Association Rules and Agrawal's Apriori Algorithm


Lots of data in tuple form...

- The cardinal example: Shopping lists
  - Grocery stores
  - Online merchants ("shopping cart")
- The focus: Transactional data consisting of tuples, such as each tuple corresponding to items which were purchased together

Find USEFUL rules

- Finding BAD rules is as easy as using keyword searches
- Amazon.com (ZShops) example
  - "When Titans Clashed", by Glantz (WWII ETO). The page also mentions…
  - "Titans: Yasmine Bleeth" issue of *Entertainment Weekly"
More useful…

- Rules that…
  - Apply reasonably often
  - Are unusually reliable
  - Make interesting predictions
- Such as “People who buy *When Titans Clashed* also may be interested in *The Road to Stalingrad*”

The Association Rule

- Simplify our perspective
  - Treat each tuple member as a binary predicate -- only presence or absence matters, not quantity.
  - Then, for disjoint predicate sets \{a_i\} and \{b_i\}, the statement “\{a_i\} implies \{b_i\}” is an association rule.

...but not all are interesting

- Rules which seldom occur -- the combination of referenced items is unusual -- are less interesting
- Stipulate a minimum support, where support is the fraction of transactions that have all the required and predicted items.
...or accurate

- Rules which are rarely correct -- the implication seldom holds -- are also less useful.
- Stipulate a minimum confidence, where confidence is the fraction of relevant cases (conditions met) where the implication is correct.

More (probably) bad rules

- Too specific (low support): “People who buy books on Irish mythology, science fiction, Perl, and Chinese classics might also buy books on the KGB.”
- Too inaccurate (low confidence): “People who breathe oxygen might also buy AA batteries.”

Problem Summary

- Given a database consisting of tuples, find association rules that frequently and reliably predict which items occur together.
- It's akin to deducing Horn clauses in Prolog, but rules are not expected to always hold: exceptions are the rule.
Search Space

- Naively, the search space of possible rules is $O(3^n)$ for $n$ items: each may be a
  - Requirement for rule
  - Prediction by rule
  - Or… entirely absent
- And $n$ can be big (hundreds of thousands, or more) so brute-force is impractical

Monotonicity of Support

- A key property:
  $$A \subseteq B \text{ implies that } \text{support}(A) \geq \text{support}(B)$$
- Call a set "large" if it has sufficient support. Then all the subsets of a large predicate set must also be large!

And support is what counts

- If $A$ implies $B$ has sufficient support and confidence, then...
  - So do both $A$ and $B$, individually
  - The confidence is the ratio of the support of their union to the support of just $A$
  - Computing support of all large sets will suffice
Really.

- If "A implies B" has either insufficient support or confidence, then adding items to the RHS won't help.
- Given such support computations, a recursive search suffices.

Finding rules, given a large set

- If A is a large set, move items over one at a time to the RHS.
- The LHS will still be large, so we can compute the confidence (the ratio of supports).
- Recurse if confidence still suffices.

The Apriori way

- Start with $L_1$, the set of 1-item large sets.
- Compute $C_k$, possible $k$-item large sets, from $L_{k-1}$.
- Prune $C_k$.
- Compute $L_k$ by checking $C_k$. 
Computing \( L_1 \)

- Iterate through all transactions, keeping statistics in a hash tree

Computing \( C_k \) from \( L_{k-1} \)

- For every pair \( A, B \) of large sets in \( L_{k-1} \), with \( A_i = B_i \) for \( i = 1..k-2 \), and \( A_{k-1} < B_{k-1} \), add \( \{A_1..A_{k-1}, B_{k-1}\} \) to \( C_k \)
- That is, extend large sets to make longer (but probably smaller) possibly large sets
- Can be implemented as a self-join on \( k-2 \) columns

Correctness of extension

- Suppose we considered every large set of size \( k-1 \), and extend with every subsequent item, to make many size \( k \) candidates?
- And then prune by checking each \( X \) to see whether \( \{x_1,..,x_{k-2},x_i\} \) is large?
Pruning $C_k$

- For each $c_i$ in $C_k$, check each $(k-1)$-item subset for membership in $L_{k-1}$
- Any $c_i$ which fails has a non-large subset, cannot itself be large, and need not be checked against the transaction list, so delete it

Computing $L_k$

- Iterate over all transactions
  - For each remaining $c_i$ in $C_k$, keep a counter of transactions where it appears
  - Can again use a hash tree-tracking statistics

Hash tree usage

- Like a trie, but only splits nodes (hash tables or leaves) when full, rather than strict "one level, one item"
- Uses an ordering on items to limit scope for subset searches
Speed?

- Pruning reduces unnecessary statistics tracking, but it still needs to perform subset checking for every single transaction for every iteration...
- ...which might be nasty if there's a long, large itemset (many iterations)

An improvement?

- What if we track transaction statistics ourselves?
- The trade-off: increased storage requirements, instead of many full transaction database scans

AprioriTid

- Stores Xtion IDs associated with each \( C_k \), as well as the derivations: which candidates were generated from which smaller candidates, and vice versa
Using the Xtion stored IDs

- Iterate over Xtions associated with \((k-1)\)-item candidates
- Examine each extension of owning \((k-1)\)-item candidate
- Check other parent of each extension for Xtion

\[ c_{k-1} \text{Tid} \]

\[ c_{k} \text{Tid} \]

\[ c_{k} \text{Tid} \]

\[ C_i \text{ gets credit for the Xtion if both parent candidates did} \]

Performance

- Agrawal compared both Apriori algorithms with AIS and SETM in terms of elapsed time for finding large sets
- AIS: Like Apriori candidate generation, but without pruning, so it tests too many candidates. Counts on the fly.
Performance

- SETM: SQL-style. Candidates are generated using joins, and also include tids. Sorting and aggregation follow.
- Apriori outperforms AIS, which outperforms SETM

Which Apriori is appropriate?

- Apriori (without transaction ID storage) does well in general...
- AprioriTid does better – when the candidate and tuple ID information fits within memory
- The usual answer is…

Use both!

- AprioriHybrid: Start with Apriori, and switch to AprioriTid when
  - An iteration reveals fewer candidates than the previous one
  - All the candidate and tuple data would fit in memory
  - Gets best of both worlds
Rule-finding

- After Apriori, prune all but maximally-large itemsets, and recursively derive rules as mentioned before
- May want additional constraints, such as some form of interesting minimum predictive power, such as confidence versus overall support of results

Rule quality

- “People who buy eggs also buy orange juice”… perhaps uninteresting unless it’s disproportionately higher
- “People who buy wrapping paper, wine and eggnog also buy Christmas cards and 35mm film”… more interesting if disproportionate (and true)

Summary

- Apriori, AprioriTid, and AprioriHybrid provide a simple, scalable approach to part of a useful real-world problem
Further Questions

• How might one handle item hierarchies?
  • Brand of bread versus type (“Rye”, “white”, “whole wheat”…) versus whole category (“bread”)
  • “Town Pride White Bread” might be too specific to have support, “bread” too broad for confidence

Further Questions

• How about non-binary attributes? In particular, what about continuous data?
  • Might “bin” continuous data, which is rather parametric
  • Discrete, non-binary attributes might be treated as one binary attribute per value

Further Questions

• Or temporal or spatial data?
  • Repeat customers might buy items predictable via previous transactions
  • …but how far back should a transaction history be used?
  • Spatial – another form of discrete non-binary data, or… ?