UCLA

Title
A scalable bottom-up data mining algorithm for relational databases

Permalink
https://escholarship.org/uc/item/2g40z8bf

Authors
Giuffrida, G
Cooper, LG
Chu, WW

Publication Date
1998

DOI
10.1109/ssdm.1998.688125

Copyright Information

Peer reviewed

A Scalable Bottom-Up Data Mining Algorithm
for Relational Databases

Giovanni Giuffrida
Computer Science Dept.
UCLA
giovanni@cs.ucla.edu

Lee G. Cooper
Anderson School
UCLA
lee.cooper@anderson.ucla.edu

Wesley W. Chu
Computer Science Dept.
UCLA
wwc@cs.ucla.edu

Abstract

Machine learning induction algorithms are difficult to scale to very large databases because of their memory-bound nature. Using virtual memory results to a significant performance degradation. To overcome such shortcomings, we developed a classification rule induction algorithm for relational databases. Our algorithm uses a bottom-up rule generation strategy that is more effective for mining databases having large cardinality of nominal variables. We have successfully used our algorithm to mine a retail grocery database containing more than 1.6 million records in about 5 hours on a dual Pentium processor PC.

1. Introduction

Machine learning practice has been based mostly on *memory-bound* techniques. The combinatorial nature of the knowledge induction process may rapidly use all the available (physical) main memory when mining very large datasets. As a result, the process relies on the *virtual memory* mechanism available in the hosting operating systems and significantly degradates the performances.

In this paper we present KDS (*K*nowledge *D*iscovery using *S*QL), a SQL-based algorithm to discover classification rules. KDS has been designed to work on top of relation DBMS. The entire learning process in KDS is a series of complex SQL queries executed on the relational database. Such queries use optimization techniques (e.g. indexing, user defined functions, etc.) extensively. We have successfully applied KDS to a real world database containing 1.6 millions records, a size that is usually prohibitive for memory-bound induction algorithms.

2. The KDS algorithm

KDS generates symbolic "if-then" classification rules. The input examples for KDS are sets of *features*. A feature is a pair $(argument, value)$. Rules are in the form "if `<condition>` then `<class-distribution>`" in the style of CN2 [4]. `condition` is a conjunction of *selectors*. A selector is an equality test of the form $a = A$, where a is an independent variable and A is one of its legal values. `class-distribution` is a counter distribution over the target variable.

KDS was designed to be implemented on top of relational databases and is based on simple concepts already exploited for other types of learning. As opposed to the majority of *memory-bound* machine learning algorithms, KDS is implemented on relational databases. Thus it is *disk-bound*. We have implemented KDS in a *tightly-coupled* mode [2] with DB2 (a commercial relational database from IBM). Optimization techniques available in DB2 have been largely exploited (e.g. indexing and User Defined Functions). Its integration with DBMS makes KDS more advantageous when very large number of records (e.g.: few millions) are involved in the mining process and/or insufficient physical memory is available to guarantee traditional learning systems to effectively process such a large amount of data.

Most mining algorithms are based on an either *divide-and-conquer* [14] or *separate-and-conquer* [9] approach. In both approaches, the input database is progressively reduced in size at each iteration. This makes rules discovered at the beginning of the process have a stronger statistical support than the ones discovered later. In turn, inducing rules from small datasets exacerbates the *small disjuncts problem* [10]. KDS uses the *conquer-without-separating* approach proposed by Domingos [7] which overcomes such a problem. Thus, *all* rules in KDS are always mined from the entire dataset.

Most induction algorithms use a *top-down* rule generation approach. In such an approach, a "for each pos-

sible selector" loop usually takes place at the time candidate rules are generated and statistically tested on the mined dataset. This can be very costly for attributes with large cardinality (large value sets) when thousands of possible values need to be considered for each attribute. Besides the complexity of testing thousands of selectors, the method also needs to test every possible combination of selectors. As a result, semantically meaningless combinations (e.g. *"STATUS=pregnant & SEX=male"*, *"RELIGION=catholic & MARITAL-STATUS=married & OCCUPATION=priest"*) with no coverage on the database are evaluated and discarded. Therefore, top-down rule induction may be very costly due to the statistical test of a very large number of meaningless combinations. In the real case of a retail grocery database, thousands of manufacturers would need to be crossed (joined) with hundreds of categories, which translates to wasting time for testing non existing patterns like: *"MANUFACTURER=Coca-Cola & CATEGORY=Baby-Supplies"*. KDS avoids this additional cost by a *bottom-up* rule generation strategy.

KDS builds rules incrementally starting from the most general rules to more specialized ones. The process starts from the most general patterns (having only one term in the conjunction: *1-term* patterns) and then progressively specializes to *2-term*, *3-term*, and so on. The specialization is always driven by the input database. By doing so, only combinations of features actually existing in the database are considered as rule specialization selectors.

The KDS algorithm is shown in Figure 1. $R[N]$ represents the set of N-term rules. The set S contains all the N-combinations of the independent variables assigned to the values of the current record. For instance, consider the input record is: $\{a = 10, b = low, c = john\}$, then the set S at the second iteration (N=2) is: $\{\{a = 10, b = low\}, \{a = 10, c = john\}, \{b = low, c = john\}\}$. Likewise, the set T is constructed from the elements of S. For instance, for the element $\{a = 10, b = low\}$ of S, T would be: $\{\{a = 10\}, \{b = low\}\}$, a set of (N-1)-term patterns. The notation $R[N].supp(X)$ specifies the popularity of the pattern X in the rule set $R[N]$. $X.class$ is the class value of the input example X, while $R[N].class(Y, C)$ is the frequency of the class C for the rule Y in the rule set $R[N]$.

2.1. Rule generation and organization

In most induction algorithms, rule generation and rule ranking phases are so tightly integrated that it is difficult to make a distinction between them. A rule scoring mechanism is used to generate the best rule at each iteration. In contrast, in KDS, there is a distinct separation between the rule generation phase and the rule (selection and) ranking phase. The KDS algorithm does not perform any rule ranking at the rule generation phase. It creates all the rules and

```
I = input database;
N = 1;
Flag = True;
While Flag
    Flag = False;
    R[ N] = {};
    For each record W in I do
        S = {N-term patterns from W};
        For each X in S do
            T = {(N-1)-term patterns from X};
            If (N=1) or (all elements in T are supported) Then
                Flag = True;
                If X ∈ R[N] then
                    R[N].supp(X) = R[N].supp(X) + 1;
                Else
                    R[N] = R[N] ∪ {X};
                    R[N].supp(X) = 1;
                End If
                R[N].class(X, X.class) ++;
            End If
        End For
    End For

    ; Pruning by minimum support
    For each Y in R[N] do
        If R[N].supp( Y) ≤ min-supp Then
            R[N] = R[N] - Y;
        End If
    End For
    N = N + 1;
End While
```

Figure 1. The KDS Algorithm

arranges them in a convenient structure. The rule selection and ranking task is postponed until classification time. This approach is also justified by the KDS goal of supporting incremental knowledge discovery. Rules have to be stored even if poorly scored; successive mining of new incoming input can update rule scores and readjust the global rank[2]. Additional study needs to be done to make KDS learn incrementally.

KDS typically generates a large set of discovered rules. These rules are stored in the DBMS in a proper structure called *pattern network*. This structure optimizes rule retrieval and speeds up the classification task. An example of a fragment of a pattern network is shown in Figure 2. The lowest levels of the pattern network contain the 1-term patterns. One level up are the 2-term patterns, and so on. Each link between lower and upper levels represents a *pattern-specialization* operator. Rules are more general in the lower

---

[2]KDS performs a rule pruning based on the minimum support concept. So, not all rules are later updateable. Value of the minimum support is a trade-off between speed of execution, storage space and the level of incrementality supported. Further study needs to be done to support full incrementality.
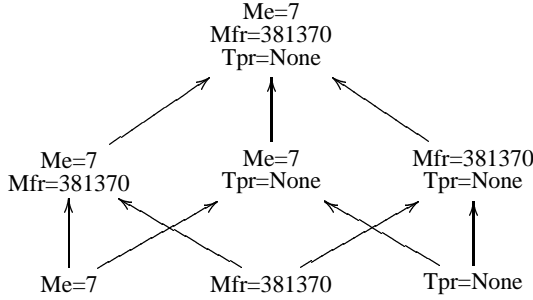
```
                Me=7
              Mfr=381370
               Tpr=None


   Me=7           Me=7          Mfr=381370
 Mfr=381370     Tpr=None         Tpr=None


   Me=7        Mfr=381370        Tpr=None
```

Figure 2. A fragment of a Pattern-Network

levels and become more specific in the upper levels. The specialization operator is a partial order on the set of discovered rules. In Figure 2 the 2-term rule "if Me=7 & Mfr=381370 then ..." is a specialization of its two children "if Me=7 then ..." and "if Mfr=381370 then ...". This architecture eases the process of selecting all rules containing a specific pattern. They are simply identified by all the ancestors of the node containing the pattern of interest. Each node of the pattern network contains the specification of the pattern itself and the class distribution vector[3].

2.2. Classification of new observations

Once the pattern network has been created, classification of new previously unseen observations can take place. Classification in KDS is performed through the following steps:

1. **Rule selection:** find all the rules covering the observation to be classified.

2. **Rule ranking:** select the best rule(s) according to the ranking criterion.

3. **Classification:** assign the class of the chosen rule(s) to the input observation.

The selection algorithm starts from the bottom of the pattern network by activating the 1-term rules corresponding to the selectors of the input example. The activation is then propagated upward, and each intermediate node is activated only if *all* its children are active as well. The activation travels to the highest nodes of the network. At this point all rules covering the input examples are selected. All selected rules are then ranked and the best one is chosen. The best class of the chosen rule is the predicted class for the input example.

The pattern network structure provides a flexible structure for developing different *ad-hoc* rule ranking criteria. Domain knowledge can be easily modeled in the ranking method.

---

[3]In the implementation of KDS other data (entropy, rule coverage, etc.) are associated to the class distribution vector.

2.3. Cost Analysis

KDS works based on a progressive *breadth-first* rule specialization. The nth iteration creates all the n-term rules existing in the input database. The rule specialization function is a monotonic operator, decreasing upon each application from more general to more specific patterns. The process is halted as soon as further specialization leads to coverage below the specified minimum support for all new generated rules. As already mentioned, KDS makes a clear distinction between the rule generation and rule selection/ranking phase. The rule generation performs a total of k iterations, where k is the maximum number of terms in the patterns before the coverage drops below the minimum support value (for all the new rules.) Actually, a maximum value of k is set. By doing so, we allow only a maximum number of terms in the rule antecedents: conjunctions with large number of terms tend to be more difficult to be interpreted by users. Therefore, the while loop in the algorithm has cost $O(k \cdot e)$ where e is the number of input examples. The nth iteration is based upon the results of the *(n-1)*th iteration. For instance, it is necessary (but not sufficient) for adding the new pattern "a&b&c" at the 3rd iteration that "a&b", "a&c", "b&c" are all supported. The cardinality s of the set S in the algorithm shown above at the nth iteration is $a!/[n!(a-n)!]$, where a is the total number of independent variables. The set S contains the candidates for new patterns to be added to the rule set. For each element of S the set of sub-patterns is generated and stored in the set T. For each element of T a lookup (with logarithmic cost) is executed until one element is not supported or all the elements have been verified to be supported. In the worst case $|T|$ lookups have to be performed for each element of S. The total cost becomes: $O(k \cdot e \cdot |S| \cdot |T| \cdot log(l))$ where l is the size of the $R[n-1]$ set at the nth iteration. Furthermore, for each iteration a pruning loop is executed to remove all new rules that are not supported. This has a minor cost that can be omitted in the computation.

2.4. KDS Application Domain

The absence of a "for each selector" loop and its linear time cost with respect to the number of input examples makes KDS well suited for mining datasets with large value sets and large number of tuples. However, KDS does not scale well to datasets with a large number of independent variables. Thus, KDS applies well to datasets with (1) a large number of records, (2) large value sets and (3) a small number of independent variables. Conversely, top-down separate-and-conquer algorithms have a better fit for datasets with (1) a small number of records, (2) small value sets and (3) a large number of independent variables.

The SQL based nature of KDS is beneficial when the size

of a problem is too big to fit in physical memory. Smaller datasets can be better processed by other memory-bound induction algorithms [13].

The execution of KDS on a real world large database (1.6 millions records, 6 independent variables for a total of 4,334 different values) required a total of about 5 hours on a dual Pentium Pro system with 128Mb of physical memory and over 30Gb of disk storage. For the sake of performance comparison, the same database was also used as input for Ripper [5]. The latter ran on the same dataset for 21 days (no other process running at the same time) without completing the task (we had to kill the process). Once Ripper exhausted the physical memory it resorted to using virtual memory (set up to 1 Gb), resulting in a tremendous performance decrease.

3. Related Work

Recently, integration of data mining algorithms with relational databases has been receiving attention. Provost and Kolluri [13] mention the problem of mining relational databases (instead of a single flat file) and the integration of KDD with DBMS as a direction in scaling up to very large datasets (when not enough main memory is available.) John and Lent [11] propose a middle layer between data mining algorithms and SQL systems. They outline an implementation of C4.5 [14] and a Bayesian classifier by using their SIP methodology. Agrawal and Shim [2] describe a methodology for developing data mining applications tightly coupled with relational systems. In their paper they describe the implementation of the *Apriori* algorithm [1] for mining association rules. Apriori is based on a bottom-up rule generation approach similar to KDS.

SLIQ [12], a classifier for disk-resident datasets, builds classification trees. SLIQ is based on the "divide-and-conquer" strategy followed by the tree induction algorithms. As reported in the paper, SLIQ scales almost linearly with the number of training examples and the number of attributes. However, no scalability report was discussed for increasing cardinality of nominal variables (the problem was however recognized by the authors as a difficult one for large value sets).

Numerous "separate-and-conquer" strategy based algorithms have been proposed in the past. Furnkranz [9] lists and classifies 40 of them. The "divide-and-conquer" approach is basically used by all the tree induction algorithms rooted in the work of Quinlan [14]. Domingos proposes a "conquer-without-separation" approach for his CWS and RISE systems [7, 8]. His "without-separation" approach is oriented to solve the problem of progressive fragmentation of the input dataset. Domingos [6] shows how such a technique achieves substantial improvements in accuracy when mining databases with large number of disjuncts (each one

covering few training records). Aronis and Provost [3] tackle the inefficiency of induction algorithms when working with large value sets in the input database. They propose a general pre-processing technique to speed up the subsequent mining task.

4. Conclusions

We presented KDS, a (classification) rule induction algorithm for relational databases. KDS uses a *bottom-up* strategy during rule specialization. This saves computation time by testing only combinations of features that exist in the mined dataset. This strategy is effective for mining large databases containing attributes with large cardinality. KDS scales linearly with the number of training records and the cardinality of nominal variables. However, it does not scale well with the number of attributes. We have successfully applied KDS to discover classification rules from a real world grocery retailer database containing about 1.6 millions records. The processing time was about 5 hours on a Dual Pentium Pro PC system.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and V. A. I. Fast discovery of association rules. *In Advances in Knowledge Discovery and Data Mining*, 1996.

[2] R. Agrawal and K. Shim. Developing tightly-coupled data mining applications on a relational database system. *KDD-96*, 1996.

[3] J. M. Aronis and F. J. Provost. Increasing the efficiency of data mining algorithms with breadth-first marker propagation. *KDD-97*, 1997.

[4] P. Clark and T. Niblett. The cn2 induction algorithm. *Machine Learning*, 1(3), 1989.

[5] W. W. Cohen. Fast effective rule induction. *Procs of 12th Int.l Conf. on Machine Learning*, 1995.

[6] P. Domingos. The rise system: Conquering without separating. *Procs. of the 6th IEEE Intern. Conf. on Tools with Artif. Intelligence*, 1994.

[7] P. Domingos. Linear-time rule induction. *KDD-96*, 1996.

[8] P. Domingos. Unifying instance-based and rule-based induction. *Machine Learning*, 24:141–168, 1996.

[9] J. Furkranz. Separate-and-conquer rule learning. *Technical Report OEFAI-TR-96-25*, 1996.

[10] R. Holte, L. E. Acker, and B. Porter. Concept learning and the problem of small disjuncts. *Procs of 11th Intern. Joint Conference on Artificial Intelligence*, 1989.

[11] G. H. John and B. Lent. Sipping from the data firehose. *KDD-97*, 1997.

[12] M. Metha, R. Agrawal, and J. Rissanen. Sliq: A fast scalable classifier for data mining. *Proc. of the Fifth Int'l Conf. on Extending Database Technology*, 1996.

[13] F. Provost and V. Kolluri. Scaling up inductive algorithms: An overview. *KDD-97*, 1997.

[14] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.