CSCI 4022 Fall 2021
A Priori Algorithm

HW 4 due Friday (not today)

p ... q

Fact: if $\overbrace{\text{an itemset } I \text{ is frequent,}}^{p}$ then $\overbrace{\text{so is every subset of } I}^{q}$.

↳ ∃ at least s baskets holding every element of I.

Proof:

Suppose p and $\neg q$

" I is frequent but there exists some subset $J \subseteq I$ such that J is not frequent

Goal: show contradiction.

CSCI 4022 Fall 2021
A Priori Algorithm

Fact: if an itemset I is frequent, then so is every subset of I.

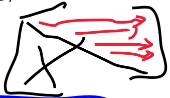
Proof: By *contradiction*

- Suppose not, or there exists frequent itemset I with threshold s and itemset J a subset of I that is not frequent.
- Then the support of J is less than s, or fewer than s baskets contain J.
- But any basket that contains I contains J, since J is a subset. So $support(J) > support(I) > s$...
- which is a contradiction!

Market Basket: Storing Counts

Preliminary step of creating a data processing/hash table for string-to-item (string-to-int) lookups. Then we store counts!

- The **triangular array** function:

C_{ij} =  stores count (i,j)

$$a[k] = \underbrace{(i)\left(n - \frac{i+1}{2}\right)}_{\text{first } i \text{ rows}} + \boxed{j - i - 1}$$

how far into row j

will (0-indexed) store item counts for the pair i, j, where $1 \leq i < j \leq n$.

- Alternatively, store counts as a list of triples $[i, j, c]$ where c is the count of $\{i, j\}$, $j > i$. Upside here: no saving "0" when i and j don't ever overlap. This also works for larger sets: $[i, j, k, c]$ could count frequent triples, and so forth.

Each of these are efforts to save on the *space* requirements for frequent item pair calculations.

Monotonicity

Fact: if an itemset I is frequent, then so is eery subset of I.

Definition: Given a support threshold s, a frequent itemset I is *maximal* if no superset of I is also frequent.

So we if we list only *maximal* itemsets, we'll know...

1. All subsets of a maximal itemset must also be frequent
2. No set that is *not* a subset of some maximal itemset can possible be frequent

In other words, the set of maximal frequent itemsets is the most concise - or minimal - way to represent all frequent items.

Maximal Sets

Example: We can't escape Walmart. For $s = 3$, what itemsets are maximal?

$B_1 =$	{m,c,b}	$B_2 =$	{m,p,b}	$B_3 =$	{m,p,j}	$B_4 =$	{m,c,b,j}
$B_5 =$	{m,c,b,n}	$B_6 =$	{c,b,j}	$B_7 =$	{c,j}	$B_8 =$	{b,c}

Maximal Sets

Example: We can't escape Walmart. For $s = 3$, what itemsets are maximal?

$B_1 =$	{m,c,b}	$B_2 =$	{m,p,b}	$B_3 =$	{m,p,j}	$B_4 =$	{m,c,b,j}
$B_5 =$	{m,c,b,n}	$B_6 =$	{c,b,j}	$B_7 =$	{c,j}	$B_8 =$	{b,c}

Solution:

1. Item sets: {{b},{c},{j},{m}, {m,b},{b,c},{c,j}, {c,m}, and {m,c,b}}

no p!

no {c, j, x}

b/c there's no x : {c, x} & {j, x}

Maximal Sets

Example: We can't escape Walmart. For $s = 3$, what itemsets are maximal?

$B_1 =$	{m,c,b}	$B_2 =$	{m,p,b}	$B_3 =$	{m,p,j}	$B_4 =$	{m,c,b,j}
$B_5 =$	{m,c,b,n}	$B_6 =$	{c,b,j}	$B_7 =$	{c,j}	$B_8 =$	{b,c}

Solution:

1. Item sets: {{b},{c},{j},{m}, {m,b},{b,c},{c,j}, {c,m}, and {m,c,b}}
2. {m,c,b} and {c,j} are the maximal itemsets.

What about bigger counts?

We talked a lot about counting pairs, but what about triples, quadruples, or larger frequent itemsets?

Count high $\Longleftrightarrow$ proportion high

1. In practice, we pick support thresholds to be high enough that we do not have too many frequent itemsets - this makes it easier to have actionable information.
2. **Monotonicity** is important! If there is a frequent triple, it must contain 3 frequent pairs.
   - And then any frequent quadruple must contain 4 frequent triples and $\binom{4}{2} = 6$ frequent pairs.
   - ... and so on
   - As a result, we expect to find more frequent pairs than triples, more triples than quadruples, and so forth.

What about bigger counts?

We talked a lot about counting pairs, but what about triples, quadruples, or larger frequent itemsets?

1. In practice, we pick support thresholds to be high enough that we do not have too many frequent itemsets - this makes it easier to have actionable information.
2. **Monotonicity** is important! If there is a frequent triple, it must contain 3 frequent pairs.
   - And then any frequent quadruple must contain 4 frequent triples and $\binom{4}{2} = 6$ frequent pairs.
   - ... and so on
   - As a result, we expect to find more frequent pairs than triples, more triples than quadruples, and so forth.

Example: We don't have to search for, allocate memory for, or even consider the frequent itemset $\{m, c, b\}$ unless we've already observed **all** of $\{m, c\}$, $\{m, b\}$, and $\{c, b\}$

What about bigger counts?

Example: We don't have to search for, allocate memory for, or even consider the frequent itemset $\{m, c, b\}$ unless we've already observed **all** of $\{m, c\}$, $\{m, b\}$, and $\{c, b\}$

...and that's a really good thing!

There are many more *candidate* triples than pairs.

Example: for $n = 10$ items, how many pairs are there? How many triples?

$\binom{10}{2} = \frac{10 \cdot 9}{2} = 45$

$\binom{10}{3} = \frac{10 \cdot 9 \cdot 8}{3 \cdot 2 \cdot 1}$

What about bigger counts?

Example: We don't have to search for, allocate memory for, or even consider the frequent itemset $\{m, c, b\}$ unless we've already observed **all** of $\{m, c\}$, $\{m, b\}$, and $\{c, b\}$

...and that's a really good thing!

There are many more *candidate* triples than pairs.

Example: for $n = 10$ items, how many pairs are there? How many triples? **Solution:** $C(10, 2) = 45$ pairs, but $C(10, 3) = 120$ triples. The order is $\mathcal{O}(10^k)$ for itemsets of size k!

at least up to $k = n/2$...

The A-Priori Algorithm

So how do we efficiently count? Starting with pairs:

- If we have enough main memory to count all pairs - via triangular array or triples - then we can go for it and not worry about A-Priori.
  1. Do one "pass" over all baskets (we assume that we **can't ever** load all baskets into main memory at once)
  2. Do a double loop within each basket to count all pairs in that basket 7 items $\to \binom{7}{2}$ pairs

     all paired items in each basket
  3. Each time you see a pair, add one to its count
  4. Check which pairs have count $\geq s$ at the end.
- If the data set is too large, we use the *A-Priori* algorithm Why? The goal is to reduce the number of pairs to count in exchange for performing two passes over the basket data.

The A-Priori Algorithm

inventory [item a, item]

row | item String | int code

hash fn.

First pass through the data: **create two data tables:**

First Table: hash table

1. If necessary, translates item names to integers
2. Allows us lookup index of any item

Second Table: counts

1. Array of counts
2. Element i count the occurrences of item numbered i.
3. Initialized as 0.

Same order.

As we read each basket...

1. Translate item name $\rightarrow$ integer in table 1
2. Use integer index into array of counts, increment

The A-Priori Algorithm

full_in = [0,

frequent = [4, 17, 23]

After the first pass we **filter** out the infrequent singletons and create *frequent item* tables.

1. Examine the count array and determine which items are frequent. The threshold s should be sufficiently high that we do not get too many frequent item sets.
   Example: consider a typical s like 1%. At the supermarket, items like milk, eggs, bread will be bought more than 1% of the time, but many other things will be much less than 1%.
2. Create a new numbering with *only* the frequent items, numbered 1 to m. This will be another length n 1-D array that holds a new numbering system:
   - 0 if the item i is not frequent
   - A unique integer 1 to m if the ith item is frequent

The A-Priori Algorithm

30,000 items → 1,000 frequent items

$\binom{1000}{2}$ possible frequent pairs

With our new tracker of the m frequent items, we do a second pass through the data. Go through each basket, **counting all pairs of two frequent items**.

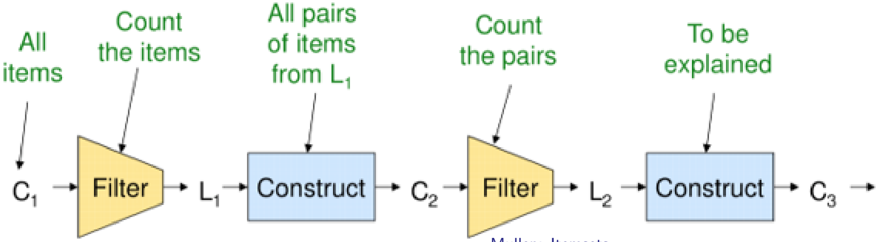
- Space required for triangular array is $\approx m^2/2$ (m choose 2) for all pairs of the m frequent items. The renumbering allowed us to make this object much smaller, as $m << n$
- For each basket...
  1. Look in frequent-items table to see which items are actually frequent
  2. In a double loop, generate all pairs of those frequent items in that basket
  3. For each pair, add one to its count in the triangular array or triples array
- At end of the second pass, examine the array of counts to determine which pairs are frequent.

if all: {ABC}, ABD, ACD, BCD ⇒ then {[A, B, C, D; count}, [A C F R, count

The A-Priori Algorithm

...and we can continue. For frequent k-tuples,

- Generate a list C_k of *candidate* k-tuples. For an item set to be a frequent k-tuple, *each* of the size $k-1$ tuples that represent subsets of that item set must be frequent.
- Use array-style counts $(i_1, i_2, \ldots, i_k, count)$, then do a pass through the data and count the support of items in C_k.
- Prune down to just L_k, the list of actually frequent $k-$tuples and repeat for $k+1$ until satisfied!



A Priori Example

Example: We can't escape Walmart. Use $s = 3$, implement A-priori.

$B_1=$	{b,c,m}	$B_2=$	{j,m,p}	$B_3=$	{b,c,m,n}	$B_4=$	{c,j}
$B_5=$	{b,m,p}	$B_6=$	{b,c,j,m}	$B_7=$	{b,c,j}	$B_8=$	{b,c}

Solution: Pass 1

size 1s: b, c, m, j

triangular array

create candidates [C_{bc}, C_{bm}, C_{bj}, C_{cm}, C_{cj}, C_{mj}]

5

get

count

Candidate[[{bcm}, C]]

A Priori Example

Example: We can't escape Walmart. Use $s = 3$, implement A-priori.

$B_1=$	{b,c,m}	$B_2=$	{j,m,p}	$B_3=$	{b,c,m,n}	$B_4=$	{c,j}
$B_5=$	{b,m,p}	$B_6=$	{b,c,j,m}	$B_7=$	{b,c,j}	$B_8=$	{b,c}

Solution: Pass 1

1. Candidates: $C_1 = \{\{b\}, \{c\}, \{j\}, \{m\}, \{n\}, \{p\}\}$. But when we actually count, we find $L_1 = \{\{b\}, \{c\}, \{j\}, \{m\}\}$
   Pass 2:

A Priori Example

Example: We can't escape Walmart. Use $s = 3$, implement A-priori.

$B_1=$	{b,c,m}	$B_2=$	{j,m,p}	$B_3=$	{b,c,m,n}	$B_4=$	{c,j}
$B_5=$	{b,m,p}	$B_6=$	{b,c,j,m}	$B_7=$	{b,c,j}	$B_8=$	{b,c}

Solution: Pass 1

1. Candidates: $C_1 = \{\{b\}, \{c\}, \{j\}, \{m\}, \{n\}, \{p\}\}$. But when we actually count, we find $L_1 = \{\{b\}, \{c\}, \{j\}, \{m\}\}$
   Pass 2:
2. Candidates: $C_2 = \{\{b, c\}, \{b, j\}, \{b, m\}, \{c, j\}, \{c, m\}, \{j, m\}\}$. But when we actually count, we find $L_2 = \{\{b, c\}, \{b, m\}, \{c, j\}, \{c, m\}\}$
   Pass 3:

A Priori Example

Example: We can't escape Walmart. Use $s = 3$, implement A-priori.

B_1=	{b,c,m}	B_2=	{j,m,p}	B_3=	{b,c,m,n}	B_4=	{c,j}
B_5=	{b,m,p}	B_6=	{b,c,j,m}	B_7=	{b,c,j}	B_8=	{b,c}

Solution: Pass 1

1. Candidates: $C_1 = \{\{b\}, \{c\}, \{j\}, \{m\}, \{n\}, \{p\}\}$. But when we actually count, we find $L_1 = \{\{b\}, \{c\}, \{j\}, \{m\}\}$
   Pass 2:
2. Candidates: $C_2 = \{\{b, c\}, \{b, j\}, \{b, m\}, \{c, j\}, \{c, m\}, \{j, m\}\}$. But when we actually count, we find $L_2 = \{\{b, c\}, \{b, m\}, \{c, j\}, \{c, m\}\}$
   Pass 3:
3. Candidates: $C_3 = \{\{b, c, m\}\}$. It is frequent, so $L_3 = \{\{b, c, m\}\}/$
   No Need for Pass 4: $C_4 = \emptyset$

We only had to look for one triple and zero quads, and even for this toy data set this is much smaller than $\binom{6}{3} = 20$ or $\binom{6}{4} = 15$ theoretical candidates!

A Priori Alternatives

What was the big cost of A-priori? If we wanted to find up to k tuples, we had to pass over the data at least k times. This is order nk, and n might be huge!

Question: could we use fewer passes over the basket data set?

Answer: Yes, but at a cost

Some algorithms use 2 or fewer passes for all sizes, but might miss some frequent itemsets. Such alternatives include:

- Random Sampling
- SON algorithm (Savasere, Omiescinski and Navathe)
- See textbook for some others.

Random Sampling

Assuming the full data set is too large to store in main memory, instead we could take a random sample of the data set of market baskets and store *that* in main memory.

Then we can run A-Priori fully in main memory to find all frequent itemsets (of desired sizes) of the sample of baskets.

- This saves on disk I/O since we only have to read the basket data once!
- But we have to reduce our support threshold proportionately to match the sample size, and hope that *extrapolating* to the rest of the data is valid. It can be hard to gather a perfectly *random* subsample!

Random Sampling

After running A-Priori to find all the frequent itemsets of the **sample** basket data, we could verify that the candidates from the subsample are present in the whole data set by running a second pass over the data.

- This means we won't have any false positives!
- ... but we might have false negatives: we would miss itemsets frequent in the whole data but **not** the sample
- A smaller threshold in the sample might catch some of these, but requires more memory to store candidates.

SON

Time for a quick discrete flashback!

Theorem: *The Generalized Pidegeonhole Principle* says that if N objects are placed into k boxes, then there is at least one box containing at least $\lceil N/k \rceil$ objects.

Idea: break the data set into m pieces. In order for an itemset to have support s, then *at least one* of those pieces has to have support s/m.

First Pass:

1. Break data into small subsets, load chunks at a time.
2. For each chunk, run A-Apriori in main memory to find all frequent itemsets for that subset of baskets.

Second Pass:

1. Now that we have candidates, allocate an object for their counts them and determine what's frequent in the entire set

SON

The big payoff to SON is that repeatedly reading small subsets of basakets into main memory does not have to be done sequentially or *in serial*. Rather, it can be done *in parallel*.

- Can distribute subsets of baskets among many different CPUs/nodes
- Compute frequent itemsets at each node
- Then distribute the candidate information among all nodes
- Accumulate counts of all candidates

We'll revisit this briefly at the end of the semester when we discuss MapReduce!

SON

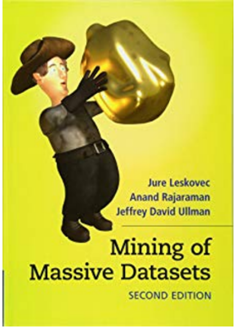
The big payoff to SON is that repeatedly reading small subsets of basakets into main memory does not have to be done sequentially or *in serial*. Rather, it can be done *in parallel*.

- Can distribute subsets of baskets among many different CPUs/nodes
- Compute frequent itemsets at each node
- Then distribute the candidate information among all nodes
- Accumulate counts of all candidates

We'll revisit this briefly at the end of the semester when we discuss MapReduce!

Acknowledgments

Next time: Graphs!
Some material is adapted/adopted from Mining of Massive Data Sets, by Jure Leskovec, Anand Rajaraman, Jeff Ullman (Stanford University) `http://www.mmds.org`



Special thanks to Tony Wong for sharing his original adaptation and adoption of slide material.