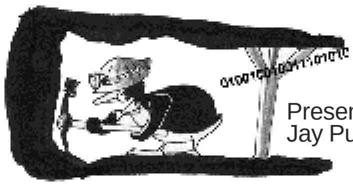
Data Mining
So you want to be a data miner?



Presented by
Jay Pujara

The Goals of Data Mining

- Find interesting data or relationships from large datasets
- This can include problems such as:
  - Find frequently occurring attributes/items
  - Clustering: group similar data together
  - Deviation Monitoring: Flag suspicious values
  - Classification – learn a function that uses data attributes to categorize the data into a class
  - Association Rules – Find correlations between frequently occurring attributes or items

Pertinent Examples of Data Mining

Sponsored Links

SAS® Data Mining
Robust, Trusted Software to Meet Your Data Mining Needs! Learn More
www.SAS.com

ata mining Guide
ee Getting Started Guide & More - white papers and research!
sinessintelligence.bitpipe.com

ata Mining
ustom Data Mining
w Cost 100% U.S.A. Programming
ww.ScrapeGoat.com

ata Mining
prove IT efficacy. Trust your
stomer data. Deploy rapid CDI.
ww.Siperian.com
ennsylvania

ata Mining
nd your answers. Search
ringer's online library now.
ringerlink.com/computerscience

ata Mining Resources
ofessional Software & Specialists
Assist with any Datamining needs
ataMining.ca

howledge on Demand
ctionable marketing & consumer
sights via offshore analytics.
ww.marketics.com

Data Mining
Stay Informed w/ Case Studies, Support & More from Baseline.
www.baselinemag.com

More Sponsored Links »

The problem statement for Mining Association Rules

- Organizations can collect and store MASSIVE amounts of sales data known as basket data.
- Basket data consists of *transactions* which consist of the *items* purchased.
- Data is often sparse, as many different items are offered.
- The rules we're interested in are *some items ? some other items* or X ? Y
- By finding association rules, companies can help people buy things they really need!

Lingo You Should Learn

- The problem requires us to find statistically frequent sets of items and find probable associations between them.
- The frequency is the <u>support</u> – the percentage of time the item(s) appear over transactions.
- Associations are judged based on <u>confidence</u> – the probability that *some items* predict *some other items*.

The *real* problem

- Given parameters *minsup* & *minconf:*
- Generate sets of items with a support value greater than *minsup* (called "<u>large</u>" itemsets)
- Use large data sets to generate association rules with a confidence value greater than *minconf.*
- Do it (a) fast and (b) over lots of data.

Lecture Roadmap

- *Introduction*
- **Paper Summary / Previous Work**
- Algorithm and Variants
- Rule Discovery
- Performance Experiments
- Optimizing Tradeoffs
- Conclusion

Paper Summary: Main Points

R. Agrawal, R. Srikant, Fast Algorithms for Mining Association Rules:

- Use clever logic about sets to quickly find large itemsets (apriori-gen) and use a similar procedure (ap-genrules) to find association rules with high confidence.
- Avoid iterating over the entire data set when checking itemsets for support (aprioriTid) and attempt to maximize performance by adapting the representation of the dataset (aprioriHybrid).
- Validate performance on synthetic and commercial datasets and show *incredible gains in performance*!

The *real* problem, formalized

Let I={i_1, i_2,...,i_m} be a set of literals called items. Let D be the set of transactions, where each transaction T is a set of items such that T µ I. Associated with each transaction is a unique identifier, called its TID.

We say that T *contains* X, a set of some items in I, if X µ T. An association rule is an implication of the form X ! Y where X ½ I, Y ½ I, and X Å Y = ;.

The rule X ! Y holds in the transaction set D with *confidence c*, if *c*% of transaction in D that contain X also contain Y. The rule X ! Y has *support s* in the transaction set D if *s*% of the transactions in D contain X U Y.

Finding Large Sets

- The algorithms of interest approach this problem in a similar manner
  - Generate a list of candidate sets
  - Check by counting candidates in xactions
- The critical difference between previous algorithms is how candidate sets are generated.

Previous Work: AIS

```
$L_1$ = {large 1-itemsets};
**for** (k=2; $L_{k-1}$ ≠ ;; k++){
  $C_k$ = ;;
    **forall** transactions t 2 D {
      $L_t$ = subset($L_{k-1}$,t);
      **forall** large itemsets $l_t$ 2 $L_t$ {
        $C_t$ = 1-extensions of $l_t$ contained in t;
        **forall** candidates c 2 $C_t$ {
          if(c 2 $C_k$)
              add 1 to the count of c in $C_k$
          else
              add c to $C_k$ with a count of 1
     } }  }
$L_k$ = {c 2 $C_k$ | c.count ¸ minsup
}
Large Itemsets = $U_k$ $L_k$
```

- Iterate on k until no large itemsets of size k are found
- For each k, find all large subsets of lengths k-1 found in a transaction and add 1-extensions of these subsets to the candidate list
- For each candidate in the list, search the transaction for the subset.

Previous Work: SETM

```
$L_1$ = {large 1-itemsets}
**$L_1$** = {Large 1-itemsets and TIDs where they appear, sorted by
        TID}
for (k=2; $L_{k-1}$ ≠ ;; k++){
        **$C_k$** = ;;
        forall transactions t 2 D {
                $L_t$ = { l 2 **$L_{k-1}$** | l.TID = t.TID};
                forall large itemsets $l_t$ 2 $L_t$ {
                                $C_t$ = 1-extensions of $l_t$ contained in t;
                                **$C_k$** += {<t.TID, c> | c 2 $C_t$}
                }
        }
        sort **$C_k$** on itemset
        delete all itemsets 2 **$C_k$** for which c.count < minup giving **$L_k$**
        $L_k$ = {<l.itemset, count of l in **$L_k$**}> | l 2 **$L_k$**}}
        sort **$L_k$** on TID;
}
Large Itemsets = [$_k$ $L_k$;
```

- Keep versions of large itemsets and candidate itemsets that include an entry for each occurrence of the itemset, along with the TID of the occurrence
- For each transaction, compute all 1-extensions of large itemsets of length k-1 found in the large-itemset-list and add them to the candidate itemsets
- Sort candidate list by itemset and compute counts
- Resort large sets by TID for the next run

Comparing AIS and SETM

- Both AIS and SETM use the same technique to generate candidates (1-extensions to large k-1 sets found in the data)
- AIS reads through the dataset every time, while SETM keeps a copy of relevant data in memory
- SETM can be implemented using only SQL commands and requires no algorithm-specific data structures, but each pass of the algorithm requires two sorts

Lecture Roadmap

- *Introduction*
- *Paper Summary / Previous Work*
- **Algorithm and Variants**
- Rule Discovery
- Performance Experiments
- Optimizing Tradeoffs
- Conclusion

Apriori Algorithm

```
$L_1$ = {large 1-itemsets}
for (k=2; $L_{k-1}$ ≠ ; ; k++){
    $C_k$ = apriori-gen($L_{k-1}$);
    forall transactions t 2 D {
        $C_t$ = subset($C_k$,t);
        forall candidates c 2 $C_t$
                c.count++
    }
    $L_k$ = {c 2 $C_k$ | c.count ¸ minsup}
}
Large Itemsets = [$_k$ $L_k$
```

- Iterate over k, and generate candidates based on L_{k-1}.
- For each candidate, go through the dataset and increment the count of candidate sets contained in that transaction
- The algorithm hinges on apriori-gen, an innovation that generates fewer candidates than 1-extension.

Apriori improves on AIS and SETM

- Intuition: If a set of length k is large, all subsets of length k-1 must also be large.
- Improve on the candidate generation of SETM and AIS by being smarter!
  - Generate candidates independent of transactions.
  - Use known large itemsets to find possible extensions that create large itemsets.
  - Prune the candidates by making sure all subsets of each candidate set are also large.
  - Fewer candidates means less memory is used!

What's behind apriori-gen?

Join Step:

```
insert into $C_k$
select p.$item_1$, p.$item_2$,...,p.$item_{k-1}$,q.$item_{k-1}$
from $L_{k-1}$ p, $L_{k-1}$ q
where p.$item_1$ = q.$item_1$, p.$item_2$ = q.$item_2$,...,
p.$item_{k-2}$ = q.$item_{k-2}$, p.$item_{k-1}$ < q.$item_{k-1}$
```

Prune Step:

```
forall itemsets c 2 $C_k$
    forall (k-1)-subsets s of c
        if (s 2 $L_{k-1}$)
            delete c from $C_k$
```

- In the join step, elements of L_{k-1} are joined with L_{k-1} on the first k-2 elements.
- Strings are kept lexicographically ordered to avoid duplicates and maintain consistency.
- In the prune step, candidates are checked to ensure all subsets with k-1 elements are in L_{k-1}

Example of apriori-gen

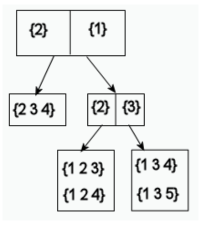
- L_3 = { {1 2 3} {1 2 4} {1 3 4} {1 3 5} {2 3 4} }
- Join Step
  - {1 2 3} joins with {1 2 4} to form {1 2 3 4}, {1 2} in common
  - {1 3 4} joins with {1 3 5} to form {1 3 4 5}, {1 3} in common
  - {2 3 4} doesn't join with anything.
- Prune Step
  - {1 2 3}, {1 2 4}, {1 3 4}, {2 3 4} are all found in L_3, so {1 2 3 4} is kept in C_k
  - {1 3 4}, {1 3 5} are found in L_3, but {1 4 5} and {3 4 5} are not, {1 3 4 5} is pruned from C_k.

Looking at Apriori Ops

- To run Apriori, many set operations on itemsets are necessary
- If these set operations are expensive, AIS and SETM would outperform Apriori
- Set operations must be fast:
  - member: Is s 2 L_{k-1}?
  - subset: Are the items in c a subset of T?

Data Structures for Fast Set Operations

- member: Use a hash table to check if an itemset is in L_{k-1}
- subset: Use a hash tree for C_k
  - Interior nodes of the tree contain hash tables whose buckets contain pointers to the next node
  - Leaves contain candidate itemsets. The answer set contains references to these sets.
  - All nodes begin as leaves and are promoted when the size of the leaf exceeds some threshold.
  - Subset is determined by hashing every item in the transaction at the root, and recursively attempting to hash any possible item at interior nodes.



Remember Memory Issues

- AIS generates candidates on the fly, requiring only the candidate list to be kept in memory.
- Apriori depends on using L_{k-1} to generate C_k. C_k, L_{k-1}, and a buffer page for D must be memory-resident
  - C_k might not fit in memory
    - Multiple passes of C_k generation and D counting
  - L_{k-1} might not fit in memory
    - Externally sort L_{k-1}
    - Bring in itemsets necessary for one join, $k-2$ common items
    - Generate candidates
    - Repeat
    - Cannot prune candidates (need all of L_{k-1})

Possible Bottlenecks in Apriori

- *Data Structures – Set operations are slow*
- *Memory – candidate sets or large itemsets may not fit.*
- **DATA – Must scan the entire dataset for each value of k for counting**

Solving the data problem

- As k increases, fewer and fewer itemsets of length k are large.
- Despite this fact, we still read every item in every transaction – millions of transactions!
- Borrow an idea from SETM - why not keep only the items in question for each transaction?
- Apriori could run with only a single, initial scan of D !

Introducing AprioriTID

```
$L_1$ = {large 1=itemsets};
$\mathbf{C}_1$ = database D;
for (k=2; $L_{k-1} \neq$ ;; k++){
    $C_k$ = apriori-gen($L_{k-1}$);
    $\mathbf{C}_k$ = ; ;
    forall entries t 2 $\mathbf{C}_{k-1}$ {
        $C_t$ = {c 2 $C_k$ |
                (c - c[k]) 2 t.itemsets) Æ
                (c - c[k-1]) 2 t.temsets}
        forall candidates c 2 $C_t$
            c.count++;
        if ($C_t \neq$ ; ) { $\mathbf{C}_k$ += <t.TID, $C_t$>;
    }
    $L_k$ = {c 2 $C_k$ | c.count > minsup}
}
Large Itemsets = $[_k L_k$
```

- If L_k can be generated by L_{k-1}, C_k can be checked using transaction information about the itemsets of C_{k-1}
- Store relevant dataset in $\mathbf{C}_{k-1}$, with candidates tagged with TID.
- If c-c[k] Æ c-c[k+1] are both in $\mathbf{C}_{k-1}$, tagged with TID, then that transaction contains c

Modifying Data Structures for AprioriTID

- No longer need to maintain a hash-tree
- Assign each candidate itemset an ID. C_k stored as an array index by ID, $\mathbf{C}_{k-1}$ has form <TID, {ID}>
- Create *generators* and *extensions*
  - Generators are the IDS to the two large (k-1) itemsets that created a candidate c_k
  - Extensions are the IDs of size k candidates created by extending a large k-1 itemset.
- Check to see if the generators of c_k show up in t.TID

Buffer Management in AprioriTID

- Candidate generation is the same, must keep L_{k-1} and C_k
- Counting is different, instead of just C_k, must also keep C_{k-1} (for ID ! itemset map), and a buffer page for each $\mathbf{C}_k$ and $\mathbf{C}_{k-1}$.
- Fill only half the buffer during candidate generation, ensuring that all itemsets generated from a single join are produced so the generators can be discarded.
- No pruning!

Lecture Roadmap

- *Introduction*
- *Paper Summary / Previous Work*
- *Algorithm and Variants*
- **Rule Discovery**
- Performance Experiments
- Optimizing Tradeoffs
- Conclusion

Rule Discovery

- For a large subset *l*, find rules for some *a* ½ *l* of the form, *a* ! (*l*-*a*).
- This occurs when $\frac{support(l)}{support(a)} \geq minconf$
- Use basic inclusion to avoid unnecessary rules: go from general to specific – if ABC **9** D, adding another item, ie. AB ! CD, will not create a valid rule.

Rule Discovery Algorithm

```
forall large itemsets $l_k$, k ¸ 2
   call genrules($l_k$, $l_k$);


genrules($l_k$, $a_m$){
   A = {(m -1 itemsets $a_{m-1}$ | $a_{m-1}$ ½ $a_m$}
   forall $a_{m-1}$ 2 A {
      conf = support($l_k$)/support($a_{m-1}$);
      if (conf ¸ minconf ) {
         output "$a_{m-1}$ ! ($l_k$ - $a_{m-1}$)
                        with conf, support($l_k$);"
         if(m-1) > 1)
            call genrules($l_k$, $a_{m-1}$);
      }
   }
```

- Recursion on a_{m-1} to create successively larger consequents.

Apply what you’ve learned: a Better Rule Discovery Algorithm

- Basic Intuition: AB ! CD holds only if ABC ! D Æ ABD ! C.
- If ABD **9** C, no reason to check AB ! CD
- Generalization: All rules involving the subsets of a consequent must hold for the consequent to hold. (Think: all subsets of an itemset must be large...)
- Idea: Use single-item consequents to generate possible two-item consequents.

Faster Rule Discovery

```
forall large itemsets $l_k$, k ¸ 2 {
   $H_1$ = {one-item consequents
            of rules derived from $l_k$};
   call ap-genrules($l_k$, $H_1$);
}

ap-genrules($l_k$, $H_m$){
   if (k > m+1){
      $H_{m+1}$ = apriori-gen($H_m$);
      forall $h_{m+1}$ 2 $H_{m+1}${
         conf = support($l_k$)/support($l_k$ - $h_{m+1}$);
         if (conf ¸ minconf)
            output "($l_k$ - $h_{m+1}$) ! $h_{m+1}$
               with conf, support($l_k$)"
         else
            delete $h_{m+1}$ from $H_{m+1}$;
      }
      call ap-genrules($l_k$, $H_{m+1}$);
   }
```

- Instead of generating all possible itemsets for the antecedent, generate longer consequents from shorter consequents using apriori-gen.

Lecture Roadmap

- *Introduction*
- *Paper Summary / Previous Work*
- *Algorithm and Variants*
- *Rule Discovery*
- **Performance Experiments**
- Optimizing Tradeoffs
- Conclusion

Comparing Performance: AIS, SETM, Apriori, AprioriTID

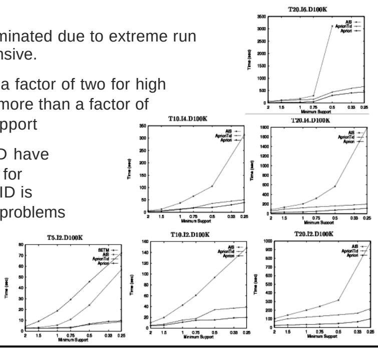
- Experiments run on IBM RS/6000, 33 MHz, 64 MB RAM, 2GB HD at 2MB/s
- Tested using synthetic data and two retail datasets.
- Naming scheme for datasets:
  - #Transactions: \cD
  - Average items in a transaction: T
  - Average size of maximal large itemset: I
  - Number of maximal large itemsets: L
  - Number of items: N
- N = 1000, L = 2000, vary T, I, D
- Naming Scheme T5.I2.D100K

Tests on Synthetic Data Apriori Wins!

• SETM was often terminated due to extreme run times. Sorts are expensive.

• Apriori beats AIS by a factor of two for high levels of support and more than a factor of 10 for low levels of support

• Apriori and AprioriTID have comparable run times for small problems, but TID is twice as slow in large problems

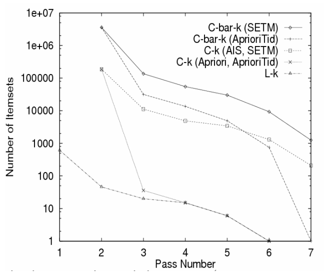


How much difference could apriori-gen make?

• Notice the *logarithmic* scale for the number of candidate itemsets generated for different values of k.

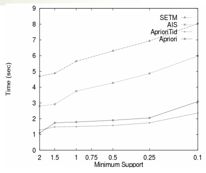
• Apriori-gen quicky drops from millions to hundreds while on-the-fly generation results in hundreds of thousands of candidates.

• SETM and APrioriTID must keep many itemsets!

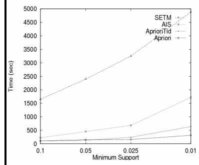


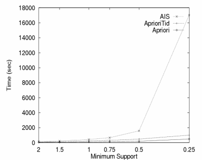
Performance Tests on Retail Data

• Left: Single orders: N = 16K, T=2.6, D = ~3M
    • AprioriTID twice as slow for low supports
    • Apriori = 2-6x AIS, 15x SETM
• Right: All customer orders: N = 16K, T=31, D = ~200K
    • AprioriTID twice as slow for low supports
    • Apriori = 3-30x AIS, SETM fills disk



N = 63, D = ~50K, T = 2.5
Aprioris = 3X AIS, 4x SETM



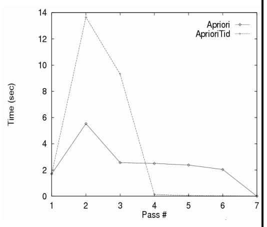


Lecture Roadmap

- *Introduction*
- *Paper Summary / Previous Work*
- *Algorithm and Variants*
- *Rule Discovery*
- *Performance Experiments*
- **Optimizing Tradeoffs**
- Conclusion

Can Apriori beat itself?

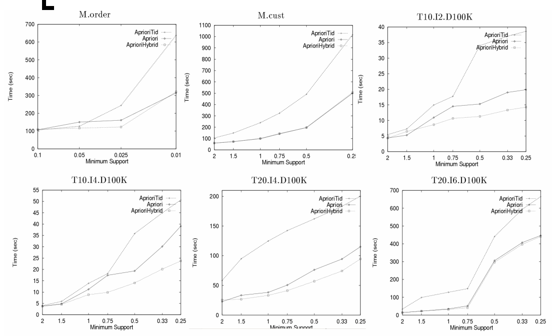
- Apriori does well, but AprioriTID didn't perform well.
- Look at the execution time vs. pass! AprioriTID is instantaneous after pass 4!
- We want the minimum of the two lines.
- How can we leverage the strengths of both these algorithms?
  - Avoid the space constraints of AprioriTID without paying the data scanning penalty of Apriori?



AprioriHybrid: best thing since sliced bread.

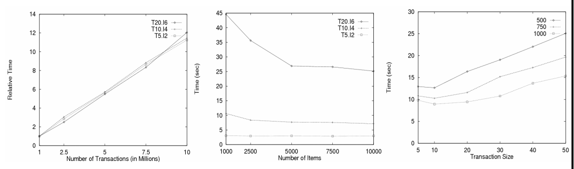
- Begin with Apriori
- When the estimated size of **C**$_k$ meets some heuristic (smaller than D or fits in memory), switch to AprioriTID
- On the next pass, create **C**$_k$ while scanning dataset- performance penalty
- Future passes will avoid scanning the entire dataset!

But does it work? Why, yes, it does! Proof by blurry graphs.



Scale-up properties: good

- Scale-up measured with respect to D, N, &T
  - Linear scale-up with increasing D
  - As N increases, faster performance, less support
  - Gradual increase as T increases



Lecture Roadmap

- *Introduction*
- *Paper Summary / Previous Work*
- *Algorithm and Variants*
- *Rule Discovery*
- *Performance Experiments*
- *Optimizing Tradeoffs*
- **Conclusion**

Conclusions about Apriori

- Generating candidate sets on-the-fly was fast, but not very smart. Fewer candidates really pays off.
- Good data structures make these algorithms possible.
- Buffer management isn't too big of a problem.
- Even today, Apriori is considered the best rule association algorithm.

Future Directions for Mining Association Rules

- Use hierarchical items
  - table is dining furniture is furniture
- Take quantities into account
- Work on finding "interesting" rules using heuristics

Other Data Mining: Classification

- Frequently use decision trees to learn F: data ! class
- Classical machine learning uses a recursive DF algorithm to generate DTs.
- Data Mining builds trees breadth first, performs split computations at once.