Analysis of Data Mining Algorithms

Final Project for Advanced Algorithms

by

Karuna Pande Joshi
March 1997

Table of Contents

1. Introduction

With an enormous amount of data stored in databases and data warehouses, it is increasingly important to develop powerful tools for analysis of such data and mining interesting knowledge from it. Data mining is a process of inferring knowledge from such huge data. Data Mining has three major components Clustering or *Classification*, *Association Rules* and *Sequence Analysis*.

By simple definition, in classification/clustering we analyze a set of data and generate a set of grouping rules which can be used to classify future data. For example, one may classify diseases and provide the symptoms which describe each class or subclass. This has much in common with traditional work in statistics and machine learning. However, there are important new issues which arise because of the sheer size of the data. One of the important problem in data mining is the Classification-rule learning which involves finding rules that partition given data into predefined classes. In the data mining domain where millions of records and a large number of attributes are involved, the execution time of existing algorithms can become prohibitive, particularly in interactive applications. This is discussed in detail in *Chapter 2*.

An association rule is a rule which implies certain association relationships among a set of objects in a database. In this process we discover a set of association rules (in the form of ``$A_1 \wedge \ldots \wedge A\ i \rightarrow B_1 \wedge \ldots \wedge B\ j$") at multiple levels of abstraction from the relevant set(s) of data in a database. For example, one may discover a set of symptoms often occurring together with certain kinds of diseases and further study the reasons behind them. Since finding interesting association rules in databases may disclose some useful patterns for decision support, selective marketing, financial forecast, medical diagnosis, and many other applications, it has attracted a lot of attention in recent data mining research . Mining association rules may require iterative scanning of large transaction or relational databases which is quite costly in processing. Therefore, efficient mining of association rules in transaction and/or relational databases has been studied substantially. This is discussed in detail in *Chapter 3*.

In *sequential Analysis*, we seek to discover patterns that occur in sequence. This deals with data that appear in separate transactions (as opposed to data that appear in the same transaction in the case of association).For e.g. : If a shopper buys item A in the first week of the month, then s/he buys item B in the second week etc. This is discussed in detail in *Chapter 4*.

There are many algorithms proposed that try to address the above aspects of data mining. Compiling a list of all algorithms suggested/used for these problems is an arduous task . I have thus limited the focus of this report to list only some of the algorithms that have had better success than the others. I have included a list of URLs in Appendix A which can be referred to for more information on data mining algorithms.

2. Classification Algorithms

In Data classification one develops a description or model for each class in a database, based on the features present in a set of class-labeled training data. There have been many data classification methods studied, including decision-tree methods, such as C4.5, statistical methods, neural networks, rough sets, database-oriented methods etc.

2.1 Data Classification Methods

In this paper, I have discussed in detail some of the *machine-learning* algorithms that have been successfully applied in the initial stages of this field. The other methods listed above are just being applied to data mining and have not been very successful. This section briefly describes these other methods. Appendix A lists the URLs which can be referred to for more information on these various methods.

- **Statistical Algorithms** Statistical analysis systems such as SAS and SPSS have been used by analysts to detect unusual patterns and explain patterns using statistical models such as linear models. Such systems have their place and will continue to be used.
- **Neural Networks** Artificial neural networks mimic the pattern-finding capacity of the human brain and hence some researchers have suggested applying Neural Network algorithms to pattern-mapping. Neural networks have been applied successfully in a few applications that involve classification.
- **Genetic algorithms** Optimization techniques that use processes such as genetic combination, mutation, and natural selection in a design based on the concepts of natural evolution.
- **Nearest neighbor method** A technique that classifies each record in a dataset based on a combination of the classes of the k record(s) most similar to it in a historical dataset. Sometimes called the k-nearest neighbor technique.
- **Rule induction** The extraction of useful *if-then* rules from data based on statistical significance.
- **Data visualization** The visual interpretation of complex relationships in multidimensional data.

2.2 Data Abstraction

Many existing algorithms suggest abstracting the test data before classifying it into various classes. There are several alternatives for doing abstraction before classification: A data set can be generalized to either a minimally generalized abstraction level, an intermediate abstraction level, or a rather high abstraction level. Too low an abstraction level may result in scattered classes, bushy classification trees, and difficulty at concise semantic interpretation; whereas too high a level may result in the loss of classification accuracy. The generalization-based multi-level classification process has been implemented in the DB- Miner system.[4]

2.3 Classification-rule learning.

Classification-rule learning involves finding rules or decision trees that partition given data into predefined classes. For any realistic problem domain of the classification-rule learning, the set of possible decision trees is too large to be searched exhaustively. In fact, the computational complexity of finding an optimal classification decision tree is NP hard.

Most of the existing induction-based algorithms use **Hunt**'s method as the basic algorithm.[2] Here is a recursive description of Hunt's method for constructing a decision tree from a set T of training cases with classes denoted $\{C_1, C_2, \ldots, C_k\}$.

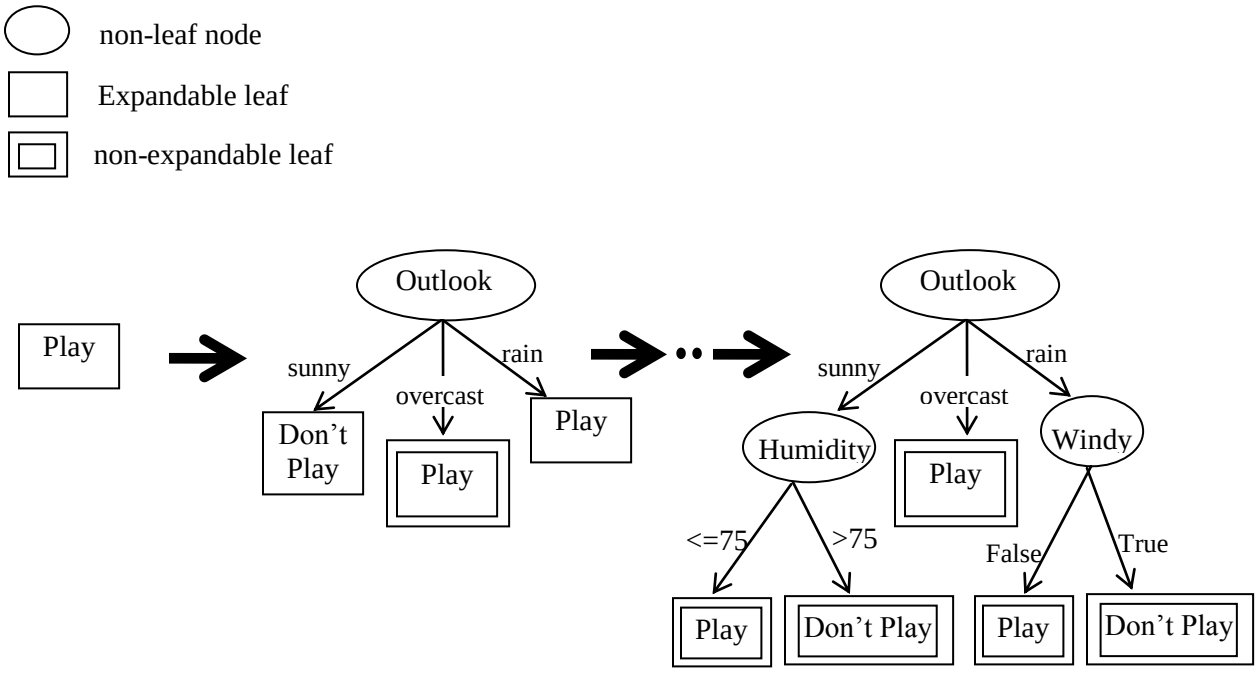
Case 1 T contains one or more cases, all belonging to a single class C_j : The decision tree for T is a leaf identifying class C_j .

Case 2 T contains no cases: The decision tree for T is a leaf, but the class to be associated with the leaf must be determined from information other than T.

Case 3 T contains cases that belong to a mixture of classes: A test is chosen, based on a single attribute, that has one or more mutually exclusive outcomes $\{O_1, O_2, .., O_n\}$. T is partitioned into subsets $T_1, T_2, \ldots, T_n$, where T_i contains all the cases in T that have outcome O_i of the chosen test. The decision tree for T consists of a decision node identifying the test, and one branch for each possible outcome. The same tree building machinery is applied recursively to each subset of training cases.

Outlook	Temp(F)	Humidity(%)	Windy?	Class
sunny	75	70	true	Play
sunny	80	90	true	Don't Play
sunny	85	85	false	Don't Play
sunny	72	95	false	Don't Play
sunny	69	70	false	Play
overcast	72	90	true	Play
overcast	83	78	false	Play
overcast	64	65	true	Play
overcast	81	75	false	Play
rain	71	80	true	Don't Play
rain	65	70	true	Don't Play
rain	75	80	false	Play
rain	68	80	false	Play
rain	70	96	false	Play

Table 2.1: A small training data set [2]



(a)Initial Classification Tree

(b)Intermediate Classification Tree

(c) Final Classification Tree

Figure 2.1: Demonstration of Hunt's Method

Attribute Value	**Class**	
	Play	**Don't Play**
sunny	2	3
overcast	4	0
rain	3	2

Table 2.2: Class Distribution Information of Attribute Outlook

Attribute Value	Binary Test	Class	
		Play	Don't Play
65	≤ >	1 8	0 5
70	≤ >	3 6	1 4
75	≤ >	4 5	1 4
78	≤ >	5 4	1 4
80	≤ >	7 2	2 3
85	≤ >	7 2	3 2
90	≤ >	8 1	4 1
95	≤ >	8 1	5 0
96	≤ >	9 0	5 0

Table 2.3: Class Distribution Information of Attribute Humidity

Table 2.1 shows a training data set with four data attributes and two classes. Figure 2.1 shows how the Hunt's method works with the training data set. In the case 3 of the Hunt's method, a test based on a single attribute is chosen for expanding the current node. The choice of an attribute is normally based on the entropy gains of the attributes. The entropy of an attribute is calculated from class distribution information. For a discrete attribute, class distribution information of each value of the attribute is required. Table 2.2 shows the class distribution information of data attribute Outlook. For a continuous attribute, binary test involving all the distinct values of the attribute is considered. Table 2.3 shows the class distribution information of data attribute Humidity. Once the class distribution information of all the attributes are gathered, the entropy is calculated based on either information theory or Gini Index. One attribute with the most entropy gain is selected as a test for the node expansion.

2.3.1 ID3 algorithm

The ID3 algorithm (**Quinlan**86) is a decision tree building algorithm which determines the classification of objects by testing the values of the their properties. It builds the tree in a top down fashion, starting from a set of objects and a specification of properties. At each node of the tree, a property is tested and the results used to partition the object set. This process is recursively done till the set in a given subtree is homogeneous with respect to the classification criteria – in other words it contains objects belonging to the same category. This then becomes a leaf node. At each node, the property to test is chosen based on information theoretic criteria that

seek to maximize information gain and minimize entropy. In simpler terms, that property is tested which divides the candidate set in the most homogeneous subsets.

2.3.2 C4.5 algorithm

This algorithm was proposed by **Quinlan** (1993). The C4.5 algorithm generates a classification-decision tree for the given data-set by recursive partitioning of data. The decision is grown using **Depth-first** strategy. The algorithm considers all the possible tests that can split the data set and selects a test that gives the best information gain. For each discrete attribute, one test with outcomes as many as the number of distinct values of the attribute is considered. For each continuous attribute, binary tests involving every distinct values of the attribute are considered. In order to gather the entropy gain of all these binary tests efficiently, the training data set belonging to the node in consideration is sorted for the values of the continuous attribute and the entropy gains of the binary cut based on each distinct values are calculated in one scan of the sorted data. This process is repeated for each continuous attributes.

2.3.3 SLIQ algorithm

SLIQ (Supervised Learning In Quest) developed by IBM's Quest project team, is a decision tree classifier designed to classify large training data [1]. It uses a pre-sorting technique in the tree-growth phase. This helps avoid costly sorting at each node. SLIQ keeps a separate sorted list for each continuous attribute and a separate list called class list. An entry in the class list corresponds to a data item, and has a class label and name of the node it belongs in the decision tree. An entry in the sorted attribute list has an attribute value and the index of data item in the class list. SLIQ grows the decision tree in **breadth-first** manner. For each attribute, it scans the corresponding sorted list and calculate entropy values of each distinct values of all the nodes in the frontier of the decision tree simultaneously. After the entropy values have been calculated for each attribute, one attribute is chosen for a split for each nodes in the current frontier, and they are expanded to have a new frontier. Then one more scan of the sorted attribute list is performed to update the class list for the new nodes.

While SLIQ handles disk-resident data that are too large to fit in memory, it still requires some information to stay memory-resident which grows in direct proportion to the number of input records, putting a hard-limit on the size of training data. The Quest team has recently designed a new decision-tree-based classification algorithm, called SPRINT (**S**calable **P**a**R**allelizable **IN**duction of decision **T**rees) that for the removes all of the memory restrictions.

2.3.4 Other Algorithms

There are many other machine learning algorithms , discussing all of them is outside the scope of this paper. Some of these algorithms are listed below [8].

Nearest-neighbor The classical nearest-neighbor with options for weight setting, normalizations, and editing (Dasarathy 1990, Aha 1992, Wettschereck 1994).

Naive-Bayes A simple induction algorithm that assumes a conditional independence model of attributes given the label (Domingos & Pazzani 1996, Langley, Iba & Thompson 1992, Duda & Hart 1973, Good 1965).
OODG Oblivous read-Once Decision Graph induction algorithm described in Kohavi (1995c).
Lazy decision trees An algorithm for building the ``best'' decision tree for every test instance described in Friedman, Kohavi & Yun (1996).
Decision Table A simple lookup table. A simple algorithm that is useful with feature subset selection

2.4 Parallel Algorithms

Most of the existing algorithms, use local heuristics to handle the computational complexity. The computational complexity of these algorithms ranges from O(AN logN) to $O(AN(\log N)^2)$ with N training data items and A attributes. These algorithms are fast enough for application domains where N is relatively small. However, in the data mining domain where millions of records and a large number of attributes are involved, the execution time of these algorithms can become prohibitive, particularly in interactive applications. Parallel algorithms have been suggested by many groups developing data mining algorithms. We discuss below two approaches that have been used.[2]

2.4.1 Basic Idea:

Initially N training data items are randomly distributed to P processors such that each processors has N=P data items. At this point, all the processors cooperate to expand the root node of a decision tree. For this, processors need to decide on an attribute to use to generate child nodes of the root. This can be done in three steps. In the first step, every processor collects the class distribution information of the local data. In the second step, the processors exchange the local class distribution information using global reduction. Finally, each processor can simultaneously compute entropy gains of the attributes and find the best attribute for splitting the root node.
There are two approaches for further progress. In *Synchronous Tree Construction Approach*, the entire set of processors synchronously expand one node of the decision tree at a time. In *partitioned Tree Construction Approach*, each new generated node is expanded by a subset of processors that helped the expansion of the parent node. These are discussed below .

2.4.2 Synchronous Tree Construction Approach

In this approach, all processors construct a decision tree synchronously by sending and receiving class distribution information of local data.
Major steps for the approach are shown below:

1. Select a node to expand according to a decision tree expansion strategy (eg. Depth-First, Breadth-First or Best-First), and call that node as the current node.
2. For each data attribute, collect class distribution information of the local data at the current node.
3. Exchange the class distribution information with all other processors and add up the class distribution information to get a complete distribution of all the attributes.

4. Calculate the entropy gains of each attribute and select the best attribute for child node expansion.
5. Based on the attribute values, create child nodes and partition the data according to the values of the attribute chosen.
6. Repeat the above steps (1--5) until no more nodes are available for the expansion.

The advantage of this approach is that it does not require any movement of the training data items. However, this algorithm suffers from high communication cost and load imbalance.

Load imbalance can be reduced if all the nodes on the frontier are expanded simultaneously, i.e. one pass of all the data at each processor is used to compute the class distribution information for all nodes on the frontier. Note that this improvement also reduces the number of times communications are done and reduces the message start-up overhead, but it does not reduce the overall volume of communications. Now the only source of load imbalance is when some leaf nodes become terminal nodes. This load imbalance can be further minimized if the training data set is distributed randomly.

2.4.3 Partitioned Tree Construction Approach

In this approach, each leaf node n of the frontier of the decision tree is handled by a distinct subset of processors $P_{(n)}$. Once the node n is expanded into child nodes, n_1, n_2,…,n_k, the processor group $P_{(n)}$ is also partitioned into k parts, P_1, P_2,…,P_k , such that P_i handle node n_i . All the data items are shuffled such that the processors in group P_i have data items that belong to the leaf n_i only. Major steps for the approach is shown below:

1. Expand a node based on the method discussed in the beginning of the Section 3 for expanding the root node.
2. (a) If the number of leaf nodes is less than $|P_{(n)}|$,
    i. Assign a subset of processor to each leaf node such that number of processors assigned to a leaf node is proportional to the number of the data items contained in the node.
    ii. Shuffle the training data such that each subset of processors has data item that belongs to the leaf nodes it is responsible for.
    iii. Processor subsets assigned to different nodes develop subtrees of the responsible nodes independently, by following the above steps recursively.

    (b) Otherwise,
    i. Partition leaf nodes into $|P_{(n)}|$ groups such that each group has about the equal number of data items. Assign each processor to one node group.
    ii. Shuffle the training data such that each processor has data item that belongs to the leaf nodes it is responsible for.
    iii. Now the expansion of the subtrees rooted at a node group proceeds completely independently at each process.
3. At the end, the whole decision tree is constructed by combining subtrees of each processor.

The advantage of this approach is that once a processor becomes solely responsible for a node, it can develop a subtree of the decision tree independently without any communication overhead. There are a number of disadvantages of this approach. First disadvantage is that it requires data movement after each node expansion until one process becomes responsible for an entire subtree. The communication cost is particularly expensive in the expansion of the upper part of the decision tree. Second disadvantage is due to load balancing.

3. Association Rule Algorithms

An association rule is a rule which implies certain association relationships among a set of objects (such as ``occur together" or ``one implies the other") in a database. Given a set of transactions, where each transaction is a set of literals (called items), an **association rule** is an expression of the form $X \Rightarrow Y$, where X and Y are sets of items. The intuitive meaning of such a rule is that transactions of the database which contain X tend to contain Y . An example of an association rule is: ``30% of transactions that contain beer also contain diapers; 2% of all transactions contain both of these items". Here 30% is called the confidence of the rule, and 2% the support of the rule. The problem is to find all association rules that satisfy user-specified minimum support and minimum confidence constraints.

3.1 Apriori Algorithm

An association rule mining algorithm, **Apriori** has been developed for rule mining in large transaction databases by IBM's Quest project team[3] . A *itemset* is a non-empty set of items.

They have decomposed the problem of mining association rules into two parts

- Find all combinations of items that have transaction support above minimum support. Call those combinations frequent itemsets.
- Use the frequent itemsets to generate the desired rules. The general idea is that if, say, ABCD and AB are frequent itemsets, then we can determine if the rule $AB \Rightarrow CD$ holds by computing the ratio r = support(ABCD)/support(AB). The rule holds only if r >= minimum confidence. Note that the rule will have minimum support because ABCD is frequent. The Apriori algorithm used in Quest for finding all frequent itemsets is given below

```
procedure AprioriAlg()
begin
        L1 := {frequent 1-itemsets};
        for ( k := 2; Lk-1≠ 0; k++ ) do {
        Ck= apriori-gen(Lk-1) ;  // new candidates
        for all transactions t in the dataset do {
                for all candidates c ∈ Ck contained in t do
                c:count++
        }
        Lk = { c ∈ Ck | c:count >= min-support}
}
Answer := ∪k Lk
end
```

It makes multiple passes over the database. In the first pass, the algorithm simply counts item occurrences to determine the frequent 1-itemsets (itemsets with 1 item). A subsequent pass, say pass k, consists of two phases. First, the frequent itemsets L_{k-1} (the set of all frequent (k-1)-itemsets) found in the (k-1)th pass are used to generate the candidate itemsets C_k, using the apriori-gen() function. This function first joins L_{k-1} with L_{k-1}, the joining condition being that the

lexicographically ordered first k-2 items are the same. Next, it deletes all those itemsets from the join result that have some (k-1)-subset that is not in L_{k-1} yielding C_k.
The algorithm now scans the database. For each transaction, it determines which of the candidates in C_k are contained in the transaction using a hash-tree data structure and increments the count of those candidates. At the end of the pass, C_k is examined to determine which of the candidates are frequent, yielding L_k . The algorithm terminates when L_k becomes empty.

3.2 Distributed/Parallel Algorithms

Databases or data warehouses may store a huge amount of data to be mined. Mining association rules in such databases may require substantial processing power . A possible solution to this problem can be a distributed system.[5] . Moreover, many large databases are distributed in nature which may make it more feasible to use distributed algorithms.

Major cost of mining association rules is the computation of the set of large itemsets in the database. Distributed computing of large itemsets encounters some new problems. One may compute locally large itemsets easily, but a locally large itemset may not be globally large. Since it is very expensive to broadcast the whole data set to other sites, one option is to broadcast all the counts of all the itemsets, no matter locally large or small, to other sites. However, a database may contain enormous combinations of itemsets, and it will involve passing a huge number of messages.

A distributed data mining algorithm FDM (Fast Distributed Mining of association rules) has been proposed by [5], which has the following distinct features.

1. The generation of candidate sets is in the same spirit of Apriori. However, some relationships between locally large sets and globally large ones are explored to generate a smaller set of candidate sets at each iteration and thus reduce the number of messages to be passed.
2. After the candidate sets have been generated, two pruning techniques, local pruning and global pruning, are developed to prune away some candidate sets at each individual sites.
3. In order to determine whether a candidate set is large, this algorithm requires only O(n) messages for support count exchange, where n is the number of sites in the network. This is much less than a straight adaptation of Apriori, which requires $O(n^2)$ messages.

4. Sequential Analysis

4.1 Sequential Patterns

The input data is a set of sequences, called data-sequences. Each data sequence is a ordered list of transactions(or itemsets), where each transaction is a sets of items (literals). Typically there is a transaction-time associated with each transaction. A sequential pattern also consists of a list of sets of items. The problem is to find all sequential patterns with a user-specified minimum support, where the support of a sequential pattern is the percentage of data sequences that contain the pattern.

An example of such a pattern is that customers typically rent ``Star Wars", then ``Empire Strikes Back", and then ``Return of the Jedi". Note that these rentals need not be consecutive. Customers who rent some other videos in between also support this sequential pattern. Elements of a sequential pattern need not be simple items. ``Fitted Sheet and flat sheet and pillow cases", followed by ``comforter", followed by ``drapes and ruffles" is an example of a sequential pattern in which the elements are sets of items. This problem was initially motivated by applications in the retailing industry, including attached mailing, add-on sales, and customer satisfaction. But the results apply to many scientific and business domains. For instance, in the medical domain, a data-sequence may correspond to the symptoms or diseases of a patient, with a transaction corresponding to the symptoms exhibited or diseases diagnosed during a visit to the doctor. The patterns discovered using this data could be used in disease research to help identify symptoms/diseases that precede certain diseases.

4.2 Algorithms for Finding Sequential Patterns

Various groups working in this field have suggested algorithms for mining sequential patterns. Listed below are two algorithms proposed by IBM's Quest data team.[6]

Terminology : The length of a sequence is the number of itemsets in the sequence. A sequence of length k is called a k-sequence. The sequence formed by the concatenation of two sequences x and y is denoted as x.y. The support for an itemset i is defined as the fraction of customers who bought the items in i in a single transaction. Thus the itemset i and the 1-sequence $\langle i \rangle$ have the same support. An itemset with minimum support is called a large itemset or *litemset*. Note that each itemset in a large sequence must have minimum support. Hence, any large sequence must be a list of litemsets. In the algorithms, L_k denotes the set of all large k-sequences, and C_k the set of candidate k-sequences.

4.2.1 Algorithm

The problem of mining sequential patterns can be split into the following phases:

1. *Sort Phase*. This step implicitly converts the original transaction database into a database of sequences.
2. *Litemset Phase*. In this phase we find the set of all litemsets L. We are also simultaneously finding the set of all large 1-sequences, since this set is just $\{\langle l \rangle \mid l \in L\}$.
3. *Transformation Phase*. We need to repeatedly determine which of a given set of large sequences are contained in a customer sequence. To make this test fast, we transform each

customer sequence into an alternative representation. In a transformed customer sequence, each transaction is replaced by the set of all litemsets contained in that transaction. If a transaction does not contain any litemset, it is not retained in the transformed sequence. If a customer sequence does not contain any litemset, this sequence is dropped from the transformed database. However, it still contributes to the count of total number of customers. A customer sequence is now represented by a list of sets of litemsets.

4. *Sequence Phase*. Use the set of litemsets to find the desired sequences. Algorithms for this phase below.
5. *Maximal Phase*. Find the maximal sequences among the set of large sequences. In some algorithms this phase is combined with the sequence phase to reduce the time wasted in counting non maximal sequences.

The general structure of the algorithms for the sequence phase is that they make multiple passes over the data. In each pass, we start with a seed set of large sequences. We use the seed set for generating new potentially large sequences, called *candidate sequences*. We find the support for these candidate sequences during the pass over the data. At the end of the pass, we determine which of the candidate sequences are actually large. These large candidates become the seed for the next pass. In the first pass, all 1-sequences with minimum support, obtained in the litemset phase, form the seed set.

There are two families of algorithms- *count-all* and *count-some*. The count-all algorithms count all the large sequences, including non-maximal sequences. The non-maximal sequences must then be pruned out (in the maximal phase). **AprioriAll** listed below is a count-all algorithm, based on the Apriori algorithm for finding large itemsets presented in chapter2. **Apriori- Some** is a count-some algorithm. The intuition behind these algorithms is that since we are only interested in maximal sequences, we can avoid counting sequences which are contained in a longer sequence if we first count longer sequences. However, we have to be careful not to count a lot of longer sequences that do not have minimum support. Otherwise, the time saved by not counting sequences contained in a longer sequence may be less than the time wasted counting sequences without minimum support that would never have been counted because their subsequences were not large.

4.2.2 Algorithm AprioriAll

L_1 = {large 1-sequences}; // Result of litemset phase
for (k = 2; $L_{k-1} \neq 0$; k++) **do**
begin
    C_k = New Candidates generated from L_{k-1} (see below)
    foreach customer-sequence *c* in the database **do**
        Increment the count of all candidates in C_k that are contained in *c*.
    L_k = Candidates in C_k with minimum support.
    end
Answer = Maximal Sequences in $\cup_k L_k$;

AprioriAll Algorithm

The algorithm is given above. In each pass, we use the large sequences from the previous pass to generate the candidate sequences and then measure their support by making a pass over the database. At the end of the pass, the support of the candidates is used to determine the large sequences. In the first pass, the output of the litemset phase is used to initialize the set of large 1-sequences. The candidates are stored in *hash-tree* to quickly find all candidates contained in a customer sequence.

Apriori Candidate Generation
The apriori-generate function takes as argument L_{k-1}, the set of all large (k-1)-sequences. It works as follows. First join L_{k-1} with L_{k-1}

insert into C_k
select $p.litemset_1$, ..., $p.litemset_{k-1}$, $q.litemset_{k-1}$
from L_{k-1} p, L_{k-1} q
where $p.litemset_1 = q.litemset_1$, . . .,
$p.litemset_{k-2} = q.litemset_{k-2}$;

Next delete all sequences $c \in C_k$ such that some (k-1)-subsequence of c is not in L_{k-1}

Example

Consider a database with the customer-sequences shown below in Fig4.1. We have not shown the original database in this example. The customer sequences are in transformed form where each transaction has been replaced by the set of litemsets contained in the transaction and the litemsets have been replaced by integers. The minimum support has been specified to be 40% (i.e. 2 customer sequences). The first pass over the database is made in the litemset phase, and we determine the large 1-sequences shown in Fig. 4.2. The large sequences together with their support at the end of the second, third, and fourth passes are also shown in the same figure. No candidate is generated for the fifth pass. The maximal large sequences would be the three sequences ⟨1 2 3 4 ⟩, ⟨1 3 5 ⟩ and ⟨ 4 5 ⟩.

⟨ {1 5} {2} {3} {4}⟩
⟨ {1} {3} {4} {3 5}⟩
⟨{1} {2} {3} {4}⟩
⟨{1} {3} {5}⟩
⟨{4} {5}⟩
Fig4.1: Customer Sequences

4.2.3 Algorithm AprioriSome

This algorithm is given below. In the forward pass, we only count sequences of certain lengths. For example, we might count sequences of length 1, 2, 4 and 6 in the forward phase and count sequences of length 3 and 5 in the backward phase. The function next takes as parameter the length of sequences counted in the last pass and returns the length of sequences to be counted in the next pass. Thus, this function determines exactly which sequences are counted, and balances the tradeoff between the time wasted in counting non-maximal sequences versus counting extensions of small candidate sequences. One extreme is next(k) = k + 1 (k is the length for which candidates were counted last), when all non-maximal sequences are counted, but no

extensions of small candidate sequences are counted. In this case, AprioriSome degenerates into AprioriAll. The other extreme is a function like next(k) = 100 * k, when almost no non-maximal large sequence is counted, but lots of ex- tensions of small candidates are counted.

```
// Forward Phase
L1 = {large 1-sequences};        // Result of litemset phase
C1 = L1 ;          // so that we have a nice loop condition
last = 1;          // we last counted Clast
for ( k = 2; Ck-1 ≠ 0 and Llast ≠ 0;  k++) do
begin
   if (Lk-1 known) then
        Ck= New candidates generated from Lk-1 ;
   else
        Ck= New candidates generated from Ck-1 ;
   if (k == next(last) ) then begin
        foreach customer-sequence c in the database do
          Increment the count of all candidates in Ck that are contained in c.
   Lk = Candidates in Ck with minimum support.
   last = k;
   end
end

// Backward Phase
for ( k-- ; k >=1; k==) do
   if (Lk not found in forward phase) then begin
        Delete all sequences in Ck contained in some Li , i>k;
        foreach customer-sequence in DT do
            Increment the count of all candidates in Ck that are contained in c.
            Lk = Candidates in Ck with minimum support.
        end
    else          // Lk already known
        Delete all sequences in Lk contained in some Li , i > k.

Answer = ∪k Lk ;
```

AprioriSome Algorithm

Let hit_k denote the ratio of the number of large k-sequences to the number of candidate k-sequences (i.e., $|L_k| / |C_k|$). The *next* function we used in the experiments is given below. The intuition behind the heuristic is that as the percentage of candidates counted in the current pass which had minimum support increases, the time wasted by counting extensions of small candidates when we skip a length goes down.

```
function next(k: integer)
begin
        if (hit_k < 0.666)      return k + 1;
        elsif (hit_k <0.75)     return k + 2;
        elsif (hit_k < 0.80)    return k + 3;
        elsif (hit_k < 0.85)    return k + 4;
        else                    return k + 5;
end
```

We use the apriori-generate function given above to generate new candidate sequences. However, in the kth pass, we may not have the large sequence set L_{k-1} available as we did not count the (k-1)-candidate sequences. In that case, we use the candidate set C_{k-1} to generate C_k. Correctness is maintained because $C_{k-1} \supseteq L_{k-1}$

In the backward phase, we count sequences for the lengths we skipped over during the forward phase, after first deleting all sequences contained in some large sequence. These smaller sequences cannot be in the answer because we are only interested in maximal sequences. We also delete the large sequences found in the forward phase that are non-maximal.

Example

Using again the database used in the example for the AprioriAll algorithm, we find the large 1-sequences (L_1) shown in Fig. 4.2 below in the litemset phase (during the first pass over the database). Take for illustration simplicity, f(k) = 2k. In the second pass, we count C_2 to get L_2 (Fig. 4.2). After the third pass, apriori-generate is called with L_2 as argument to get C_3 . We do not count C_3 , and hence do not generate L_3 . Next, apriori-generate is called with C_3 to get C_4. After counting C_4 to get L_4 (Fig. 4.2), we try generating C_5, which turns out to be empty.
We then start the backward phase. Nothing gets deleted from L_4 since there are no longer sequences. We had skipped counting the support for sequences in C_3 in the forward phase. After deleting those sequences in C_3 that are subsequences of sequences in L_4, i.e., subsequences of ⟨ 1 2 3 4⟩ , we are left with the sequences ⟨ 1 3 5 ⟩ and ⟨ 3 4 5⟩. These would be counted to get ⟨1 3 5⟩ as a maximal large 3-sequence. Next, all the sequences in L_2 except ⟨ 4 5⟩ are deleted since they are contained in some longer sequence. For the same reason, all sequences in L_1 are also deleted.

L 1

1-Sequences	Support
⟨1⟩	4
⟨2⟩	2
⟨3⟩	4
⟨4⟩	4
⟨5⟩	4

L 2

2-Sequences	Support
⟨1 2⟩	2
⟨1 3⟩	4
⟨1 4⟩	3
⟨1 5⟩	3
⟨2 3⟩	2
⟨2 4⟩	2
⟨3 4⟩	3
⟨3 5⟩	2
⟨4 5⟩	2

L 3

3-Sequences	Support
⟨1 2 3⟩	2
⟨1 2 4⟩	2
⟨1 3 4⟩	3
⟨1 3 5⟩	2
⟨2 3 4⟩	2

L 4

4-Sequences	Support
⟨1 2 3 4⟩	2

Fig 4.3 : **Large Sequences**

4.2.4 Relative Performance of the two Algorithms

As expected, the execution times of all the algorithms increase as the support is decreased because of a large increase in the number of large sequences in the result. The apriori-generate does not count any candidate sequence that contains any subsequence which is not large. The major advantage of AprioriSome over AprioriAll is that it avoids counting many non-maximal sequences. However, this advantage is reduced because of two reasons. First, candidates C_k in AprioriAll are generated using L_{k-1}, whereas AprioriSome sometimes uses C_{k-1} for this purpose. Since $C_{k-1} \supseteq L_{k-1}$, the number of candidates generated using AprioriSome can be larger. Second, although AprioriSome skips over counting candidates of some lengths, they are generated nonetheless and stay memory resident. If memory gets filled up, AprioriSome is forced to count the last set of candidates generated even if the heuristic suggests skipping some more candidate sets. This effect decreases the skipping distance between the two candidate sets that are indeed counted, and AprioriSome starts behaving more like AprioriAll. For lower supports, there are longer large sequences, and hence more non-maximal sequences, and AprioriSome does better.

5. Conclusion

After studying through the vast resources of technical papers, white papers written on data mining, here are some of the conclusions that I have made regarding the Algorithms used for data mining.

5.1 Comparing Algorithms

There is a theoretical result that no single learning algorithm can outperform any other when the performance measure is the expected generalization accuracy. This result, sometimes called the No Free Lunch Theorem or Conservation Law (Wolpert 1994, Schaffer 1994), assumes that all possible targets are equally likely.

Averaging an algorithm's performance over all target concepts, assuming they are all equally likely, would be like averaging a car's performance over all possible terrain types, assuming they are all equally likely. This assumption is clearly wrong in practice; for a given domain, it is clear that not all concepts are equally probable.

In medical domains, many measurements (attributes) that doctors have developed over the years tend to be independent: if the attributes are highly correlated, only one attribute will be chosen. In such domains, a certain class of learning algorithms might outperform others. For example, Naive-Bayes seems to be a good performer in medical domains (Kononenko 1993). Quinlan (1994) identifies families of parallel and sequential domains and claims that neural-networks are likely to perform well in parallel domains, while decision-tree algorithms are likely to perform well in sequential domains. Therefore, although a single induction algorithm cannot build the most accurate classifiers in all situations, some algorithms will perform better in specific domains.

5.2 Drawbacks of Existing Algorithms

This field is still in it’s infancy and is constantly evolving. The first people who gave a serious thought to the problem of data mining were those researching in the Database field , since they were the first to face this problem. Whereas most of the tools and techniques used for data mining come from other related fields like pattern recognition, statistics and complexity theory. It is only recently that the researchers of these various fields have been interacting to solve the mining issue.

Data Size

Most of the traditional data mining techniques failed because of the sheer size of the data. New techniques will have to be developed to store this huge data. Any algorithm that is proposed for mining data will have to account for out of core data structures. Most of the existing algorithms haven’t addressed this issue. Some of the newly proposed algorithms like parallel algorithms (sec. 2.4) are now beginning to look into this.

Data Noise

Most of the algorithms assume the data to be noise-free. As a result, the most time-consuming part of solving problems becomes data preprocessing. Data formatting and experiment/result management are frequently just as time-consuming and frustrating.

The concept of noisy data can be understood by the example of mining logs. A real life scenario can be if one wants to mine information from web logs. A user may have gone to a web site by mistake - incorrect URL or incorrect button press. In such a case, this information is useless if we are trying to deduce a sequence in which the user accessed the web pages. The logs may contain many such data items . These data items constitute data noise. A database may constitute upto 30-40% such Noisy data and pre-processing this data may take up more time than the actual algorithm execution time.

References

[1] R. Agrawal, A. Arning, T. Bollinger, M. Mehta, J. Shafer, R. Srikant: "*The Quest Data Mining System*", Proc. of the 2nd Int'l Conference on Knowledge Discovery in Databases and Data Mining, Portland, Oregon, August, 1996.

[2] Eui-Hong (Sam) Han, Anurag Srivastava and Vipin Kumar: "*Parallel Formulations of Inductive Classification Learning Algorithm*" (1996).

[3] Agrawal, R. Srikant: ``*Fast Algorithms for Mining Association Rules*", Proc. of the 20th Int'l Conference on Very Large Databases, Santiago, Chile, Sept. 1994.

[4] J. Han, J. Chiang, S. Chee, J. Chen, Q. Chen, S. Cheng, W. Gong, M. Kamber, K. Koperski, G. Liu, Y. Lu, N. Stefanovic, L. Winstone, B. Xia, O. R. Zaiane, S. Zhang, H. Zhu, *`DBMiner: A System for Data Mining in Relational Databases and Data Warehouses''*, Proc. CASCON'97: Meeting of Minds, Toronto, Canada, November 1997.

[5] Cheung, J. Han, V. T. Ng, A. W. Fu an Y. Fu, `` *A Fast Distributed Algorithm for Mining Association Rules*", Proc. of 1996 Int'l Conf. on Parallel and Distributed Information Systems (PDIS'96), Miami Beach, Florida, USA, Dec. 1996.

[6] R. Agrawal, R. Srikant, ``Mining Sequential Patterns", Proc. of the Int'l Conference on Data Engineering (ICDE), Taipei, Taiwan, March 1995.

[7] R. Srikant, R. Agrawal: ``Mining Sequential Patterns: Generalizations and Performance Improvements", Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT), Avignon, France, March 1996.

[8] Ron Kohavi, Dan Sommerfield, James Dougherty, "Data Mining using MLC++ : A Machine Learning Library in C++", Tools with AI, 1996

Appendix A

URL Listing

Listed below are all the web sites that I referred to for this project report. Some of these sites containing research papers of the various teams that I referred to for my report (see the References).

University / Non-profit Research Groups

- Simon Fraser University's database group : Knowledge Discovery in Databases and Data Mining (Research papers of Prof. Jiawei Han)
  http://fas.sfu.ca/cs/research/groups/DB/sections/publication/kdd/kdd.html

- University of Minnesota : Parallel computing research group , (Research papers of Prof. Vipin Kumar)
  ftp://ftp.cs.umn.edu/users/kumar/WEB/papers.html#bbbb

- The Cooperative Research Centre for Advanced Computational Systems , Australia
  http://www.dit.csiro.au/~gjw/dataminer/dmpapers.html

- UCLA Data Mining Laboratory
  http://nugget.cs.ucla.edu:8001/main.html

Corporate Research Groups

- IBM QUEST Data Mining Project
  http://www.almaden.ibm.com/cs/quest/index.html

- Data Mining with Silicon Graphics Technology / Machine Learning Library in C++ classes
  http://www.sgi.com/Technology/data-mining.html / http://www.sgi.com/Technology/mlc/

- Data Mining at Dun & Bradstreet
  http://www.santafe.edu/~kurt/text/wp9501/wp9501.shtml

Other Sites of Interest

- The Corporate KDD Bookmark **< *Exhaustive Listing* >**
  http://www.cs.su.oz.au/~thierry/ckdd.html

- dbProphet: Data Mining with Neural Networks
  http://www.trajecta.com/white.htm

- PRW - Not Just Another Neural Network Tool
  http://www.unica-usa.com/nntool.htm