Research Report

Constraint-Based Rule Mining in Large, Dense Databases

Roberto J. Bayardo Jr.
Rakesh Agrawal
Dimitrios Gunopulos

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, California 95120

Research Division
Yorktown Heights, New York ❋ San Jose, California ❋ Zurich, Switzerland

Constraint-Based Rule Mining in Large, Dense Databases

Roberto J. Bayardo Jr.
Rakesh Agrawal
Dimitrios Gunopulos*

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, California 95120

ABSTRACT:

Constraint-based rule miners find all rules in a given data-set meeting user-specified constraints such as minimum support and confidence. We describe a new algorithm that exploits all user-specified constraints including minimum support, minimum confidence, and a new constraint that ensures every mined rule offers a predictive advantage over any of its simplifications. Our algorithm maintains efficiency even at low supports on data that is dense (e.g. relational data). Previous approaches such as Apriori and its variants exploit only the minimum support constraint, and as a result are ineffective on dense data due to a combinatorial explosion of “frequent itemsets”.

*Current affiliation: University of California at Riverside

1. Introduction

Mining rules from data is a problem that has attracted considerable interest because a rule provides a concise statement of potentially useful information that is easily understood by end users. In the database literature, the focus has been on developing association rule [2] algorithms that identify all conjunctive rules meeting user-specified constraints such as minimum support (a statement of generality) and minimum confidence (a statement of predictive ability). The completeness guarantee provided by association rule miners is what distinguishes them from other rule-mining methods such as decision-tree induction. This completeness guarantee provides a high level of comfort to the analyst who uses rules for decision support (end-user understanding), as opposed to building a predictive model for performing automated classification tasks.

Association rule algorithms were initially developed to tackle data-sets primarily from the domain of market-basket analysis. In market-basket analysis, one problem is to mine rules that predict the purchase of a given set of store items based on other item purchases made by the consumer. Though the dimensionality of market-basket data is quite high (equal to the total number of distinct items), the number of items appearing in a typical record (or transaction) is tiny in comparison. This sparsity is exploited by algorithms such as Apriori for efficient mining. Unlike data from market-basket analysis, data-sets from several other domains including telecommunications data analysis [29], census data analysis [10], and classification and predictive modeling tasks in general tend to be *dense* in that they have any or all of the following properties:[1]

- many frequently occurring items (e.g. sex=male);
- strong correlations between several items;
- many items in each record.

These data-sets cause an exponential blow-up in the resource consumption of standard association rule mining algorithms including Apriori [3] and its many variants. The combinatorial explosion is a result of the fact that these algorithms effectively mine all rules that satisfy only the minimum support constraint, the number of which is exorbitant [6,7,18]. Though other rule constraints are specifiable, they are typically enforced solely during a post-processing filter step.

In this paper, we directly address the problem of constraint-based rule mining in dense data. Our approach is to enforce all user-specified rule constraints during mining. For example, most association rule miners allow users to set a minimum on the predictive ability of any mined rule specified as either a minimum confidence [2] or an alternative measure such as lift [9,15] or conviction [10]. We present an algorithm that can exploit such minimums on predictive ability during mining for vastly improved efficiency.

---

[1] Market-basket data is sometimes dense, particularly when it incorporates information culled from convenience card applications for mining rules that intermix personal attributes with items purchased.

Even given strong minimums on support and predictive ability, the rules satisfying these constraints in a dense data-set are often too numerous to be mined efficiently or comprehended by the end user. A constraint-based rule miner that can be effectively applied to dense data must therefore provide alternative or additional constraints that the user may specify. Ideally, the constraints should be easy to specify, and further, eliminate only those rules that are uninteresting. To this end, we present and incorporate into our algorithm a new constraint that eliminates any rule that can be simplified to yield an equally or more predictive rule. This constraint is motivated by the principle of Occam's Razor, which states that plurality should not be posited without necessity. To motivate this concept, first consider the example rule given below.

`Bread & Butter → Milk (Confidence = 80%)`

The rule has a confidence of 80%, which means that 80% of the people who purchase bread and butter also purchase the item in the *consequent* of the rule, which is milk. Because of its high confidence, one might be inclined to believe that this rule is an interesting finding if the goal is to, say, understand the population of likely milk buyers in order to make better stocking and discounting decisions. However, if 85% of the population under examination purchased milk, this rule is actually quite uninteresting for this purpose since it characterizes a population that is even less likely to buy milk than the average shopper. Put more concretely, this "wordy" rule offers no advantage over the simple rule predicting milk whose antecedent is empty (always evaluating to true).

This point has already motivated additional measures for identifying interesting rules, including lift and conviction. Both lift and conviction represent the predictive advantage a rule offers over simply guessing based on the frequency of the consequent. But both measures still fail to fully enforce Occam's Razor, as illustrated by the next two rules.

`Eggs & Cereal → Milk (Confidence = 95%)`
`Cereal → Milk (Confidence = 99%)`

Because the confidence of the first rule (95%) is significantly higher than the frequency with which milk is purchased (85%), the rule will have lift and conviction values that could imply to the end-user that it is interesting for understanding likely milk buyers. But note that the second rule tells us that the purchase of cereal alone implies that milk is purchased with 99% confidence. We thus have that the first rule actually represents a significant decrease in predictive ability over the second, more concise rule which is more broadly applicable (because there are more people who buy cereal than people who buy both cereal and eggs).

The algorithm we describe in this paper directly allows the user to eliminate unnecessarily complex rules by specifying a *minimum improvement* constraint. The idea is to mine only those rules whose confidence is at least *minimp* greater than the confidence of *any* of its simplifications, where a simplification of a rule is formed by removing one or more conditions from its antecedent. Any positive setting of minimp would prevent the unnecessarily complex rules from the examples above from being generated by our algorithm. By making this constraint a threshold, the user is free to define what is considered to be a “significant” improvement in predictive ability. This feature remedies the rule explosion problem resulting from the fact that in dense data-sets, the confidence of many rules can often be marginally improved upon in an overwhelming number of ways by adding conditions. For example, given the rule stating that cereal implies milk with 99% confidence, there may be hundreds of rules of the form below with a confidence between 99% and 99.1%.

$$\texttt{Cereal \& } I_1 \texttt{ \& } I_2 \texttt{ \& } \ldots \texttt{ \& } I_n \rightarrow \texttt{Milk}$$

The improvement constraint allows the user to trade away such marginal benefits in predictive ability for a far more concise set of rules, with the added property that every returned rule consists entirely of items that are strong contributors to its predictive ability. We feel this is a worthwhile trade-off in most situations where the mined rules are used for end-user understanding.

For rules to be comparable in the above-described context, they must have equivalent consequents. For this reason, our work is done in the setting where the consequent of the rules is fixed and specified in advance. This setting is quite natural in many applications where the goal is to discover properties of a specific class of interest. This task is sometimes referred to as partial-classification [5]. Some example domains where it is applicable include failure analysis, fraud detection, and targeted marketing among many others.

1.1 Paper overview

Section 2 summarizes related work. Section 3 formally defines and motivates the problem of mining rules from dense data subject to minimum support, confidence, and/or improvement constraints. Section 4 begins with an overview of the general search strategy, and then presents pseudo-code for the top level of our algorithm. Section 5 provides details and pseudo-code for the pruning functions invoked by the algorithm body. Section 6 details an item-reordering heuristic for improving pruning performance. Section 7 describes the rule post-processor, which is used to fully enforce the minimum improvement constraint. Some additional optimizations are discussed by Section 8, after which the algorithm is empirically evaluated in Section 9. Section 10 concludes with a summary of the contributions.

2. Related work

Previous work on mining rules from data is extensive. We will not review the numerous proposals for greedy or heuristic rule mining (e.g. decision tree induction) and focus instead on constraint-based algorithms. We refer the reader interested in heuristic approaches to mining large data-sets to the scalable algorithms proposed in [12] and [27].

There are several papers presenting improvements to the manner in which the Apriori algorithm [3] enumerates all frequent itemsets (e.g. [10,21,24,31]), though none address the problem of combinatorial explosion in the number of frequent itemsets that results from applying these techniques to dense data. Other works (e.g. [7,14,17]) show how to identify all maximal frequent itemsets in data-sets where the frequent itemsets are long and numerous. Unfortunately, all association rules cannot be efficiently extracted from maximal frequent itemsets alone, as this would require performing the intractable task of enumerating and computing the support of all their subsets.

Srikant et al. [29] and Ng et al. [20] have investigated incorporating item constraints on the set of frequent itemsets for faster association rule mining. These constraints, which restrict the items or combinations of items that are allowed to participate in mined rules, are orthogonal to those exploited by our approach. We believe both classes of constraints should be part of any rule-mining tool or application.

There is some work on ranking association rules using interest measures [10,15,16], though this work gives no indication of how these measures could be exploited to make mining on dense data-sets feasible. Smythe and Goodman [28] describe a constraint-based rule miner that exploits an information theoretic constraint which heavily penalizes long rules in order to control model and search complexity. We incorporate constraints whose effects are easily understood by the end user, and allow efficient mining of long rules should they satisfy these constraints.

There are several proposals for constraint-based rule mining with a machine-learning instead of data-mining focus that do not address the issue of efficiently dealing with large data-sets. Webb [30] provides a good survey of this class of algorithms, and presents the OPUS framework which extends the set-enumeration search framework of Rymon [22] with additional generic pruning methods. Webb instantiates his framework to produce an algorithm for obtaining a single rule that is optimal with respect to the Laplace preference function. We borrow from this work the idea of exploiting an optimistic pruning function in the context of searching through a power set. However, instead of using a single pruning function for optimization, we use several for constraint enforcement. Also, because the itemset frequency information required for exploiting pruning functions is expensive to obtain from a large data-set, we frame our pruning functions so that they can accommodate restricted availability of such information.

3. Definitions and problem statement

A *transaction* is a set of one or more items obtained from a finite item domain, and a *data-set* is a collection of transactions. A set of items will be referred to more succinctly as an *itemset*. The *support* of an itemset I, denoted $\sup(I)$, is the number of transactions in the data-set to contain I. An *association rule,* or just *rule* for short, consists of an itemset called the antecedent, and an itemset disjoint from the antecedent called the *consequent*. A rule is denoted as $A \rightarrow C$ where A is the antecedent and C the consequent. The *support* of an association rule is the support of the itemset formed by taking the union of the antecedent and consequent ($A \cup C$). The *confidence* of an association rule is the probability with which the items in the antecedent A appear together with items in the consequent C in the given data-set. More specifically:

$$\mathrm{conf}(A \rightarrow C) = \frac{\sup(A \cup C)}{\sup(A)}$$

The association rule mining problem [2] is to produce all association rules present in a data-set that meet specified minimums on support and confidence. In this paper, we restrict the problem in two ways in order to render it solvable given dense data.

3.1 The consequent constraint

We require mined rules to have a given consequent C specified by the user. This restriction is an *item constraint* which can be exploited by other proposals [20, 29], but only to reduce the set of frequent itemsets considered. A frequent itemset is a set of items whose support exceeds the minimum support threshold. Frequent itemsets are too numerous in dense data even given this item constraint. Our algorithm instead leverages the consequent constraint through pruning functions for enforcing confidence, support, and improvement (defined next) constraints during the mining phase.

3.2 The minimum improvement constraint

While our algorithm runs efficiently on many dense data-sets without further restriction, the end-result can easily be many thousands of rules, with no indication of which ones are “good”. On some highly dense data-sets, the number of rules returned explodes as support is decreased, resulting in unacceptable algorithm performance and a rule-set the end-user has no possibility of digesting. We address this problem by introducing an additional constraint.

Let the *improvement* of a rule be defined as the minimum difference between its confidence and the confidence of any proper sub-rule with the same consequent. More formally, for a rule $A \rightarrow C$:

$$\mathrm{imp}(A \rightarrow C) = \min(\forall A' \subset A, \mathrm{conf}(A \rightarrow C) - \mathrm{conf}(A' \rightarrow C))$$

If the improvement of a rule is positive, then removing any non-empty combination of items from its antecedent will drop its confidence by at least its improvement. Thus, every item and every combination of items present in the antecedent of a large-improvement rule is an important contributor to its predictive ability. A rule with negative improvement is typically undesirable because the rule can be simplified to yield a proper sub-rule that is more predictive, and applies to an equal or larger population due to the antecedent containment relationship. An improvement greater than 0 is thus a desirable constraint in almost any application of association rule mining. A larger minimum on improvement is also often justified because most rules in dense data-sets are not useful due to conditions or combinations of conditions that add only a marginal increase in confidence. Our algorithm allows the user to specify an arbitrary positive minimum on improvement.

3.3 Problem statement

We develop an algorithm for mining all association rules with consequent C meeting user-specified minimums on support, confidence, and improvement. The algorithm parameter specifying the minimum confidence bound is known as *minconf*, and the minimum support bound *minsup*. We call the parameter specifying a minimum bound on improvement *minimp*. A rule is said to be *confident* if its confidence is at least minconf, and *frequent* if its support is at least minsup. A rule is said to have a *large improvement* if its improvement is at least minimp.

Other measures of predictive ability that are sometimes used to rank and filter rules in place of confidence include *lift* [9,15] (which is also known as *interest* [10] and strength [13]) and conviction [10]. Below we show that these values can each be expressed as a function of the rule's confidence and the frequency of the consequent; further, note that both functions are monotone in confidence:

$$\begin{aligned}
\mathrm{lift}(A \to C) &= \frac{P(A \wedge C)}{P(A)P(C)} \\
&= \frac{\mathrm{sup}(A \cup C) / \mathrm{sup}(\emptyset)}{(\mathrm{sup}(A) / \mathrm{sup}(\emptyset))(\mathrm{sup}(C) / \mathrm{sup}(\emptyset))} \\
&= \frac{\mathrm{sup}(\emptyset)}{\mathrm{sup}(C)} \cdot \mathrm{conf}(A \to C)
\end{aligned}$$

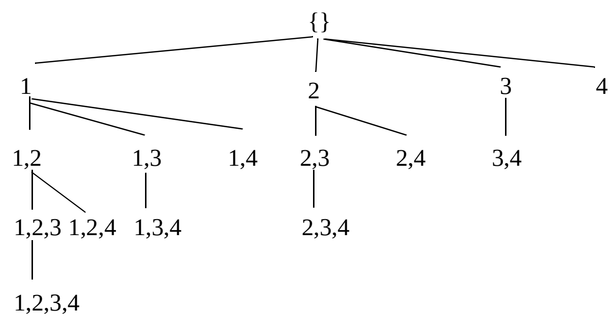
$$\begin{aligned}
\mathrm{conviction}(A \to C) &= \frac{P(A)P(\neg C)}{P(A \wedge \neg C)} \\
&= \frac{(\mathrm{sup}(A) / \mathrm{sup}(\emptyset))\left(\dfrac{\mathrm{sup}(\emptyset) - \mathrm{sup}(C)}{\mathrm{sup}(\emptyset)}\right)}{\left(\dfrac{\mathrm{sup}(A) - \mathrm{sup}(A \cup B)}{\mathrm{sup}(\emptyset)}\right)} \\
&= \frac{\mathrm{sup}(\emptyset) - \mathrm{sup}(C)}{\mathrm{sup}(\emptyset)[1 - \mathrm{conf}(A \to C)]}
\end{aligned}$$

Though we frame the remainder of this work in terms of confidence alone, it can be recast in terms of these alternative measures. This is because, given a fixed consequent, each measure ranks rules identically.

4. Set-enumeration search in large data-sets

From now on, we will represent a rule using only its antecedent itemset since the consequent is assumed to be fixed to itemset C. Let U denote the set of all items present in the database except for those in the consequent. The rule-mining problem is then one of searching through the power set of U for rules which satisfy the minimum support, confidence, and improvement constraints. Rymon's set-enumeration tree framework [22] provides a scheme for representing a subset search problem as a tree search problem, allowing pruning rules to be defined in a straightforward manner in order to reduce the space of subsets (rules) considered. The idea is to first impose an ordering on the set of items, and then enumerate sets of items according to the ordering as illustrated in Figure 1.

FIGURE 1. A completely expanded set-enumeration tree over $U = \{1, 2, 3, 4\}$ with items ordered lexically.



4.1 Terminology

We draw upon and extend the machinery developed in previous work where we framed the problem of mining maximal frequent itemsets from databases as a set-enumeration tree search problem [7]. Each node in the tree is represented by two itemsets called a *group*. The first itemset, called the *head,* is simply the itemset (rule) enumerated at the given node. The second itemset, called the *tail,* is actually an ordered set, and consists of those items which can be potentially appended to the head to form any viable rule enumerated by a sub-node. For example, at the root of the tree, the head itemset is empty and the tail itemset consists of all items in U.

The head and tail of a group g will be denoted as $h(g)$ and $t(g)$ respectively. The order in which tail items appear in $t(g)$ is significant since it reflects how its children are to be expanded. Each child g_c of a group g is formed by taking an item $i \in t(g)$ and appending it to $h(g)$ to form $h(g_c)$. Then, $t(g_c)$ is made to contain all items in $t(g)$ that follow i in the ordering. Given this child expansion policy, without any pruning of nodes or tail items, the set-enumeration tree enumerates each and every subset of U exactly once.

We say a rule r is *derivable* from a group g if $h(g) \subset r$, and $r \subseteq h(g) \cup t(g)$. By definition, any rule that can be enumerated by a descendent of g in the set-enumeration tree is derivable from g.

Define the *candidate set* of a group g to be the set consisting of the following itemsets:

- $h(g)$ and $h(g) \cup C$;
- $h(g) \cup \{i\}$ and $h(g) \cup \{i\} \cup C$ for all $i \in t(g)$;
- $h(g) \cup t(g)$ and $h(g) \cup t(g) \cup C$.

A group is said to be *processed* once the algorithm has computed the support of every itemset in its candidate set.

FIGURE 2. Dense-Miner at its top level. The input parameters minconf, minsup, minimp, and C are assumed global.

```
DENSE-MINER(Set of Transactions T)
  ;; Returns all frequent, confident, large
  ;; improvement rules present in T
  Set of Rules R ← ∅
  Set of Groups G ← GENERATE-INITIAL-GROUPS(T, R)
  while G is non-empty do
      scan T to process all groups in G
      PRUNE-GROUPS(G, R)                ;; Section 5
      G ← GENERATE-NEXT-LEVEL(G)
      R ← R ∪ EXTRACT-RULES(G)
      PRUNE-GROUPS(G, R)                ;; Section 5
  return POST-PROCESS(R, T)             ;; Section 7
```

4.2 Top-level algorithm description

It is now possible to provide a top-level description of the algorithm, which we call Dense-Miner. The body (Figure 2) implements a breadth-first search of the set enumeration tree with Generate-Initial-Groups seeding the search. The groups representing an entire level of the tree are processed together in one pass over the data-set. Though any systematic traversal of the set-enumeration tree could be used, Dense-Miner uses a breadth-first traversal to limit the number of database passes to at most the length of the longest frequent itemset. To support efficient processing of these groups, a hash tree [4] or a trie is first used to index the head of each group in a set of groups. Then, for each transaction in the data-set, any group whose head is contained by the transaction is quickly identified using this data-structure. For each such group, tail items are scanned and a counter associated with a tail item is incremented should the tail item be found within the transaction. Each tail item is paired with two count values, one for when the consequent itemset is present in the transaction and one for otherwise. A pair of counters is also maintained for when every tail item is found to reside within the transaction for computing the support of the long itemsets. Due to good locality, this scheme significantly outperforms individually indexing each candidate set member within a hash-tree [7].

Generate-Initial-Groups could simply produce the root node which consists of an empty head and a tail containing all items from U. However, our implementation seeds the search at the second level of the tree after an optimized phase that rapidly computes the support of all 1 and 2 item rules and their antecedents using array data-structures instead of hash trees (a similar optimi-

FIGURE 3. Procedure for expanding the next level of the set-enumeration tree.

```
GENERATE-NEXT-LEVEL(Set of groups G)
  ;; Returns a set of groups representing the next level
  ;; of the set-enumeration tree
  Set of Groups G_c ← ∅
  for each group g in G do
          reorder the items in t(g)          ;; Section 6
          for each item i in t(g) do
                let g_c be a new group
                   with h(g_c) = h(g) ∪ {i} and
                   t(g_c) = {j | j follows i in the ordering}
                G_c ← G_c ∪ {g_c}
  return G_c
```

zation is used in the Apriori implementation [4]). This is why the pseudo-code function call accepts the set of rules R (which is passed by reference) -- any of these short rules which are found to satisfy the input constraints are added to R before returning.

Generate-Next-Level (Figure 3) generates the groups that comprise the next level of the set-enumeration tree. Note that the tail items of a group are reordered before its children are expanded. This reordering step is a crucial optimization designed to maximize pruning efficiency. We delay discussing the details of item reordering until after the pruning strategies are described, because the particular pruning operations greatly influence the reordering strategy. After child expansion, any rule represented by the head of a group is placed into R by Extract-Rules if it is frequent, confident, and potentially has a large improvement. The support information required to check if the head of a group g represents a frequent or confident rule is provided by the parent of g in the set-enumeration tree because $h(g)$ and $h(g) \cup C$ are members of its candidate set. As a result, this step can be performed before g is processed. To check if a rule potentially has a large improvement at this point in the algorithm, Extract-Rules simply compares its confidence to the confidence of rules enumerated by ancestors of the rule in the set-enumeration tree. A post processing phase (the POST-PROCESS function) later determines the precise improvement value of each rule extracted by this step. The remaining algorithmic details, which include node pruning (the PRUNE-GROUPS function), item-reordering, and post-processing, are the subjects of the next three sections.

5. Pruning

This section describes how Dense-Miner prunes both processed and unprocessed groups. In Figure 2, note that groups are pruned following tree expansion as well as immediately after they are processed. Because groups are unprocessed following tree expansion, in order to determine if they are prunable, Dense-Miner uses support information gathered during previous database passes.

FIGURE 4. Top level of the pruning function.

```
PRUNE-GROUPS(Set of groups G, Set of rules R)
    ;; Prunes groups and tail items from groups within G
    ;; G and R are passed by reference
    for each group g in G do
            do
                try_again ← false
                if IS-PRUNABLE(g)
                   then remove g from G
                   else for each i ∈ t(g) do
                            let g' be a group
                               with h(g') = h(g) ∪ {i}
                               and t(g') = t(g) − {i}
                            if IS-PRUNABLE(g')
                               then remove i from t(g)
                                    put h(g) ∪ {i} in R if it
                                      is a frequent and
                                      confident rule
                                    try_again ← true
            while try_again = true
```

5.1 Applying the pruning strategies

Dense-Miner applies multiple strategies to prune nodes from the search tree. These strategies determine when a group g can be pruned because no derivable rule can satisfy one or more of the input constraints. When a group g cannot be pruned, the pruning function checks to see if it can instead prune some items i from $t(g)$. Pruning tail items reduces the number of children generated from a node, and thereby reduces the search space. An added benefit of pruning tail items is that it can increase the effectiveness of the strategies used for group pruning. The observation below, which follows immediately from the definitions, suggests how any method for pruning groups can also be used to prune tail items.

OBSERVATION 5.1: Given a group g and an item $i \in t(g)$, consider the group g' such that $h(g') = h(g) \cup \{i\}$ and $t(g') = t(g) - \{i\}$. If no rules derivable from g' satisfy some given constraints, then except for rule $h(g) \cup \{i\}$, no rules r derivable from g such that $i \in r$ satisfy the given constraints.

The implication of this fact is that given a group g and tail item i with the stated condition, we can avoid enumerating many rules which do not satisfy the constraints by simply removing i from $t(g)$ after extracting rule $h(g) \cup \{i\}$ if necessary. The implementation of Prune-Groups, described in Figure 4, exploits this fact.

The group pruning strategies are applied by the helper function Is-Prunable which is described next. Because fewer tail items can improve the ability of Is-Prunable to determine whether a group can be pruned, whenever a tail item is found to be prunable from a group, the group and all tail items are checked once more (due to the outer while loop in the pseudo-code).

5.2 Pruning strategies

The function Is-Prunable computes the following values for the given group g:

- an upper-bound $\mathrm{uconf}(g)$ on the confidence of any rule derivable from g,
- an upper-bound $\mathrm{uimp}(g)$ on the improvement of any rule derivable from g that is frequent,
- an upper-bound $\mathrm{usup}(g)$ on the support of any rule derivable from g.

Note that a group g can be pruned without affecting the completeness of the search if one of the above bounds falls below its minimum allowed value as specified by minconf, minimp, and minsup respectively. The difficulty in implementing pruning is in how to compute these bounds given that acquiring support information from a large data-set is time consuming. We show how to compute these bounds using only the support information provided by the candidate set of the group, and/or the candidate set of its parent.

In establishing these bounding techniques in the remaining sub-sections, for a given item i, we sometimes assume the existence of an item $\neg i$ contained only by those transactions that do not contain i. Given an itemset I, we similarly assume the existence of a derived item $\neg I$ that is contained only by those transactions in the data-set that do not contain all items in I. These derived items need not actually be present in the data-set, since the support of any itemset that contains one or more derived items can be computed using itemsets which contain no derived items. This is because for disjoint itemsets I_1 and I_2, we have that $\mathrm{sup}(I_1 \cup \{\neg I_2\}) = \mathrm{sup}(I_1) - \mathrm{sup}(I_1 \cup I_2)$. Note also that $I_1 \subset I_2 \Rightarrow \mathrm{sup}(I_1) \geq \mathrm{sup}(I_2)$, which holds whether or not I_1 and/or I_2 contain derived items.

5.3 Bounding confidence

THEOREM 5.2: The following expression provides an upper-bound on the confidence of any rule derivable from a given group g:

$$\frac{x}{x+y}$$

where x and y are non-negative integers such that $y \leq \mathrm{sup}(h(g) \cup t(g) \cup \{\neg C\})$ and $x \geq \mathrm{sup}(h(g) \cup C)$.

Proof: Recall that the confidence of a rule r is equal to $\mathrm{sup}(r \cup C) / \mathrm{sup}(r)$. This fraction can be rewritten as follows:

$$\frac{x'}{x' + y'} \text{ where } x' = \mathrm{sup}(r \cup C) \text{ and } y' = \mathrm{sup}(r) - \mathrm{sup}(r \cup C).$$

Because this expression is monotone in x' and anti-monotone in y', we can replace x' with a greater or equal value and y' with a lesser or equal value without decreasing the value of the expression. Consider replacing x' with x and y' with y. The claim then follows if we establish that for any rule r derivable from g, (1) $x \geq x'$, and (2) $y \leq y'$. For (1), note that $h(g) \subset r$. It follows that $\sup(r \cup C) \le \sup(h(g) \cup C)$, and hence $x \ge x'$. For (2), note that $r \subseteq h(g) \cup t(g)$. Because $r \cup \{\neg C\} \subseteq h(g) \cup t(g) \cup \{\neg C\}$, we have $y \le \sup(h(g) \cup t(g) \cup \{\neg C\}) \le \sup(r \cup \{\neg C\}) = \sup(r) - \sup(r \cup C) = y'$. □

Theorem 5.2 is immediately applicable for computing $\mathrm{uconf}(g)$ for a processed group g since the following itemsets needed to compute tight values for x and y are all within its candidate set: $h(g)$, $h(g) \cup C$, $h(g) \cup t(g)$, and $h(g) \cup t(g) \cup C$. There are $2^{|t(g)|} - 1$ rules derivable from a given group g, and the support of these four itemsets can be used to potentially eliminate them all from consideration. Note that if $h(g) \cup t(g) \cup C$ were frequent, then an algorithm such as Apriori would enumerate every derivable rule.

We have framed Theorem 5.2 in a manner in which it can be exploited even when the exact support information used above is not available. This is useful when we wish to prune a group before it is processed by using only previously gathered support information. For example, given an unprocessed group g, we cannot compute $\sup(h(g) \cup t(g) \cup \{\neg C\})$ to use for the value of y, but we can compute a lower-bound on the value. Given the parent node g_p of g, because $h(g_p) \cup t(g_p)$ is a superset of $h(g) \cup t(g)$, such a lower-bound is given by the observation below.

OBSERVATION 5.3: Given a group g and its parent g_p in the set-enumeration tree, $\sup(h(g_p) \cup t(g_p) \cup \{\neg C\}) \le \sup(h(g) \cup t(g) \cup \{\neg C\})$.

Conveniently, the support information required to apply this fact is immediately available from the candidate set of g_p.

In the following observation, we apply the support lower-bounding theorem from [7] to obtain another lower-bound on $\sup(h(g) \cup t(g) \cup \{\neg C\})$, again using only support information provided by the candidate set of g_p.

OBSERVATION 5.4: Given a group g and its parent g_p in the set-enumeration tree,

$$\sup(h(g) \cup \{\neg C\}) - \sum_{i \in t(g)} \sup(h(g_p) \cup \{\neg i, \neg C\}) \le \sup(h(g) \cup t(g) \cup \{\neg C\}).$$

When attempting to prune an unprocessed group, Dense-Miner computes both lower-bounds and uses the greater of the two for y in Theorem 5.2.

5.4 Bounding improvement

We propose two complementary methods to bound the improvement of any (frequent) rule derivable from a given group g. The first technique uses primarily the value of $\mathrm{uconf}(g)$ described above, and the second directly establishes an upper-bound on improvement from its definition. Dense-Miner computes $\mathrm{uimp}(g)$ by retaining the smaller of the two bounds provided by these techniques.

Bounding improvement using the confidence bound

The theorem below shows how to obtain an upper-bound on improvement by reusing the value of $\mathrm{uconf}(g)$ along with another value z no greater than the confidence of the sub-rule of $h(g)$ with the greatest confidence.

THEOREM 5.5: The value of $\mathrm{uconf}(g) - z$ where $z \le \max(\forall r \subseteq h(g), \mathrm{conf}(r))$ is an upper-bound on the improvement of any rule derivable from g.

Proof: Let r_s denote the sub-rule of $h(g)$ with the greatest confidence. Because r_s is a proper sub-rule of any rule r_d derivable from g, we know that $\mathrm{conf}(r_d) - \mathrm{conf}(r_s)$ is an upper-bound on $\mathrm{imp}(r_d)$. Because $\mathrm{uconf}(g) \ge \mathrm{conf}(r_d)$ and $z \le \mathrm{conf}(r_s)$, we have:

$$\begin{aligned} \mathrm{imp}(r_d) &\le \mathrm{conf}(r_d) - \mathrm{conf}(r_s) \\ &\le \mathrm{conf}(r_d) - z \\ &\le \mathrm{uconf}(g) - z\,. \ \square \end{aligned}$$

Dense-Miner uses the previously described method for computing $\mathrm{uconf}(g)$ when applying this result. Computing a tight value for z requires knowing the sub-rule r_s of $h(g)$ with the greatest confidence. Because r_s is not known, Dense-Miner instead sets z to the value of the following easily computed function:

$$f_z(g) = \max(f_z(g_p), \mathrm{conf}(h(g))) \text{ if } g \text{ has a parent } g_p,$$
$$f_z(g) = \mathrm{conf}(h(g)) \text{ otherwise.}$$

The fact that $f_z(g) \le \max(\forall r \subseteq h(g), \mathrm{conf}(r))$ follows from its definition. Its computation requires only the value of $f_z(g_p)$ where g_p is the parent of g, and the supports of $h(g)$ and $h(g) \cup C$ in order to compute $\mathrm{conf}(h(g))$. The value can be computed whether or not the group has been processed because this information can be obtained from the parent group.

Bounding improvement directly

A complementary method for bounding the improvement of any frequent rule derivable from g is provided by the next theorem. This technique exploits strong dependencies between head items.

THEOREM 5.6: The following expression provides an upper-bound on the improvement of any frequent rule derivable from a given group g:

$$\frac{x}{x+y} - \frac{x}{x+y+\beta}$$

where x, y and β are non-negative integers such that $y \le \sup(h(g) \cup t(g) \cup \{\neg C\})$, $\beta \ge \min(\forall i \in h(g), \sup((h(g) - \{i\}) \cup \{\neg C, \neg i\}))$, and $x = \min(\max(\sqrt{y^2 + y\beta}, \text{minsup}), \sup(h(g) \cup C))$

Proof sketch: For any frequent rule r derivable from g, note that $\text{imp}(r)$ can be written as:

$$\frac{x'}{x'+y'} - \frac{x'+\alpha'}{x'+y'+\alpha'+\beta'}$$

where the first term represents $\text{conf}(r)$ (as in Theorem 5.2) and the subtractive term represents the confidence of the proper sub-rule of r with the greatest confidence. To prove the claim, we show how to transform this expression into the expression from the theorem statement, arguing that the value of the expression never decreases as a result of each transformation.

To begin, let the subtractive term of the expression denote the confidence of r_s, a proper sub-rule of r such that $r_s = r - \{i_m\}$ where i_m denotes the item i from $h(g)$ that minimizes $\sup((h(g) - \{i\}) \cup \{\neg C, \neg i\})$. Since we can only decrease the value of the subtractive term by such a transformation, we have not decreased the value of the expression.

Now, given r and r_s, it is easy to show that $\alpha' \ge 0$, $y' \ge y$, and $\beta' \le \beta$. Because the expression is anti-monotone in α' and y' and monotone in β', we can replace α' with 0, β' with β, and y' with y without decreasing its value.

We are now left with an expression identical to the expression in the theorem, except for x' occurring in place of x. Taking the derivative of this expression with respect to x' and solving for 0 reveals it is maximized when $x' = \sqrt{y^2 + y\beta}$. Note that for any rule derivable from g, x' must fall between $\sup(h(g) \cup C)$ and minsup. Given this restriction on x', the equation is maximized at $x' = \min(\max(\sqrt{y^2 + y\beta}, \text{minsup}), \sup(h(g) \cup C)) = x$. We can therefore replace x' with x without decreasing its value. The resulting expression, identical to that in the theorem statement, is thus an upper-bound on $\text{imp}(r)$. □

To apply this result to prune a processed group g, Dense-Miner sets y to $\sup(h(g) \cup t(g) \cup \{\neg c\})$ since the required supports are known. Computing a tight value for β ($\sup((h(g) - i_m) \cup \{\neg i_m, \neg C\})$ where i_m is the item in $h(g)$ that minimizes this support value) is not possible given the support values available in the candidate set of g and its ancestors. Dense-Miner therefore sets β to an upper-bound on $\sup((h(g) - i_m) \cup \{\neg i_m, \neg C\})$ as computed by the following function:

$f_\beta(g) = \min(f_\beta(g_p), \sup(h(g_p) \cup \{\neg i, \neg C\}))$ when g has a parent g_p and where i denotes the single item within the itemset $h(g) - h(g_p)$,

$f_\beta(g) = \infty$ otherwise.

This computation requires only the value of $f_\beta(g_p)$ which was previously computed by the parent, and the supports of candidate set members $h(g)$, $h(g) \cup C$, $h(g_p)$, and $h(g_p) \cup C$ in order to compute $\sup(h(g_p) \cup \{\neg i, \neg C\})$.

In applying theorem 5.6 to prune an unprocessed group g, Dense-Miner computes β as above. For y, it lacks the necessary support information to compute $\sup(h(g) \cup t(g) \cup \{\neg C\})$, so instead it computes a lower-bound on the value as described in section 5.3.

There are a few interesting properties that should be noted about this particular bounding technique. First, because we incorporate the minsup parameter into the bounding function, it exploits both frequency and improvement constraints simultaneously, which provides more pruning power than exploiting each of them completely independently. Second, note that in special case where we have a rule for which $\beta = 0$, the resulting bound on improvement is always zero. Also note that if $\beta = 0$ for a given rule r, then $\beta = 0$ for any superset of r. In this case, the bound given by this technique is thus *anti-monotone* with respect to rule containment, which allows it to be straightforwardly exploited by algorithms such as Apriori. Unfortunately, the more common case where $\beta > 0$ does not give rise to an anti-monotone bound, so it cannot be exploited by an Apriori-like algorithm.

5.5 Bounding support

The value of $\mathrm{usup}(g)$ is comparatively easy to compute and exploit because support is anti-monotone with respect to rule containment. Any such anti-monotone rule value function requires we simply compute the value of that function on the rule corresponding to $h(g)$ in order to obtain an upper-bound. For $\mathrm{usup}(g)$, Dense-Miner thus uses the value of $\sup(h(g) \cup C)$. Other anti-monotone constraints, e.g. those discussed in [20], can be exploited by Dense-Miner with similar ease.

6. Item ordering

The motivation behind reordering tail items in the Generate-Next-Level function is to, in effect, force unpromising rules into the same portion of the search tree. The reason this strategy is critical is that in order for a group to be prunable, *every* sub-node of the group must represent a rule that fails to satisfy one or more of the constraints. An arbitrary ordering policy will result in a roughly even distribution of rules that satisfy the constraints throughout the search tree, yielding little pruning opportunities.

We experimented with several different ordering policies intended to tighten the bounds provided by the pruning functions. These policies included the obvious ones such as ordering tail items according to their support, rule support, and confidence, computed respectively as follows:

- $\sup(h(g) \cup \{i\})$;
- $\sup(h(g) \cup \{i\} \cup C)$; and

- $\sup(h(g) \cup \{i\} \cup C) / \sup(h(g) \cup \{i\})$).

We also tried several more obscure policies. The strategy we found to work best by a considerable margin exploits the fact that the computations for $\mathrm{uconf}(g)$ and $\mathrm{uimp}(g)$ both require a value $y \leq \sup(h(g) \cup t(g) \cup \{\neg C\})$, and the larger the value allowed for y, the tighter the resulting bound. The idea then is to reorder tail items so that many sub-nodes will have a large value for $\sup(h(g) \cup t(g) \cup \{\neg C\})$. This is achieved by positioning tail items which contribute to a large value of $\sup(h(g) \cup t(g) \cup \{\neg C\})$ last in the ordering, since tail items which appear deeper in the ordering will appear in more sub-nodes than those tail items appearing earlier. We have found that the tail items which contribute most to this value tend to be those with small values for $\sup(h(g) \cup \{\neg i, \neg C\})$. This can be seen from Observation 5.4 which yields a larger lower-bound on $\sup(h(g) \cup t(g) \cup \{\neg C\})$ when the value of $\sup(h(g) \cup \{\neg i, \neg C\})$ summed over every tail item is small. The policy used by Dense-Miner is therefore to arrange tail items in decreasing order of $\sup(h(g) \cup \{\neg i, \neg C\})$. Compared to a simple lexographic ordering of the items, this policy reduces runtime (and search tree size) by an order of magnitude or more when mining in highly dense data-sets such as those used in the upcoming evaluation section.

7. Post-processing

The fact that Dense-Miner finds all frequent, confident, large-improvement rules and places them into R follows from the completeness of a set-enumeration tree search and the correctness of our pruning rules, as established by the theorems from Section 5. Dense-Miner must still post-process R because it could contain some rules that do not have a large improvement.

Removing rules without a large improvement is non-trivial because improvement is defined in terms of all of the (exponentially many) proper sub-rules of a rule, and all such rules are not necessarily generated by the algorithm. A naive post-processor for removing rules without a large improvement might, for every mined rule, explicitly compute its improvement by generating and testing every proper sub-rule. Because Dense-Miner is capable of mining many long rules, such an approach would be too inefficient.

Instead, the post-processor first identifies some rules that do not have a large improvement by simply comparing them to the other rules in the mined rule set R. It compares each rule $r_1 \in R$ to every rule r_2 such that $r_2 \in R$ and $r_2 \subset r_1$. If ever it is found that $\mathrm{conf}(r_1) - \mathrm{conf}(r_2) < \mathrm{minimp}$, then rule r_1 is removed because its improvement is not large. This step alone requires no database access, and removes almost all rules that do not have a large improvement. Note that a hash-tree can be used to efficiently implement this step by indexing every rule in R in order to quickly identify all sub-rules of any given rule.

To remove any remaining rules, the post-processor performs a set-enumeration tree search for rules that could potentially prove some rule in R does not have a large improvement. The main difference between this procedure and the mining phase is in the pruning strategies applied. For this search problem, a group g is prunable when none of its derivable rules can prove that some rule in R lacks a large improvement. This is determined by either of the following conditions:

- There exists no rule $r \in R$ for which $h(g) \subset r$;
- $\mathrm{conf}(r) - \mathrm{uconf}(g) \geq \mathrm{minimp}$ for all rules $r \in R$ such that $h(g) \subset r$.

After groups are processed, the post-processor removes any rule r from R if there exists some group g such that $h(g) \subset r$ and $\mathrm{conf}(r) - \mathrm{conf}(h(g)) < \mathrm{minimp}$. Because the search explores the set of all rules that could potentially prove some rule in R does not have a large improvement, all rules without a large improvement are identified and removed.

Our post-processor includes some useful yet simple extensions of the above for ranking and facilitating the understanding of rules mined by Dense-Miner as well as other algorithms. The improvement of a rule is useful as an interestingness and ranking measure to be presented to the user along with confidence and support. It is also often useful to present the proper sub-rule responsible for a rule's improvement value. Therefore, given an arbitrary set of rules, our post-processor determines the exact improvement of every rule, and associates with every rule its proper sub-rule with the greatest confidence (whether or not this sub-rule is in the original rule set). In rule-sets that are not guaranteed to have high-improvement rules (such as those extracted from a decision tree), the sub-rules can be used to potentially simplify, improve the generality of, and improve the predictive ability of the originals.

To compute the exact improvement value of every rule in R, we must modify the post-processing strategy from above only slightly. First, each rule in r needs to maintain an upper-bound $\mathrm{imp\_bound}(r)$ on its improvement. This upper-bound is initialized to the value of $f_z(g)$ (from Section 5) where $h(g) = r$. Each time a rule r_{sub} is enumerated by the set-enumeration tree, the confidence of this rule is compared against the confidence of any rule r in R that is a superset of r_{sub}. If $\mathrm{conf}(r) - \mathrm{conf}(r_{\mathrm{sub}})$ is less than $\mathrm{imp\_bound}(r)$, then we make $\mathrm{imp\_bound}(r) = \mathrm{conf}(r) - \mathrm{conf}(r_{\mathrm{sub}})$. The pruning conditions given above must next be weakened so that a group is pruned if and only if it cannot possibly lead to a rule which will affect the value of $\mathrm{imp\_bound}(r)$ for some rule r in R. These conditions are as follows:

- There exists no rule $r \in R$ for which $h(g) \subset r$;
- $\mathrm{conf}(r) - \mathrm{uconf}(g) \geq \mathrm{imp\_bound}(r)$ for all rules $r \in R$ such that $h(g) \subset r$.

A rule r is removed from R whenever $\mathrm{imp\_bound}(r) < \mathrm{minimp}$. To maintain the proper sub-rule responsible for the improvement value of a rule r, one simply has to maintain a pointer to the rule r_{sub} which most recently caused a modification of $\mathrm{imp\_bound}(r)$.

Though post-processing involves scanning the database, we have found this phase accounts for at most 15% of the total execution time of our algorithm (whether or not we compute the exact improvement value for each rule). The reason is that the set of rules R and the initial bounds computed on the improvement of each rule in R strongly constrain the search space.

8. Tree-Traversal Strategies

While Dense-Miner is described in Section 4 as performing a breadth-first search of the tree, any tree-traversal policy is compatible with the techniques proposed up until this point. This is because we have described how each node in the tree can be processed and pruned using only local information, and/or information provided by its ancestors in the tree. In this section we describe optimizations that become possible if we restrict ourselves to a particular tree-traversal strategy. These optimizations can then be mixed and matched if hybrid traversal strategies are used.

8.1 Breadth-First Optimizations

A breadth-first traversal strategy is often appropriate since it limits the number of database passes that must be performed to one for each level of the tree. Since the tree-height is usually well under 20, a breadth-first traversal strategy is very accommodating of large data-sets and access of the data through cursors provided by a database engine. Given a breadth-first traversal strategy, a mining algorithm can exploit several of what we call “cross-tree” optimizations. The idea is to use not only information provided by ancestors of a given node, but also information provided by nodes in other branches of the tree. As an example, consider the Apriori algorithm which generates a candidate k-itemset if and only if every one of its $(k-1)$-item subsets is known to be frequent. This pruning technique can be described in the set-enumeration tree framework only if we make use of a cross-tree optimization that searches across branches for each of the $(k-1)$-item subsets of a particular k-item candidate.

In the context of Dense-Miner, we could employ a cross-tree support-pruning optimization that, for each tail item i in a group g, checks to see if $(h(g) - \{u\}) \cup \{i\}$ for each $u \in h(g)$ is infrequent. Such information is often provided by the candidate set of some other node at the same level of g in the set-enumeration tree. Since support is such a weak constraint in dense data-sets, this particular optimization has negligible benefits. However, other cross-tree optimizations with stronger effects are possible. For example, consider the value β from Section 4 which we use when pruning with the improvement constraint.

$$\beta \geq \min(\forall i \in h(g), \sup((h(g) - \{i\}) \cup \{\neg C, \neg i\}))$$

Note that this value depends only on the items within the head of a candidate group, and further, this value is anti-monotone with set containment. A value computed for β for some group g_1 can thus be used by another group g_2 if $h(g_1) \subset h(g_2)$. Note that we have a similar case for the value z in Theorem 5.5 which is monotone with set containment since it reflects the best-known confidence of any sub-rule of $h(g)$. The cross-tree optimization then is as follows: when generating a new group g, find all groups g_s in the current level of the tree such that $h(g_s) \subset h(g)$. For β, keep the minimum of $f_\beta(g)$ and $f_\beta(g_s)$ over all such groups g_s, and for z, keep the maximum of $f_z(g)$ and $f_z(g_s)$ over all such groups.

8.2 Depth-First Optimizations

A depth-first traversal also allows for different cross-tree optimizations. For example, one can be used to get a better value for y in the pruning functions from Section 4, where $y \le \sup(h(g) \cup t(g) \cup \{\neg C\})$. Quite often, given two siblings g_1 and g_2 in the set-enumeration tree, we have that $h(g_2) \cup t(g_2) \subset t(g_1) \cup t(g_2)$. Thus, if a depth-first traversal descends and processes node g_1 before g_2, we can use the value of $\sup(h(g_1) \cup t(g_1) \cup \{\neg C\})$ to potentially obtain a tighter bound on y in order to try and prune g_2 before it is processed.

A pure depth-first traversal of the set-enumeration tree requires a pass over the data-set for each node in the tree. While this drawback seems difficult to overcome, if we are willing to cache "projections" of the data-set, a depth-first strategy can sometimes be advantageous. This idea was originally suggested in [1] in the context of mining frequent itemsets, though it also applies here with only simple modifications. The idea is based on the following fact: in order to process a node g, we need only consider those transactions of the data-set that contain $h(g)$. By virtue of our candidate generation procedure, the transactions required to process a node g are a subset of the transactions required to process its parent g_p in the tree. Exploiting this fact, an algorithm can progressively filter down the data-set as it descends the tree by projecting out any transaction that fails to contain $h(g)$ while processing node g. This optimization is beneficial not only because far fewer records need to be considered when processing a node (particularly at the deeper levels), but also because we know that each transaction already contains all items in the head of $h(g)$ except one item i (the item contained within the set $h(g) - h(g_p)$). Thus, there is no need to test for the presence of these items when counting support. We only test for the presence of item i, and then test for each of the tail items exactly as before. If i is present, then the transaction is placed into a new cache that will be provided to the children of the current node. Additionally, item i can be removed from these transactions since its presence will be assumed.

While this transaction projection optimization can be used for any traversal strategy, because a subset of the data-set must be cached with each open node, it is only practical when the set of open nodes is small. In a depth-first strategy, the number of open nodes is bounded by the height of the tree h, which results in an $O(h|D|)$ bound on the number of cached transactions. Further, since support usually drops significantly with each level, this worst-case estimate is extremely conservative. Note also that caching need not be performed exclusively in main memory. Each of the caches, since they are produced as well as accessed in sequential order, could be stored on disk without complication.

9. Evaluation

This section provides an evaluation of Dense-Miner using two real-world data-sets which were found to be particularly dense in [7].[1] The first data-set is compiled from PUMS census data obtained from http://augustus.csscr.washington.edu/census/comp_013.html. It consists of 49,046 transactions with 74 items per transaction. Each transaction represents the answers to a census questionnaire, including the age, tax-filing status, marital status, income, sex, veteran status, and location of residence of the respondent. Similar data-sets are used in targeted marketing campaigns for identifying a population likely to respond to a particular promotion. Continuous attributes were discretized as described in [10], though no frequently occurring items were discarded. The second data-set is the connect-4 data-set from the Irvine machine learning database repository (http://www.ics.uci.edu/~mlearn/MLRepository.html). It consists of 67,557 transactions and 43 items per transaction. This data-set is interesting because of its size, density, and a minority consequent item ("tie games") that is accurately predicted only by rules with low support. All experiments presented here use the "unmarried partner" item as the consequent with the pums data-set, and the "tie games" item with the connect-4 data-set; we have found that using other consequents consistently yields qualitatively similar results.

The implementation of Dense-Miner used here does not exploit any of the traversal-specific optimizations suggested in Section 8 in order to provide a traversal-neutral assessment of its performance. Preliminary experimental results suggest that breadth-first cross-tree optimizations can improve on the reported runtimes by a factor of two at the lower support values. We have not yet experimented with any of the depth-first optimizations. Execution times are reported in seconds on an IBM IntelliStation M Pro running Windows NT with a 400 MHZ Intel Pentium II Processor and 128MB of SDRAM. Execution time includes runtime for both the mining and post-processing phases. The post-processor is the one which computes the exact improvement value of every rule in the result.

The minsup setting used in the experiments is specified as a value we call *minimum coverage*, where $\text{minimum coverage} = \text{minsup} / \text{sup}(C)$. In the context of consequent constrained association rule mining, minimum coverage is more intuitive than minimum support, since it specifies the smallest fraction of the population of interest that must be characterized by each mined rule.

[1] Both data-sets are available in the form used in these experiments from http://www.almaden.ibm.com/cs/quest.

9.1 Effects of minimum improvement

The first experiment (Figure 5) shows the effect of different minimp settings as minsup is varied. Minconf in these experiments is left unspecified, which disables pruning with the minimum confidence constraint. The graphs of the figure plot execution time and the number of rules returned for several algorithms at various settings of minimum support. Dense-miner is run with minimp settings of .0002, .002, and .02 (dense_0002, dense_002, and dense_02 respectively). We compare its performance to that of the Apriori algorithm optimized to exploit the consequent constraint (apriori_c). This algorithm materializes only those frequent itemsets that contain the consequent itemset.

The first row of graphs from the figure reveals that apriori_c is too slow on all but the greatest settings of minsup for both data-sets. In contrast, very modest settings of minimp allow Dense-Miner to efficiently mine rules at far lower supports, even without exploiting the minconf constraint. A natural question is whether mining at low supports is necessary. For these data-sets, the answer is yes simply because rules with confidence significantly higher than the consequent frequency do not arise unless minimum coverage is below 20%. This can be seen from Figure 7, which plots the confidence of the best rule meeting the minimum support constraint for any given setting.[1] This property is typical of data-sets from domains such as targeted marketing, where response rates tend to be low without focusing on a small but specific subset of the population.

The graphs in the second row of Figure 5 plot the number of rules satisfying the input constraints. Note that runtime correlates strongly with the number of rules returned for each algorithm. For apriori_c, the number of rules returned is the same as the number of frequent itemsets containing the consequent because there is no minconf constraint specified. Modest settings of minimp dramatically reduce the number of rules returned because most rules in these data-sets offer only insignificant (if any) predictive advantages over their proper sub-rules. This effect is particularly pronounced on the pums data-set, where a minimp setting of .0002 is too weak a constraint to keep the number of such rules from exploding as support is lowered. The increase in runtime and rule-set size as support is lowered is far more subdued given the larger (though still small) minimp settings.

9.2 Effects of minimum confidence

The next experiment (Figure 6) shows the effect of varying minconf while fixing minimp and minsup to very low values. With connect-4, we used a minimum coverage of 1%, and with pums, a minimum coverage of 5%. Minimp was set to .0002 with both data-sets. As can be extrapolated from the previous figures, the number of rules meeting these weak minimp and min-

---

[1] The data for this figure was generated by a version of Dense-Miner that prunes any group that cannot lead to a rule on the depicted support/confidence border [8]. This optimization criteria is enforced during mining using the confidence and support bounding techniques from section 5.

FIGURE 5. Execution time and rules returned versus minimum coverage for the various algorithms.

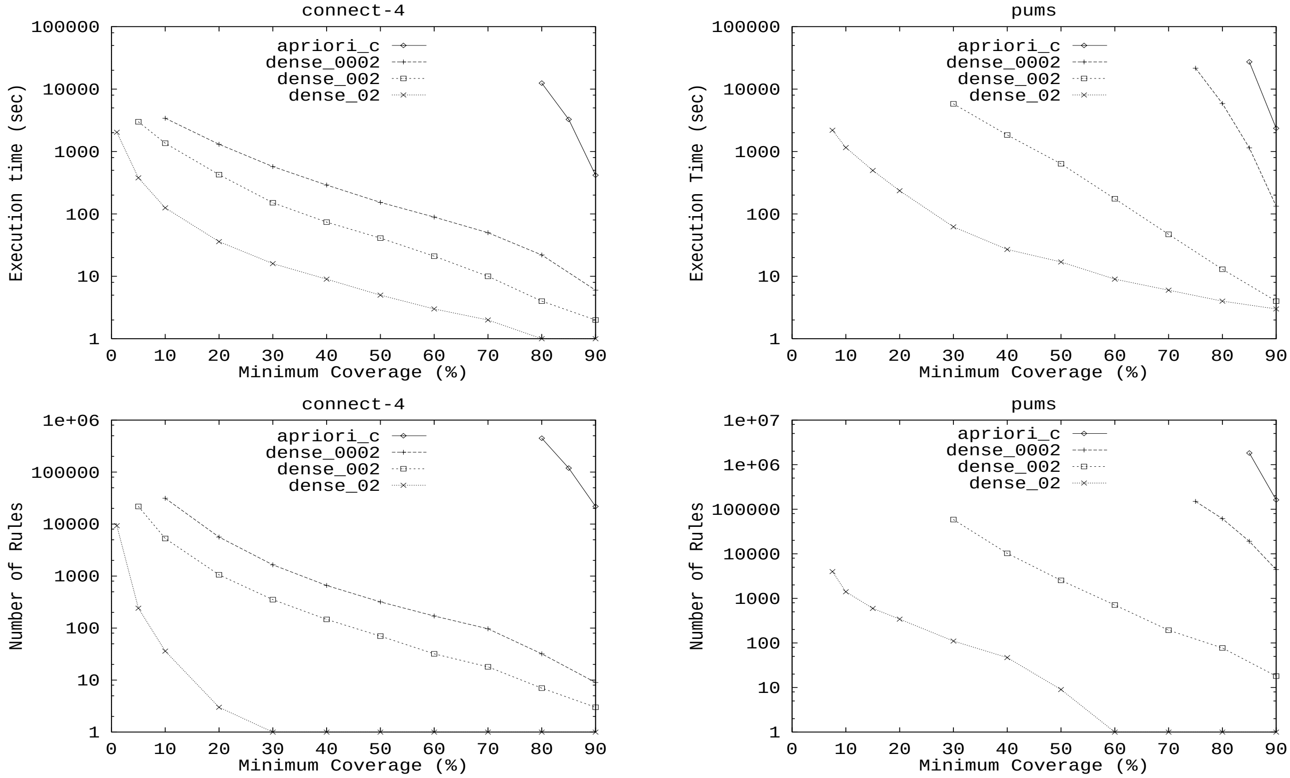


FIGURE 6. Execution time of dense_0002 as minconf is varied for both data-sets. Minimum coverage is fixed at 5% on pums and 1% on connect-4.

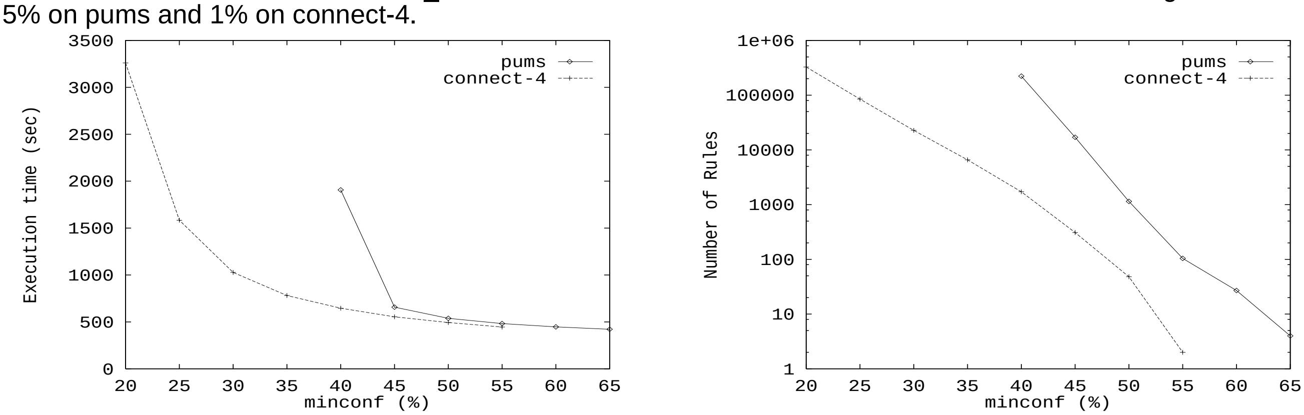
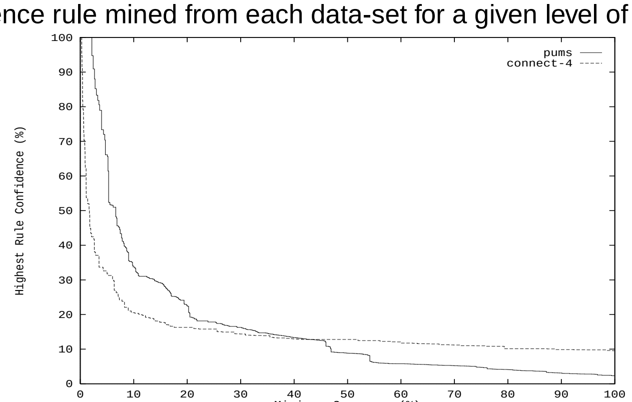


FIGURE 7. Maximum confidence rule mined from each data-set for a given level of minimum coverage.



sup constraints would be enormous. As a result, with these constraints alone and no minimum confidence specification, Dense-Miner exceeds the available memory of our machine.

The efficiency of Dense-Miner when minimum confidence is specified shows that it is effectively exploiting the confidence constraint to prune the set of rules explored. We were unable to use lower settings of minconf than those plotted because of the large number of rules. As minconf is increased beyond the point at which fewer than 100,000 rules are returned, the run-time of Dense-Miner rapidly falls to around 500 seconds on both data-sets.

9.3 Summary of experimental findings

These experiments demonstrate that Dense-Miner, in contrast to approaches based on finding frequent itemsets, achieves good performance on highly dense data even when the input constraints are set conservatively. Minsup can be set low (which is necessary to find high confidence rules), as can minimp and minconf (if it is set at all). This characteristic of our algorithm is important for the end-user who may not know how to set these parameters properly. Low default values can be automatically specified by the system so that all potentially useful rules are produced. Refinements of the default settings can then be made by the user to tailor this result. In general, the execution time required by Dense-Miner correlates strongly with the number of rules that satisfy all of the specified constraints.

10. Conclusions

We have shown how Dense-Miner exploits rule constraints to efficiently mine consequent-constrained rules from large and dense data-sets, even at low supports. Unlike previous approaches, Dense-Miner exploits constraints such as minimum confidence (or alternatively, minimum lift or conviction) and a new constraint called minimum improvement during the mining phase. The minimum improvement constraint prunes any rule that does not offer a significant predictive advantage over its proper sub-rules. This increases efficiency of the algorithm, but more importantly, it presents the user with a concise set of predictive rules that are easy to comprehend because every condition of each rule strongly contributes to its predictive ability.

The primary contribution of Dense-Miner with respect to its implementation is its search-space pruning strategy which consists of the three critical components: (1) functions that allow the algorithm to flexibly compute bounds on confidence, improvement, and support of any rule derivable from a given node in the search tree; (2) approaches for reusing support information gathered during previous database passes within these functions to allow pruning of nodes before they are processed; and (3) an item-ordering heuristic that ensures there are plenty of pruning opportunities. In principle, these ideas can be retargeted to exploit other constraints in place of or in addition to those already described.

We lastly described a rule post-processor that Dense-Miner uses to fully enforce the minimum improvement constraint. This post-processor is useful on its own for determining the improvement value of every rule in an arbitrary set of rules, as well as associating with each rule its proper sub-rule with the highest confidence. Improvement can then be used to rank the rules, and the sub-rules used to potentially simplify, generalize, and improve the predictive ability of the original rule set.

References

[1] Agarwal, R.; Aggarwal, C.; Prasad, V. V. V.; and Crestana, V. 1998. *A Tree Projection Algorithm for Generation of Large Itemsets for Association Rules*. IBM Research Report RC21341, Nov, 1998.

[2] Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Mining Associations between Sets of Items in Massive Databases. In *Proc. of the 1993 ACM-SIGMOD Int'l Conf. on Management of Data*, 207-216.

[3] Agrawal, R.; Mannila, H.; Srikant, R.; Toivonen, H.; and Verkamo, A. I. 1996. Fast Discovery of Association Rules. In *Advances in Knowledge Discovery and Data Mining*, AAAI Press, 307-328.

[4] Agrawal, R., and Srikant, R. 1994. *Fast Algorithms for Mining Association Rules.* IBM Research Report RJ9839, June 1994. IBM Almaden Research Center, San Jose, CA.

[5] Ali, K.; Manganaris, S.; and Srikant, R. 1997. Partial Classification using Association Rules. In *Proc. of the 3rd Int'l Conference on Knowledge Discovery in Databases and Data Mining*, 115-118.

[6] Bayardo, R. J. 1997. Brute-Force Mining of High-Confidence Classification Rules. In *Proc. of the Third Int'l Conf. on Knowledge Discovery and Data Mining*, 123-126.

[7] Bayardo, R. J. 1998. Efficiently Mining Long Patterns from Databases. In *Proc. of the 1998 ACM-SIGMOD Int'l Conf. on Management of Data*, 85-93.

[8] Bayardo, R. J. and Agrawal, R. 1999. Mining the Most Interesting Rules. In *Proc. of the ACM SIGKDD Conf. on Knowledge Discovery and Data Mining*, to appear.

[9] Berry, Michael J. A. and Linoff G. S. 1997. *Data Mining Techniques for Marketing, Sales and Customer Support*, John Wiley & Sons, Inc.

[10] Brin, S.; Motwani, R.; Ullman, J.; and Tsur, S. 1997. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proc. of the 1997 ACM-SIGMOD Int'l Conf. on the Management of Data*, 255-264.

[11] Clearwater, S. H., & Provost, F. J. 1990. RL4: A tool for knowledge-based induction. In *Proc. of the Second Int'l IEEE Conf. on Tools for Artificial Intelligence*, 24-30.

[12] Cohen, W. W. 1995. Fast Effective Rule Induction. In *Proc. of the 12th Int'l Conf. on Machine Learning*, 115-123.

[13] Dhar, V. and Tuzhilin, A. 1993. Abstract-driven pattern discovery in databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(6).

[14] Gunopulos, G.; Mannila, H.; and Saluja, S. 1997. Discovering All Most Specific Sentences by Randomized Algorithms. In *Proc. of the 6th Int'l Conf. on Database Theory*, 215-229.

[15] International Business Machines, 1996. *IBM Intelligent Miner User's Guide*, Version 1, Release 1.

[16] Klemettinen, M.; Mannila, P.; Ronkainen, P.; and Verkamo, A. I. 1994. Finding Interesting Rules from Large Sets of Discovered Association Rules. In *Proc. of the Third Int'l Conf. on Information and Knowledge Management*, 401-407.

[17] Lin, D.-I and Kedem, Z. M. 1998. Pincer-Search: A New Algorithm for Discovering the Maximum Frequent Set. In *Proc. of the Sixth European Conf. on Extending Database Technology*, 105-119.

[18] Liu, B.; Hsu, W.; and Ma, Y. 1998. Integrating Classification and Association Rule Mining. In *Proc. of the Fourth Int'l Conf. on Knowledge Discovery and Data Mining*, 80-86.

[19] Murphy, P. and Pazzani, M. 1994. Exploring the decision forest: An empirical investigation of Occam's Razor in decision tree induction. In *J. of Artificial Intelligence Research*, 1, 257-275.

[20] Ng, R. T.; Lakshmanan, V. S.; Han, J.; and Pang, A. 1998. Exploratory Mining and Pruning Optimizations of Constrained Association Rules. In *Proc of the 1998 ACM-SIGMOD Int'l Conf. on the Management of Data*, 13-24.

[21] Park, J. S.; Chen, M.-S.; and Yu, P. S. 1996. An Effective Hash Based Algorithm for Mining Association Rules. In *Proc. of the 1995 SIGMOD Conf. on the Management of Data*, 175-186.

[22] Rymon, R. 1992. Search through Systematic Set Enumeration. In *Proc. of Third Int'l Conf. on Principles of Knowledge Representation and Reasoning*, 539-550.

[23] Rymon, R. 1994. On Kernel Rules and Prime Implicants. In *Proc. of the Twelfth Nat'l Conf. on Artificial Intelligence*, 181-186.

[24] Savasere, A.; Omiecinski, E.; and Navathe, S. 1995. An Efficient Algorithm for Mining Association Rules in Large Databases. In *Proc. of the 21st Conf. on Very Large Data-Bases*, 432-444.

[25] Segal, R., & Etzioni, O. 1994. Learning decision lists using homogeneous rules. In *Proc. of the Twelfth Nat'l Conf. on Artificial Intelligence*, 619-625.

[26] Schlimmer, J. C. 1993. Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In *Proc. of the Tenth. Int'l Conf. on Machine Learning*, 284-290.

[27] Shafer, J.; Agrawal, R.; and Mehta, M. 1996. SPRINT: A Scalable Parallel Classifier for Data-Mining. In *Proc. of the 22nd Conf. on Very Large Data-Bases*, 544-555.

[28] Smythe, P. and Goodman, R. M. 1992. An Information Theoretic Approach to Rule Induction from Databases. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):301-316.

[29] Srikant, R.; Vu, Q.; and Agrawal, R. 1997. Mining Association Rules with Item Constraints. In *Proc. of the Third Int'l Conf. on Knowledge Discovery in Databases and Data Mining*, 67-73.

[30] Webb, G. I. 1995. OPUS: An Efficient Admissible Algorithm for Unordered Search. In *Journal of Artificial Intelligence Research*, 3:431-465.

[31] Zaki, M. J.; Parthasarathy, S.; Ogihara, M.; and Li, W. 1997. New Algorithms for Fast Discovery of Association Rules. In *Proc. of the Third Int'l Conf. on Knowledge Discovery in Databases and Data Mining*, 283-286.