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
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 support – the percentage of time the item(s) appear over transactions.
- Associations are judged based on confidence – 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 “large” 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.
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 dataset 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 $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals called items. Let $\mathcal{D}$ be the set of transactions, where each transaction $T$ is a set of items such that $T \subseteq \mathcal{I}$. Associated with each transaction is a unique identifier, called its TID.

We say that $T$ contains $X$, a set of some items in $\mathcal{I}$, if $X \subseteq T$. An association rule is an implication of the form $X \Rightarrow Y$ where $X \subseteq \mathcal{I}$, $Y \subseteq \mathcal{I}$, and $X \cap Y = \emptyset$.

The rule $X \Rightarrow Y$ holds in the transaction set $\mathcal{D}$ with confidence $c$, if $c\%$ of the transactions in $\mathcal{D}$ that contain $X$ also contain $Y$. The rule $X \Rightarrow Y$ has support $s$ in the transaction set $\mathcal{D}$ if $s\%$ of the transactions in $\mathcal{D}$ contain $X \cup Y$.
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 transactions
- The critical difference between previous algorithms is how candidate sets are generated.

Previous Work: AIS

\[ L_1 = \{\text{large 1-itemsets}\}; \]
\[
\text{for } (k=2; L_{k-1} \neq \emptyset ; k++)\{
C_k = \{\};
\text{forall transactions } t \in D \{
L_t = \text{subset}(L_{k-1}, t); \]
\[
\text{forall large itemsets } l \in L_t \{
C_t = \text{1-extensions of } l \text{ contained in } t; \]
\[
\text{if}(l \subseteq C_k) \text{ add } 1 \text{ to the count of } c \text{ in } C_k \text{ else add } c \text{ to } C_k \text{ with a count of } 1 \}
\}
\]
\[
L_k = \{c \in C_k \mid c \text{.count } \geq \text{minsup}\}
\]
Large Itemsets = \[L_1 \cup L_k\]

- 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

\[ L_1 = \{\text{large 1-itemsets}\}; \]
\[ L'_1 = \{\text{large 1-itemsets and TIDs where they appear, sorted by TID}\}; \]
\[
\text{for } (k=2; L_{k-1} \neq \emptyset ; k++)\{
C_k = \{\};
\text{forall transactions } t \in D \{
L_t = \{ l \in L_{k-1} \mid \text{TID} = \text{TID}\}; \]
\[
\text{forall large itemsets } l \in L_t \{
C_t = \text{1-extensions of } l \text{ contained in } t; \]
\[
C_t += \{<\text{t.TID}, c> \mid c \in C_t \}
\}
\}
\]
\[
\text{sort } C_k \text{ on itemset}
\text{delete all itemsets } 2 C_k \text{ for which c.count < minsup giving } L_k
\]
\[
L_k = \{<\text{itemset, count of } l \text{ in } L_k> \mid (2 \ L_k)\}
\]
\[
\text{sort on TID;}
\]
Large Itemsets = \[L_1 \cup L_k\]

- 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

\[ L_k = \{ \text{large 1-itemsets} \} \]

\[ \text{for (} k=2, L_k = ; k++ ; \} \]

\[ C_k = \text{apriori-gen}(L_{k-1}); \]

\[ \text{forall transactions } t \in D \{ \]

\[ C_t = \text{subset}(C_{k}, t); \]

\[ \text{forall candidates } c \in C_t, \]

\[ c \cdot \text{count}++; \]

\[ L_k = \{ c \in C_t | c \cdot \text{count} \geq \text{minsup} \} \]

Large Itemsets = \[ I_k \]

- 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 3, L 3 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

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 4
- {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 4
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+1}, 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, k2 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

\[ L_1 = \text{large 1-itemsets}; C_1 = \text{database D}; \]
\[ \textbf{for} \{k=2; L_{k-1} \neq \emptyset ; k+1\}: \]
\[ C_k = \text{apriori-gen}(L_{k-1}); \]
\[ \text{forall entries } t \geq C_{k-1} \{ \]
\[ C_t = \{c \in C_k | \text{c-itemsets} \in L_{k-1} \}; \]
\[ \text{forall candidates } c \in C_t \{ \]
\[ \text{c.count}++; \]
\[ \text{if}(C_t \neq \emptyset) \{ \]
\[ <t, \text{TID}, C_t >; \]
\[ \} \]
\[ L_k = \{ c \in C_k | \text{c.count} > \text{minsup} \}; \]
\[ \text{Large itemsets } = L_1, L_k \]

- 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 \( C_{k-1} \), with candidates tagged with TID.
- If \( c \in \mathbb{N} \& c < k+1 \) are both in \( C_{k-1} \), 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, $C_{k-1}$ 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 $L_{TID}$.

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 $C_k$ and $C_{k-1}$.
- 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
For a large subset \( l \), find rules for some \( a \subseteq l \) of the form, \( a \rightarrow (l-a) \).

This occurs when \( \frac{\text{support}(l)}{\text{support}(a)} \geq \minconf \).

Use basic inclusion to avoid unnecessary rules: go from general to specific – if ABC \( \rightarrow \) D, adding another item, i.e. AB \( \rightarrow \) CD, will not create a valid rule.

### Rule Discovery Algorithm

forall large itemsets \( l_k, k \geq 2 \)
call genrules\( (l_k, l_k) \);

\[
\text{genrules}(l_k, a_{m-1}) \{
A = \{ \text{itemsets} a_{m-1} | a_{m-1} \subseteq a_m \}
\text{forall } a_{m-1} \in A \{
\text{conf} = \frac{\text{support}(l_k)}{\text{support}(a_{m-1})};
\text{if (conf} \geq \minconf) \{
\text{output } "a_{m-1} \rightarrow (l_k - a_{m-1})" \text{ with conf, support}(l_k) ;
\text{if}(m-1) > 1)
call genrules\( (l_k, a_{m-1}) ;
\}
\}
\]

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

- Basic Intuition: AB \( \rightarrow \) CD holds only if ABC \( \rightarrow \) D \( \rightarrow \) ABC \( \rightarrow \) CD.
- If ABD \( \rightarrow \) C, no reason to check AB \( \rightarrow \) 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

\[
\text{forall large itemsets } l_k, k \geq 2 \{
H_1 = \{\text{one-item consequents of rules derived from } l_k\};
call \text{ap-genrules}(l_k, H_1);
\}
\text{ap-genrules}(l_k, H_m) |
\text{if } (k > m + 1) |
H_{m+1} = \text{apriori-gen}(H_m);
\text{forall } h_{m+1} \in H_{m+1} |
\text{conf} = \text{support}(l_k)/\text{support}(l_k - h_{m+1});
\text{if } (\text{conf} \geq \text{minconf}) |
\text{output } "(l_k - h_{m+1}) \Rightarrow h_{m+1} \text{ with conf, support}(l_k)"
\text{else } |
delete h_{m+1} \text{ from } H_{m+1};
\}
call \text{ap-genrules}(l_k, H_{m+1});
\]

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: D
  - 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 quickly 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
  - Apriori-gen = 3X AIS, 4x SETM

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