Frequent Itemsets:** An important property of itemsets is that if a set of items is frequent, then so are all its subsets. We exploit this property to eliminate the need to count certain itemsets by using its contrapositive: if an itemset is not frequent, then neither are its supersets.
- ▶ **The A-Priori Algorithm for Pairs:** We can find all frequent pairs by making two passes over the baskets. On the first pass, we count the items themselves, and then determine which items are frequent. On the second pass, we count only the pairs of items both of which are found frequent on the first pass. Monotonicity justifies our ignoring other pairs.
- ▶ **Finding Larger Frequent Itemsets:** A-Priori and many other algorithms allow us to find frequent itemsets larger than pairs, if we make one pass over the baskets for each size itemset, up to some limit. To find the frequent itemsets of size k , monotonicity lets us restrict our attention to only those itemsets such that all their subsets of size $k-1$ have already been found frequent.

Frequent Pattern Mining

- ▶ Efficient algorithm for mining frequent itemsets without candidate generation
- ▶ Proposed as an improvement over the Apriori algorithm
- ▶ Uses a compact data structure called the FP-tree
 - ▶ Built by inserting transactions one by one
- ▶ The tree can often be held in main memory even when the database cannot
- ▶ For each frequent item, a conditional pattern base is created
- ▶ Useful for large datasets where Apriori becomes inefficient

E.g., transactions $\{A, B, C, D\} \times 3$, $\{A, B, C, E\} \times 2$, $\{A, E\} \times 2$, and $\{D\} \times 1$:



The FP-tree is a *compressed summary* of the transaction database.

Construction of an FP-tree

- ▶ First database scan:
 - ▶ Determine frequent 1-itemsets (based on a given minsupp parameter)
 - ▶ Sort items by descending frequency (to get smaller trees later)
- ▶ Second database scan:
 - ▶ Read one transaction at a time
 - ▶ Prune non-frequent items from the transaction
 - ▶ Sort items based on support counts in decreasing order
 - ▶ Insert into the FP-tree

<https://medium.com/@karna.sujan52/frequent-pattern-mining-fp-growth-numerical-bfcc6c7cd1f5>

Advantages of an FP-tree

- ▶ Contains all information necessary for frequent pattern mining
- ▶ Does not include irrelevant information (non frequent items)
- ▶ Smaller than original database
- ▶ Best case: tree only has a single branch of nodes
- ▶ Spark.mlib implements a parallel version of the FP-growth algorithm called PFP (<https://spark.apache.org/docs/3.5.4/ml-frequent-pattern-mining.html>)

Exercise [optional] ●

Apply the FP-Growth algorithm in PySpark to extract frequent itemsets and generate association rules from the transactions of the Apriori exercise.

Display frequent itemsets and association rules.

