Reference copy

REPLACED SHEETS SG
☑ Description : paged ....1.... to paged ....27....
☑ Claim(s) : paged ....28.... to paged ....32....
☐ Drawing(s) : paged ..... to paged .....
☑ Abstract : paged ....33.... to paged ....-....

CLASSIFICATION BY AGGREGATING EMERGING PATTERNS

FIELD OF THE INVENTION

The present invention relates to the field of data mining and the classification of data.

BACKGROUND

A number of systems for classifying data and performing data mining have been proposed. A number of these techniques involve decision trees, linear classification trees, and association rules. However, each of these techniques has significant disadvantages. Data mining techniques have also been proposed. Han, J., and Fu, Y., "Exploration of the Power of Attribute-Oriented Induction in Data Mining", *Advances in Knowledge Discovery in Databases and Data Mining*, AAAI/MIT Press, 1996, pp. 399-421, disclose one such technique utilising a discriminant rule involving an assertion that holds on instances of a class, but that does not hold on instances of other classes. An evolution of regularity is either a characteristic rule or a discriminant rule when the classes represent datasets at two different times. The discriminating rule is found by merging values according to some concept hierarchies and removing overlapping tuples in the two classses. However, this is disadvantageous because of exclusiveness restraints. Also this technique requires a concept hierarchy and merges values.

Conventional techniques do not handle well biased sample data which has a large volume and high dimensions. They tend to ignore weak signals.

Thus a need clearly exists for an improved system of data mining and classifying data.

9903958-8
16 AUG 1999

SUMMARY

The aspects of the invention are directed towards a system of extracting rules in the form of emerging patterns (EP) and constructing a classifier from correctly labelled data (e.g., DNA sequences) to decide which category a sample belongs to and/or making a prediction on the sample. The system discovers features or signals that differentiate one category of data from another and builds a system to classify such data.

The system is able to handle well biased data that has a large volume and high dimensions. Also the system does not ignore weak signals. The EPs are associated with the ratio of change in supports. The system is robust in the presence of biased sample data and is scalable in terms of large numbers of samples and in terms of dimensions in practical situations.

- An EP is a signal/itemset whose supports increase significantly from one class of data to the next. In other words, it is a differentiating factor between the two classes. The use of EPs is novel.

- The aggregation of the differentiating strengths, in terms of their supports and ratio of change, of all of some set of discovered EPs (whose cardinality is not bounded before classifier construction) that occur in a new case in a decision step is novel.

- The normalization by dividing by a base score chosen at the same percentile (such as 50%) across training instances of all classes is novel.

- Our way to finding emerging pattern is based on a novel border-based representation of very large collections of itemsets, and some novel algorithms which derive EPs by operating (such as differentials) on some borders; these algorithms were recently designed by two of the current inventors. These borders are first efficiently discovered using Max-Miner [R.J. Bayardo. Efficiently mining long patterns from databases.*Proceedings of the 1998 ACM-SIGMOD International Conference on Management of Data* , pages 85--93, 1998.] These algorithms are scalable in terms of large number of tuples and high dimensions in practical situations.

- The use of EPs is a novel idea in the protein translation start-site identification problem. (This is an example application of CAEP to datamining in Molecular Biology.)

Our CAEP classifier (i) extracts emerging patterns (EPs), (ii) uses each of these EPs as a multiple-attribute test, (iii) aggregates the power of individual EPs to get raw scores, (iv) normalizes the raw scores by dividing them using some base scores chosen from a certain percentile of the scores of the training instances. CAEP has near equal prediction accuracy on both the positive and the negative classes. CAEP is based on a novel border-based representation of very large collections of itemsets. It derives EPs by operating on some borders (which can also be efficiently discovered). It is scalable in terms of large number of tuples and high dimensions in practical situations.

DETAILED DESCRIPTION

1 Introduction

Classification is an important problem in data mining and machine learning, aimed at building a classifier from training instances for predicting the classes of new instances. Recently, datasets are becoming increasingly larger in both volume and dimensionality (number of attributes); a new challenge is the ability to efficiently build highly accurate classifiers from such datasets. In this paper we propose a new classifier, CAEP (Classification by Aggregating Emerging Patterns), which is suitable for many applications, even those with large volumes of high dimensional data. The classifier is highly accurate, and is usually equally accurate on all classes even if their populations are unbalanced. These advantages are achieved without dimension reduction on data.

CAEP is based on the following two main new ideas:

(i) We use a new type of knowledge, the so-called *emerging patterns* (EPs),

, to build CAEP. Roughly speaking, EPs are those itemsets whose supports (i.e. frequencies) increase significantly from one class of data to another. For example, the itemset {odor=none, stalk-surface-below-ring = smooth, ring-number=one} in the Mushroom dataset [10] is a typical EP, whose support increases from 0.2% in the poisonous class to 57.6% in the edible class, at a growth rate of 288 ($= \frac{57.6\%}{0.2\%}$). For us, an item is a simple test on an attribute, and an EP is a multi-attribute test. Each EP can have very strong power for differentiating the class membership of some instances. Indeed, if a new instance s contains the above EP, then with odds of 99.6% we can

claim that s belongs to the edible class. In general, the differentiating power of an EP is roughly proportional to the growth rate of its supports and its support in the target class.

(ii) An individual EP is usually sharp in telling the class of only a very *small* fraction (e.g. 3%) of all instances, and thus it will have very poor overall classification accuracy if it is used by itself on all instances. To build an accurate classifier, we first find, for each class C, all the EPs meeting some support and growth rate thresholds, from the (opponent) set of all none-C instances to the set of all C instances. Then we aggregate the power of the discovered EPs for classifying an instance s: We derive an aggregate differentiating score for each class C, by summing the differentiating power of all EPs of C that occur in s; the score for C is then normalized by dividing it by some base score (e.g. median) of the training instances of C. Finally, we let the largest normalized score determine the winning class. Normalization is done to reduce the effect of unbalanced distribution of EPs among the classes (classes with more EPs frequently give higher scores to instances, even to those from other classes).

CAEP achieves very good predictive accuracy on *all* the datasets we tested, and it gives better accuracy than C4.5 and CBA on all except one of these datasets. (Note: we reported all datasets that we tested!) We believe that the high accuracy is achieved because we are using a new high dimensional method to solve a high dimensional problem: Each EP is a multi-attribute test and CAEP is using the combined power of an unbounded set of EPs to arrive at a classification decision.

Being equally accurate on all classes is very useful for many applications, where there are a dominant class (e.g. 98% of all instances) and a minority class, and the sole purpose of classification is to accurately catch instances of the minority class. Classification accuracy is not the desired measure, as we would then consider the classifier which classifies all instances as in the dominant class a very good classifier. In this paper we also measure classifiers using *sensitivity* and *precision*, which reward classifiers that correctly label more minority instances and do not mislabel many other instances.

The CAEP classifier can be efficiently built for large high dimensional training datasets in a scalable way, since EPs can be discovered efficiently, using border-based algorithms [3] and Max-Miner [1]. We can quickly produce CAEP classifiers for datasets such as Mushroom, whose records consist of 21 attributes. (Figure 9 of [1] shows that CPU time is 100 seconds for support threshold of 0.1%. Border-based algorithms [3] then can find the EPs in around 0.5 hour.)

There are several parameters that need to be selected. All these are done automatically, using the performance of the resulting classifier on the training instances as guidance. Because of the use of aggregation and perhaps normalization, we have not encountered the traditional overfitting problem in our experiments.

Organization: We compare CAEP with related work below. §2 introduces EPs and preliminaries. §3 presents our main ideas on how to build CAEP. §4 discusses how to efficiently discover the EPs, their supports and growth rates. §5 discusses how to reduce the number of EPs and §6 on the automatic selection of CAEP's parameters. §7 discusses the rice-DNA dataset and the need to use *sensitivity* and *precision* to measure classifiers. §8 contains the experimental results, and §9 offers some concluding remarks.

Related Work: CAEP is fundamentally different from previous classifiers in its use of the new knowledge type of EPs. To arrive at a score for decision making, CAEP uses

a set of multi-attribute tests (EPs) for each class. Most previous classifiers consider only one test on one attribute at a time; a few exceptions, X-of-N [14], CBA [9] and linear decision trees [2], consider only one multi-attribute test to make a decision.

Aggregation of the differentiating power of EPs is different from bagging or boosting [13], which manipulate the training data to generate different classifiers and then aggregate the votes of several classifiers. With CAEP, each EP is too weak as a classifier and all the EPs are more easily obtained.

Loosely speaking, our aggregation of the power of EPs in classification is related to the Bayesian prediction theory. For an instance t viewed as an itemset, Bayesian prediction would label t as C_k, where the probability $Pr(t|C_k) * Pr(C_k)$ is the largest among the classes. The optimal Bayesian classifier needs to "know" the probability $Pr(t|C_k)$ for all possible t, which is clearly impractical for high dimensional datasets. Roughly speaking, CAEP "approximates" $Pr(t|C_k) * Pr(C_k)$ using the normalized score.

CAEP is the first application of EPs to classification. Partially influenced by CAEP, [8] proposes a different classifier, JEP-Classifier, also based on aggregated power of EPs. Major differences include: (i) CAEP uses general EPs, whereas JEP-Classifier uses exclusively jumping EPs (i.e. EPs whose support increases from zero in one dataset to non-zero in the other dataset). (ii) For datasets with more than two classes, CAEP uses the classes in a symmetric way, whereas JEP-Classifier uses them in an ordered way. (iii) In aggregating the differentiating power of all EPs, CAEP uses factors based on both support and support growth rate, whereas JEP-Classifier uses only the supports. (iv) As CAEP uses EPs with mixed growth rates, the reduction of the EPs is more complicated; for JEP-Classifier, all jumping EPs have infinite growth rates and the reduction is simpler. (v) The normalization idea is used in CAEP but not in JEP-Classifier. The two classifiers offer their own advantages: CAEP is better for cases with few or even no jumping EPs whose supports meet a reasonable threshold (such as 1%), whereas JEP-Classifier is better when there are many jumping EPs. Each of them is almost consistently better than C4.5 and CBA.

2 Emerging Patterns and Preliminaries

Assume the original data instances have m attribute values. Each instance in the training dataset $\mathcal{D}$ is associated with a class label, out of a total of p class labels: $C_1, C_2, ..., C_p$. We partition $\mathcal{D}$ into p sets, $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_p$, with $\mathcal{D}_i$ containing all instances of class C_i.

Emerging patterns are defined for binary transaction databases. To find them, we may need to encode a raw dataset into a binary one: We discretize the value range of each continuous attribute into intervals [7]. Each $(attribute, interval)$ pair is called an *item* in the binary (transaction) database, which will be represented as an integer for convenience. An instance t in the raw dataset will then be mapped to a transaction of the binary database: t has the value 1 on exactly those items (A, v) where t's A-value is in the interval v. We will represent this new t as the set of items for which it takes 1, and we will assume henceforth the datasets $\mathcal{D}, \mathcal{D}_1, ..., \mathcal{D}_p$ are binary.

Let I be the set of all items in the encoding. An *itemset* X is a subset of I, and its *support* in a dataset $\mathcal{D}'$, $supp_{\mathcal{D}'}(X)$, is $\frac{|\{t \in \mathcal{D}' | X \subseteq t\}|}{|\mathcal{D}'|}$. Given two datasets $\mathcal{D}'$ and $\mathcal{D}''$, the *growth rate* of an itemset X from $\mathcal{D}'$ to $\mathcal{D}''$ is defined as $growth_rate_{\mathcal{D}' \to \mathcal{D}''}(X) = \frac{supp_{\mathcal{D}_2}(X)}{supp_{\mathcal{D}_1}(X)}$ if $supp_{\mathcal{D}_1}(X) \neq 0$; $= 1$ if $supp_{\mathcal{D}_1}(X) = supp_{\mathcal{D}_2}(X) = 0$; and $= \infty$ if $supp_{\mathcal{D}_1}(X) = 0 \neq supp_{\mathcal{D}_2}(X)$.

Emerging patterns [3] are itemsets with large growth rates from $\mathcal{D}'$ to $\mathcal{D}''$.

Definition 1. *Given $\mathcal{D}'$, $\mathcal{D}''$ and a growth rate threshold $\rho > 1$, an* emerging pattern *(ρ-EP or simply EP) from $\mathcal{D}'$ to $\mathcal{D}''$ is an itemset e where $growth_rate_{\mathcal{D}' \to \mathcal{D}''}(e) \geq \rho$.*

Example 1. Consider the following training dataset with two classes, $\mathcal{P}$ and $\mathcal{N}$ (this is actually an encoding of the Saturday morning activity example from [11]).

$\mathcal{P}$			$\mathcal{N}$	
{2,6,7,10}	{3,5,7,10}	{3,4,8,10}	{1,6,7,10}	{1,6,7,9}
{2,4,8,9}	{1,4,8,10}	{3,5,8,10}	{3,4,8,9}	{1,5,7,10}
{1,5,8,9}	{2,5,7,9}	{2,6,8,10}	{3,5,7,9}	

Then $\{1, 9\}$ is an EP from class $\mathcal{P}$ to class $\mathcal{N}$ with a growth rate $\frac{9}{5}$; it is also an ρ-EP for any $1 < \rho \leq \frac{9}{5}$. Some other EPs are given later.

3 Classification by Aggregating EPs

We now describe the major ideas and components of the CAEP classifier: (1) how to partition the dataset to derive the EPs for use in CAEP, (2) how individual EPs can differentiate class memberships, (3) how to combine the contribution of individual EPs to derive the *aggregate scores*, and (4) how to normalize the aggregate scores for deciding class membership. We also give an overview on how to construct and use CAEP.

3.1 Partitioning dataset to get EPs of classes

For each class C_k, we will use a set of EPs to contrast its instances, $\mathcal{D}_k$, against all other instances: We let $\mathcal{D}'_k = \mathcal{D} - \mathcal{D}_k$ be the opposing class, or simply opponent, of $\mathcal{D}_k$. We then mine (discussion on how is given later) the EPs from $\mathcal{D}'_k$ to $\mathcal{D}_k$; we refer to these EPs as *the EPs of class C_k*, and sometimes refer to C_k as the *target* class of these EPs.

For Example 1, some EPs of class $\mathcal{N}$ (i.e. from $\mathcal{P}$ to $\mathcal{N}$) are $(e : \{1\}, supp_{\mathcal{N}}(e) : 0.6, growth_rate_{\mathcal{P} \to \mathcal{N}} : 2.7)$, $(\{1,7\}, 0.6, \infty)$, $(\{1,10\}, 0.4, 3.6)$, $(\{3,4,8,9\}, 0.2, \infty)$. Similarly, some EPs of $\mathcal{P}$ are $(\{2\}, 0.44, \infty), (\{8\}, 0.67, 3.33), (\{4,8\}, 0.33, 1.67)$.

3.2 Differentiating power of individual EPs

Each EP can sharply differentiate the class membership of a fraction of instances which contain the EP, and this sharp differentiating power is derived from the big difference between its supports in the opposing classes. Continuing with Example 1, consider the EP $(\{1,10\}, 0.40, 3.60)$ for class $\mathcal{N}$. Suppose s is an instance containing this EP. What is the odds that s belongs to $\mathcal{N}$, given that s contains this EP? To simplify the discussion, we assume all classes have roughly equal population counts; then the answer is $\frac{supp_{\mathcal{N}}}{supp_{\mathcal{N}}+supp_{\mathcal{P}}} = \frac{3.60*supp_{\mathcal{P}}}{3.60*supp_{\mathcal{P}}+supp_{\mathcal{P}}} = \frac{3.60}{3.60+1} = 78\%$, since $supp_{\mathcal{N}} = 3.60 * supp_{\mathcal{P}}$. Without this assumption, we need to replace supports (e.g. $supp_{\mathcal{N}}$) by counts (e.g. $supp_{\mathcal{N}} * count_{\mathcal{N}}$ where $count_{\mathcal{N}}$ is the number of instances of class $\mathcal{N}$), and similar odds can be obtained. Observe that this EP has no differentiating power on instances s' that do not contain the EP. Therefore, assuming the population ratio in the training data accurately reflects the ratio in test instances and all classes have roughly equal population counts, this EP can be used to differentiate the class membership with the probability of 78% for roughly $\frac{supp_{\mathcal{N}}+supp_{\mathcal{P}}}{2} = 0.5 * (1 + \frac{1}{3.60}) * supp_{\mathcal{N}} = 25\%$ of the total population.

The fraction of instances which contain an EP may be a very small fraction (25% above, but much smaller, e.g. 3%, in many examples) of all instances. Hence, it cannot yield very accurate predictions if it is used by itself on all instances. For example, if we applied the above EP on all instances, we would arrive at an overall predictive accuracy of roughly $0.25 * 0.78 = 19.5\%$. This would be much lower if coverage is only 3%.

3.3 Better overall accuracy by aggregated score

We noticed above that a single EP is sharp on predicting class membership of a small fraction of instances, but not on all instances. We now show how to combine the strength of a set of EPs in order to produce a classifier with good overall accuracy.

Roughly speaking, given a test instance s, we let all the EPs of a class C_i that s contains contribute to the final decision of whether s should be labelled as C_i. This gives us the advantage of covering more cases than each single EP can cover, because different EPs complement each other in their applicable populations. To illustrate, consider Example 1. The largest fraction of population that a single EP (e.g. $\{8\}$) can cover is around 50%, whereas the seven EPs given above in §3.1 have a much larger combined coverage, around $\frac{12}{14} \approx 85.7\%$.

How do we combine the differentiating power of a set of EPs? A natural way is to sum the contributions of the individual EPs. (Other possibilities exist, but are beyond the scope of this paper.) Now, how do we formulate the contribution of a single EP? Roughly, we use a product of the odds discussed earlier and the fraction of the population of the class that contain the EP. More specifically, let e be an EP of class C, we let e's contribution be given by $\frac{growth_rate(e)}{growth_rate(e)+1} * supp_C(e)$. Observe that the first term is roughly the conditional probability that an instance is in class C given that the instance contains this EP e, and the second term is the fraction of the instances of class C that this EP applies. The contribution is proportional to both $growth_rate(e)$ and $supp_C(e)$. We now define scores of instances for the classes.

Definition 2. *Given an instance s and a set $E(C)$ of EPs of a class C discovered from the training data, the* aggregate score *(or* score*) of s for C is defined as*

$$score(s, C) = \sum_{e \subseteq s, e \in E(C)} \frac{growth_rate(e)}{growth_rate(e) + 1} * supp_C(e).$$

We now illustrate the calculation of contributions of EPs and scores of instances using Example 1 and the instance $s = \{1, 5, 7, 9\}$. Among EPs of the growth rate threshold of 1.1, s contains 2 of class $\mathcal{P}$: ($\{5\}$, 44%, 1.11), ($\{1, 5, 9\}$, 11%, ∞); it contains 10 of class $\mathcal{N}$: ($\{1\}$, 60%, 2.7), ($\{7\}$, 80%, 2.4), ($\{1, 5\}$, 20%, 1.8), ($\{1, 7\}$, 60%, ∞), ($\{1, 9\}$, 20%, 1.8), ($\{5, 7\}$, 40%, 1.8), ($\{7, 9\}$, 40%, 3.6), ($\{1, 5, 7\}$, 20%, ∞), ($\{1, 7, 9\}$, 20%, ∞), ($\{5, 7, 9\}$, 20%, 1.8). The aggregate score of s for $\mathcal{P}$ is: $score(s, \mathcal{P}) = \frac{1.11}{1.11+1} * 0.44 + \frac{\infty}{\infty+1} * 0.11 = 0.52 * 0.44 + 1 * 0.11 = 0.33$. Similarly, the contributions of the 10 EPs for $\mathcal{N}$ are respectively 0.41, 0.56, 0.12, 0.60, 0.12, 0.24, 0.31, 0.20, 0.20, 0.12, and their sum is $score(s, \mathcal{N}) = 2.88$.

3.4 Normalizing the scores to make decision

For each instance s, how do we use the p scores for all classes to predict its class?

One might be tempted to assign to s the class label C, for which the score of s is the largest. This turns out to be a bad strategy. The main reason for this is that the numbers of EPs for different classes may not be balanced, which is a frequent scenario for applications where some classes may have more random (uniform) distributions of values and consequently fewer EPs. If a class C has many more EPs than another class C', then instances usually get higher scores for C than for C', even for training instances of class C'. This indeed happens, for example in the rice-DNA dataset (see §6), which consists of a positive class and a negative class. The negative class contains mostly "random" instances, and the ratio of the number of EPs of the positive to that of the negative is 28:1 when the support threshold is 3% and the growth rate threshold is 2.

Our solution to this problem is to "normalize" the scores, by dividing them using a score at a fixed percentile for the training instances of each class. More specifically, a *base score* for each class C, $base_score(C)$, should be first found from the training instances of the class. The *normalized score* of an instance s for C, $norm_score(s, C)$, is defined as the ratio $score(s, C)/base_score(C)$. (Observe that our use of the term "normalized" is a slight abuse, since our normalized scores may be > 1.) Instead of letting the class with the highest raw score win, we let the class with the largest normalized score win. (We break tie by letting the class with the largest population win.)

How do we determine the base scores? We can let $base_score(C)$ be the median of the scores of the training instances class C; that is, exactly 50% of the training instances of C have scores larger than or equal to $base_score(C)$. We do not have to use 50%; in fact, other percentiles between 50%–85% give roughly similar results. The CAEP construction program should automatically choose a good percentile in this range, by testing the performance of the constructed classifier on the training instances. We do not want to use percentage on the two extreme ends (e.g. 3%), because the training instances usually contain some outliers, and if we use such a choice we let the outliers give too much influence.

Example 2. For a simple illustration of the decision process, assume there are 5 training instances from each of the positive (+ve) and negative (-ve) classes, and their scores are:

+ve training instances		*-ve training instances*	
score(s,+ve)	score(s,-ve)	score(s,+ve)	score(s,-ve)
18.44	0.31	4.89	5.51
16.65	0.39	8.37	5.47
15.76*	0.05	2.8	5.4*
15.28	0.21	9.93	4.97
14.52	0.41	10.31	4.8

The (median) base scores for the positive and negative classes are respectively 15.76 and 5.4. Given a test instance s (known to be from the negative class) with scores 7.07 and 4.82 for the positive and negative classes respectively, we have

norm_score(s,+ve)=7.07/15.76=0.45 norm_score(s,-ve)=4.82/5.4=0.89

s is thus labelled as negative. Observe that this decision is made even when s has a higher raw score for the positive class.

3.5 The entire process

The entire process for building and using CAEP is summarized below, assuming that the original dataset is partitioned according to the class labels.

CAEP (training datasets $\mathcal{D}_1, \ldots, \mathcal{D}_p$ for p classes $C_1, \cdots, C_p$)
;; *training phase*
1) Mine the EP set E_i from $\cup_{j=1}^{p} \mathcal{D}_j - \mathcal{D}_i$ to $\mathcal{D}_i$ for each $1 \leq i \leq p$;
;; *A growth rate threshold is given, or set to a default e.g.* 2
2) Optionally, reduce the number of EPs in each of $E_1, \cdots, E_p$;
3) Calculate the aggregate scores of all training instances for all classes;
4) Get the base scores $base_score(C_i)$ for each class C_i;
;; *testing phase*
5) For each test instance s do:
6) Calculate aggregate and normalized scores of s for each class C_i;
7) Assign to s the class C_j for which s has the largest normalized score.

4 Efficient Mining of EPs

For the discovery of EPs we will be using methods introduced in [3]. A key tool used by the efficient methods of [3] is that of borders, useful for the concise representation and efficient manipulation of large collections of itemsets.

Example 3. An example border is $<\mathcal{L} = \{\{22\}, \{57\}, \{61\}\}, \mathcal{R} = \{\{22, 34, 36, 57, 61, 81, 85, 88\}\}>$. The collection of itemsets represented by this border is $\{Y \mid \exists X \in \mathcal{L}, \exists Z \in \mathcal{R}$ such that $X \subseteq Y \subseteq Z\}$. Representative itemsets covered in the border include $\{22\}, \{22, 57\}, \{36, 57, 81, 88\}$. For interested readers, this is actually a border for the EPs from the edible to the poisonous class, of an encoding of the Mushroom dataset, at support threshold $\delta = 40\%$ in the poisonous and growth rate threshold $\rho = 2$.

To calculate the aggregate score contributed by all the EPs (meeting some thresholds) of a class C_i, we need to (i) find the EPs of C_i and (ii) discover their supports and growth rates. We now list two possible methods:

The large-border based approach: Max-Miner [1] is first used to efficiently discover the border of the large itemsets from $\mathcal{D}_i$. (Such a border is called a large border, hence the word "large" in the title of this approach.) If the large itemsets represented by the border can be enumerated in memory, then with one more scan of $\mathcal{D}_i$ and $\mathcal{D}_i'$ we can get the supports and growth rates of the EPs of C_i. If it can be applied, this approach can discover all EPs whose supports in $\mathcal{D}_i$ are larger than the given support threshold. However, because some larger borders may represent "exponentially" many candidate itemsets, only a small portion of these candidates can be held in memory; we need then to use the next approach.

The border differential based approach: We first use Max-Miner [1] to discover the two large borders of the large itemsets in $\mathcal{D}_i$ and the opponent $\mathcal{D}_i'$ having certain support thresholds. Then we use the *MBD-LLborder* (multiple-border differential) algorithm of [3] to find all the EP borders. Finally, we enumerate the EPs contained in the EP borders, and go through $\mathcal{D}_i$ and $\mathcal{D}_i'$ to check their supports and growth rates. With 13 EP borders of the Mushroom dataset for some support thresholds, using this approach we quickly found the supports and growth rates of 4692 EPs. Since *MBD-LLborder* only finds EPs whose supports in the second dataset are $\geq$ one support threshold and in the first dataset are $<$ another support threshold, we need to apply this method multiple times on multiple pairs [3] of large borders, or combine it with the previous method, to get the important EPs satisfying the given support and growth rate thresholds.

5 Reduction of EPs Used

Given a class C, we would like to find as many EPs as possible to give good coverage of the training instances; at the same time, we prefer EPs that have relatively large supports and growth rates, as these characteristics correspond to larger coverage and stronger differentiating power. Very often, many of the EPs can be removed without loss of too much accuracy, by exploiting relationships between the EPs. Reduction can increase understandability of the classifier, and it may even increase predictive accuracy.

The reduction step is optional, and it should not be done if it leads to poor classification of the training instances. This is a training time decision.

Our method to reduce the number of EPs uses these factors: the absolute strength of EPs, the relationships between EPs, and the relative difference between their supports and growth rates. We measure the absolute strength of EPs using a new growth rate threshold ρ', which should be larger than the growth rate threshold ρ for the EPs. The main idea is to select the strong EPs and remove the weaker EPs which have strong close relatives. We will refer to the selected EPs as the essential EPs.

To reduce the set of EPs, we first sort the mined EPs into a list E, in decreasing order on $(growth_rate, support)$. The set of essential EPs, $essE$, is initialized to contain the first EP in E. For each next EP e in E we do 1 and then 2:

1. For each EP x in $essE$ such that $e \subset x$, replace x by e if 1.a or 1.b is true:
   1.a. $growth_rate(e) \geq growth_rate(x)$
   1.b. $supp(e) >> supp(x)$ and $growth_rate(e) \geq \rho'$
2. Add e to $essE$ if both 1.a and 1.b are false, and e is not a superset of any x in $essE$.

We select EPs this way because: When condition 1.a is true, e definitely covers more instances than x since $e \subset x$, and e has a stronger differentiating power than x because e has a higher growth rate. A typical situation captured by condition 1.b is when x is an EP with growth rate ∞ but a very small support, whereas e is an EP whose growth rate is less than that of x but e has a much larger support than x. In this case, we prefer to have e since it covers many more cases than x and since it has a relatively high differentiating power already due to its growth rate being larger than ρ'. To illustrate this point, consider these two EPs of the *Iris-versicolor* class from the Iris dataset [10]:

$$e_1 = (\{1, 5, 11\}, 3\%, \infty) \qquad e_2 = (\{11\}, 100\%, 22.25)$$

e_2 is clearly more useful than e_1 for classification, since it covers 32 times more instances and its associated odds, 95.7%, is also very near that of the other EP, e_1. In our experiments, for 1.b, we set the default value of ρ' to 20 and the default interpretation of the condition "$supp(e) >> supp(x)$" is $\frac{supp(e)}{supp(x)} \geq 30$. These parameters can be tuned based on coverage on training instances.

6 Selection of Thresholds and Base Scores

To build a classifier from a training dataset, we need to select two thresholds (one for support and one for support growth rate), and a base score for each class.

The selection of these can be done *automatically* with the guidance of the *training* data: We start with some default thresholds and percentiles for the base scores. Then a

classifier is built and its performance on the training instances is found. We then let the program try several alternatives and see if significant improvements are made. The best choice is then selected.

We observe from our experiments that the lower the support threshold δ, the higher predictive accuracy the classifier achieves; and for each support threshold, the higher the growth rate threshold, the higher predictive accuracy the classifier achieves. Once δ is lowered to 1%-3%, the classifier usually becomes stable in predictive accuracy; if the growth rate threshold is then raised to the highest possible (see discussion below), CAEP almost always has better predictive accuracy than C4.5 and CBA. In our experiments reported later, δ is always between 1%–3%; the exact choice of δ is influenced by the how much running time is allowed for CAEP construction.

The growth rate threshold also has strong effect on the quality of the classifier produced. Our general principle is to (a) mine EPs with a small initial growth rate threshold such as 2, and (b) automatically select a larger final growth rate threshold guided by the coverage of selected EPs on the training instances. Generally, if the growth rate threshold is too high, the classifier would contain too few EPs and the classifier may have low accuracy because of poor coverage of the training instances. (Coverage of a set of EPs is measured by the number of zero scores it produces on the training instances: The fewer the number of zero scores the better the coverage.) On the other hand, if it can be done without lowering the coverage of training instances, raising the growth rate threshold would always results in a classifier with higher predictive accuracy. Our experiments show that with support threshold 1%–3%, the UCI datasets usually yield a huge number of EPs with growth rates from 1 to ∞ (Rice-DNA data is an exception, see Section 7). The automatically chosen growth rate threshold is usually around 15.

7 Rice-DNA, Sensitivity and Precision

Our motivation for more accurate measure of classifiers comes from the rice-DNA dataset (available at http://adenine.krdl.org.sg:8080/limsoon/kozak/rice), containing rice-DNA Kozak sequences.

A genomic DNA is a string over the alphabet of {A, C, T, G}. The context surrounding the protein translation start site of a gene is called the Kozak sequence [5]. Correct identification of such start sites from a long genomic DNA sequence can save a lot of labor and money in identifying genes on that sequence. The start site is always the A-T-G sequence. The context surrounding the A-T-G has been the most important information to distinguish real start sites from non-start sites. A context is typically taken from up to 15 bases up stream of A-T-G to 10 bases down stream. So a Kozak sequence—for the purpose of this work—consists of 25 letters (excluding the A-T-G).

In the genomic DNA sequences, non-start sites (negative) overwhelm real start sites (positive) typically at a ratio of 24:1 or more. So a distinctive feature of the rice-DNA dataset is that the number of instances of the two datasets are very unbalanced. What makes the treatment of this dataset more difficult is that the number of positives, which is more important in reality, is far more the minority. With this very unbalanced dataset, even the *just-say-no* classifier, which always predicts an instance to be negative, will have an overall accuracy of $\frac{24}{25} = 96\%$. Unfortunately, the fact is that not a single real start site has been identified, which is against our aim of classification.

From this analysis we can see that in evaluating a classification method more meaningful measures than accuracy or error rate on the whole dataset are desirable. We will use a measure in terms of two parameters, namely *sensitivity* and *precision*, for each class, which have long been used in the signals world and in information retrieval [4].

Definition 3. *Given N instances whose class is known to be C, for a classifier P, if P labels N' instances as of class C, of which N_1 are indeed to be of class C, then N_1/N is called P's* sensitivity on C, *denoted* sens(C), *and N_1/N' is called P's* precision on C, *denoted* prec(C). *For the special case of $N' = 0$, we define $sens(C) = 0, prec(C) = 0$.*

From the above definition we can see that *sensitivity* on a class C is an indication of the strength of a classifier on C and *precision* tells us how much confidence the classifier has on C. Using the "minority" vs "majority" terminology, sensitivity for the minority class represents the percentage of the minorities caught by the classifier, precision for the minority class represents the percentage of those claimed to be of the minority class are indeed in the minority class. Clearly, these are a more accurate description of the performance of the classifier on class C.

Example 4. Given the ratio of negative to positive instances of 24:1, let us examine the performance of the *just-say-no* classifier on the rice-DNA dataset:

$$sens(+ve) = 0 \qquad prec(+ve) = 0$$
$$sens(-ve) = \tfrac{24}{24} = 100\% \qquad prec(-ve) = \tfrac{24}{25} = 96\%$$

Since the sensitivity and precision on the positive class are more important for the rice-DNA dataset, we can conclude that the *just-say-no* classifier performs very badly on the rice-DNA dataset and we should seek better classifiers.

8 Experimental Results

We compare CAEP with the state-of-the-art classifiers C4.5 and CBA. (CBA was reported to beat C4.5 [9].) Except for rice-DNA (see §6 for its description), the datasets we use are from the UCI machine learning repository [10]. Furthermore, we test CAEP on datasets with a large number of records and where each record is long, where no results of C4.5 or CBA are known to us.

Discretization of continuous attributes is done by the entropy method [7], using code from the MLC++ machine learning library [6]. All the results are obtained by 10-fold cross validation.

Table 1 compares the overall predictive accuracy of CAEP, C4.5 (with discretization) and CBA. Dashes indicate that results are unavailable.

Columns 2, 3 and 4 describe the datasets: the numbers of records, of attributes and of classes respectively. Rice-DNA and Mushroom are the most challenging datasets, having both a large number of instances and high dimensionality. Observe that datasets of 2, 3 and even more classes are included, and that CAEP performs equally well.

Columns 5 and 6 give the average predictive accuracy of C4.5 and CBA over 10-fold cross-validation, and Columns 7 and 8 are that of CAEP, both before and after reduction. Columns 9 and 10 give the average number of EPs in the classifier, before and after reduction. It can be seen that although the number of EPs has been dramatically reduced after the reduction process, there is no big loss in predictive accuracy and often there is an increase in accuracy.

Table 1: Accuracy Comparison

Dataset	#records	#attributes	#classes	C4.5 (disc.)	CBA	CAEP Accuracy w/o red.	CAEP Accuracy red.	#EPs/class w/o red.	#EPs/class red.
breast-w	699	10	2	96.1%	96.1%	97.85%	97.70%	7647	233
ionosphere	351	34	2	92.0%	92.1%	92.39%	91.41%	1813464	7906
iris	150	4	3	94.7%	92.9%	98.62%	100%	40	4
mushroom	8124	22	2	—	—	98.82%	98.93%	823649	2738
pima	768	8	2	72.5%	73.0%	96.23%	96.56%	1427	57
rice-DNA	15760	28	2	—	—	70.92%	**	42576	**
sonar	208	60	2	72.2%	78.3%	83.90%	90%	651864	4608
tic-tac-toe	958	9	2	99.4%	100%	99.06%	96.87%	5707	682
vehicle	846	18	4	66.4%	68.8%	76.28%	76.17%	324048	14857
wine	178	13	3	92.1%	91.6%	99.38%	99.44%	175384	1859

Table 2: Precision and Sensitivity of CAEP

Dataset	class(instance dist.)	sensitivity w/o red.	sensitivity red.	precision w/o red.	precision red.
breast-w	benign (66%)	96.72%	97.17%	100%	99.34%
	malignant (34%)	100%	98.75%	94.26%	95.00%
ionosphere	good (64%)	94.54%	92.9%	94.23%	94.15%
	bad (36%)	88.33%	88.65%	91.33%	87.68%
iris	Setosa (35%)	100%	100%	100%	100%
	Versicolour (31%)	100%	100%	96.33%	100%
	Virginica (34%)	96.00%	100%	100%	100%
mushroom	edible (52%)	99.43%	99.61%	98.32%	98.46%
	poisonous (48%)	98.16%	98.35%	99.38%	99.34%
pima	positive (29%)	95.52%	94.93%	92.27%	93.60%
	negative (71%)	96.54%	97.23%	98.18%	97.95%
rice-DNA	positive (4%)	77.75%	**	9.64%	**
	negative (96%)	70.65%	**	98.75%	**
sonar	R (46%)	83.33%	93.33%	84.62%	87.32%
	M (54%)	84.09%	87.27%	88.46%	94.29%
tic-tac-toe	positive (65%)	99.52%	96.01%	99.07%	99.21%
	negative (35%)	98.19%	98.49%	99.13%	93.12%
vehicle	van (24%)	48.68%	53.95%	64.22%	62.94%
	saab (25%)	65.62%	65.00%	59.23%	61.16%
	bus (27%)	95.40%	90.72%	89.01%	88.93%
	opel (24%)	92.93%	93.06%	88.31%	90.54%
wine	class 1 (40%)	98.57%	100%	100%	98.75%
	class 2 (33%)	100%	100%	100%	100%
	class 3 (27%)	100%	98.00%	98.00%	100%

We point out that it takes the CAEP classifier almost no time to decide the class of an instance. Even for a classifier with 10000 EPs, it takes only 0.01 second to decide the class label of an instance.

The rice-DNA dataset is quite different from datasets in the UCI repository, and we do not recommend reduction of EPs of either class as this leads to poor coverage of the training instances. This decision is made at training time.

Table 2 gives a more detailed characterization of CAEP on the datasets; sensitivity and precision on each class, before and after reduction, are listed. It shows that CAEP generally has good sensitivity and precision on each class.

The positive sensitivity and precision on the rice-DNA dataset are better than the best neural-net results known to us.

9 Conclusion

In this paper we have proposed a classification method which is fundamentally different from previous classifiers, including C4.5 or CBA. It is based on a new kind of knowledge mined from the training dataset: *emerging patterns* (EPs).

Specifically, our classifier CAEP is based on aggregating the contribution of all EPs for differentiating instances of different classes. The aggregate score is proposed to quantify the overall contribution of all the EPs. The contribution of an EP takes into account both the support and growth rate of the EP. We then normalize the aggregate scores to classify instances, by dividing the scores by the base scores (chosen at a certain percentile of training instance scores) of the corresponding classes. The resulting classifier CAEP is in general more accurate than C4.5 and CBA, is a lot more accurate than them over datasets where they do not have good performance. CAEP is equally accurate on all classes, and can be built efficiently from large, even high dimensional datasets. Observing that accuracy on the whole dataset is too coarse description of classifiers, we also used a more accurate measure, *sensitivity* and *precision*, to better characterize the performance of classifiers. CAEP is also very good under this measure.

References

1. Roberto J. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of the 1998 ACM-SIGMOD International Conference on Management of Data*, pages 85–93, 1998.
2. L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth International Group, 1984.

4. William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, 1992.
5. Mary Kozak. An analysis of 5'-noncoding sequences from 699 vertebrate messenger RNAs. *Nucleic Acids Research*, 15:8125–8148, 1987.
6. R. Kohavi, G. John, R. Long, D. Manley, and K. Pfleger. MLC++: a machine learning library in C++. In *Tools with artificial intelligence*, pages 740 – 743, 1994.
7. R Kohavi, M Sahami. Error-based and Entropy-based Discretization of Continuous Features. In *KDD'96*, 1996.
8. J. Li, G. Dong, and K. Ramamohanarao. JEP-Classifier: Classification by Aggregating Jumping Emerging Patterns. Submitted to CIKM'99.
9. B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proc. of 4th KDD*, 1998.
10. P.M. Murphy and D.W. Aha. UCI repository of machine learning database. In [http://www.cs.uci.edu/ mlearn/mlrepository.html].
11. J R Quinlan. Induction of decision trees. In *Machine Learning*, Vol 1, pages 81–106, 1986.
12. J.R. Quinlan. *C4.5: program for machine learning*. Morgan Kaufmann, 1992.
13. R.E. Schapire. *The strength of weak learnability*. Machine Learning, 5(2):197-227, 1990.
14. Z. Zheng, Constructing X-of-N attributes for decision tree learning. *To appear in Machine Learning*, 1999.

APPENDIX A

"Efficiently Mining Long Patterns from Databases"

Roberto J Bayardo Jr.

The disclosure of the accompanying materials is hereby incorporated in this application.

Appears in Proc. of the 1998 ACM-SIGMOD Int'l Conf. on Management of Data, 85-93.

Efficiently Mining Long Patterns from Databases

Roberto J. Bayardo Jr.
IBM Almaden Research Center
http://www.almaden.ibm.com/cs/people/bayardo/
bayardo@alum.mit.edu

Abstract

We present a pattern-mining algorithm that scales roughly linearly in the number of maximal patterns embedded in a database irrespective of the length of the longest pattern. In comparison, previous algorithms based on Apriori scale exponentially with longest pattern length. Experiments on real data show that when the patterns are long, our algorithm is more efficient by an order of magnitude or more.

1. Introduction

Finding patterns in databases is the fundamental operation behind several common data-mining tasks including association rule [1] and sequential pattern mining [4]. For the most part, pattern mining algorithms have been developed to operate on databases where the longest patterns are relatively short. This leaves data outside this mold unexplorable using current techniques. Interesting data-sets with long patterns include those composed of questionnaire results (people tend to answer similarly to many questions), sales transactions detailing the purchases made by regular customers over a large time window, and biological data from the fields of DNA and protein analysis. Most categorically-valued data-sets used for classification problems (e.g. targeted marketing campaigns) also tend to have long patterns because they contain many frequently occurring items and have a wide average record length.

Almost every recently-proposed pattern-mining algorithm is a variant of Apriori [2]. Two recent papers have demonstrated that Apriori-like algorithms are inadequate on data-sets with long patterns. Brin et al. [6] applied their association-rule miner DIC to a data-set composed of PUMS census records. To reduce the difficulty of this data-set, they removed all items appearing in over 80% of the transactions yet still could only mine efficiently at high support. We [5] previously applied an Apriori-inspired algorithm to several data-sets from the Irvine Machine Learning Database Repository. In order to mine efficiently, this algorithm had to sometimes apply pruning strategies that rendered the search incomplete.

Apriori involves a phase for finding patterns called *frequent itemsets*. A frequent itemset is a set of items appearing together in a number of database records meeting a user-specified threshold. Apriori employs a bottom-up search that enumerates every single frequent itemset. This implies in order to produce a frequent itemset of length l, it must produce all 2^l of its subsets since they too must be frequent. This exponential complexity fundamentally restricts Apriori-like algorithms to discovering only short patterns.

To address this problem, this paper proposes the Max-Miner algorithm for efficiently extracting only the maximal frequent itemsets, where an itemset is maximal frequent if it has no superset that is frequent. Because any frequent itemset is a subset of a maximal frequent itemset, Max-Miner's output implicitly and concisely represents all frequent itemsets. Max-Miner is shown to result in two or more orders of magnitude in performance improvements over Apriori on some data-sets. On other data-sets where the patterns are not so long, the gains are more modest. In practice, Max-Miner is demonstrated to run in time that is roughly linear in the number of maximal frequent itemsets and the size of the database, irrespective of the size of the longest frequent itemset.

Max-Miner is successful because it abandons a strict bottom-up traversal of the search space, and instead always attempts to "look ahead" in order to quickly identify long frequent itemsets. By identifying a long frequent itemset early on, Max-Miner can prune all its subsets from consideration. Max-Miner uses a heuristic to tune its search in an effort to identify long frequent itemsets as early as possible. It also uses a technique that can often determine when a new candidate itemset is frequent before accessing the database. The idea is to use information gathered during previous database passes to compute a good lower-bound on the number of transactions that contain the itemset.

The techniques we introduce in this paper are flexible and can be extended in various ways and applied to other algorithms. To demonstrate this point, we optimize Apriori with the lower-bounding technique mentioned above. While the fundamental limitations of Apriori with respect to pattern length remain, performance is improved by an order of magnitude on several data-sets. We also show how Max-Miner can be extended to exploit additional pattern constraints during its search by creating a variant that identifies only the longest of the maximal frequent itemsets in a data-set. This algorithm efficiently identifies all of the longest maximal frequent itemsets even when the space of all maximal frequent itemsets is itself intractably large.

1.1 Related Work

There are many variants of Apriori that differ in how they check "candidate" itemsets against the database. Apriori in its purest form checks itemsets of length l for frequency during database pass l. DIC [6] is more eager and begins checking an itemset shortly after all its subsets have been determined frequent, rather than waiting until the database pass completes. Partition [11] identifies all frequent-itemsets in memory-sized partitions of the database, and then checks these against the entire database during a final pass. DIC considers the same number of candidate itemsets as Apriori, and Partition can consider more but never fewer candidate itemsets than Apriori, potentially exacerbating problems associated with long patterns.

Park et al. [9] enhance Apriori with a hashing scheme that can identify (and thereby eliminate from consideration) some candidates that will turn up infrequent if checked against the database. It also uses the hashing scheme to re-write a smaller database after each pass in order to reduce the overhead of subsequent passes. Still, like Apriori, it considers every frequent itemset.

Gunopulos et al. [7] present a randomized algorithm for identifying maximal frequent itemsets in memory-resident databases. Their algorithm works by iteratively attempting to extend a working pattern until failure. A randomized version of the algorithm that does not guarantee every maximal frequent itemset will be returned is evaluated and found to efficiently extract long frequent itemsets. Unfortunately, it is not clear how this algorithm would be scaled to disk resident data-sets since each attempt at extending an itemset requires a scan over the data. It also remains to be seen how the proposed complete version of the algorithm would perform in

practice.

Zaki et al. [16] present the algorithms MaxEclat and MaxClique for identifying maximal frequent itemsets. These algorithms are similar to Max-Miner in that they also attempt to look ahead and identify long frequent itemsets early on to help prune the space of candidate itemsets considered. The important difference is that Max-Miner attempts to look ahead throughout the search, whereas MaxEclat and MaxClique look ahead only during an initialization phase prior to a purely bottom-up Apriori-like search with exponential scaling. The initialization phase of MaxClique is also prone to problems with long frequent itemsets since it uses a dynamic programming algorithm for finding maximal cliques in a graph whose largest clique is at least as large as the length of the longest frequent itemset.

Concurrent to our work, Lin and Kedem [8] have proposed an algorithm called Pincer-Search for mining long maximal frequent itemsets. Like Max-Miner, Pincer-Search attempts to identify long patterns throughout the search. The difference between these algorithms is primarily in the long candidate itemsets considered by each. Max-Miner uses a simple, polynomial time candidate generation procedure directed by heuristics, while Pincer-Search uses an NP-hard reduction phase to ensure no long candidate itemset contains any known infrequent itemset. Our comparison with Pincer-Search is thus far only preliminary, and more work is needed to fully understand the advantages offered by each technique.

1.2 Overview

Section 2 begins with an introduction into the basic search strategy used by Max-Miner and the techniques used to restrict the search space. Section 3 formalizes these techniques through pseudo-code, provides implementation details, and establishes Max-Miner's correctness and efficiency characteristics. Section 4 discusses and exploits a technique for lower-bounding the support of candidate itemsets. Section 5 discusses the integration of additional pattern constraints into the search, culminating in a description of the Max-Miner-LO algorithm for finding only the longest maximal frequent itemsets. The algorithms are evaluated experimentally in Section 6 followed by a summary of contributions and avenues for future work in Section 7.

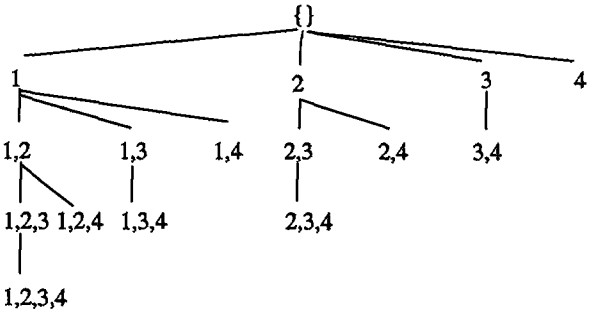
2. Introducing Max-Miner

We begin with defining the necessary terminology for describing the Max-Miner algorithm. For simplicity of presentation, we will be dealing only with the problem of identifying frequent itemsets. The application of our techniques to finding other patterns (e.g. sequential patterns) is similar.

A *data-set* is a set of *transactions* that are sets over a finite item domain. Transactions can represent things such as the supermarket items purchased by a customer during a shopping visit, or the characteristics of a person as described by his or her replies in a census questionnaire. A set of items is more succinctly called an *itemset*, and a *frequent itemset* is one that is contained in a number of transactions above or equal to the minimum support (*minsup*) specified by the user. An itemset with k items will be more succinctly referred to as a k-itemset. The *support* of an itemset I, denoted $sup(I)$, is the number of transactions that contain it. The minsup parameter will sometimes be specified as a percentage of the transactions in the data-set instead of as an absolute number of transactions.

Max-Miner can be described using Rymon's generic set-enumeration tree search framework [10]. The idea is to expand sets over an ordered and finite item domain as illustrated in Figure 1 where four items are denoted by their position in the ordering. The particular ordering imposed on the item domain affects the parent/child relationships in the set-enumeration tree but not its completeness. The figure assumes a static lexical ordering of the items, but later we describe an optimization that dramatically improves performance by heuristically ordering the items and dynamically reordering them on a per-node basis. Set-enumeration trees are not data-structures like the hash tree or trie, but instead are used to illustrate how sets of items are to be completely enumerated in a search problem. Note that the tree could be traversed depth-first, breadth first, or even best-first as directed by some heuristic. Max-Miner employs a purely breadth-first search of the set-enumeration tree in order to limit the number of passes made over the data.

Figure 1. Complete set-enumeration tree over four items.



The key to an efficient set-enumeration search is the pruning strategies that are applied to remove entire branches from consideration. Without pruning, a set-enumeration tree search for frequent itemsets will consider every itemset over the set of all items. Max-Miner uses pruning based on subset infrequency, as does Apriori, but it also uses pruning based on superset frequency.

To aid in our pruning efforts, we will represent each node in the set enumeration tree by what we call a *candidate group*. A candidate group g consists of two itemsets. The first, called the head and denoted $h(g)$, represents the itemset enumerated by the node. The second itemset, called the tail and denoted $t(g)$, is an ordered set and contains all items not in $h(g)$ that can potentially appear in any sub-node. For example, the node enumerating itemset $\{1\}$ in the figure has $h(g) = \{1\}$ and $t(g) = \{2, 3, 4\}$. The ordering of tail items reflect how the sub-nodes are to be expanded. In the case of a static lexical ordering without pruning, the tail of any candidate group is trivially the set of all items following the greatest item in the head according to the item ordering. When we are applying pruning and dynamic item reordering, it becomes necessary to make the tail items explicit.

When we say we are *counting the support* of a candidate group g, we are computing the support of itemsets $h(g)$, $h(g) \cup t(g)$, and $h(g) \cup \{i\}$ for all $i \in t(g)$. The supports of itemsets other than $h(g)$ are used for pruning. For example, consider first the itemset $h(g) \cup t(g)$. Since $h(g) \cup t(g)$ contains every item that appears in any viable sub-node of g, if it is frequent, then any itemset enumerated by a sub-node will also be frequent but not maximal. Superset-frequency pruning can therefore be implemented by halting sub-node expansion at any candidate group g for which $h(g) \cup t(g)$ is frequent. Consider next the itemset $h(g) \cup \{i\}$ for some $i \in t(g)$. If $h(g) \cup \{i\}$ is infrequent, then any head of a sub-node that contains item i will also be infrequent. Subset-infrequency pruning can therefore be implemented by simply removing any such tail item from a candidate group before expanding its sub-nodes.

3. Formalizing Max-Miner

We now provide a pseudo-code description of Max-Miner followed by the motivation behind and description of the item ordering policy. Implementation details describing how Max-Miner can use the same data-structures as Apriori are provided in the following subsection, and the last subsection provides correctness and efficiency arguments.

3.1 Pseudo-Code for Max-Miner

The pseudo-code description of Max-Miner appears in figures 2 through 4. The body (Figure 2) accepts a data-set and (implicitly) the minimum support specified by the user. The while loop

implements a breadth-first search of the set-enumeration tree that maintains every frequent itemset encountered so long as it is potentially maximal. The function Gen-Initial-Groups (Figure 3) performs the initial scan over the data-set to identify the item domain and seed the search at the second level of the tree. Superset-frequency based pruning is performed by only expanding the sub-nodes of a candidate g if $h(g) \cup t(g)$ is infrequent. Another instance of superset-frequency pruning is any candidate group g is pruned if $h(g) \cup t(g)$ is a subset of some already-know-to-be frequent itemset I.

Sub-nodes are generated by Gen-Sub-Nodes in Figure 4. Subset-infrequency pruning is performed here through the removal of any tail item i from a candidate group g if $h(g) \cup \{i\}$ is infrequent. Gen-Sub-Nodes and Gen-Initial-Groups return the sub-node with an empty tail as a frequent itemset instead of a candidate group since its frequency is already known and it has no children in the search tree.

Figure 2. Max-Miner at its top level.

```
MAX-MINER(Data-set T)
    ;; Returns the set of maximal frequent itemsets present in T
    Set of Candidate Groups C ← { }
    Set of Itemsets F ← {GEN-INITIAL-GROUPS(T, C)}
    while C is non-empty do
        scan T to count the support of all candidate groups in C
        for each g ∈ C such that h(g) ∪ t(g) is frequent do
            F ← F ∪ {h(g) ∪ t(g)}
        Set of Candidate Groups C_new ← { }
        for each g ∈ C such that h(g) ∪ t(g) is infrequent do
            F ← F ∪ {GEN-SUB-NODES(g, C_new)}
        C ← C_new
        remove from F any itemset with a proper superset in F
        remove from C any group g such that h(g) ∪ t(g)
                has a superset in F
    return F
```

Figure 3. Generating the initial candidate groups.

```
GEN-INITIAL-GROUPS(Data-Set T, Set of Candidate Groups C)
    ;; C is passed by reference and returns the candidate groups
    ;; The return value of the function is a frequent 1-itemset
    scan T to obtain F_1, the set of frequent 1-itemsets
    impose an ordering on the items in F_1 ;; see section 3.2
    for each item i in F_1 other than the greatest item do
        let g be a new candidate with h(g) = {i}
            and t(g) = {j | j follows i in the ordering}
        C ← C ∪ {g}
    return the itemset in F_1 containing the greatest item
```

Figure 4. Generating sub-nodes.

```
GEN-SUB-NODES(Candidate Group g, Set of Cand. Groups C)
    ;; C is passed by reference and returns the sub-nodes of g
    ;; The return value of the function is a frequent itemset
    remove any item i from t(g) if h(g) ∪ {i} is infrequent
    reorder the items in t(g) ;; see section 3.2
    for each i ∈ t(g) other than the greatest do
        let g' be a new candidate with h(g') = h(g) ∪ {i}
            and t(g') = {j | j ∈ t(g) and j follows i in t(g)}
        C ← C ∪ {g'}
    return h(g) ∪ {m} where m is the greatest item in t(g),
            or h(g) if t(g) is empty.
```

3.2 Item Ordering Policies

The motivation behind item (re)ordering is to increase the effectiveness of superset-frequency pruning. Recall that superset-frequency pruning can be applied when a candidate group g is found such that $h(g) \cup t(g)$ is frequent. We therefore want to make it likely that many candidate groups will have this property. A good heuristic for accomplishing this is to force the most frequent items to appear in the most candidate groups. This is simply because items with high frequency are more likely to be part of long frequent itemsets. Items that appear last in the ordering will appear in the most candidate groups. For instance, item 4 from Figure 1 appears either in the head or tail of every single node. Item ordering is therefore used to position the most frequent items last.

Gen-Initial-Groups orders the items in increasing order of $\sup(\{i\})$. Items in the tails of candidate groups are re-ordered prior to sub-node generation in Gen-Sub-Nodes. This function orders the tail items of a group g in increasing order of $\sup(h(g) \cup \{i\})$. This strategy tunes the frequency heuristic by having it consider only the subset of transactions relevant to the given node.

Interestingly, the same item reordering heuristic is used by Slagel et al. [12] in a set-enumeration algorithm for identifying prime implicants in CNF propositional logic expressions. The fact that the same policy works well for both problems is likely due to their close relationship. Finding prime implicants in CNF expressions is similar to the problem of generating hypergraph transversals, and Gunopulos et al. [7] have previously shown that hypergraph transversal algorithms can be used as a component of an algorithm for mining maximal frequent itemsets.

3.3 Implementation Details

Max-Miner can use the same data-structures as Apriori (as detailed in [3]) for efficiently computing itemset supports. The primary data-structure used by Apriori is the hash tree to index candidate itemsets. Max-Miner uses the hash tree to index only the head of each candidate group. For each transaction in the data-set, Max-Miner uses the hash tree to quickly look up all candidate groups whose head appears in the transaction. Then, for each candidate group g identified, it traverses down its tail items one by one, incrementing the support of $h(g) \cup \{i\}$ if tail item i is present in the transaction. If every tail item appears in the transaction, then the support of $h(g) \cup t(g)$ is also incremented. We found this implementation to be substantially faster than individually storing each itemset within the hash tree. Hash trees are also used by our implementation of Max-Miner for efficiently identifying the subsets of frequent itemsets in F and C.

Our implementation of Max-Miner diverges from the pseudo-code description in only one way. During the second pass over the data-set, we use a two-dimensional array for quickly computing the support of all 2-itemsets as suggested in [3], and do not compute the support of the long itemsets $h(g) \cup t(g)$. We have found that the long itemsets $h(g) \cup t(g)$ almost always turn up infrequent at this stage, so computing their support offers no benefit.

3.4 Correctness and Efficiency Claims

THEOREM (CORRECTNESS): Max-Miner returns all and only the maximal frequent itemsets in the given data-set.

Proof: The fact that Max-Miner identifies and stores every maximal frequent itemset follows from the completeness of a set-enumeration tree search and the fact that branches of the set-enumeration tree are pruned if and only if they lead to only infrequent itemsets or non-maximal frequent itemsets. The fact that Max-Miner returns only those itemsets that are maximal frequent follows from the operation within the body of Max-Miner that continuously removes any itemset I if a frequent superset of I is known. □

Max-Miner, like Apriori, easily handles disk-resident data because it requires only one transaction at a time in memory. Also like Apriori, the number of passes over the data made by Max-Miner is bounded by the length of the longest frequent itemset. For Apriori

this bound is nearly tight. Max-Miner, on the other hand, often performs substantially fewer database passes, as will be demonstrated in our evaluation.

THEOREM (EFFICIENCY): Max-Miner makes at most $l+1$ passes over the data-set, where l denotes the length of the longest frequent itemset.

Proof: We establish the claim by demonstrating why Max-Miner terminates after pass $l+1$ if it happens to get that far. By the manner in which Max-Miner generates candidate groups, it is easy to see that during pass k, the head of any candidate group has exactly $k-1$ items. After pass $l+1$, for any candidate group g and any of its tail items i, $h(g) \cup \{i\}$ will be infrequent otherwise we would have a frequent itemset longer than l. This implies that the next set of candidate groups will be empty, in which case the algorithm terminates. □

4. Support Lower-Bounding

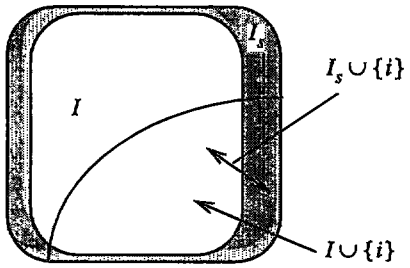
This section first describes a general technique that can be used to obtain a lower-bound on the support of an itemset by exploiting the support information provided by its subsets. The subsequent subsections describe how this technique is used to improve the performance of Max-Miner and Apriori.

4.1 Computing Support Lower-Bounds

The idea behind our method of computing a lower-bound on the support of an itemset is to exploit, as much as possible, the available support information provided by its subsets. The following function is useful in this endeavor, as established by the subsequent theorem. The function computes the number of transactions that are "dropped" from the supporting set of an itemset when it is extended with a given item.

DEFINITION: $\text{drop}(I_s, i) = \text{sup}(I_s) - \text{sup}(I_s \cup \{i\})$, where i is an item not in itemset I_s.

Figure 5. Illustration of support drop resulting from extending itemsets I and I_s with i.



Because $\text{sup}(I \cup \{i\}) = \text{sup}(I) - \text{drop}(I, i)$, we can get a lower-bound on the value of $\text{sup}(I \cup \{i\})$ given the value of $\text{sup}(I)$ and an upper-bound on $\text{drop}(I, i)$. This fact is exploited by our next theorem.

THEOREM (SUPPORT LOWER-BOUNDING): $\text{sup}(I) - \text{drop}(I_s, i)$ is a lower-bound on the support of itemset $I \cup \{i\}$ when $I_s \subset I$.

Proof: We show that $\text{drop}(I_s, i)$ is an upper-bound on $\text{drop}(I, i)$ from which the claim follows. The argument is geometric and is illustrated in Figure 5. The outer square represents the space of all transactions supporting itemset I_s, and the inner square the space of all transactions supporting itemset I. Because the space occupied by I is entirely contained within the space occupied by I_s, the set of transactions dropped by extending itemset I with i must be a subset of the transactions dropped by extending itemset I_s with i. The fact that $\text{drop}(I_s, i)$ is an upper-bound on $\text{drop}(I, i)$ is immediate. □

The support lower-bounding theorem, as stated, can only be applied to lower-bound the support of a k-itemset if we know the support of two $k-1$ item subsets. Below we generalize the theorem to apply when only the supports of smaller subsets are available.

THEOREM (GENERALIZED SUPPORT LOWER-BOUNDING): The following equation computes a lower-bound on the support of itemset $I \cup T$ where T is an itemset disjoint from I and $I_s \subset I$.

$$\text{sup}(I) - \sum_{i \in T} \text{drop}(I_s, i)$$

Proof: The equation is simply the result of applying the previous theorem repeatedly for each item i in T. □

4.2 Support Lower-Bounding in Max-Miner

Support lower-bounding can be used by Max-Miner within the Gen-Sub-Nodes function for additional superset-frequency pruning (Figure 6). If sub-nodes of a candidate group are generated while traversing the tail items in item order, a sub-node g_2 generated after g_1 will have the property that $h(g_2) \cup t(g_2) \subset h(g_1) \cup t(g_1)$. This means if $h(g_1) \cup t(g_1)$ is a frequent itemset, then any sub-node generated after g_1 can lead only to non-maximal itemsets. Max-Miner thus avoids generating sub-nodes following any sub-node g for which $h(g) \cup t(g)$ can be lower-bounded above minsup.

Figure 6. Generating sub-nodes with support lower-bounding.

```
GEN-SUB-NODES(Candidate Group g, Set of Cand. Groups C)
    ;; C is passed by reference and returns the sub-nodes of g
    ;; The return value of the function is a frequent itemset
    remove any item i from t(g) if h(g) ∪ {i} is infrequent
    reorder the items in t(g)
    for each i ∈ t(g) in increasing item order do
        let g' be a new candidate with h(g') = h(g) ∪ {i}
            and t(g') = {j | (j ∈ t(g)) and j follows i in t(g)}
        if COMPUTE-LB(g', h(g)) ≥ minsup
            then return h(g') ∪ t(g') ;; this itemset is frequent
        else C ← C ∪ {g'}
    return h(g) ;; This case arises only if t(g) is empty
```

Figure 7. Computing the support lower-bound.

```
COMPUTE-LB(Candidate Group g, Itemset I_s)
    ;; Returns a lower-bound on the support of h(g) ∪ t(g)
    ;; Itemset I_s is a proper subset of h(g)
    Integer d ← 0
    for each i ∈ t(g) do
        d ← d + drop(I_s, i)
    return sup(h(g)) - d
```

Figure 7 shows how Max-Miner computes a lower-bound on $h(g) \cup t(g)$ using only support information provided by its subsets. The function directly applies the generalized support lower-bounding theorem. Note that the support values required for this computation are available through the itemsets whose supports were computed while counting the support of the parent node.

4.3 Support Lower-Bounding in Apriori

Apriori generates candidate itemsets of length k whose supports are computed during database pass k. An itemset is made a candidate if every $k-1$ item subset was found frequent during the previous database pass. Suppose we are using Apriori to identify the maximal frequent itemsets. If Apriori can be made to lower-bound the support of a candidate itemset above or equal to minsup, then its support does not have to be explicitly counted. However, the candidate itemset cannot be entirely pruned because it must remain present for the subsequent candidate generation phase. Nevertheless, the savings that result from counting the support of fewer candidate itemsets can be substantial. Also, for a particular database pass, if ever Apriori can lower-bound the support of all candidate itemsets above minsup, then it can skip the database pass altogether.

A support lower-bound can be obtained on a candidate itemset I_c by plugging into the support lower-bounding theorem any proper subset I of I_c, and any proper subset I_s of I. The tightest bounds will be obtained by using only subsets with one less item. If the candidate itemset I_c has k items, there are then $(k-1)(k-2)$ subset combinations for lower-bounding the support of I_c. Our lower-bounding enhanced Apriori algorithm, Apriori-LB, considers every one of these possibilities and compares the best (largest) bound obtained to minsup.

Apriori-LB does not always have access to the exact support of every proper subset of a candidate because one or more subsets of a candidate itemset may have been left uncounted due to previous lower-bounding efforts. When this is the case, Apriori-LB uses a new function to bound support drop. The new function, drop-b, uses support bounds instead of exact support values. Because drop-b clearly computes an upper-bound on the value of drop given the same arguments, it can be used in place of drop in the lower-bounding theorems.

DEFINITION: drop-b(I_s, i) = ub-sup(I_s) − lb-sup$(I_s \cup \{i\})$, where ub-sup(I) is the best-known upper-bound on sup(I), and lb-sup(I) is the best-known lower-bound on sup(I).

The support of an itemset can be upper-bounded by the support (or upper-bound on support) of any of its subsets. Apriori-LB sets ub-sup(I) of an uncounted itemset I with k items to the smallest of the support upper-bounds or (if computed) support values of each of its $k-1$ item subsets.

5. Exploiting Additional Constraints

Though the frequent patterns themselves are often of interest to the end-user, often they need to be digested further before being presented. *Association rules* are digested frequent itemsets that can be useful for prediction. The *confidence* of an association rule $i_1, i_2, \ldots, i_k \rightarrow i_c$ is equal to the support of the itemset $\{i_1, i_2, \ldots, i_k\}$ divided by the support of the itemset $\{i_1, i_2, \ldots, i_k, i_c\}$. Typically the user is interested in finding only those association rules with high confidence and support, and these are produced by searching the entire space of frequent itemsets. Another database pass is required after finding all maximal frequent itemsets in order to obtain the supports of all frequent itemsets for producing association rules. If the frequent itemsets are long, even if implemented efficiently using specialized data structures, this step is hopelessly intractable.

Max-Miner can be used to identify many, though not all, high-confidence association rules during its execution. After counting the support of a candidate group g, Max-Miner has all the supports necessary to compute the confidence of any association rule of the form $h(g) \rightarrow i_c$ where $i_c \in t(g)$. Another approach at incomplete rule-mining could be to use the set of maximal frequent itemsets to define the space of rules searched by a randomized algorithm.

Though incomplete techniques at association rule mining may sometimes be sufficient, completeness is more desirable. We believe that incorporating additional constraints into the search for frequent patterns is the only way to achieve completeness on complex data. Association rule confidence is a constraint that Bayardo [5] uses to prune some itemsets from consideration. Other constraints that have been used during the search for patterns include item constraints [15] and information-theoretic constraints [13]. Interestingness constraints thus far applied only during post-processing (e.g. [6]) might also be exploitable during search to improve efficiency.

Max-Miner provides a framework in which additional constraints can often be easily integrated into the search. Consider as an example the problem of finding only the longest frequent itemsets. This constraint is quite powerful because data sets with long frequent itemsets usually have very many maximal frequent itemsets, of which only a small fraction are longest. We have implemented a version of Max-Miner that exploits this constraint called Max-Miner-LO ("Longest Only"). It determines the cardinality l of the longest itemset in the set of frequent itemsets F after each database pass. Any frequent itemsets in F that are shorter than l are then pruned, as are any candidate groups g in C such that $|h(g) \cup t(g)| < l$.

6. Evaluation

We selected data-sets from several domains for evaluating the performance of Max-Miner, all but one of them being publicly accessible through the web. The data-set that is not publicly available was provided by a retailer. This data-set contains records listing all of the items purchased by a customer over a period of time. We produced another of our data-sets from PUMS census data available at http://augustus.csscr.washington.edu/census/comp_013.html. Following Brin et al. [6], we discretized continuous attributes and removed uninformative attributes. We created another version of the data-set where all items with 80% or greater support were discarded. The raw data from the web consists of both housing and person records, but we extracted only the person records. The remaining data-sets used in our evaluation were taken from the Irvine Machine Learning Database Repository (http://www.ics.uci.edu/~mlearn/MLRepository.html). We favored those with categorically-valued attributes, relatively wide record length, and a substantial number of records. These data-sets include connect-4, chess, splice, and mushroom. The splice data set contains DNA data, the mushroom database contains records describing the characteristics of various mushroom species, and Connect-4 and chess are compiled from game state information. Table 1 lists the width and height of each of these data-sets. Pumsb* is the same data set as pumsb minus all items with 80% or more support.

Table 1. Width and height of the evaluation data-sets.

Data-set	Records	Avg. Record Width
chess	3,196	37
connect-4	67,557	43
mushroom	8,124	23
pumsb	49,046	74
pumsb*	49,046	50
retail	213,972	31
splice	3,174	61

The data-set with the longest patterns was pumsb. This data-set also had the most maximal frequent patterns -- intractably many even at relatively high levels of support. We thus focus on pumsb* for evaluating Max-Miner, and use pumsb only to evaluate Max-Miner-LO. Even though items with 80% or more support are removed, pumsb* still a challenging data set with long frequent itemsets. A similar but smaller data-set was used by Brin et al. to evaluate DIC, though only at supports of 36% and higher. Their data-set was not available to us at the time of this writing, so we created pumsb* to use in its place.

All experiments were performed on a lightly loaded 200MHz Power-PC with 256 megabytes of RAM. The algorithms were implemented in C++ atop the same optimized hash tree implementation. Apriori and Apriori-LB were optimized for finding only maximal frequent itemsets by having them discard from memory any frequent itemset that was found to be non-maximal. The Max-Miner algorithm evaluated here uses support lower-bounding. In subsection 6.3 we describe the effects of disabling this and other optimizations.

Figure 8. CPU time on pumsb*.

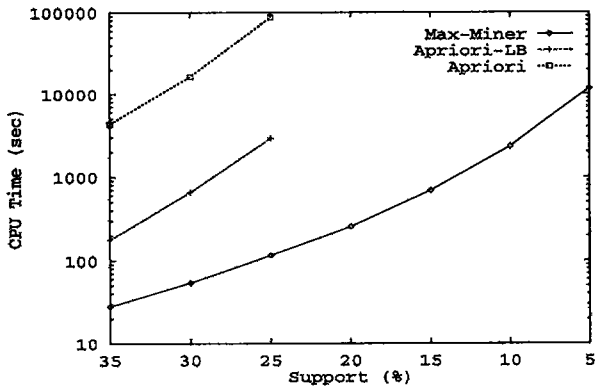


Figure 9. CPU time on mushroom.

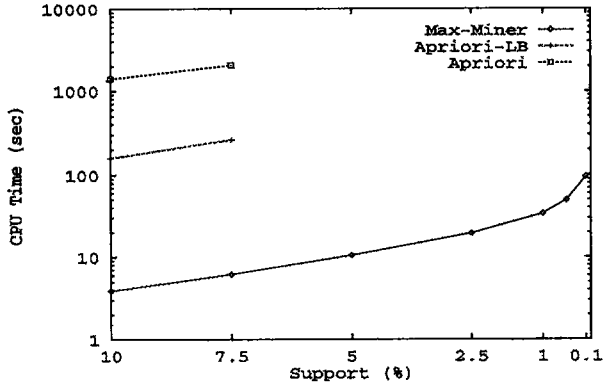


Figure 10. CPU time on chess.

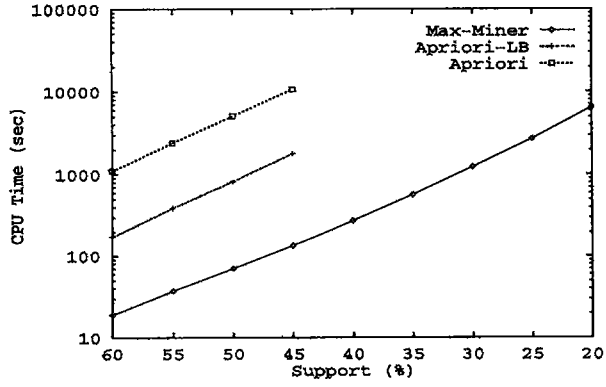
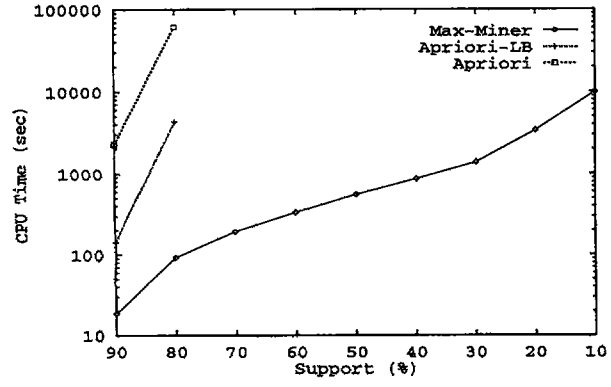


Figure 11. CPU time on connect-4.



6.1 Max-Miner versus Apriori and Apriori-LB

Figures 8 through 11 compare the performance of Max-Miner, Apriori-LB, and Apriori on the most difficult of the evaluation data-sets. While Apriori-LB is performing far better than Apriori with respect to run-time (note the logarithmically scaled y axes), because of its space complexity, we were unable to run it at lower support than Apriori without exceeding the memory of our machine. At several data points, Max-Miner is over two orders of magnitude faster than Apriori at identifying maximal frequent patterns, and an order of magnitude faster than Apriori-LB. Had we allowed Apriori to perform candidate paging, the speedups at lower supports would be even more substantial. Even though these data sets are from distinctly different domains, the performance trends are all identical. What these data-sets all have in common are long patterns at relatively high values of support.

Figure 12. CPU time on retail.

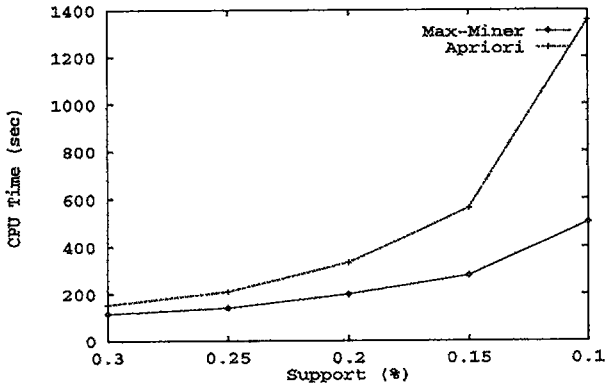
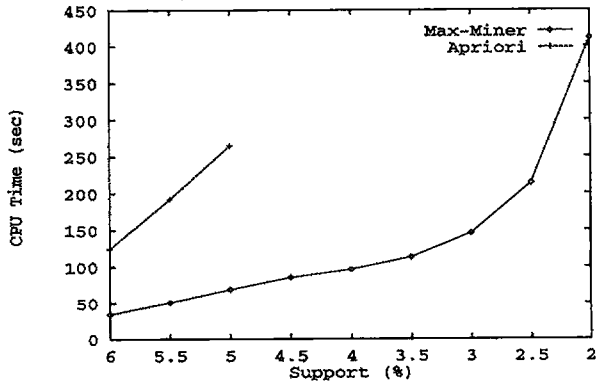
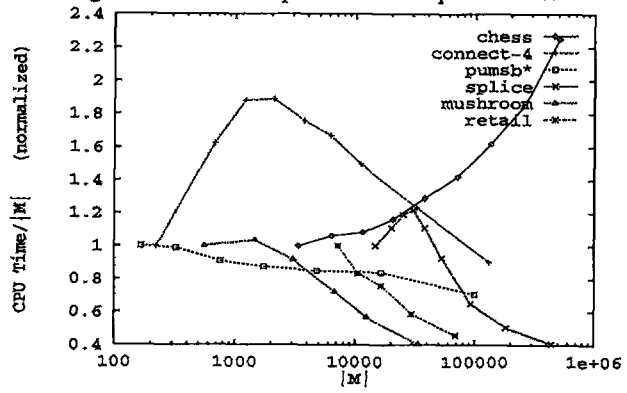


Figure 13. CPU time on splice.



The two remaining evaluation data-sets, retail and splice, have comparatively small patterns at low support levels. Even so, Max-Miner still outperforms both Apriori and Apriori-LB. We plot only the performance of Apriori on these data sets, since Apriori-LB offered no advantages. Apriori-LB appears to be more effective at tightly lower-bounding support when the patterns are long. Most of the maximal frequent itemsets in these data sets are of length 2 to 4, and the longest are of length 8 in splice and 6 in retail at the lowest support values. The superior performance of Max-Miner on these data-sets arises from considering fewer candidate itemsets and reduced index overhead resulting from the indexing of candidate groups using only head items. This suggests that Apriori should also index candidate itemsets that share a common prefix as a group rather than individually. We are currently investigating whether this optimization pays off more generally.

Figure 14. CPU time per maximal frequent itemset.

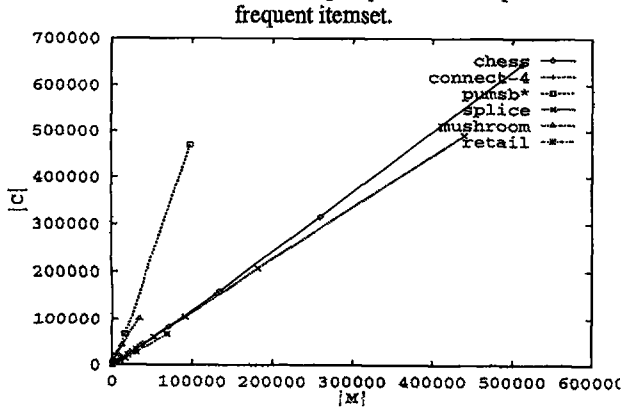


6.2 Max-Miner Scaling

For every data set at the support levels we looked at, the lower the support, the more maximal-frequent patterns were found. Figure 14 plots the amount of time spent by Max-Miner per maximal frequent itemset against the number of maximal frequent itemsets found during a given run of the algorithm. The support values used to generate this data are the same ones that appear in the previous graphs. Max-Miner's performance per maximal frequent itemset remains relatively constant as the number of maximal frequent itemsets increases, even though the size of the longest itemsets varies significantly. For instance, for the pumsb* data set, the left-most point arises from a run at 35% support where the longest frequent itemsets contain 15 items. The right-most point is for a run at 5% support where the longest frequent itemsets contain 40 items. Even so, the performance per maximal frequent itemset varies no more than 25%, indicating Max-Miner is scaling roughly linearly with the number of maximal frequent itemsets.

The chess data-set exhibits the most non-linear increase in difficulty with the number of maximal frequent itemsets, though this increase is still relatively subdued (within a factor of 2.5) considering the large increase in the number of maximal frequent itemsets and their length. It is possible that the curve for the chess data-set would begin to decrease had we mined at even lower levels of support. We were unable to determine this because of the explosion in the number of maximal frequent itemsets at low supports -- already there are over half a million maximal frequent itemsets at a minsup of 20%. The other data-sets all begin to get easier as the number of maximal frequent patterns increases beyond some point. This is certainly in part due to the fact that itemsets with lower support incur less overhead since they are contained by fewer database transactions. However, as support decreases, maximal frequent itemsets become longer, which leads to an increase in storage and indexing overhead.

Figure 15. Number of candidate groups considered per maximal frequent itemset.



To remove the effect of indexing and support-counting overhead on runtime scaling, and to demonstrate how Max-Miner scales in its space consumption, we also compared the number of candidate groups whose support was counted to the number of maximal frequent sets identified (Figure 15)[1]. For every data-set there is a strong linear relationship. In fact, the number of candidates considered is always within a small (at most 3.7 on pumsb*) constant of the number of maximal frequent itemsets.

Figure 16. Database passes performed by Max-Miner compared to the longest frequent itemset.

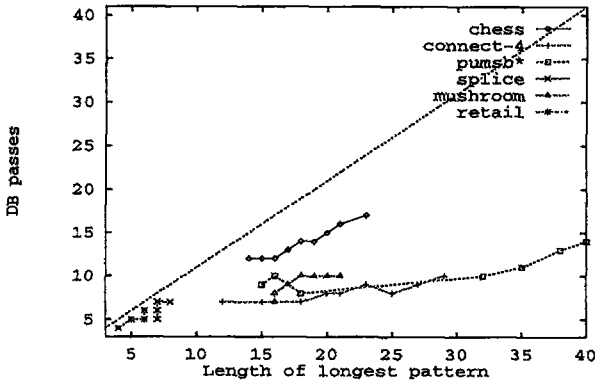


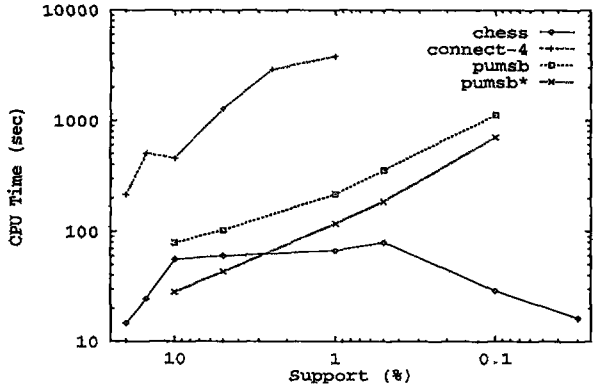
Figure 16 plots the number of database passes performed by Max-Miner against the length of the longest patterns identified during each run. The worst-case number of passes is represented by the diagonal. While the number of database passes increases with pattern length, it is usually far from the worst-case. Though not illustrated, Apriori-LB was often able to skip one or two database passes because all candidates were lower-bounded above minsup. This had a relatively small effect on performance compared to the amount of reduction in overhead from computing the support of fewer candidates. The reductions in database passes tended not to have as dramatic an effect on performance as might be expected from experiments on small-pattern data because the overhead of candidate itemset lookup largely exceeded that of data access.

6.3 Effects of Max-Miner Optimizations

Support lower-bounding is a beneficial optimization on the data-sets with long patterns for Max-Miner as well as Apriori. For example, after turning off support lower-bounding, Max-Miner's performance dropped by approximately four-fold at all levels of support on the chess data-set due to a four-fold increase in the number of candidate groups to be counted. On the retail data-set, the optimization's effects were negligible. The item-(re)ordering heuristic was beneficial, usually dramatically so, on every data-set we looked at. For example, at 80% support on the chess data set, turning off item ordering (which caused the algorithm to default to a static lexical item ordering) resulted in an order of magnitude decrease in performance, with the performance gap widening even further as support was decreased. The decrease in performance was due to an increase in candidate groups resulting primarily from less opportunities for superset-frequency pruning.

[1] The plot for the connect-4 data-set is difficult to make out because it lies directly along the lower-left portion of the plot for chess.

Figure 17. Performance of Max-Miner-LO



6.4 Max-Miner-LO

Figure 17 shows the runtime of Max-Miner-LO on the data sets with the longest frequent itemsets. We chose very low support values for these runs to illustrate how additional constraints can make mining feasible even when the number of maximal frequent itemsets is too large. Note that Max-Miner-LO can mine on the pumsb data-set without removing items with 80% or more support. The longest frequent itemsets in pumsb at .1% support contain 72 items. Frequent itemsets were very long even at 10% support where they contained 61 items. For most of these data points, the number of longest frequent itemsets was under 300.

6.5 Comparison with Pincer-Search

Because our work was done concurrently, we have thus far only had the opportunity to perform a preliminary comparison with Lin and Kedem's Pincer-Search algorithm [8]. We have run Max-Miner on the data-set from their evaluation with the longest frequent itemsets (T20.I15.D100K). Unfortunately, it was not challenging enough for either algorithm to draw any solid conclusions. The only clear case of one algorithm appearing superior over the other is at 8% minsup where Pincer-Search requires over 20,000 seconds and considers over 150,000 candidate itemsets. At this same support value Max-Miner requires 125 seconds and considers less than 22,000 candidate itemsets (within 4,894 candidate groups)[1].

Pincer-Search uses a candidate itemset generation procedure that couples Apriori candidate generation with another technique for generating long candidate itemsets used for superset-frequency pruning. The long candidate generation procedure is seeded with one itemset that contains every frequent item. For each infrequent itemset encountered after a database pass, any long itemset containing the infrequent itemset is replaced by itemsets that do not contain the infrequent itemset. The new itemsets are formed by removing a single item from the original, and keeping only those itemsets that are not subsets of any other long candidate itemset. The process is iterated until no long itemset contains any known infrequent itemset.

After the second database pass, the Pincer-Search long candidate itemset generation procedure amounts to identifying all the maximal cliques in a graph where the nodes are the frequent 1-itemsets and the edges are the frequent 2-itemsets, much as Zaki's MaxClique algorithm [16]. Unlike MaxClique, Pincer-Search uses a top-down instead of bottom-up approach for finding the cliques. But due to the fact that the problem is NP-hard, any approach at generating candidates in this manner may be prone to performance problems when the frequent itemsets (and hence the maximal cliques) are long.

[1] Candidate itemset counts for both Pincer-Search and Max-Miner do not include the 1 and 2-itemsets. Runtimes are not directly comparable due to differences in hardware and implementation details.

Though similar in principle, Max-Miner and Pincer-Search are quite different in their details. We look forward to performing a more exhaustive comparison and feel it is likely that the algorithms will prove complementary.

7. Conclusions

We have presented and evaluated the Max-Miner algorithm for mining maximal frequent itemsets from large databases. Max-Miner applies several new techniques for reducing the space of itemsets considered through superset-frequency based pruning. The result is orders of magnitude in performance improvements over Apriori-like algorithms when frequent itemsets are long, and more modest though still substantial improvements when frequent itemsets are short. Max-Miner is also easily made to incorporate additional constraints on the set of frequent itemsets identified. Incorporating these constraints into the search is the only way to achieve tractable completeness at low supports on complex data-sets. It is therefore these extensions which we feel warrant the most future work. In particular, we hope there exists a clean way to exploit many of the wide variety of interestingness constraints during the search rather than applying them only in a post-processing filtering step.

Acknowledgments

I thank Ramakrishnan Srikant, Rakesh Agrawal, Sunita Sarawagi, and Dimitrios Gunopulos for many helpful suggestions and comments, Robert Schrag for directing me to the work of Ron Rymon, and Dao-I Lin for sharing details of the experiments used to evaluate Pincer-Search.

References

[1] Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Mining Association Rules between Sets of Items in Large Databases. In *Proc. of the 1993 ACM-SIGMOD Conf. on Management of Data*, 207-216.

[2] 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.

[3] 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.

[4] Agrawal, R. and Srikant, R. 1995. Mining Sequential Patterns. In *Proc. of the 11th Int'l Conf. on Data Engineering*, 3-14.

[5] 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.

[6] 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 Conf. on Management of Data*, 255-264.

[7] 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.

[8] 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*, to appear.

[9] 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 ACM-SIGMOD Conf. on Management of Data*, 175-186.

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

[11] 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.

[12] Slagel, J. R.; Chang, C.-L.; and Lee, R. C. T. 1970. A New Algorithm for Generating Prime Implicants. *IEEE Trans. on Computers*, C-19(4):304-310.

[13] 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.

[14] Srikant, R. and Agrawal, R. 1996. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. of the Fifth Int'l Conf. on Extending Database Technology*, 3-17.

[15] 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.

[16] 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.

APPENDIX B

Border-Based Mining of Emerging Patterns

The disclosure of the accompanying materials is hereby incorporated in this application.

Border-Based Mining of Emerging Patterns

A collection $\mathcal{S}$ of sets is called *interval closed* if, for all $X, Z \in \mathcal{S}$ and for all Y, it is the case that $Y \in \mathcal{S}$ whenever $X \subseteq Y \subseteq Z$. For example, $\mathcal{S} = \{\{1,2\}, \{2,3\}, \{1,2,3\}, \{1,2,4\}, \{2,3,4\}, \{1,2,3,4\}\}$ is interval-closed.

The collection of all large itemsets w.r.t. any fixed threshold (itemsets whose supports are larger than the threshold) is interval closed. Borders offer succinct representation of such collections, which are typicall very large.

An ordered pair $<\mathcal{L}, \mathcal{R}>$ is called a *border*, $\mathcal{L}$ the *left-hand bound* of this border and $\mathcal{R}$ the *right-hand bound*, if (a) each of $\mathcal{L}$ and $\mathcal{R}$ is an antichain[1] collection of sets, and (b) each element of $\mathcal{L}$ is a subset of some element in $\mathcal{R}$ and each element of $\mathcal{R}$ is a superset of some element in $\mathcal{L}$. The collection of sets *represented* by, or the *set interval* of, a border $<\mathcal{L}, \mathcal{R}>$, is $[\mathcal{L}, \mathcal{R}] = \{Y \mid \exists X \in \mathcal{L},\ \exists Z \in \mathcal{R}$ such that $X \subseteq Y \subseteq Z\}$. The collection $[\mathcal{L}, \mathcal{R}]$ is said to *have* $<\mathcal{L}, \mathcal{R}>$ as border.

The set interval of $<\{\{1\}, \{2,3\}\}, \{\{1,2,3\}, \{2,3,4\}\}>$ is $\{\{1\}, \{1,2\}, \{1,3\}, \{1,2,3\}, \{2,3\}, \{2,3,4\}\}$. The set interval of $<\{\{1,2\}\}, \{\{1,2,3,4,5\}, \{1,2,4,5,6\}\}>$ consists of 12 itemsets: all sets that are both supersets of $\{1,2\}$ and subsets of either $\{1,2,3,4,5\}$ or $\{1,2,4,5,6\}$. Many simple borders' set intervals are very large collections; e.g. there are 2^{11} itemsets in the set interval of $<\{\{1\}\}, \{\{1,2,3,4,5,6,7,8,9,10,11,12\}\}>$.

There is a one-to-one correspondence between borders and interval-closed collections: Each interval-closed collection $\mathcal{S}$ of sets has a unique border $<\mathcal{L}, \mathcal{R}>$, where $\mathcal{L}$ is the collection of minimal sets in $\mathcal{S}$ and $\mathcal{R}$ is the collection of maximal sets in $\mathcal{S}$.

Our border-based algorithms can discover *all* EPs satisfying a certain support constraint, and they do this without enumerating elements of very large collections. In fact, the entire process of discovering EPs in this rectangle only needs to deal with borders: They will take borders, that represent collections of large itemsets, as inputs; they only manipulate borders in the internal process; and they produce as output borders that represent EPs. As a consequence, the running time of this process is short for most practical situations, even when the number of EPs is huge.

We concentrate our discussion on the situation when two large borders, one from $\mathcal{D}_1$ and one from $\mathcal{D}_2$, are available. The differential procedure, called BORDER-DIFF, is given first. Then the main algorithm, MBD-LLBORDER, using BORDER-DIFF as a subroutine, is given.

BORDER-DIFF aims to derive the differential between a pair of borders with a special form: Given a pair of borders $<\{\emptyset\}, \{U\}>$ and $<\{\emptyset\}, \mathcal{R}_1>$, BORDER-DIFF derives another border $<\mathcal{L}_2, \{U\}>$ such that $[\mathcal{L}_2, \{U\}] = [\{\emptyset\}, \{U\}] - [\{\emptyset\}, \mathcal{R}_1]$.

BORDER-DIFF($<\{\emptyset\}, \{U\}>$, $<\{\emptyset\}, \{S_1, S_2, \cdots, S_k\}>$) ;; improved version
;; *return border of* $[\{\emptyset\}, \{U\}] - [\{\emptyset\}, \{S_1, S_2, \cdots, S_k\}]$
1) initialize $\mathcal{L}$ to $\{\{x\} \mid x \in U - S_1\}$;
2) **for** $i = 2$ to k **do**
3) $\quad \mathcal{L} \leftarrow \{X \cup \{x\} \mid X \in \mathcal{L}, x \in U - S_i\}$;
4) $\quad$ remove all Y in $\mathcal{L}$ that are not minimal;
5) **return** $<\mathcal{L}, \{U\}>$;

We now illustrate how BORDER-DIFF works on the arguments of $<\{\emptyset\}, \{1,2,3,4\}>$ and $<\{\emptyset\}, \{\{3,4\}, \{2,4\}, \{2,3\}\}>$. It first initializes $\mathcal{L}$ to $\{\{1\}, \{2\}\}$ since $U - S_1 = \{1,2\}$. Then it updates $\mathcal{L}$ to $\{\{1\}, \{1,2\}, \{1,3\}, \{2,3\}\}$ since $U - S_2 = \{1,3\}$, and reduces it to $\{\{1\}, \{2,3\}\}$. Finally it updates $\mathcal{L}$ to $\{\{1\}, \{1,2,3\}, \{1,4\}, \{2,3,4\}\}$ since $U - S_3 = \{1,4\}$, and reduces it to $\{\{1\}, \{2,3,4\}\}$.

[1] A collection $\mathcal{S}$ of sets is an *antichain* if X and Y are incomparable sets (i.e. $X \not\subseteq Y$ and $Y \not\subseteq X$) for all $X, Y \in \mathcal{S}$.

Assume we have found LARGEBORDER$_\delta(\mathcal{D}_1^*) = <\{\emptyset\}, \{C_1, C_2, \cdots, C_m\}>$ (which represents all δ-large itemsets from $\mathcal{D}_1^*$) and LARGEBORDER$_\theta(\mathcal{D}_2^*) = <\{\emptyset\}, \{D_1, D_2, \cdots, D_n\}>$ for some δ and θ satisfying $\theta = \rho * \delta$. The MBD-LLBORDER algorithm will find all EPs satisfying: their supports in $\mathcal{D}_2^*$ are $\geq \delta * \rho$ but their supports in $\mathcal{D}_1^*$ are $< \delta$.

MBD-LLBORDER(LARGEBORDER$_\delta(\mathcal{D}_1^*)$, LARGEBORDER$_\theta(\mathcal{D}_2^*)$)
;; *return all emerging patterns satisfying (*) by multiple calls of* BORDER-DIFF
1) *EPborders*$\leftarrow$ {};
2) **for** j from 1 to n **do**
3) if some C_i is a superset of D_j then **continue**;
4) $\{C_1', \cdots, C_m'\} \leftarrow \{C_1 \cap D_j, \cdots, C_m \cap D_j\}$;
5) *RightBound* $\leftarrow$ the set of all maximal itemsets in $\{C_1', \cdots, C_m'\}$;
6) add BORDER-DIFF($<\{\emptyset\}, D_j>, <\{\emptyset\}, RightBound>$) into *EPborders*;
7) **return** *EPborders*;

We now illustrate how MBD-LLBORDER works for

$$LargeBorder_\delta(\mathcal{D}_1) = <\{\emptyset\}, \{\{2,3,5\}, \{3,4,6,7,8\}, \{2,4,5,8,9\}\}>$$

$$LargeBorder_\theta(\mathcal{D}_2) = <\{\emptyset\}, \{\{1,2,3,4\}, \{6,7,8\}\}>.$$

For $D_1 = \{1,2,3,4\}$, $C_1 = \{2,3,5\}$, $C_2 = \{3,4,6,7,8\}$, $C_3 = \{2,4,5,8,9\}$, step 4 of MBD-LLBORDER produces $C_1' = \{2,3\}$, $C_2' = \{3,4\}$, $C_3' = \{2,4\}$; it then calls BORDER-DIFF with $<\{\emptyset\}, \{\{1,2,3,4\}\}>$ and $<\{\emptyset\}, \{\{2,3\}, \{3,4\}, \{2,4\}\}$, since $C's$ are all minimal. BORDER-DIFF algorithm then returns $<\{\{1\}, \{2,3,4\}\}$, $\{\{1,2,3,4\}\}>$. Since $\{6,7,8\} \subseteq \{3,4,6,7,8\} = D_2$, MBD-LLBORDER does not call BORDER-DIFF for D_2 since the answer is known to be empty.

CLAIMS:

1. A system for extracting emerging patterns from data comprising the following components:

   An input component for reading samples from different input categories (say C_1, ..., C_n).

   An adjustable discretizing component for discretizing the samples read.

   An optional component for the storage and management of samples.

   An optional adjustable component for reading optional conditions for derivation of emerging patterns. These optional conditions are given below.

   - A threshold on support level. The support level of a pattern (some time called an itemset) I in a category C is defined as

     $$supp_C(I) = \text{the percentage of samples in } C \text{ that exhibit that } I$$

     If this threshold is not given, a default will be used or derived from the input data. For example, 1%.

   - A threshold on growth rate. Given two categories C_a and C_b of samples, the growth rate of a pattern (or itemset) I from C_b to C_a is defined as

* C L M S *

$$growthrate_{C_b \to C_a}(I) = \begin{cases} 0, & \text{if } supp_{C_b}(I) = 0 \text{ and } supp_{C_a}(I) = 0 \\ \text{infinity}, & \text{if } supp_{C_b}(I) = 0 \text{ but not } supp_{C_a}(I) = 0 \\ supp_{C_a}(I) / supp_{C_b}(I), & \text{otherwise} \end{cases}$$

If this threshold is not given, a default will be chosen or derived from the input training data. For example, 5.

Given a category C, its opponent category is C' which is defined to contain all instances not in category C.

A pattern or itemset whose growth rate exceeds this threshold is considered an emerging pattern from *C'* to *C*, or simply an emerging pattern of category *C*.

- A function *weight(g)* for weighting growth rate *g*. For growth rate > 1, this function should be a monotonic increasing function which always takes nonnegative values.

  If this function is not given, a default will be chosen. For example,

  $$weight(g) = g/(g+1)$$

- A function *score(s,C)* specifying the aggregate differentiating score of a sample *s* and a set *E(C)* of emerging patterns of a category *C*. This function should involve (i) the support of emerging patterns in *E(C)* and (ii) weighted growth rates of emerging patterns in *E(C)*.

  If this function is not given, a default will be chosen. For example,

  $$score(s,C) = \text{sum of } supp_C(I) * weight(growthrate(I)) \text{ over all pattern } I \text{ that appears in } s \text{ and in } E(C)$$

- A base score *base_score(C)* on the aggregate differentiating score for each category *C*. This base score can be given as a percentile of the range of aggregate differentiating scores of the training samples of that category.

  If this threshold is not given, a default will be chosen. For example, the 50th percentile.

- An indication of whether the set of derived emerging patterns should be reduced.

  If this indication is not given, a default decision will be made. For example, some related emerging patterns will be eliminated unless the reduction leads to a poor coverage (i.e., leads to a larger number of zero scores) on training samples.

An optional component for the storage and management of the extracted emerging

patterns and their derivation conditions.

A component for displaying or printing of extracted emerging patterns.

A component for mining emerging patterns for all categories.

An optional component for reducing the number of related emerging patterns. Two emerging patterns are related if one is a sub-pattern (or subset) of the other.

A component for computing aggregate differentiating scores for all samples and categories.

A component for computing the base scores for all categories.

2. A system for extracting emerging patterns from data as given in *Claim 1*, with the component for reducing the number of related emerging patterns replaced by one that specifically prefers patterns that cover more training samples and have stronger differentiating power.

   An instantiation of such a component is the following process. We first sort the emerging patterns from C' to C into a list E in decreasing order of growth rate and support. The set of essential emerging patterns, *essE*, is initialized to contain the first emerging pattern in E. Then for each next pattern I in E we do the following in order:

   For each J in *essE* such that I is a sub-pattern (or subset) of J, we replace J by I if either of the following is true:

   - $growthrate_{C'->C}(I)$ exceeds $growthrate_{C'->C}(J)$
   - $supp_C(I)$ greatly exceeds $supp_C(J)$ and $growthrate_{C'->C}(I)$ exceeds the threshold on growth rate.

   We add I to *essE* if both of the above two conditions are false and I is not a super-pattern (or superset) of any pattern in essE.

3. A system for extracting emerging patterns from data as given in *Claim 1*, with the component for mining of emerging patterns replaced by one that uses manipulation of borders.

   An ordered pair (L,R) is called a border, if each of L and R is an antichain collection of sets, each element of L is a subset of some element of R, and each element of R is a superset of some element in L. The collection of sets represented by (or the set interval of) such a border are those sets Y such that Y is superset of some element of L and is subset of some element of R.

4. A system for extracting emerging patterns from data as given in *Claim 1*, with the component for mining of emerging patterns replaced by one that uses the following specific algorithm.

   Given the categories C' and C of samples. First use MaxMiner [R. J. Bayardo. Efficiently mining long patterns from databases *Proceedings of ACM-SIGMOD International Conference on Management of Data* , 85--93, June 1998.] to discover the two large borders of the large itemsets in C' and C having appropriate support thresholds. Then we use *MBD-LLBORDER* to find all the emerging pattern borders. Finally, we enumerate the emerging patterns contained in these borders, and go through samples in C' and C to check their actual supports and growth rates.

   Assume we have found $LARGEBORDER_d(C')$ and $LARGEBORDER_t(C)$ for some d and t

satisfying $t = p * d$, where p is an appropriate threshold on growth rate. The *MBD-LLBORDER* algorithm will find all emerging patterns such that their supports in C exceed $d * p$ but their supports in C' are less than d. The *MBD-LLBORDER* algorithm is as follows. Let $LARGEBORDER_d(C')$ be $(\{\{\}\}, \{C_1, C_2, ..., C_m\})$. Let $LARGEBORDER_t(C)$ be $(\{\{\}\}, \{D_1, D_2, ..., D_n\})$.

MBD-LLBORDER($LARGEBORDER_d(C')$, $LARGEBORDER_t(C)$)

- *EPBORDERS <- {};*
- *for j from 1 to n do*
  - *if some C_i is superset of D_j then continue;*
  - *$\{C_1', ..., C_m'\}$ <- $\{C_1$ intersect $D_j, ..., C_m$ intersect $D_j\}$;*
  - *RIGHTBOUND <- the set of all maximal itemsets in $\{C_1', ..., C_m'\}$;*
  - *add BORDER-DIFF($(\{\{\}\}, D_j)$, $(\{\{\}\}$, RIGHTBOUND)) into EPBORDERS;*
- *return EPBORDERS;*

The function *BORDER-DIFF* is employed to derived differentials between a pair of borders of special forms: Given a pair of borders $(\{\{\}\}, \{U\})$ and $(\{\{\}\}, R)$, *BORDER-DIFF* derives another border $(L, \{U\})$ such that the collection of sets represented is exactly those sets represented by $(\{\{\}\}, \{U\})$ but not represented by $(\{\{\}\}, R)$. Importantly, *BORDER-DIFF* achieves this by manipulating only the itemsets in the borders.

BORDER-DIFF($(\{\{\}\}, \{U\})$, $(\{\{\}\}, \{S_1, ..., S_k\})$)

- *initialize L to $\{\{x\} \mid x$ in $U - S_1\}$;*
- *for i from 2 to k do*
  - *L <- $\{X$ union $\{x\} \mid X$ in L, x in $U - S_i\}$;*
  - *remove all itemsets Y in L that are not minimal;*
- *return (L, {U});*

5. A system for classifying data comprising the following components:

   A means for extracting emerging patterns from data such as that described in *Claim 1* above.

   A component for input of samples to be classified.

   A component for output of classification decisions on the samples.

   A component for computing the base scores of aggregate differentiating scores for all samples and categories.

   A component to assign to a category to each sample. The category that is assigned to that sample is the one whose normalized score (ie. aggregate differentiating score divided by the corresponding base score) is maximal for that sample.

6. A system for ranking and classifying data comprising the following components:

   A means for extracting emerging patterns from data such as that described in *Claim 1* above.

* C L M S *

A component for input of samples to be classified.

A component for output of ranked classification decisions on the samples.

A component for computing the base scores of aggregate differentiating scores for all samples and categories.

A component that ranks each category against each sample. The rank by measuring the normalized score (ie. aggregate differentiating score divided by the corresponding base score) for that sample. The greater the normalized score, the higher the rank.

* C L M S *