

# Efficient Multi-Relational Data Mining

Jan Struyf and Hendrik Blockeel

Katholieke Universiteit Leuven, Dept. of Computer Science,  
Celestijnenlaan 200A, B-3001 Leuven, Belgium  
{*Jan.Struyf, Hendrik.Blockeel*}@cs.kuleuven.ac.be

## Abstract

Multi-relational data mining algorithms search a large hypothesis space in order to find a suitable model for a given data set. During this search, a huge number of complex queries has to be evaluated on the data set. This explains why multi-relational data mining algorithms (e.g. ILP algorithms) typically have high run times. In this text we give an overview of two techniques designed to reduce these run times. We show that this is possible by exploiting similarities in both queries and data sets. The first technique is query-pack evaluation and the second one is parallel cross-validation. This paper is a summary of (Blockeel and Struyf, 2001), (Struyf and Blockeel, 2001) and (Blockeel et al., 2000).

## 1 Introduction

Multi-relational data mining algorithms search a large hypothesis space in order to find a suitable model for a given data set. During this search, a huge number of queries has to be evaluated on the data set. Evaluating a query that uses and combines information from different relations is far more complex compared to evaluating a test on a single table, i.e. what propositional data mining algorithms do. The main reason for this is that in the multi-relational context, when executing queries top-down (as is the case for ILP algorithms), backtracking can occur over different relations. Because query evaluation is more complex, multi-relational data mining algorithms typically have high run times and it becomes interesting to do research for techniques that speed up model building. In this text we give an overview of two such techniques. The first one is query-pack evaluation (Blockeel et al., 2000) and the second one is parallel cross-validation (Blockeel and Struyf,

2001), (Struyf and Blockeel, 2001). The basic idea underlying both techniques is that it is possible to eliminate redundant computations by exploiting similarities in the queries or in the data sets.

Query-pack evaluation can be used when a set of similar queries has to be evaluated on a data set. These similar queries are typically generated by the refinement operator of the data mining algorithm. A first order decision tree induction system (e.g. TILDE (Blockeel and De Raedt, 1998), S-Cart (Kramer, 1996)) considers refinements of the query from the parent node when selecting a query for a new node. Because each refinement is obtained by extending the parent query with a few new literals, different refinements are highly similar (i.e. share literals). Running these similar queries separately on the data set results in redundant computations because the common parts are evaluated more often than necessary. By integrating the different similar queries in a query-pack (Blockeel et al., 2000), it is possible to eliminate these redundant computations.

Cross-validation is a technique used in many different machine learning approaches, such as instance based learning, artificial neural networks or decision tree induction, to tune parameters (e.g. using a wrapper method (Kohavi and John, 1995)), select relevant features or to estimate predictive accuracies. Running an  $n$ -fold cross-validation consists of partitioning the data set  $D$  into  $n$  subsets  $D_i$  and then running the machine learning algorithm  $n$  times, each time using a different training set  $T_i = D - D_i$  and validating the results on  $D_i$ . The results on each  $D_i$  are averaged to provide a reliable estimate of the induced model's performance on unseen cases. The obvious disadvantage of cross-validation is the computational over-

head of running the machine learning algorithm  $n$  times. However, for some machine learning techniques (e.g. decision tree induction, rule induction), this overhead can be reduced significantly, again by removing redundancies. With query-pack execution, redundancies occur when evaluating similar queries separately on a data set. Here the redundancies stem from running queries on similar data sets. Note that the training sets  $T_i = D - D_i$  used in the different cross-validation folds are highly similar because each example from  $D$  is replicated  $n - 1$  times in the different  $T_i$ . By building the  $n$  cross-validation models in parallel it is possible to remove these redundancies.

This paper is organised as follows. Section 2 summarises logical decision tree induction. Section 3 discusses query-packs, Section 4 discusses parallel cross-validation. Section 5 presents experimental results and Section 6 states the conclusions.

## 2 Logical Decision Tree Induction

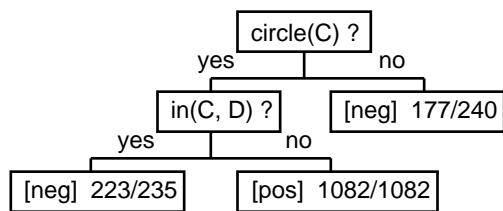


Figure 1: A first order decision tree.

A first order decision tree (Blockeel and De Raedt, 1998) is a binary decision tree with conjunctions of first order literals in the nodes. Different nodes of the tree can be linked by sharing variables. The leaves contain class values in case of a classification task or (vectors of) real values in case of a regression task. An example tree grown on one of the Bongard data sets (De Raedt and Van Laer, 1995a) is shown in Figure 1. The prediction task for this set is classifying pictures containing circles, squares and triangles as positive or negative. We use the learning from interpretations setting (De Raedt and Džeroski, 1994) in which each example is given by a set of (Prolog) facts.

First order decision trees are grown top down. The induction algorithm continues to add new nodes to the tree until a stop criterion is met.

Examples sorted into a certain node are characterized by the conjunction of all succeeding literals in the path from the root to the node; call this the “current query” for that node. The best query for a new node is selected by a greedy algorithm. It first generates refinements of the current query by extending it with new literals. In the Bongard example possible refinements of the query ‘circle(C)’ are ‘circle(C), triangle(D)’, ‘circle(C), square(D)’, ‘circle(C), in(C,D)’, ... The algorithm computes a quality measure such as information gain (Quinlan, 1993) for each refinement. The refinement that maximises this quality is used to create the new node.

## 3 Query-packs

```

circle(C), in(C,D), triangle(D)
circle(C), in(C,D), square(D)
circle(C), in(C,D), circle(D)
circle(C), in(C,D), triangle(E)
...
  
```

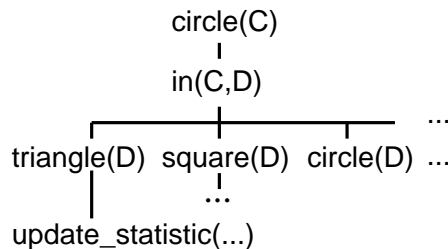


Figure 2: A query-pack.

Consider again the Bongard example from Figure 1. The refinements for the left-left subtree (query ‘circle(C), in(C,D)’) are shown in Figure 2. All these refinements have the first two literals in common. Executing these queries separately on the training set will cause redundant computations. This is because the first two literals will be (re-)evaluated for each query. By integrating the queries in a query-pack (Blockeel et al., 2000) as shown in the lower part of Figure 2, this redundancy can be removed.

A query-pack is a tree structure with literals or conjunctions of literals in the nodes. Each path from the root to some node represents a conjunctive query. Decision tree query-packs can be compared to brooms. The current query, its length being proportional to the current tree depth, forms the stick of the broom. It is shown

in (Blokkeel et al., 2000) that the speed-up factor  $T_{\text{sequential}}/T_{\text{pack}}$  ranges from 1 to  $\min(c+1, b)$  where  $b$  is the branching factor of the pack and  $c$  is the ratio of the computational complexity in the shared part over the complexity in the non-shared part. Because a broom has a long shared part and a high branching factor, one can expect high speed-ups.

To select the best query from a query-pack, the algorithm needs to keep track of some information (i.e. a statistic) for each query. In the case of a classification task, the natural choice is a class distribution. This is because quality measures as information gain or gain ratio (Quinlan, 1993) can be easily calculated from a positive (i.e. the query succeeds) and a negative (i.e. the query fails) class distribution. In case of a regression task we can use  $\sum(y_i^2, y_i, 1)$  with  $y_i$  the target value for  $e_i$ . This statistic makes it possible to calculate a variance based quality measure (Breiman et al., 1984).

Each leaf of the query-pack contains a call to `update_statistic`. If a query  $q$  from the pack succeeds for a given example  $e$ , then the positive statistic for  $q$  is updated. The negative statistic  $NS$  can be calculated based on the positive  $PS$  and the total  $TS$  statistic (i.e. the distribution of the examples before the node is split) by applying additivity:  $NS = TS - PS$  (e.g. for a class distribution one subtracts the class counts component-wise).

Note that query packs are very general and can be used in all settings where a number of similar queries have to be evaluated on a data set. Applications are first order decision tree induction (e.g. TILDE (Blokkeel and De Raedt, 1998), S-Card (Kramer, 1996)), first order rule induction (e.g. FOIL (Quinlan, 1990), Progol (Muggleton, 1995)) and finding first order association rules (e.g. WARMR (Dehaspe and Toivonen, 1999)).

## 4 Parallel Cross-validation

This section discusses parallel cross-validation. We introduce parallel cross-validation in the context of first order decision tree induction. We show how parallel cross-validation can be combined with query-packs and how it can be implemented for rule induction.

As explained briefly in the introduction, decision tree cross-validation involves growing  $n$

different trees, each on a slightly different training set  $T_i = D - D_i$ . Because the training sets  $T_i$  are highly similar, one can expect that the trees will be highly similar too, especially near the root. Parallel cross-validation (Blokkeel and Struyf, 2001) exploits this similarity while growing  $n+1$  different trees at once: one tree for each cross-validation fold and one tree grown on the whole data set  $D = T_0$ . We call this set of  $n+1$  trees the cross-validation forest (See Figure 3).

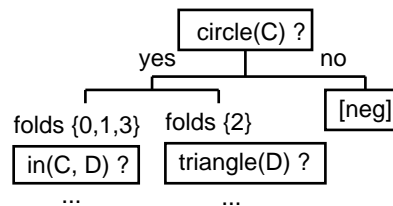


Figure 3: A 3-fold cross-validation forest.

A cross-validation forest is not a set of disjoint trees, there is some amount of sharing between the trees. The forest is grown top-down. As long as the same best query  $q^*$  is selected for each tree, the trees continue to share nodes. In Figure 3 all trees share the ‘circle(C)’ node. A group of trees or a single tree can be split off if a different  $q^*$  is selected for this group. At level two of the example forest, the tree for training set  $T_2$  is split off and stops sharing computations with group  $\{0, 1, 3\}$ .

The algorithm for refining a node  $N$  of the cross-validation forest is shown in Figure 4. It keeps track of a tuple  $(q^*, Q^*)$  for each tree in the group of trees  $G$  that shares  $N$ . The first component of this tuple,  $q_i^*$ , is the best query found so far for tree  $i$  and the second component,  $Q_i^*$ , is  $q_i^*$ ’s quality. In each iteration a new refinement  $q$  is evaluated on the data (Lines 3 - 6). The tuple  $(q^*, Q^*)_i$  is updated (Line 11) if  $q$  is better than  $q_i^*$ .

Lines 3 - 6 compute statistics  $S_i^D$  for query  $q$  on each set  $D_i$  of the cross-validation partition. Because these sets are disjoint, the query is evaluated only once on each example from  $D$ . This assures that no redundant computations occur. To estimate the quality  $Q$  (Line 10) we need statistics  $S_i^T$  on the training sets  $T_i$ , which are derived from the previously calculated  $S_i^D$  (Line 7 - 8). This step is only possible if the statistics are additive, meaning that  $S^{A_1 \cup A_2} = S^{A_1} + S^{A_2}$

1.  $(q^*, Q^*)_i = (\text{none}, -\infty)$ ,  $i \in G$
2. **for each** refinement  $q$
3.      $S_{0..n}^D = 0$
4.     **for each**  $i \in 1 \dots n$
5.         **for each**  $e \in D_i$
6.             update\_statistics( $S_i^D$ ,  $q(e)$ ,  $e$ )
7.      $S^D = \sum_{i=1}^n S_i^D$
8.     **for each**  $i \in G$
9.          $S_i^T = S^D - S_i^D$
10.          $Q = \text{compute\_quality}(S_i^T)$
11.         **if**  $Q > Q_i^*$  **then**  $(q^*, Q^*)_i = (q, Q)$
12. **for each** different  $q^* \in \{q_i^*\}$
13.     partition  $D_i$  according to  $q^*$

Figure 4: Parallel cross-validation.

if  $A_1 \cap A_2 = \emptyset$ . Each statistic  $S$  is a tuple with two components  $PS$  and  $NS$ . The positive component  $PS$  is updated if  $q(e)$  succeeds (Line 6) and the negative component  $NS$  is updated if  $q(e)$  fails.

The last two lines of the algorithm add new nodes to the forest. One for each different  $q^*$ . Each new node has two children: a positive child (i.e.  $q^*$  succeeds) and a negative child (i.e.  $q^*$  fails). The parallel cross-validation algorithm is called recursively for each child until a stopping criterion is met.

It is shown in (Blockeel and Struyf, 2001) that the speed-up factor  $T_{\text{serial}}/T_{\text{parallel}}$  is given by

$$\frac{n \cdot t_r(1) + n \cdot t_r(2) + n \cdot t_r(3) + \dots}{t_r(1) + f(1) \cdot t_r(2) + f(2) \cdot t_r(3) + \dots} \quad (1)$$

where  $t_r(i)$  is the average time for growing one level of a single tree from  $D$  and  $f(i)$  is the average number of tree groups that have been split off in the forest. Because tree groups usually split off at lower levels of the forest, where only a few examples are left, speed-up is in most cases quite good.

The algorithm of Figure 4 shares only computations within a tree group. Under some conditions its also possible to share computations among different tree groups. This is discussed in more detail in (Struyf and Blockeel, 2001).

Note the similarity between a query-pack and a cross-validation forest. The structure and the basic idea are the same. A cross-validation forest represents the shared part of similar decision trees and a query-pack represents the shared

part of similar queries. Parallel cross-validation can be seen as parallel model building on similar data and query-pack evaluation as parallel query evaluation with similar queries.

#### 4.1 Combination with Query-packs

Parallel cross-validation can be combined with query-pack evaluation. The algorithm consists of two parts. In the first part, it evaluates the queries on the data and calculates statistics for each query. In the second part, it calculates for each fold the quality measure for all queries and selects the best query  $q_i^*$  for each fold. Evaluating the queries can be done in using query-pack evaluation. The difference with the original parallel cross-validation algorithm is that now all queries are run in parallel on the data. This implies that we have to keep statistics for each query and for each fold. These can be stored in an  $(n+1) \times s$  statistic matrix ( $s$  is the number of queries in the pack; the “pack-size”). The rows of this matrix correspond to the different trees in the forest and the columns correspond to the different queries in the pack. Another difference is that we have to switch from positive and negative statistics to positive and total statistics as discussed in Section 3. This is because it is difficult to know if some query in the pack fails for a given example.

By combining parallel cross-validation with query-packs, we remove the two types of redundancies at once. Similar queries are evaluated efficiently using query-pack execution. Similar training sets are handled efficiently by parallel cross-validation. The obtained speed-up will be maximal if query complexity is high (due to query-pack execution) or if model stability is high (due to parallel cross-validation). Note that high query complexity can imply low model stability. Combining query-pack execution and parallel cross-validation results in a robust algorithm that scores reasonable on a wide range of data sets.

#### 4.2 Rule induction

Although we focused on decision tree induction, almost everything said so far also applies to top down rule induction (e.g. FOIL (Quinlan, 1990), Progol (Muggleton, 1995)) and to top down constraint induction (e.g. ICL (De Raedt and Van Laer, 1995b)). A top down rule induction system tries to cover all positive examples

by learning a set of conjunctive rules. Each time a new rule is learned, the system removes the covered positive examples from the data set and tries to learn a next rule until all positive examples are covered or no more good rules can be found. To learn one rule, the system starts from the most general rule ‘true’ and keeps adding the best literal according to some quality measure as long as the rule’s quality improves.

We can use the ideas from the parallel decision tree cross-validation algorithm to build a parallel rule cross-validation algorithm. This parallel rule cross-validation algorithm builds a tree that represents the shared part between rules for different folds. If one or more folds select a different literal then the rule for this group of folds is split off and forms a new branch. This can be compared to the cross-validation forest from Section 4. As long as a group of rules remains together, computations can be shared by evaluating the rule for this group on the disjoint sets  $D_i$  and not on the overlapping training sets  $T_i$ .

Before a rule induction system starts to learn a new rule, it removes all positive examples covered by the previous rule. As long as the same rules are selected for different folds of the cross-validation (i.e. the tree has no branches) the same examples are removed from the data set  $D$  and  $D$  remains equal for all folds. If fold  $i$  selects a different rule then different examples will be removed from  $D$  and the data set for fold  $i$  will differ from the data sets of the other folds. If we partition  $D$  in  $(D^s, D^1 \dots D^n)$ , where  $D^s$  contains the shared examples and  $D^i$  contains the examples for fold  $i$  not in  $D^s$ , then it is possible to share computations over  $D^s$ .

## 5 Experimental Results

For our experiments we implemented parallel cross-validation as a module of TILDE, the first order decision tree learner of the ACE data mining tool<sup>1</sup>. ACE is built on top of ILPROLOG, a Prolog engine dedicated to data mining that supports query-pack evaluation. ACE and ILPROLOG are discussed in (Blockeel et al., 2000).

We compare execution times of a 10-fold run for serial cross-validation (no optimisations), serial cross-validation with query-packs, paral-

<sup>1</sup>ACE is available for academic purposes upon request: <http://www.cs.kuleuven.ac.be/~dtai/ACE/>

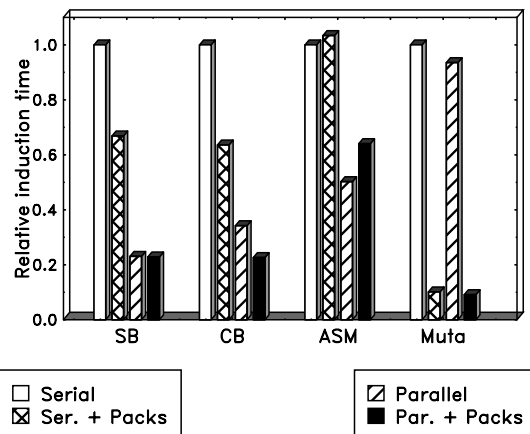


Figure 5: Different cross-validation algorithms (time relative to serial 10-fold).

lel cross-validation and parallel cross-validation with query-packs. All experiments are performed on an Intel P3 850Mhz with 256MB RAM running Linux kernel 2.2.20.

The data sets used are:

- The simple (SB) and complex (CB) Bongard data set (De Raedt and Van Laer, 1995a) (this set was also used as running example in this text). SB contains 1453 examples with a simple underlying theory, CB contains 1521 examples with a more complex theory.
- A subset (ASM) of 999 examples sampled from the ‘‘Adaptive Systems Management’’ data set, kindly provided to us by Perot Systems Nederland.
- The Mutagenesis (Muta) data set (Srinivasan et al., 1996), an ILP benchmark (230 examples).

As can be seen in Figure 5 and Table 1 the results are not uniform for all data sets. This is because we deliberately selected different types of data sets. For SB parallel cross-validation performs very well. No tree groups are split off. The packs version does not perform much better because the queries are too short to have many literals in common. CB is the least surprising data set: some tree groups are split off and the queries become longer. The ASM data set contains many propositional numeric attributes. Parallel cross-validation works rea-

Table 1: Timings (seconds) comparing parallel to serial cross-validation (10-folds), once with packs disabled and a second time with packs enabled.

| <b>SB</b>   | Packs off | Packs on |       |
|-------------|-----------|----------|-------|
| Serial      | 16        | 11       | 1.5×  |
| Parallel    | 3.6       | 3.5      | 1.0×  |
|             | 4.4×      | 3.1×     |       |
| <b>CB</b>   |           |          |       |
| Serial      | 24        | 15       | 1.6×  |
| Parallel    | 8.2       | 5.4      | 1.5×  |
|             | 2.9×      | 2.8×     |       |
| <b>ASM</b>  |           |          |       |
| Serial      | 4100      | 4300     | 0.95× |
| Parallel    | 2100      | 2600     | 0.81× |
|             | 2.0×      | 1.7×     |       |
| <b>Muta</b> |           |          |       |
| Serial      | 5000      | 500      | 10×   |
| Parallel    | 4600      | 450      | 10×   |
|             | 1.09×     | 1.11×    |       |

sonably well on this data set. Query-packs perform bad on ASM because the pack-size is high (it contains a huge number of tests comparing numeric attributes to each of their discretised values). The time necessary for compiling a pack depends on the pack-size. Near the leaves this compilation time dominates the execution time which is linear in the number of examples. The Mutagenesis data set is not suited for parallel cross-validation because some queries, generated near the leaves of the forest, dominate the total execution time (this happens when the algorithm looks for circular substructures in the molecules). Query-packs on the other hand perform very well for Mutagenesis (10 times faster). This is because query-pack execution shares computations among the different refinements of these few complex queries.

In the previous experiment we’ve set the number of cross-validation folds to 10 because this value is used often in practice. It is however interesting to see what happens if we increase the number of folds. Figure 6 plots the time for run-

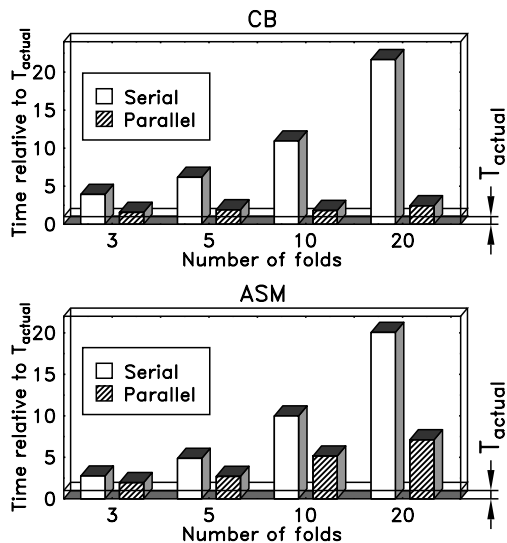


Figure 6: Induction time in function of the number of folds (CB and ASM).

ning a parallel cross-validation (no query packs) in function of the number of folds for the CB and ASM data sets. The scale is relative to  $T_{actual}$ , the time for building one single tree from the data set  $D$ .

The parallel cross-validation time for CB is almost independent of the number of folds. This is because model stability is high for this data set. Tree groups split off at low levels. For ASM, the parallel cross-validation time has a linear component in the number of folds. In general, we can expect that the time for parallel cross-validation varies sub-linear with the number of folds. This is because training set similarity increases with the number of folds. If training sets are more similar, tree groups will split off at lower levels of the forest and the linear component of the parallel cross-validation time reduces.

## 6 Conclusions

We summarised two techniques for speeding up model building in the context of multi-relational data mining. The first one was query-pack evaluation and the second one parallel cross-validation. Query-pack evaluation is parallel evaluation of similar queries and parallel cross-validation is building similar models in parallel from similar training sets.

The experimental results show that we obtain a robust algorithm that scores well on a wide

range of data sets if we combine parallel cross-validation with query pack evaluation.

Parallel cross-validation can be compared to incremental cross-validation as discussed by Utgoff (Utgoff, 1997) and Kohavi (Kohavi, 1995). The idea is to use an incremental machine learning algorithm to perform leave-one-out cross-validation by first building a model from  $D$  and then using the in(de-)cremental algorithm to remove one example  $e_i$  for each fold  $i$ .

Note that the parallel cross-validation idea can also be used for other applications (e.g. bagging) where one needs to build a set of similar models from similar training data.

## Acknowledgments

The authors are a research assistant, respectively post-doctoral fellow, of the Fund for Scientific Research of Flanders (Belgium). They thank Perot Systems Nederland / Syllogis for providing the ASM data. The cooperation between Perot Systems Nederland and the authors was supported by the European Union's Esprit Project 28623 (Aladin).

## References

- H. Blockeel and L. De Raedt. 1998. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June.
- H. Blockeel and J. Struyf. 2001. Efficient algorithms for decision tree cross-validation. In *Proceedings of ICML-2001 - Eighteenth International Conference on Machine Learning*, pages 11–18. Morgan Kaufmann.
- H. Blockeel, B. Demoen, L. Dehaspe, G. Janssens, J. Ramon, and H. Vandecasteele. 2000. Executing query packs in ILP. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference in Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 60–77, London, UK, July. Springer.
- L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. 1984. *Classification and Regression Trees*. Wadsworth, Belmont.
- L. De Raedt and S. Dzeroski. 1994. First order  $jk$ -clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392.
- L. De Raedt and W. Van Laer. 1995a. Inductive constraint logic. In Klaus P. Jantke, Takeshi Shinohara, and Thomas Zeugmann, editors, *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*, pages 80–94. Springer-Verlag.
- L. De Raedt and W. Van Laer. 1995b. Inductive constraint logic. Unpublished.
- L. Dehaspe and H. Toivonen. 1999. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36.
- Ron Kohavi and George John. 1995. Automatic parameter selection by minimizing estimated error. In Armand Frieditis and Stuart Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 304–312. Morgan Kaufmann, July.
- Ron Kohavi. 1995. The power of decision tables. In Nada Lavrac and Stefan Wrobel, editors, *Proceedings of the European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence 914, pages 174–189, Berlin, Heidelberg, New York. Springer Verlag.
- Stefan Kramer. 1996. Structural regression trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 812–819, Cambridge/Menlo Park. AAAI Press/MIT Press.
- S. Muggleton. 1995. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286.
- J.R. Quinlan. 1990. Learning logical definitions from relations. *Machine Learning*, 5:239–266.
- J. R. Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in machine learning. Morgan Kaufmann.
- A. Srinivasan, S.H. Muggleton, M.J.E. Sternberg, and R.D. King. 1996. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1,2).
- J. Struyf and H. Blockeel. 2001. Efficient cross-validation in ILP. In *Proceedings of the Eleventh International Conference on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence, pages 228–239. Springer-Verlag.
- Paul E. Utgoff. 1997. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29:5.