Speeding up Knowledge Discovery in Large Relational Databases by Means of a New Discretization Algorithm.

Alex Alves Freitas and Simon H. Lavington
{freial,lavis}@essex.ac.uk
University of Essex, Dept. of Computer Science
Wivenhoe Park, Colchester, CO4 3SQ, UK

Abstract

Most of the KDD (Knowledge Discovery in Databases) algorithms proposed in the literature have been applied to relatively small datasets and do not permit any integration with a DBMS. Hence, the application of these algorithms to the huge amounts of data found in current databases and data warehouses faces serious scalability problems, particularly the problem of excessive learning time. This paper investigates a way of improving the scalability of KDD algorithms, via discretization of ordinal or continuous attributes. This work has two novel aspects. First, we map a generic discretization primitive into an SQL query. Second, we propose a new discretization algorithm for classification tasks. We show how the new discretization algorithm can be implemented with good effect via the SQL primitive.

1 Introduction: the Problem.

Current databases and data warehouses contain huge amounts of data. Clearly, there is a strong need for extracting interesting, useful information from such large datasets [Piatetsky-Shapiro & Frawley 91], [Fayyad et al. 96].

There are many hundreds of KDD algorithms which have emerged from research areas such as Machine Learning and Statistics, to which may be added more recent KDD activity in Neural Networks and Genetic Algorithms. However, there is no single KDD algorithm that is ideally satisfactory for all classes of database and all kinds of KDD task [Schaffer 94], [Michie et al. 94]. Therefore, any practical KDD system has to have a range of available techniques that cover classification, clustering, association and summarisation on data that is both categorical and numeric. Furthermore, any practical KDD system has to be able to deal efficiently with large quantities of data within a commercial-strength DBMS.

Despite the demands of large databases and data warehouses, “conventional” Machine Learning (ML) algorithms have been applied mainly to relatively small datasets (typically less than 10,000 tuples) and do not have any integration at all with RDBMS. Hence, the application of these algorithms to very large amounts of data faces serious scalability problems, particularly the problem of excessive learning time. For instance, [Catlett 91] estimated that C4.5 would take several months to learn from 1,000,000 tuples by using state-of-the-art hardware at that time. More recently, [Provost & Aronis 96] report that a standard version of the RL algorithm is impractical (i.e. take too long to run) on datasets of more than 70k tuples.

In this paper we address an important aspect of efficiency which relates to data that is numeric - i.e. data whose attributes are ordinal (integer-valued) or continuous (real-valued) variables. Some existing KDD algorithms (e.g. ITRULE [Smyth &

Goodman 91]) cannot handle such attributes unless the attributes are first converted into a small set of discrete values (or categories). Furthermore, for those ML algorithms that can handle ordinal/continuous attributes directly, the processing of these attributes is much slower than processing categorical attributes [Paliouras & Bree 95], [Catlett 91a]. Finally, discretization usually improves the comprehensibility of the discovered knowledge [Catlett 91a].

Therefore, on the grounds of speed and ease of use, there is much to be gained from developing efficient and generic discretization techniques which can be used by a range of KDD systems. However, there are two caveats. Firstly, an inherent disadvantage of discretization is that it may reduce the accuracy of the discovered knowledge since discretization, along with any kind of summarization process, may cause some relevant detailed information to be lost. Secondly, any new discretization technique should for preference be useable within a RDBMS environment so as to be compatible with current commercial data processing practice.

The challenge is therefore to develop a generic discretization technique that is SQL-compatible, improves KDD run-times, and does not unduly diminish the accuracy of the discovered knowledge.

By way of further justification for emphasizing SQL, we have shown in a recent paper [Freitas & Lavington 96] that other generic KDD activities may be mapped into SQL. In particular, we have implemented a decision-tree learner using generic KDD primitives expressed as SQL queries, and evaluated its performance on a parallel DB server. It is our belief that the use of suitable SQL primitives is a promising strategy for developing efficient KDD on large volumes of real-world data.

The rest of this paper is organized as follows. Section 2 presents our approach to the discretization process. Section 3 reviews other work on discretization algorithms. Then in Section 4 we propose a generic SQL-based primitive which underpins the discretization process. In Section 5 we show how this primitive may be used to implement a new class-driven, information-theoretic discretization algorithm. In Section 6 we give the results of running this new algorithm on test datasets, and finally Section 7 presents some conclusions.

2 Approach to Discretization.

We follow a Machine Learning (ML) approach for KDD. However, many arguments of this paper also apply to a Statistics-based approach for KDD, since the core of ML and Statistics methods are often very similar. We focus on relational databases (RDBMS) and on classification tasks (also called supervised learning), where each tuple belongs to a class, among a pre-defined set of classes. Tuples consist of a set of predicting attributes and a class attribute. For instance, a tuple may represent an employee record, where the predicting attributes are the employee's age, gender, educational level, etc; and the class attribute is the managerial level of the employee. The goal of the classification algorithm is to find some relationship (e.g. in the form of a decision tree or a set of rules [Quinlan 93]) between the predicting attributes and the class one. Thus, the classification process can use this relationship to predict the class of a new, unknown-class tuple.

We present a class-driven discretization algorithm, i.e. an algorithm that takes into account the class of a tuple when carrying out the discretization procedure. Hence, this algorithm was designed particularly for classification tasks. Our work has two novel aspects. First, we map a generic primitive operation of discretization algorithms into an SQL query. Second, we propose a bottom-up, class-driven discretization algorithm based on information theory. This algorithm differs from other bottom-up, class-driven discretization algorithms, which are typically based on a statistical measure of association (or significance test) - see Section 3.

We favour a tight integration between the KDD system and the DB system. This allows the KDD system to benefit from several important database facilities, such as integrity maintenance, recovery, security and concurrency control. In addition, it helps to scale up KDD systems for very large data sets, by using efficiency-oriented database facilities such as automatic query optimization. See e.g. [Provost & Aronis 96] for an overview of the reasons why it is desirable that a ML algorithm learns from very large datasets. Here we briefly remark that increasing the size of a dataset may be seen as a *domain-independent* way of achieving higher accuracy. This is in contrast with most "improvements" of machine learning algorithms proposed in the literature, which in essence are a kind of bias, and so have a *domain-dependent* effectiveness [Schaffer 94], [Michie et al. 94].

The use of SQL offers the benefit of portability across a wide range of database systems and parallel database computers, from symmetric shared-memory multiprocessors to massively-parallel computers. Note that the use of parallel processing as one approach for improving the scalability of ML algorithms is complementary to the discretization approach investigated in this paper. Naturally, both approaches may be used together to improve the scalability of ML algorithms.

3 Related Work on Discretization of Continuous Attributes.

Discretization algorithms accept as input the values of an ordinal - i.e. integer-valued or possibly continous (real-valued) - attribute (henceforth the input attribute) and output a relatively small list of ordered intervals in the form [X_{lower}-X_{upper}], where X_{lower} and X_{upper} are the lower and upper bounds associated to the input attribute's values within the interval.

Discretization algorithms can be classified in two broad groups: class-blind and class-driven algorithms. Class-blind algorithms discretize a given attribute without considering the value of the class (goal) attribute [Mannino et al. 88]. For instance, the equal-width method simply divides the range of values of an attribute into a pre-specified number of equal-width intervals, where width is computed as X_{upper} minus X_{lower}.

Class-driven algorithms consider the class value when discretizing the input attribute. The input for this kind of algorithm is a relation with two attributes: the input attribute (to be discretized) and a class attribute, whose value indicates the class the tuple belongs to. It is well established that class-driven discretization algorithms work better when the output intervals will be used to pre-process a dataset to be used by a classification algorithm, which is the case in our research. In this paper, we are interested in discretization algorithms which work in two steps. First, the values of the input attribute are sorted. Second, a discretization procedure is applied to the sorted values to produce a set of ordered intervals. This kind of algorithm can be further divided into two groups: top-down and bottom-up algorithms.

An example of a top-down algorithm is D-2 [Catlett 91a]. After sorting the values, D-2 first chooses a threshold T to partition the set of values S into two subsets S_1 (with values ≤ T) and S_2 (with values > T). T is chosen in such a way to maximize the information gain after the binary partition, as in ID3/C4.5 [Quinlan 93]. Then D-2 recursively applies this method to choose thresholds for subsets S_1 and S_2, and so on, until a stopping criteria is satisfied. Hence, D-2 avoids the computationally expensive sort operation in all subsets, unlike ID3/C4.5, where the sort is done in all tree nodes.

An example of a bottom-up algorithm is ChiMerge [Kerber 92]. It first sorts the input attribute's values, and assigns each value to a (single-value) interval. Then ChiMerge iteratively computes the χ^2 (Chi-squared) value - concerning the class frequency distribution - for every group of N adjacent intervals and merges the N adjacent intervals with the lowest χ^2 value, where N is a user-specified parameter. The rationale behind this is that the lower the χ^2 value, the higher the similarity in the class frequency distribution of the N intervals, and so the higher the motivation to merge those intervals. This iterative process continues until all groups of N adjacent intervals have a χ^2 value above a given threshold. As an additional merging-stopping criterion, the user can specify the minimum and maximum number of intervals that can be created by the algorithm.

ChiMerge has two drawbacks. First, the χ^2 significance test is not reliable when the number of values in the intervals being merged is very small. This is a serious problem for large datasets with a real-valued input attribute, where the number of distinct values of that attribute can be equal to the number of tuples in the input relation. In this case, the early iterations of the merging process will consider intervals with very few values, once each input attribute value will very likely be associated with just one class value. In one extremum, the first iteration will consider single-value intervals. [Kerber 92] partially alleviated this problem by modifying the χ^2 formula so that the denominator of every term of the χ^2 formula has a value of at least 0.5 (to avoid division by a very small number). Although this author reports that this (rather arbitrary) value works well, there is no statistically-sound justification for it. Actually, the statistics theory recommends to use another significance test rather than χ^2, in these situations.

Second, statistical measures of association, such as χ^2, were not designed for *classification*. Rather, they were designed for measuring the association (or dependency) between two attributes in a *symmetric* way, i.e. none of the two attributes being analyzed is given special treatment when computing the χ^2 value. On the other hand, classification is an *assymetric* task with respect to the two attributes being analyzed. We want to predict the value of the class attribute given the input attribute, not the other way round.

Another bottom-up algorithm is StatDisc [Richeldi & Rossotto 95], which produces a hierarchy of intervals. First, the input attribute values are sorted. The initial intervals are assigned to leaf nodes of the interval hierarchy. Then StatDisc iteratively computes the Φ (Phi) value, a statistical measure of association, for every group of N adjacent intervals (where N is a user-specified parameter) and merges the N adjacent intervals with the lowest Φ value. When intervals are merged, a new internal node (parent of the merged intervals) is added to the interval hierarchy. The merging process continues until all groups of N adjacent intervals have a Φ value greater than a given threshold. When the discretization process is over, the interval hierarchy can be explored to select a suitable set of intervals according to specific characteristics of the target application and of the dataset.

Before considering primitives for discretization, we should briefly justify the claim that in general a reduction in discovered-knowledge accuracy due to discretization is more than compensated for by the improvements in KDD efficiency. This claim is supported by some empirical evidence. For instance, [Catlett 91a] applied a discretization algorithm to several datasets, as a pre-processing step for ID3. He reported that ID3 applied to the discretized data achieved a speed up (over ID3 applied to the raw, non-discretized data) of more than 10 in datasets with many continuous attributes, with little or no loss of accuracy. In one dataset (with 92 real-valued attributes) the speed up achieved was more than 50, without significant decrease in classification accuracy. Actually, in some datasets the discretization process even slightly increased the classification accuracy. One possible explanation for this is that the discretization process may reduce the amount of noise in the continuous attributes.

4 Mapping a Generic Discretization Primitive into SQL.

A primitive should satisfy at least two requirements, viz generality (i.e. it should find use in several algorithms) and significant time saving (i.e. in a given algorithm, it should occur frequently and/or be computationally significant). The use of primitives is not very common in the area of AI, probably due to the great difficult of finding domain-independent primitives in AI (see e.g. [Stolfo et al. 95]). Fortunately, however, we have found a generic, domain-independent primitive for discretization algorithms. Developing generic primitives is important because no algorithm can be expected to perform well across all domains [Schaffer 94], [Michie et al. 94].

Our primitive was designed to perform the initialization step of class-driven discretization algorithms (see Section 3), where the algorithm needs to sort tuples according to the input attribute (to be discretized) and compute the class frequency distribution for each tuple (initial interval). Note that, although this procedure is an initialization step, it is a computationally expensive step, taking O(n log n), where n is the number of tuples. Actually, it is usually possible to implement an optimized version of the discretization process itself in time O(n log n) as well [Kerber 92], so that the time taken for performing the initialization step is very significant.

In essence, our primitive (which will be called Count by Ordered Group) counts the number of tuples in each partition of an *ordered group by* statement. The result of this primitive can be represented in the form of the two-dimensional matrix shown in Figure 1. This is an n x k matrix extended with totals of rows and columns, where n is the cardinality of the input attribute and k is the number of classes. Each cell C_{ij} (i=1,..,n, j=1,..,k) of this matrix contains the number of tuples in the input relation with input attribute's value equal to A_i and goal (class) attribute's value equal to G_j. The summation of the count values along the i-th row (i=1,..,n) is denoted C_{i+}, and the summation along the j-th column (j=1,..,k) is denoted C_{+j}. The total number of tuples is denoted by C_{++}. Count by Ordered Group is implemented in a declarative style by Query 1, followed by a trivial computation of rows and columns totals.

Note that in one extremum, if each tuple has a unique value of the input attribute (typically in the case of real-valued attributes rather than integer-valued ones), n is equal to the number of tuples in the input relation. In this case Query 1 degenerates into a relational projection on the input and class attributes, since the *group by* and the *count(*)* clauses become irrelevant.

	G_1 . . . . . G_k	Total
A_1	C_{11} . . . . C_{1k}	C_{1+}
.	. . . . . .	.
.	. . . . . .	.
.	. . . . . .	.
A_n	C_{n1} . . . . C_{nk}	C_{n+}
Total	C_{+1} . . . . C_{+k}	C_{++}

Fig. 1: Structure of the result of Count by Ordered Group.

```
SELECT input_attribute, class_attribute, COUNT(*)
FROM     relation_name
GROUP BY  input_attribute, class_attribute
ORDER BY  input_attribute
```

Query 1: SQL query underlying the Count by Ordered Group primitive.

5 A Class-Driven, Information-Theoretic Discretization Algorithm.

We propose a new discretization algorithm that we will call InfoMerge. First, the input attribute values are sorted and the class frequency distribution for each value of the input attribute is computed (via the SQL primitive shown in Query 1).

Second, InfoMerge iteratively computes a measure of the *damage* associated with the interval-merging process for every group of N adjacent intervals (where N is a user-specified parameter) and merges the N adjacent intervals with the *lowest* value of that measure, following the bottom-up paradigm of ChiMerge and StatDisc. The difference is that both ChiMerge and StatDisc use a statistical measure of class-frequency-distribution similarity, respectively χ^2 and Φ values. In contrast, we use an information-theoretic approach as follows, in order to address the problems with statistical measures of association discussed in section 3.

In each iteration of the merging process, InfoMerge first computes the entropy of each interval by using the class frequency distribution within that interval. Then InfoMerge computes, for each group of N adjacent intervals, the information *loss* associated with the merging of the N intervals. The information loss is computed as:

$$\text{Info\_Loss} = \text{Info\_After\_Merge} - \text{Info\_Before\_Merge} \qquad (1)$$

where Info_After_Merge is the amount of information necessary to identify the class of a tuple after merging, measured by the information-theoretic Equation (2); and Info_Before_Merge is the corresponding amount of information before merging, measured by Equation (3). In these equations, k denotes the number of classes, and N denotes the number of adjacent intervals considered for merging. C_{ij}, C_{i+}, C_{+j} and C_{++} have the meaning defined in Section 4, with the difference that here the row index i varies in the range 1..N rather than 1..n (where n is the cardinality of the input attribute). Thus, in Equations (2) and (3) C_{++} is the total number of tuples in all the N intervals considered for merging.

$$\text{Info\_After\_Merge} = \sum_{j=1}^{k} - (C_{+j} / C_{++}) * \log(C_{+j} / C_{++}) \qquad (2)$$

$$\text{Info\_Before\_Merge} = \sum_{i=1}^{N} (C_{i+} / C_{++}) * \sum_{j=1}^{k} - (C_{ij} / C_{i+}) * \log(C_{ij} / C_{i+}) \qquad (3)$$

Note that the minimization of information *loss* when merging intervals can be thought of as the dual operation of maximizing the information *gain* when splitting a node of a decision tree [Quinlan 93]. Actually, decision-tree learners are top-down algorithms, where the splitting process aims at improving the separation of the classes as much as possible in the new “intervals” (branches of the tree). On the other hand, InfoMerge is a bottom up algorithm, where the merging process aims at getting larger intervals, loosing as little class separation as possible in this process.

However, when merging intervals we should consider not only the information loss, but also the size of the intervals to be merged. For instance, assume that the merging of intervals I1 and I2 results in an information loss smaller than the merging of intervals I3 and I4. Now assume that intervals I1 and I2 are much larger (have a higher frequency of tuples) than I3 and I4. Intuitively, it is better to merge I3 and I4, rather than I1 and I2, since this will tend to produce “less harm” (smaller information loss) in the dataset as a whole. Hence, we weight the information loss in Equation (1) with the interval’s size (relative frequency of tuples within the interval, in relation to the whole dataset). The final formula for computing the Weighted_Info_Loss in the InfoMerge algorithm is shown in Equation (4), where Info_Loss is computed according to Equation (1) and n is the total number of tuples in the whole dataset. In each iteration, InfoMerge merges the group of N adjacent intervals with the minimum value of Weighted_Info_Loss.

$$\text{Weighted\_Info\_Loss} = (C_{++} / n) * \text{Info\_Loss} \qquad (4)$$

Note that this weighting criterion for the use of information loss in a interval-merging process has no counterpart in the use of information gain in a data subset-splitting process. In this latter case, the classification algorithm (e.g. a decision-tree learner) typically computes the information gain for each candidate attribute, but in each computation the number of tuples in the data subset to be split is the same. Hence, information gain is used in a local context (the current data subset to be split). There is no need to make global decisions involving comparison of information gain across different data subsets. In contrast, the discretization algorithm does need to compare the information loss for different groups of intervals, where the size of the intervals (i.e. the number of tuples contained in each interval) can be very different from each other.

6 Results.

We evaluated the performance of InfoMerge and ChiMerge in large datasets. Each dataset was randomly partitioned into a training set, used for learning, and a test set, used to measure the classification accuracy of the discovered knowledge. We report results for a single partition of the dataset, following the methodology used in the Esprit project Statlog to measure classification accuracy in large datasets [Michie et al. 94].

In all the experiments, N (the number of adjacent intervals considered for merging) was set to 2 (following [Kerber 92]), both for InfoMerge and ChiMerge. In both algorithms, the stopping criterion of the merging process depends on user-specified maximum and minimum number of intervals to be produced and on a given threshold. In ChiMerge, this threshold was determined by looking up a χ^2 table to find the χ^2 value for a significance level of 95% (see [Kerber 92] for details). In InfoMerge this threshold was empirically determined as a Weighted_Info_Loss value of 0.001. Hence, InfoMerge halts if Weight_Info_Loss is greater than 0.001 and the current number of intervals is smaller than or equal to the maximum specified; or if the current number of intervals is equal to the minimum specified (remember that the number of intervals is reduced at each iteration, by the merging process). In all the experiments the minimum and maximum number of intervals to be produced was set to 5 and 12, for both InfoMerge and ChiMerge.

The Labour Force Survey (LFS) data is produced by the UK's Department of Employment. The LFS dataset used in our experiments has 11 predicting attributes: Sex, Age, Ethnic code, Full-time/Part-time job, Permanent/Temporary job, Training status, Time as Employee in the company (denoted by EmpTime), Educational level, Public/Private Job, Industry-Area code, and Number of Employees at workplace. Attributes Age and EmpTime are ordinal (integer-valued). The other attributes are categorical. The class attribute is the Managerial status of an employee, with two possible values (indicating whether or not the employee has some managerial status, e.g. manager or supervisor). This dataset has 84532 tuples. We used 74,532 randomly-selected tuples as the training set and the remaining 10,000 tuples as the test set.

The Shuttle data has 58,000 tuples and 9 ordinal (integer-valued) attributes. There are 7 classes. The attributes concern the position of radiators in a NASA space shuttle, and the classes are the appropriate actions to be taken during a space shuttle flight [Catlett 91]. We used the same training/test set partition used in the Esprit project Statlog, viz. 43,500 tuples in the training set and 14,500 tuples in the test set. Although the LFS and the Shuttle dataset are not huge, they are large enough for testing purposes.

In order to evaluate the effectiveness of InfoMerge, we compared the classification accuracy achieved and the time taken by C4.5 [Quinlan 93] on the raw data and on the discretized data. Note that there is no straightforward, “optimal” way of evaluating the performance of a discretization algorithm. This performance depends on the classification algorithm which will be applied to the discretized data. We chose C4.5 because it is a state-of-the-art decision-tree learner algorithm, and decision-tree learners are in general the most used kind of ML algorithm. We used all the default parameters of C4.5.

The results are shown in Table 1. The first column indicates the dataset used in the experiments. The second and third columns show the size of the produced decision tree (number of tree nodes) and its associated error rate (in %), when C4.5 learns from the raw (non-discretized) data. The fourth and fifth columns show the correspondingly tree size and error rate when C4.5 learns from the data discretized by the ChiMerge algorithm. The sixth column shows the speed up associated with the discretization performed by ChiMerge, i.e. the ratio of the time taken by C4.5 to learn from the raw data divided by the time to learn from the discretized data. The seventh, eigth and ninth columns show the corresponding tree size, error rate and speed up associated with the discretization performed by our new InfoMerge.

For the LFS dataset the discretization performed by both ChiMerge and InfoMerge increased the error rate a little (the increase was 1.8% for ChiMerge and 1.0% for InfoMerge). However, there was a great gain in computational efficiency, resulting in speed ups by a factor of about 30. There was also a gain in simplicity and comprehensibility of the discovered knowledge, since the trees produced from discretized data are much smaller than the tree produced from the raw data. The discretizaton performed by InfoMerge resulted in smaller tree size, smaller error rate and greater speed up than the discretization performed by ChiMerge.

For the Shuttle dataset the discretization process did not result in any significant increase in error rate. On the other hand, discretization resulted in increased tree size. However, discretization still results in a speed up by a factor of about 10. The discretization performed by InfoMerge resulted in smaller tree size and greater speed up than the discretization performed by ChiMerge.

Table 1: Results running C4.5 on raw data and discretized data.

Dataset	Raw Data		ChiMerge Discretization			InfoMerge Discretization		
	Size	Error	Size	Error	Sp	Size	Error	Sp
LFS	7501	27.3	1643	29.1	27.7	1096	28.3	31.0
Shuttle	45	0.0	253	0.0	8.9	164	0.0	9.9

7 Conclusion.

This paper investigated a way of improving the scalability of KDD algorithms, viz. the discretization of ordinal or continuous attributes. This paper has two novel aspects. First, we mapped a generic primitive of discretization algorithms into an SQL query, which is another step towards the goal of integrating the areas of Machine Learning (ML) and RDBMS. See [Freitas & Lavington 96] for a discussion of how other KDD stages can be mapped into SQL and how the use of parallel SQL servers speeds up a KDD algorithm.

Second, we proposed a bottom-up, class-driven, information-theoretic discretization algorithm, called InfoMerge. This algorithm was compared to the ChiMerge algorithm, based on the χ^2 significance test, and the former was shown to achieve better results (i.e. smaller decision tree sizes, smaller error rates and greater speed up associated with the discretized data) than the latter.

In our future work, we intend to apply InfoMerge to other large datasets and develop other SQL-based KDD primitives to integrate ML and RDBMS.

Acknowledgments.

The first author is supported by Brazilian government's CNPq, grant number 200384/93-7. We thank D.R. Thoen for his help in the experiments. We also thank N.E.J. Dewhurst and R.A. Gamble for their support in early phases of this research. Finally, we thank the anonymous referees for their valuable comments.

References.

[Catlett 91] J. Catlett. Megainduction: a test flight. Proc. 8th Int. Workshop on Machine Learning, 596-599. 1991.

[Catlett 91a] J. Catlett. On changing continuous attributes into ordered discrete attributes. Proc. European Working Session on Learning (EWSL-91). LNAI-482, 164-178.

[Fayyad et al. 96] U.M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth R. Uthurusamy. (Eds.) Advances in Knowledge Discovery and Data Mining. AAAI/MIT, 1996.

[Freitas & Lavington 96] A.A. Freitas and S.H. Lavington. Using SQL primitives and parallel DB servers to speed up knowledge discovery in large relational databases. Accepted for Symposium on Knowledge Discovery in Databases, 13th European Meeting on Cybernetics and Systems Research (EMCSR'96). Vienna, Apr./96.

[Kerber 92] R. Kerber. ChiMerge: Discretization of numeric attributes. Proc. 1992 Conf. American Assoc. for AI (AAAI-92), 123-128.

[Mannino et al. 88] M.V. Mannino, P. Chu and T. Sager. Statistical profile estimation in database systems. ACM Computing Surveys, 20(3), Sep./88, 191-221.

[Michie et al. 94] D. Michie, D.J. Spiegelhalter and C.C. Taylor. Machine Learning, Neural and Statistical Classification. New York: Ellis Horwood, 1994.

[Paliouras & Bree 95] G. Paliouras and D.S. Bree. The effect of numeric features on the scalability of inductive learning programs. Proc. 8th European Conf. Machine Learning (ECML-95). LNAI-912, 218-231. 1995.

[Provost & Aronis 96] F.J. Provost and J.M. Aronis. Scaling up inductive learning with massive parallelism. To appear in Machine Learning, 1996.

[Piatetsky-Shapiro & Frawley 91] G. Piatetsky-Shapiro and W.J. Frawley. (Eds.) Knowledge Discovery in Databases. Menlo Park,CA: AAAI, 91.

[Quinlan 93] J.R. Quinlan. C4.5: Programs for Machine Learning. San Mateo, CA: Morgan Kaufmann, 1993.

[Richeldi & Rossotto 95] M. Richeldi and M. Rossotto. Class-driven statistical discretization of continuous attributes. (Extended Abstract) Proc. 8th European Conf. Machine Learning (ECML-95). LNAI-912, 335-338.

[Schaffer 94] C. Schaffer. A conservation law for generalization performance. Proc. 11th Int. Conf. Machine Learning, 259-265. 1994

[Smyth & Goodman 91] P. Smyth and R.M. Goodman. Rule induction using information theory. In G. Piatetsky-Shapiro and W.J. Frawley. (Eds.) Knowledge Discovery in Databases. Menlo Park, CA: AAAI Press, 1991.

[Stolfo et al. 95] S.J. Stolfo, H.M. Dewan, D.Ohsie and M.Hernandez. A parallel and distributed environment for database rule processing: open problems and future directions. In: M. Abdelguerfi & S. Lavington. (Ed.) Emerging Trends in Database and Knowledge-Based Machine, 225-253. IEEE, 1995.