part1 Frequent Itemset Mining & Association Rules

关联规则挖掘

Goal： 挖掘频繁被消费者同时购买的商品集合

Input:

Basket	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

Goal : 发现商品basket中的frequent items

定义 Support of itme I (支持度)：几个关联的数据在数据集中出现的次数占总数据集的比重

我们将support超过了阈值s的itemsete称之为frequent itemsets

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

Support of
{Beer, Bread} = 2

例如

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

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}

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

Association Rules

$\{i_1, i_2, ..., i_k\} \to j$ 表示如果某个basket中已经包括了所有的i_1到i_k个item，那么这个basket很有可能包括i_j

$Confidence\ of\ association\ rules$表示给出basket $I = \{i_1, ..., i_k\}$时，含有j的概率

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

我们并不是关注所有的高概率关联规则，比如$X \to milk$可能发生的概率很高，但是这仅仅是因为milk会被经常购买，即milk和X是独立的。因此我们需要定义$Interest\ of\ an\ association\ rule I \to j$ 来平衡item本身被购买的概率和他成为某个basket的frequent item的概率,即找到那些“高价值的Association Rules”。

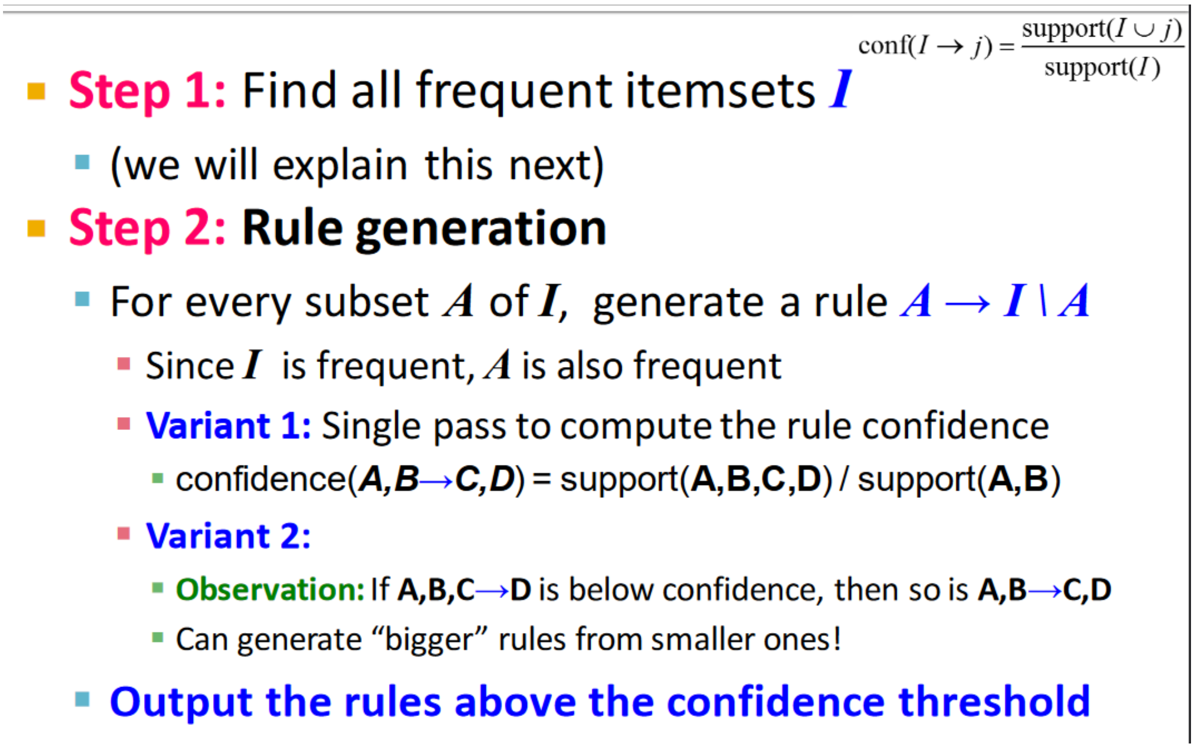
$$Interest(I \to j) = |conf(I \to j) - Pr[j]|$$

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} →c**
  - **Support** = 2
  - **Confidence** = 2/4 = 0.5
  - **Interest** = |0.5 – 5/8| = 1/8
    - Item ***c*** appears in 5/8 of the baskets
    - The rule is not very interesting!

Association Rule Mining

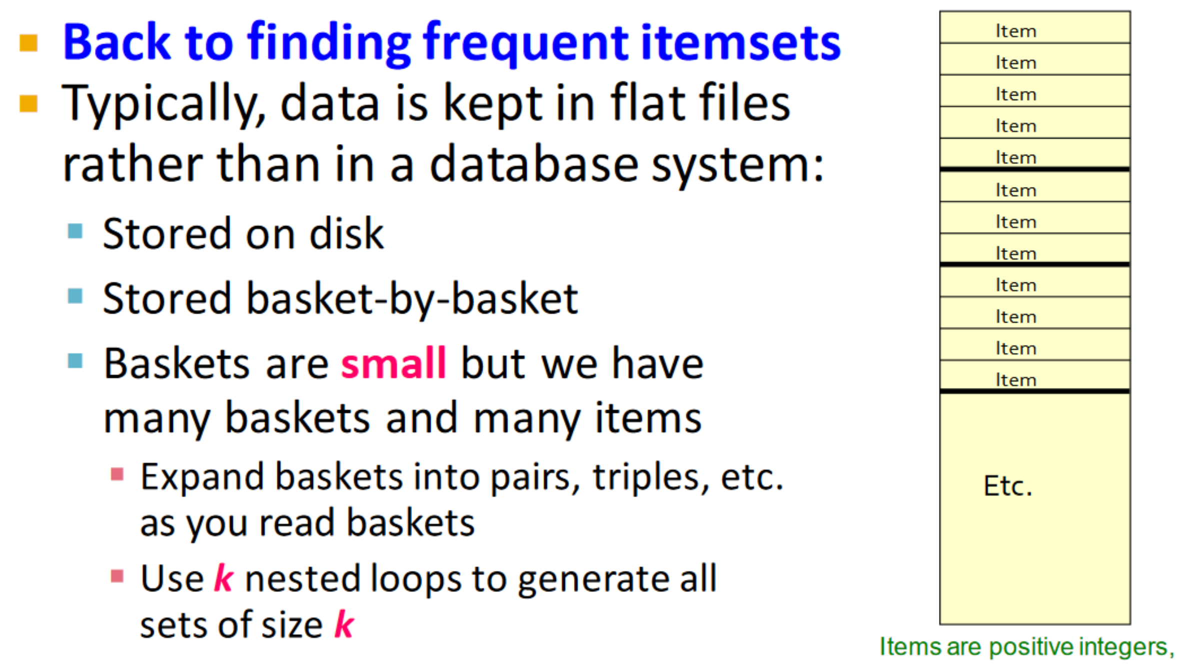
我们现在的目标为找到$support >= s$且$confidence >= c$的关联规则



第一步首先找到所有的频繁项集。第二步生成对应的关联规则。

Finding Frequent Itemsets

如果我们希望挖掘frequent itemsets，那么我们就需要去对每一个item进行count，那么我们就需要对于整个数据集进行遍历。



通常来说我们的数据集会以basket-by-basket的方式存放在disk中，那么对于这些数据而言，花费时间最多的步骤就变成了I/Os的时间。在实践中，我们以某种顺序依次读取basket中的数据，称之为passes。我们对于某种关联挖掘算法的代价的定义按照passes的数目来衡量。

- 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

因为main-memory的限制，所以我们不能以简单遍历的方式对于频繁项集进行挖掘。

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

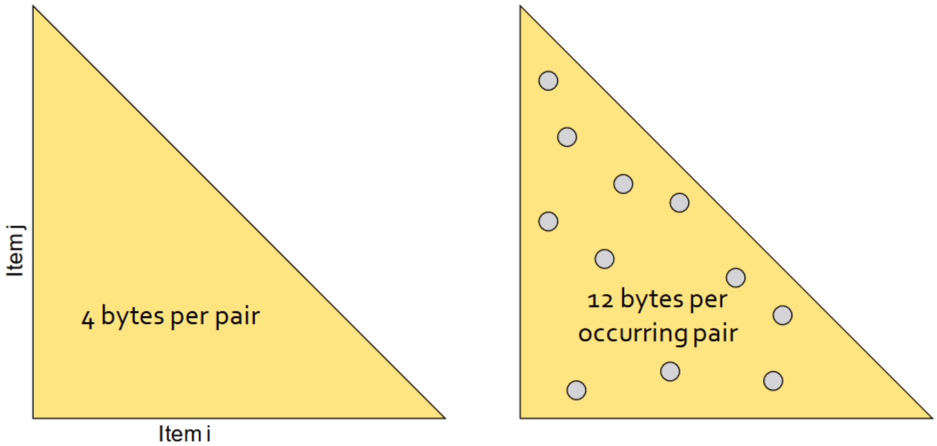
假设我们对于某个basket寻找他们的frequent pairs，那么我们需要首先枚举所有潜在的frequent pairs可能。通过两层嵌套的循环，我们可以遍历basket来枚举所有可能的pairs。但是在大数据量的情况下，我们需要（$\#items$）^{2}的内存容量来保存循环的结果，而这往往会超出内存限制。

- **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

对于上述情况，可以给出两种解决方法。

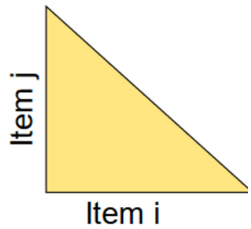
1. 使用矩阵来保存所有可能出现的pairs数目
2. 使用三元元组的形式保存出现的pairs结果，即（item i, item j, 共现次数）



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:
    - $\{1,2\}, \{1,3\},\ldots, \{1,n\}, \{2,3\}, \{2,4\},\ldots,\{2,n\}, \{3,4\},\ldots$
  - Pair $\{i, j\}$ is at position: $[n(n-1) - (n-i)(n-i+1)]/2 + (j-i)$
  - 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