universidade de aveiro

Mining Large Scale Datasets

Frequent Itemsets and Association Rules

(Adapted from CS246@Starford.edu; http://www.mmds.org)

Sérgio Matos - aleixomatos@ua.pt

Motivation

- Classical data mining problem
- Finding sets of items that appear in (or are related to) many of the same baskets
- Frequent itemsets $\rightarrow$ association rules

Market-Basket Model

- Goal: Identify items that are bought together by many customers
- Approach: Process sales data to find dependencies among items

Classical example

If someone buys diaper and milk, then he/she is likely to buy beer

- Place beer next to diapers, maybe place chips nearby as well :)
- Lower price on diapers, increase price on beer

Market-Basket Model

- Many-many relationships between <u>items</u> and <u>baskets</u>
- A large set of items

  e.g. all items sold by a e-commerce site, $\sim$100K
- A very large number of baskets

  each customer transaction at the site, many millions
- Each basket consists of a small subset of items

  items bought by a costumer on one day
- Discover association rules:

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

Market-Basket Model

- Many-many relationships between items and baskets

- Items and baskets are abstract:

  products/shopping basket

  words/documents

  basepairs/genes

  drugs/patients

- Search for connections among “items”, not “baskets”

Example applications

- Related words: items are words, baskets are documents
- Plagiarism: items are documents, baskets are sentences
- Biomarkers: items are diseases and biomarkers (genes/blood proteins), baskets are sets of data about each patient
- Side-effects: items are drugs and side-effects, baskets are patients

Note: baskets should contain small number of items; items can be in a large number of baskets

Outline

- Frequent itemsets
- Association rules
  - Support, Confidence, Interest

- Algorithms for finding frequent itemsets
  - Finding frequent pairs
  - A-Priori algorithm
  - PCY algorithm
  - PCY extensions
  - Frequent itemsets in ≤ 2 passes

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**

Frequent itemsets: example

Support threshold = 3 baskets

B1: {beer, coke, milk}
B2: {juice, milk, pepsi}
B3: {beer, milk}
B4: {coke, juice}
B5: {beer, milk, pepsi}
B6: {beer, coke, juice, milk}
B7: {beer, coke, juice}
B8: {beer, coke}

Frequent itemsets: example

Support threshold = 3 baskets

B1: {beer, coke, milk}
B2: {juice, milk, pepsi}
B3: {beer, milk}
B4: {coke, juice}
B5: {beer, milk, pepsi}
B6: {beer, coke, juice, milk}
B7: {beer, coke, juice}
B8: {beer, coke}

Frequent itemsets: {beer}, {coke}, {juice}, {milk}, {beer, coke}, {beer, milk}, {coke, juice}

Association rules: Confidence

$I \to j$

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

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

$$confidence(I \to j) = support\Big(I \cup \{j\}\Big)/support(I)$$

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

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

Not all high-confidence rules are interesting

- Rule $X \rightarrow$ *milk* may have high confidence for many itemsets X, because milk is purchased very often (independent of X) and the confidence will be high

Association rules: Interest

$I \rightarrow j$

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

Interest of a rule $I \rightarrow j$ is given by the probability of j given I minus the probability of j

$$interest(I \rightarrow j) = p(j|I) - p(j)$$

$interest(I \rightarrow j) = confidence(I \rightarrow j) -$ baskets containing $j/$ baskets

high positive interest: presence of I indicates the presence of j
high negative interest: presence of I discourages the presence of 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

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

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.

Association rules: Standardised lift

$I \rightarrow j$

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

$$Std\ Lift\big(I \rightarrow j\big) = \frac{Lift\big(I \rightarrow j\big) - \frac{max\{P(I)+P(j)-1, 1/n\}}{P(I)P(j)}}{\frac{1}{max\{P(I), P(j)\}} - \frac{max\{P(I)+P(j)-1, 1/n\}}{P(I)P(j)}}$$

n is the number of baskets.
Standardised lift ranges from 0 to 1.

This facilitates setting a fixed threshold for selecting the rules.

Association rules: example

B1: {beer, coke, milk}
B2: {juice, milk, pepsi}
B3: {beer, milk}
B4: {coke, juice}
B5: {beer, milk, pepsi}
B6: {beer, coke, juice, milk}
B7: {beer, coke, juice}
B8: {beer, coke}

Association rule: {beer, milk} $\rightarrow$ *coke*

Association rules: example

B1: {beer, coke, milk}
B2: {juice, milk, pepsi}
B3: {beer, milk}
B4: {coke, juice}
B5: {beer, milk, pepsi}
B6: {beer, coke, juice, milk}
B7: {beer, coke, juice}
B8: {beer, coke}

Association rule: {beer, milk} $\rightarrow$ *coke*
Support = 2
Confidence = 2/4 = 0.5

Association rules: example

B1: {beer, coke, milk}
B2: {juice, milk, pepsi}
B3: {beer, milk}
B4: {coke, juice}
B5: {beer, milk, pepsi}
B6: {beer, coke, juice, milk}
B7: {beer, coke, juice}
B8: {beer, coke}

Association rule: {beer, milk} $\rightarrow$ *coke*
Support = 2
Confidence = 2/4 = 0.5
Interest = | 2/4 - 5/8 | = 1/8

Item *coke* appears in 5/8 of the baskets (independ. of other items)
Rule is not very interesting!

Mining association rules

Find all association rules with support $\geq s$ and confidence $\geq c$

- Find all itemsets I with support $\geq c \cdot s$
- For each j in I

  if $support\big(I - \{j\}\big) \geq s$

  then $I - \{j\} \rightarrow j$ is an acceptable association rule with

  $confidence = support(I)/support\big(I - \{j\}\big) \geq c$

Note: if $\{i_1, i_2, ..., i_k\} \rightarrow j$ has high support and confidence, then both $\{i_1, i_2, ..., i_k\}$ and $\{i_1, i_2, ..., i_k, j\}$ are frequent

$\rightarrow$**Hard part: Finding the frequent itemsets**

Mining association rules

- Step 1: Find all frequent itemsets I
  (next slides)

- Step 2: Rule generation
  - For every subset A of I, generate a rule $A \to I \setminus A$
    - Since I is frequent, A is also frequent

  - Calculate the confidence of the rules

    $confidence(\{k, l, m\} \to \{j\}) = \frac{support(\{j,k,l,m\})}{support(\{k,l,m\})}$

    Note:
    If $\{k, l, m\} \to j$ is below confidence, so is $\{k, l\} \to \{m, j\}$

- Output the rules above the confidence threshold

Mining association rules: Example

B1 = {b, c, m}	B2 = {j, m, p}
B3 = {b, c, m, n}	B4= {c, j}
B5 = {b, m, p}	B6 = {b, c, j, m}
B7 = {b, c, b}	B8 = {b, c}

- Support threshold s = 3, confidence c = 0.75

1) Frequent itemsets:
   {b,c} {b,m} {c,j} {c,m} {b,c,m}

2) Generate rules:

$b \rightarrow c$	$b \rightarrow m$	$m \rightarrow b$	$b, c \rightarrow m$	$b, m \rightarrow c$	$b \rightarrow c, m$
c=5/6	c=4/6	**c=4/5**	c=3/5	**c=3/4**	c=3/6

Compacting the output

We can post-process to reduce the number of rules and only output

- Maximal frequent itemsets: no immediate superset is frequent
  - Gives more pruning

OR

- Closed itemsets: no immediate superset has same count (>0)
  - Stores not only frequent information, but exact counts

Compacting the output: Example

	Support	Maximal(s=3)	Closed
A	4	No	No
B	5	No	Yes
C	3	No	No
AB	4	Yes	Yes
AC	2	No	No
BC	3	Yes	Yes
ABC	2	No	Yes

Frequent, but superset BC also frequent.

Superset BC has same count.

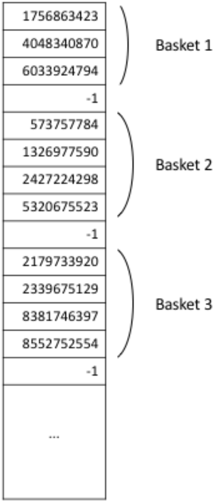
Frequent, and its only superset, ABC, not freq.

Its only superset, ABC, has smaller count.

Finding frequent itemsets

- Data stored on disk in flat files, rather than on a database system
- Organized basket-by-basket
- Baskets are small (few items) but there are many baskets and many different items
- Expand baskets into pairs, triples, etc. as you read baskets
- Use k nested loops to generate all sets of size k

Market-basket model: Flat file structure



- items are positive integers
- -1 is used to separate baskets

Finding frequent itemsets

- The true cost of mining disk-resident data is usually the number of disk I/O's
- In practice, algorithms for finding frequent itemsets read the data in passes – all baskets read in turn
- The basket file is read sequentially
- Running time is proportional to the number of passes made through the basket file times the size of the file
- Cost can be measured by the number of passes an algorithm makes over the data

Finding frequent itemsets

- For many frequent-itemset algorithms, main-memory is the critical resource
- As we read baskets, we need to keep counts, e.g. the occurrences of pairs of items
- The number of different things we can count is limited by main memory
- Swapping counts in/out to disk is not feasible

Finding frequent itemsets

- The hardest problem often turns out to be finding the frequent pairs of items $\{i_1, i_2\}$

- We can usually count all the singleton sets in main memory
- Support threshold is set high enough so that we don't end up with too many frequent itemsets
  $\hookrightarrow$ Frequent pairs are common, but frequent triples are rare
  $\hookrightarrow$ Probability of being frequent drops exponentially with size

Finding frequent itemsets

Naïve approach to finding frequent pairs

- Read file once, count occurrences of each pair in main memory

  For each basket if the file, use two nested loops to generate all item pairs and add 1 to each pair's count

- Fails if (#items)^2 exceeds main memory

  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 needed

Finding frequent itemsets: Counting Pairs in Memory

- **Approach 1:** Count all pairs using a triangular 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 each pair with count >0, plus some additional overhead for the hashtable

  Note: It is more space-efficient to represent items by consecutive integers from 1 to n. If required, we use a hash table to translate items to integers.

Finding frequent itemsets: Counting Pairs in Memory

Triangular matrix

- n = total number items
- use a 2-dimensional array
- order the pairs and only use the entries [i,j] for i <j

- half of the 2-d array contains zeroes, but consumes memory!

Finding frequent itemsets: Counting Pairs in Memory

Triangular matrix

- n = total number items
- Use a 1-d array counts[i, j] for all i,j, i <j
- counts[k] contains the count for pair {i, j}, with *

$$k = (i-1)(n-i/2)+j-i$$

- Pair counts stored in lexicographical order:
  {**1**,2}, {**1**,3}, ..., {**1**,n}, {**2**,3}, {**2**,4}, ..., {**2**,n}, {**3**,4}, ..., {**3**,n}, ..., {**n-1**,n}

*Exercise

Finding frequent itemsets: Counting Pairs in Memory

- Triangular matrix

  Total number of pairs n(n −1)/2

  4 bytes per count; total bytes $\approx 2n^2$

- Triples

  12 bytes per occurring pair (pairs with count >0)

Triples are preferable if $< 1/3$ of possible pairs actually occur

Finding frequent itemsets: Counting Pairs in Memory

- Triangular matrix

  Total number of pairs n(n −1)/2

  4 bytes per count; total bytes $\approx 2n^2$

- Triples

  12 bytes per occurring pair (pairs with count >0)

Triples are preferable if $<$ 1/3 of possible pairs actually occur

→What if the pairs do not fit into memory?

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

- Takes advantage of the monotonicity property to reduce the number of pairs that must be counted
- Requires two passes over data to find frequent pairs

  First pass: count occurrences of single items

  Second pass: count occurrences of pairs of frequent items
- In general, uses k passes to find frequent sets of size k

A-Priori algorithm

- **First pass**

  Initiate to 0 an array of counts with size n (number of items)

  For each basket read, loop through items and

  if necessary, map item name to integer in range $1..n$

  add 1 to the corresponding value in the array of counts

  At the end, create a frequent items table

- **Second pass**

  For each basket read,

  Generate all pairs of items using a double loop

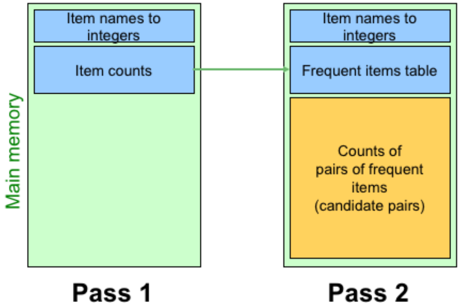
  For each pair, add 1 to its count

  only if both items are frequent

A-Priori algorithm

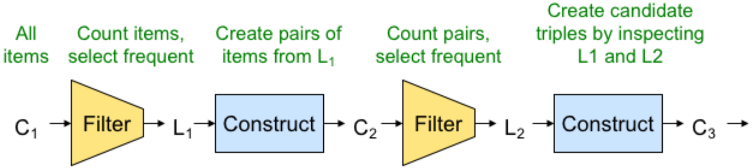
```
%first pass
for (each basket)
    for (each item i in basket)
        item_counts[i] += 1

%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: memory use



- Frequent items table can be a set (item is or not frequent)
- Can also use triangular matrix to store pair counts
  - Frequent items table maps from range 1..n to 1..m, or 0 if item is not frequent, with m = number of frequent items

A-Priori algorithm: k >2



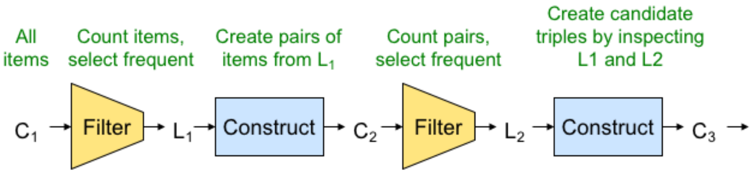
- For each k, construct two sets of k-sets (sets of size k):

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

  L_k = the set of truly frequent k-sets

- $C_2, C_3, ...$ are 'constructed' implicitly from $L_1, L_2, ...$

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

PCY algorithm

Observation

In pass 1 of A-Priori, only individual item counts are stored

$\hookrightarrow$ Use the idle memory to reduce memory required in pass 2

The PCY Algorithm uses an array of integers that generalizes the idea of a Bloom filter

PCY algorithm

- Pass 1 of PCY

  In addition to item counts, keep an array of bucket counts

  Hash each pair of items in the basket into a bucket

  Keep count of how many pairs of items hash into each bucket
- Buckets with count greater than the support threshold s are called *frequent buckets*
- Any bucket containing at least one frequent pair is surely a frequent bucket
- Not all frequent buckets contain frequent pairs, but...

  $\hookrightarrow$ **Any pair that hashes to a non-frequent bucket can not be a frequent pair**

PCY algorithm

```
%first pass
for (each basket)
    for (each item i in basket)
        increment item count

    for (each pair of items)
        hash the pair to a bucket
        increment bucket count
```

Note: we are not keeping counts for each individual pair.

Note: can stop incrementing when count reaches s.
Why? What is gained?

PCY algorithm

- Between passes

  Replace bucket counts by a bit-vector:

  1 if the bucket is frequent (count $\geq s$); 0 if not

  4-byte integers are replaced by bits; takes 1/32 of memory

  Also create a frequent items table, as in A-Priori

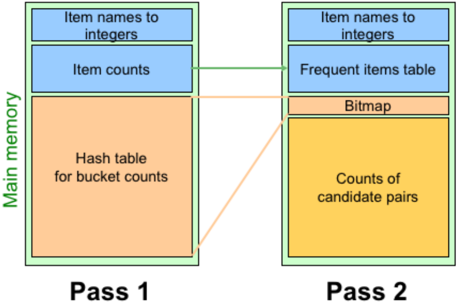
- Pass 2 of PCY

  Count only pairs $\{i, j\}$ that meet the conditions for being a candidate pair:

  Both i and j are frequent items

  The pair $\{i, j\}$ hashes to a frequent bucket (bit set to 1)

PCY algorithm: memory use

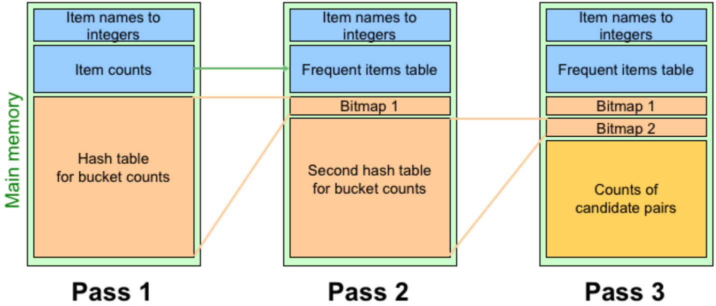


- Why should hash table take most of the available memory?
- Why is hash table bigger than bitmap?
- In PCY we cannot use a triangular matrix in pass 2. Why?

PCY algorithm: buckets

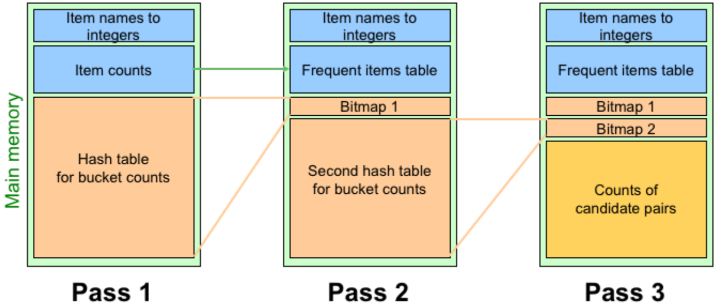
- If a frequent pair hashes to bucket b, bucket b will be frequent
  - Other pairs that hash to this bucket will be counted in the second pass, unless one of its items is not frequent
- A bucket can be frequent event if all pairs that hash into it are not frequent
  - Again, these pairs cannot be eliminated from the second pass
- Best case occurs when the final bucket count is less than the support threshold s
  - No need to count any of the pairs that hash into such bucket

Multistage algorithm



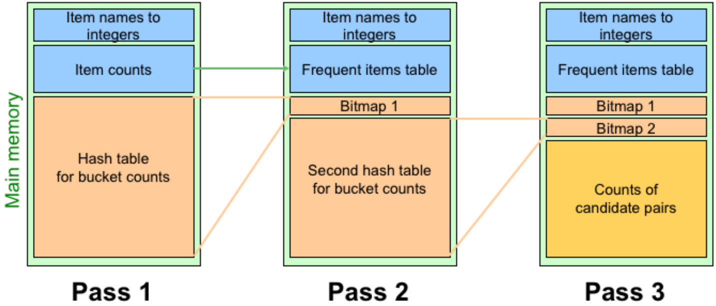
- In Pass 2, only rehash pairs that qualify for Pass 2 of PCY
  - Both items are frequent
  - Pair hashes to a frequent bucket
- Fewer pairs will contribute to bucket counts in Pass 2, so fewer false positives (infrequent pairs hashed to frequent buckets)

Multistage algorithm



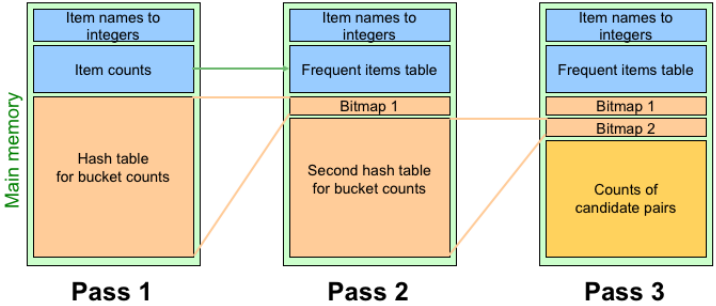
- Uses two different and independent hash functions
- What are the conditions for being a candidate pair?

Multistage algorithm



- Uses two different and independent hash functions
- What are the conditions for being a candidate pair?
  - Both items are frequent
  - Pair hashes to frequent bucket (bit set to 1) in bitmap 1
  - Pair hashes to frequent bucket (bit set to 1) in bitmap 2

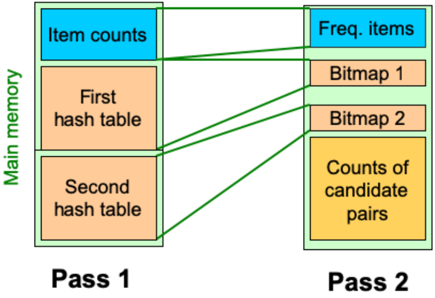
Multistage algorithm



- Uses two different and independent hash functions
- What are the conditions for being a candidate pair?
  - Both items are frequent

  ?? Pair hashes to frequent bucket (bit set to 1) in bitmap 1
  - Pair hashes to frequent bucket (bit set to 1) in bitmap 2

Multihash



- Each hash table has half the number of buckets
- Must ensure that the count in most buckets does not reach s
- Pair is candidate if both items are frequent and if pair hashes to frequent buckets according to both functions
- j hash functions $\approx$ benefits of j stages of multistage

Frequent Itemsets in ≤ 2 Passes

Previous algorithms (A-Priori, PCY, etc.) need k passes over data to find frequent itemsets of size k

Can we use fewer passes?

Some algorithms use 2 or fewer passes for all sizes, at the expense of possibly missing some frequent itemsets

- Random sampling
- SON (Savasere, Omiecinski, and Navathe)
- Toivonen

[Note: we don't always need to find all frequent itemsets. Why?]

Random sampling

- Take a random sample of all baskets, keep sample in memory
- Run A-Priori (or one of its improvements) in main memory
- Run the algorithm over the sample for each itemset size, until no frequent items are found

  Note: no disk I/O required since sample is in memory

- Support threshold is reduced proportionally to match the sample size

  $\hookrightarrow$ if the sample is a fraction p of the baskets, use $p \cdot s$ as the support threshold instead of s

- May generate false positives and false negatives

  $\hookrightarrow$ How to deal with this?

Random sampling

- To avoid false positives:

  Do a second pass over the full data to verify that the candidate pairs found from the sample are truly frequent in the entire data set

  $\hookrightarrow$ Can we do this in one pass?

Random sampling

- To avoid false positives:

  Do a second pass over the full data to verify that the candidate pairs found from the sample are truly frequent in the entire dataset

  $\hookrightarrow$ We know which pairs to count

  $\hookrightarrow$ No need to keep the sample in memory

Random sampling

- What about false negatives?

  Frequent sets that were missed in the sample will not be found in this second pass

  We can use a smaller threshold for the sample to lower the false negatives, for example $0.8 \cdot p \cdot s$

  Requires more memory

SON algorithm

- Avoids both false negatives and false positives
- First pass

  Repeatedly read small subsets of the baskets into main memory and run an in-memory algorithm to find all frequent itemsets

  Not sampling – entire data processed in memory-sized chunks

  Itemsets frequent in at least one subset become candidates
- Second pass

  Count all the candidate itemsets and determine which are frequent in the entire dataset

Key "monotonicity" idea

$\hookrightarrow$ An itemset cannot be frequent in the entire set of baskets unless it is frequent in at least one subset

SON algorithm through MapReduce

Each phase of the SON algorithm can be implemented as a MapReduce operation

- First phase

  Chunks can be processed in parallel, to find frequent itemsets

  The frequent itemsets found for each chunk are combined to form the candidates

- Second phase

  Candidates can be distributed, with each processor counting the support of each candidate in a subset of the baskets

  Supports for each candidate are summed to get the total support in the whole dataset

SON algorithm through MapReduce

- First phase

  Map: Takes a chunk and applies in-memory algorithm; produces a set of key-value pairs $(F, 1)$ where F is a frequent itemset from the chunk

  Reduce: Each task is assigned a set of keys (itemsets) and produces those itemsets that appear one or more times

- Second phase

  Map: Takes all candidate itemsets (from first Reduce) and a chunk of the data; counts occurrences of each candidate in the chunk; produces a set of key-value pairs (C, v) where C is one of the candidates and v is its support among the baskets in the chunk

  Reduce: Takes a set of keys (itemsets) and sums the values to obtain the total support for each; selects itemsets with support $> s$

Toivonen's algorithm

- Pass 1:

  Start with a random sample, but lower the threshold slightly for the sample

  For example, use threshold $0.8 \cdot p \cdot s$ instead of $p \cdot s$, where p is the fraction of baskets in the sample

  Find frequent itemsets in the sample

  Add itemsets that are in the negative border of the frequent itemsets

Negative border

$\hookrightarrow$ An itemset is in the negative border if it is not frequent in the sample, but all its immediate subsets are

$\hookrightarrow$ Immediate subsets are obtained by deleting one element

$\{A, B, C, D\} \rightarrow \{A, B, C\}, \{A, B, D\}, \{A, C, D\}, \{B, C, D\}$

Toivonen's algorithm

- Pass 1:

  Start with a random sample, but lower the threshold slightly for the sample

  For example, use threshold $0.8 \cdot p \cdot s$ instead of $p \cdot s$, where p is the fraction of baskets in the sample

  Find frequent itemsets in the sample

  Add itemsets that are in the negative border of the frequent itemsets

- Pass 2:

  Count all candidate frequent itemsets from the first pass, and also sets in their negative border

Toivonen's algorithm

Theorem

If there is an itemset S that is frequent in the full data, but not frequent in any sample, then the negative border must contain at least one itemset that is frequent in the whole.

- If no itemset from the negative border is frequent in the whole dataset, we found all the frequent itemsets
- If an itemset in the negative border is frequent in the whole dataset

    Must start over again with another sample

    Choose the support threshold so the probability of failure is low, while the number of itemsets checked on the second pass fits in main-memory

Wrap up: 2 full passes

- A-Priori

  Uses monotonicity property to define candidate pairs
- PCY algorithm

  Hash pairs and create a bit array of frequent buckets
- Multistage algorithm

  Extension of PCY with multiple hash functions

Wrap up: < 2 full passes

- Random sampling

  Run in-memory algorithm on a sample, counting itemsets of all sizes

  Do a second pass through entire data to eliminate false positives
- SON algorithm

  Process entire data in memory-sized chunks, counting itemsets of all sizes

  Count all candidates in second pass through entire data
- Toivonen's algorithm

  Count itemsets in negative border

  Must repeat with different sample if itemsets in negative border are frequent in the full dataset