6a. Data Mining - association rules

Objective of association rules: **extraction** of frequent correlations or pattern from a transactional database.

Association rules of transactions

A transaction is a **set of items** where **items** in said transaction are **not ordered.**
Given $A, B \Rightarrow C$, A and B are the items in the rule body while C is the item in the rule head. $\Rightarrow$ is a relation of *co-occurrence* and not *causality* thus it should be read as "whenever there is A and B, there's also C".

Definitions

Itemset: a set including one or more items. E.g.: {Beer,diapers}
k-itemset: an itemset containing **k** items
Support count(#): frequency of occurence of an itemset in the database
Frequent itemset: itemset whose *support* is $\geq$ than *minsup* threshold

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

#{Beer,Diapers}=2 since it's contained in TID=3 and TID=4. It's sup is 2/5

Given $A \Rightarrow B$, we have that
Support: is the fraction of transactions containing both A and B and is computed as follows $\frac{\#\{A,B\}}{|T|}$ where $|T|$ is the cardinality of the transactional database
Confidence: frequency of B in transactions containing A. It represents the *strength* of $\Rightarrow$ and is computed as follows: $\frac{sup(A,B)}{sup(B)}$

Association rule mining: Extraction of rules satisfying the *minsup* and *minconf* thresholds. The result is **complete**(*all* rules satisfying both constraints) and **correct**(only the correct rules satisfies the constraints)

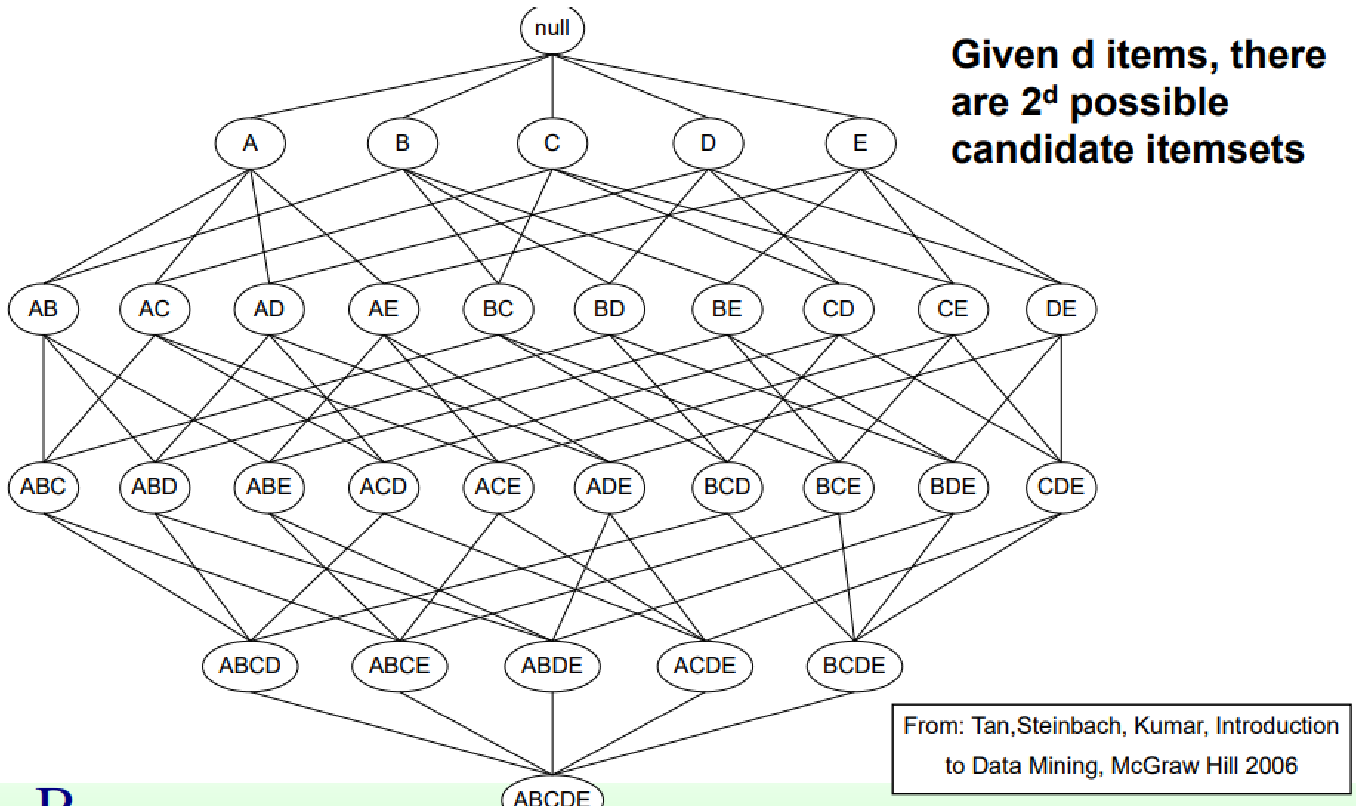
Approaches for association rule mining

Bruteforce but it's computationally unfeasible: one must enumerate all possible permutations and for each compute support and confidence.
Thus a first concrete approach might be:

1. Generation of *frequent*(sup$\geq$ minsup) itemsets
2. Generation of rules from frequent itemsets

Extraction of frequent itemsets

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



The image right above is the lattice of all possible itemsets.
The number of candidates is exponential to the number of items.

Bruteforce

Each itemset in the lattice is a ***candidate*** frequent itemset.
It's a rather simple approach: just scan the whole database to compute support for each candidate.
Complexity of $O(|T|2^d w)$ with w being the transaction length

Improving the efficiency

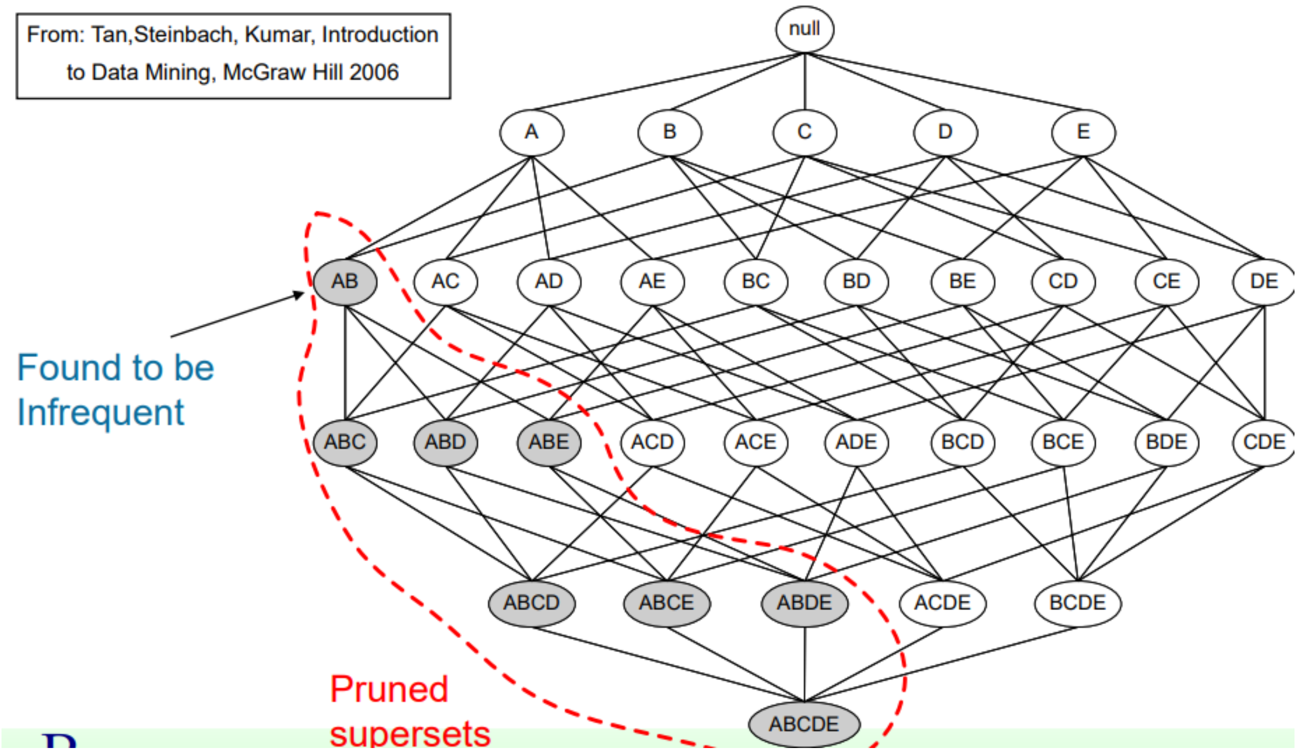
It can be achieved by:

- reducing the **number of candidates**(done by pruning the search space)
- reducing the **number of transactions**(done by pruning the transactions)
- reducing the **number of comparison**

Apriori algorithm

Based on the **Apriori principle** that states:
if an itemset is frequent, then all of its subsets must also be frequent
It holds due to the antimonotone property of the support measure:

- Given two arbitrary itemsets A and B if $A \subseteq B$ then sup(A) ≥ sup(B)
  This allows the reduction of the number of candidates as depicted in the image below



Example of the Apriori principle applied on the lattice

Algorithm

It's level-based approach which means that at each iteration extracts itemsets of a given length k.
At each step, there are two main steps:

1. **Candidate generation**
   - **Join step:** generate candidates of length k+1 by joining frequent itemsets of length k
   - **Prune step:** application of Apriori principle by pruning length k+1 candidate itemsets that contain at least one k-itemset that is not frequent

2. **Frequent itemset generation:** Scan DB to count support for k+1 candidates and prune candidates below minsup

Pseudo-code

C_k: Candidate itemset of size k
L_k : frequent itemset of size k

L_1 = {frequent items};
for (k = 1; L_k !=$\emptyset$; k++) **do**
  begin
   C_{k+1} = candidates generated from L_k;
   for each transaction t in database do
      increment the count of all candidates in C_{k+1}
      that are contained in t
   L_{k+1} = candidates in C_{k+1} satisfying *minsup*
  end
return $\cup_k L_k$;

where $C_1 = L_1$ and the "candidates generated from L_k" is defined as follows:

Sort L_k candidates in lexicographical order

For each candidate of length k

- Self-join with each candidate sharing same L_{k-1} prefix
- Prune candidates by applying Apriori principle

Example: given L_3={*abc, abd, acd, ace, bcd*}

- Self-join
  - *abcd* from *abc* and *abd*
  - *acde* from *acd* and *ace*
- Prune by applying Apriori principle
  - *acde* is removed because *ade, cde* are not in L_3
  - C_4={*abcd*}

Performance issues

The candidate sets generated may be huge.
Also the algorithm performs multiple database scans

Factors affecting peformance

- **Min support threshold:** a lower number increase the number of frequent itemset
- **Dimensionality(number of items):** more space is needed to store support count of each item
- **Size of database:** since the algorithm scans db multiple times, performance may decrease
- **Average transaction width**

FP-Growth algorithm

Based on the **FP-Tree**, a *main memory* compressed representation of the database.
It uses a *Divide-et-impera* recursive approach.
It's composed of two parts:

- one where a FP-Tree is consturcted

- one where frequent itemsets are extracted.

FP-Tree construction

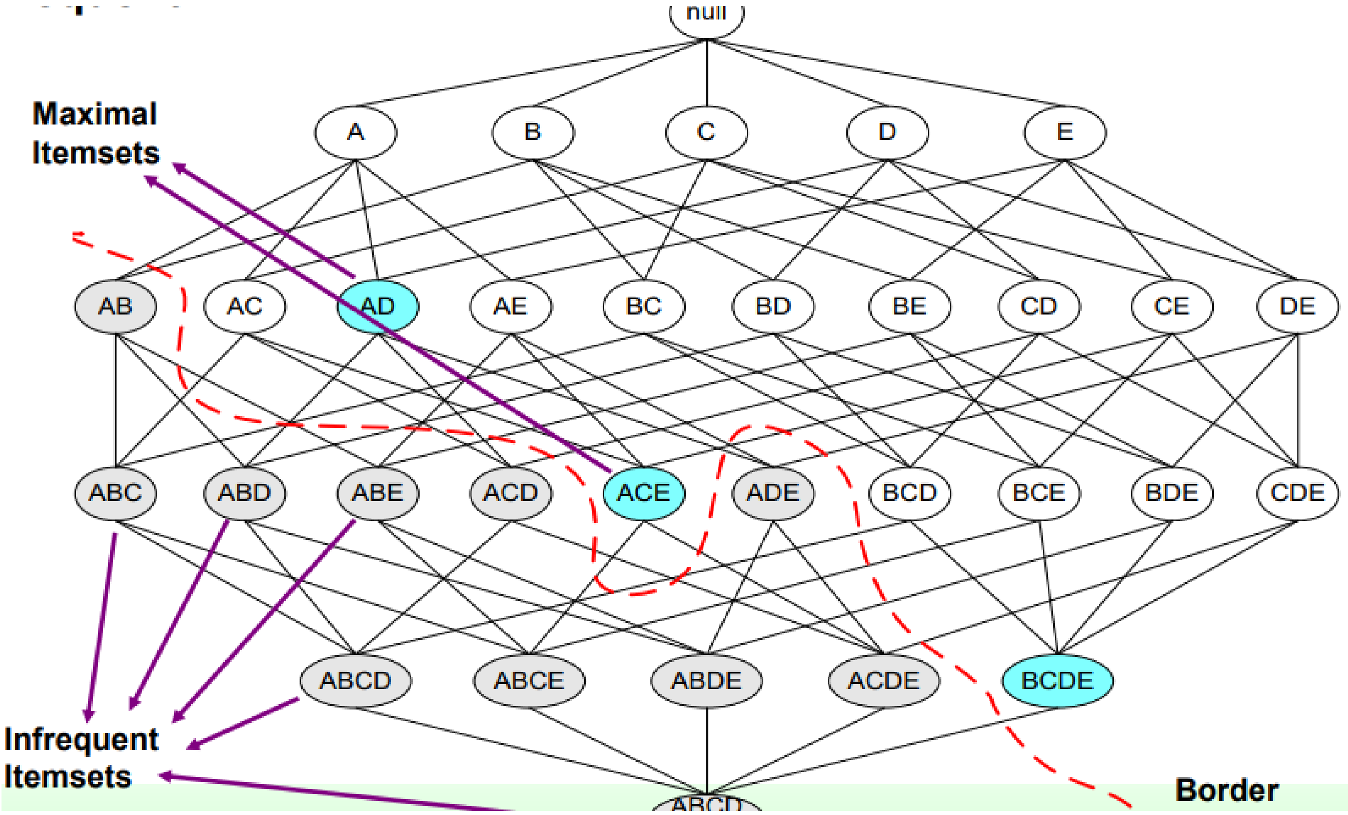
1. Count item support and prune items below minsup threshold
2. Build Header Table by sorting items in decreasing support order
   - Header table's entries are **items**, not **transactions**!
3. For each transaction *t* in the DB:
   1. Order transaction *t*'s items in decreasing support order.
   2. Insert transaction *t* in **FP-Tree**
      - Use existing path for common prefix
      - Create new branch when path becomes different
        - e.g. *A,B,C* and *A,B,E* have in common *A,B* while *A,B,C* and *B,C* dont have any prefix in common

Algorithm

1. Scan **Header Table** from lowest support item up
2. For each item *i* in **Header Table** extract frequent itemsets including item *i* preceding it in header table.
   1. Build **Conditional Pattern Base** for item *i* - **i-CPB**
      - This is done by selecting prefix-paths of item *i* from FP-Tree
   2. Recursive invocation of **FP-Growth** on *i-CPB*

Maximal Frequent Itemset

An itemset is said to be maximal if **none** of its **immediate supersets** are frequent.



In this example, **AD** is maximal as its immediate supersets **ABD**,**ACD** and **ADE** are infrequent. Same thing goes for **AC**: it's not frequent as one of its immediate, **ACE**, it's frequent(on top of being maximal).

Closed Itemset

An itemset is closed if none of its immediate supersets has the same support as the itemset

Correlation or Lift

$$r: A \Rightarrow B$$

$$\text{Correlation} = \frac{P(A,B)}{P(A)P(B)} = \frac{\text{conf(r)}}{\text{sup(B)}}$$

- Statistical independence
  - Correlation = 1
- Positive correlation
  - Correlation > 1
- Negative correlation
  - Correlation < 1

Generalizing association rules

Sometimes, an association rule may be too specific about one of its itemsets.
By generalizing one of the itemsets, new interesting properties may be discovered.
E.g *__user__: John, **time**: 6.05 p.m., **service**: Weather *may have a very low support but by generalizing* ***user*** *with* ***user:*** employee* we might have a higher support than before.

If a rule has only generalized itemsets, then we have a **high level rule** opposed to **low level rules** characterized by having only not generalized itemsets.

Extraction of association rules

Done by generating all possible binary partitioning of each frequent itemset, possibly by enforcing a confidence threshold.