Predicate Invention in Inductive Data Engineering

Peter A. Flach

ITK

Institute for Language Technology and Artificial Intelligence
Tilburg University, PObox 90153, 5000 LE Tilburg, the Netherlands
☎ +31 13 663119, fax +31 13 663069
flach@kub.nl

Abstract. By *inductive data engineering* we mean the (interactive) process of restructuring a knowledge base by means of induction. In this paper we describe INDEX, a system that constructs decompositions of database relations by inducing attribute dependencies. The system employs heuristics to locate exceptions to dependencies satisfied by most of the data, and to avoid the generation of dependencies for which the data don't provide enough support. The system is implemented in a deductive database framework, and can be viewed as an Inductive Logic Programming system with predicate invention capabilities.

1. Motivation and Scope

The application of Machine Learning techniques to databases is a subject that is receiving increasing amounts of attention (Piatetsky-Shapiro & Frawley, 1991). Databases typically contain large quantities of extensional data, while for applications like query answering and data modeling intensional data is needed. Machine Learning techniques such as Inductive Logic Programming or ILP (Muggleton, 1992) can generate intensional predicate definitions from extensional data.

In this paper, we are describing inductive techniques to obtain predicate definitions that can be used to restructure a knowledge base in a more meaningful way. What is meaningful and what is not can be partly determined by means of heuristics, but some user interaction is typically required. We use the term *inductive data engineering* to denote the interactive process of restructuring a knowledge base by means of induction.

We will introduce the main ideas by means of an example. Fig. 1 shows a fragment of a train schedule, listing the direction, departure time, and first stop of the trains leaving between 8:00 and 10:00. While such a schedule is useful from a traveler's point of view, a database designer would object to storing the schedule in a database as is: the schedule appears to be unstructured, yet contains a lot of redundancy. Instead, she would decompose

Part of this work was carried out under Esprit Basic Research Action 6020 (Inductive Logic Programming). Many thanks to Luc De Raedt, Nada Lavrac and Saso Dzeroski for stimulating discussions and helpful comments on an earlier draft. Saso also conducted the experiment with mFOIL.

```
% train(Direction,Hour,Minutes,FirstStop).
train(utrecht,8,8,den-bosch).
train(tilburg,8,10,tilburg).
train(maastricht,8,10,weert).
train(utrecht,8,13,eindhoven-bkln).
train(tilburg,8,17,eindhoven-bkln).
train(utrecht,8,25,den-bosch).
train(utrecht,8,31,utrecht).
train(utrecht,8,43,eindhoven-bkln).
train(tilburg,8,47,eindhoven-bkln).
train(utrecht,9,8,den-bosch).
train(tilburg,9,10,tilburg).
train(maastricht,9,10,weert).
train(utrecht,9,13,eindhoven-bkln).
train(tilburg,9,17,eindhoven-bkln).
train(utrecht,9,25,den-bosch).
train(utrecht,9,43,eindhoven-bkln).
train(tilburg,9,47,eindhoven-bkln).
```

Fig. 1. A train schedule.

the data into more primitive relations, and define the complete schedule as a view or intensional relation. The reader is encouraged to try to find a meaningful decomposition for herself before reading on.

Usually, the design of a database schema is based on a conceptual model of the universe of discourse, and precedes the creation of the database. Central to the research reported on in this paper is the idea, that part of a database schema can be derived from the data itself, by analysing the regularities displayed by the data. This process is inherently inductive, since we are deriving general rules (predicate definitions, integrity constraints) from specific data (instances of a database relation). We have implemented a system called INDEX that is able, with some help of the user, to construct the knowledge base in fig. 2.

The restructured knowledge base contains predicates not present in the data. Thus, INDEX is capable of *predicate invention*. This is achieved by inducing *integrity constraints* that indicate that introducing a new predicate could be meaningful. E.g., the distinction between fast and slow trains is made on the basis of a *functional dependency* from direction to first stop, which holds for fast and slow trains separately. Also, the introduction of the predicates `fast_stop1` and `slow_stop1` is justified by this dependency.

The plan of the paper is as follows. In the next section, we introduce some terminology and notation. In the two sections that follow, the main steps of our algorithm are described: the search for specific integrity constraints (section 3), and the construction

```
train(A,B,C,D) :-
     regulartrain(A,B,C,D) ; irregulartrain(A,B,C,D).
regulartrain(A,B,C,D) :-
     hour(B),
     regulartrain1(A,C,D).
regulartrain1(A,B,C) :-
     fasttrain(A,B,C) ; slowtrain(A,B,C).
fasttrain(A,B,C) :-
     fasttrain1(A,B),
     fast_stop1(A,C).
slowtrain(A,B,C) :-
     slowtrain1(A,B),
     slow_stop1(A,C).

% hour(Hour).                    % irregulartrain(Dir,Hour,Mins,Stop1).
hour(8).                         irregulartrain(utrecht,8,31,utrecht).
hour(9).
% fast_stop1(Dir,Stop1).         % slow_stop1(Dir,Stop1).
fast_stop1(tilburg,tilburg).     slow_stop1(tilburg,eindhoven-bkln).
fast_stop1(maastricht,weert).    slow_stop1(utrecht,eindhoven-bkln).
fast_stop1(utrecht,den-bosch).
% fasttrain1(Dir,Mins).          % slowtrain1(Dir,Mins).
fasttrain1(maastricht,10).       slowtrain1(tilburg,17).
fasttrain1(tilburg,10).          slowtrain1(tilburg,47).
fasttrain1(utrecht,8).           slowtrain1(utrecht,13).
fasttrain1(utrecht,25).          slowtrain1(utrecht,43).
```

Fig. 2. The restructured knowledge base for train schedules.

of decompositions that are justified by the induced constraints (section 4). In section 5, we describe the heuristics employed by INDEX to be able to deal with constraints with a limited number of exceptions, and to avoid the generation of constraints that are too specific. In section 6, we relate INDEX to other ILP systems. The concluding section contains some ideas for future work.

2. Preliminaries

Our terminology and notation will be mainly drawn from the fields of Logic Programming (Lloyd, 1987) and Deductive Databases (Minker, 1988). If `r` is an *n*-ary predicate, then an *extensional relation* is a set of ground facts $r(a_1, \ldots, a_n)$. An *intensional relation* or *predicate definition* of a predicate `p` is a set of definite clauses with `p` in the head. A

(*deductive*) *database* is a collection of intensional and extensional relations. In addition, it will be convenient to denote argument positions of predicates by *attributes*, as customary in the theory of relational databases (Maier, 1983).

Our aim is to reformulate an extensional relation as an intensional relation, defined in terms of newly introduced, more compact extensional relations. This process is referred to as *decomposition*, and it is done on the basis of integrity constraints. In general, an integrity constraint is a logical formula expressing knowledge about the database, without being part of any predicate definition. In this paper, we will only consider constraints on a single extensional relation, and we will use the following, more restricted definition. If R is an extensional relation, an *integrity constraint on R* is a logical formula containing only the predicate of R, and possibly directly evaluable predicates like = and <. An integrity constraint on R is satisfied by R if R constitutes a Herbrand model of the constraint. For instance, `C<60:-train(A,B,C,D)` is a constraint satisfied by the relation in fig. 1.

Attribute dependencies constitute a class of integrity constraints of particular interest, because they signal that the relation can be decomposed into smaller relations containing less redundancy. In this paper, we consider two types of attribute dependencies, namely functional and multivalued dependencies. A *functional dependency* (fd) is an integrity constraint like `D1=D2:-train(A,B1,C,D1),train(A,B2,C,D2)`, stating that if two trains have the same direction and leave at the same number of minutes after the hour, they will have the same first stop. Given the attributes `direction`, `minutes` and `stop1`, this fd will be written as `[direction,minutes]-->[stop1]`. It states that the value of the attribute `stop1` can be derived from the values of the attributes `direction` and `minute`. The attributes found on the lefthand side are called *antecedent attributes*, those on the righthand side *consequent attributes*.

Multivalued dependencies (mvds) generalise functional dependencies by associating a *set* of possible values of the consequent attribute to each combination of possible values for the antecedent attributes. For instance, if a relation describes describes events that occur weekly, the mvd `[day]->->[date]` holds: given the day of week, we can determine the set of dates on which the event occurs. E.g., if the Computer Science course occurs on Monday, October 5, and the AI course occurs on Monday, September 28, then the AI course also occurs on Monday, October 5 (and, by symmetry, the Computer Science course also occurs on Monday, September 28). Logically, this can be expressed as

```
event(Day,Date1,E2):-event(Day,Date1,E1),event(Day,Date2,E2)
```

Fds and mvds both describe the same phenomenon: that the consequent attribute(s) can be removed from the relation, and stored in a separate relation containing only the attributes in the dependency. The only difference is, that in the case of fds the antecedent attributes form a key in the second relation. We call this a decomposition of the relation; it

will be the subject of section 4. In the next section, we describe how attribute dependencies can be induced from an extensional relation.

3. Induction of Attribute Dependencies

In (Flach, 1990) we adressed the following problem: given an extensional relation, find all fds and mvds satisfied by it. Briefly, the approach is to order the set of dependencies by generality (implication) and to search this set in a top-down fashion, much in the spirit of Shapiro's Model Inference System (1981). To illustrate, for the train relation a most general fd would be `[]-->[direction]` (all trains go in the same direction). This fd is contradicted by many pairs of facts, e.g. `train(utrecht,8,8,den-bosch)` and `train(tilburg,8,10,tilburg)`. It can be specialised by adding attributes to the antecedent. Note that the fd `[hour]-->[direction]` is contradicted by the same pair of facts: by analysing the contradicting facts, we can avoid constructing this specialisation.

By disabling the heuristics employed by INDEX, the system will find the most general dependencies satisfied by a relation. For the train schedule in fig. 1, these are

```
[minutes]->->[hour]
[stop1]->->[hour]
[stop1,minutes]-->[direction]
[minutes,direction]-->[stop1]
```

The first two mvds are specialisations of the mvd `[]->->[hour]`, expressing that trains run every hour. There is only one fact that causes contradiction of this mvd, and that is `train(utrecht,8,31,utrecht)`. In other words, had this fact not been in the relation, then the mvd would have been satisfied. As we will see later, INDEX is able to recognise that a dependency is 'almost' satisfied, and to locate the exception(s).

The fourth fd is a specialisation of `[direction]-->[stop1]`. This dependency is contradicted by many pairs of facts, but the relation can be divided into two subrelations of approximately equal size, which both satisfy the fd (fast trains and slow trains). Since several such divisions are possible, some user interaction is required to choose a meaningful one, but INDEX is able to discover that the fd is interesting in this respect.

The heuristics used by INDEX to decide whether a dependency is almost satisfied, or whether it can lead to a useful partition of the relation, are described in section 5. In the next section, we show how dependencies can be used to decompose a relation, thereby introducing new predicates.

4. Introducing New Predicates by Decomposition

A *decomposition* of a relation R is a set of relations, such that R can be reconstructed from

```
% train1(Dir,Mins,Stop1).
train1(maastricht,10,weert).
train1(tilburg,10,tilburg).
train1(tilburg,17,eindhoven-bkln).
train1(tilburg,47,eindhoven-bkln).
train1(utrecht,8,den-bosch).
train1(utrecht,13,eindhoven-bkln).
train1(utrecht,25,den-bosch).
train1(utrecht,31,utrecht).
train1(utrecht,43,eindhoven-bkln).
```

```
% stop1hour(Stop1,Hour).
stop1hour(weert,8).
stop1hour(weert,9).
stop1hour(tilburg,8).
stop1hour(tilburg,9).
stop1hour(eindhoven-bkln,8).
stop1hour(eindhoven-bkln,9).
stop1hour(den-bosch,8).
stop1hour(den-bosch,9).
stop1hour(utrecht,8).
```

Fig. 3. A horizontal decomposition.

this set by a *composition function*. If the composition function is the join operation, the decomposition is called *horizontal*. Every attribute dependency induces a unique horizontal decomposition. A *vertical* decomposition is a partition of R into subsets, with set-theoretical union as composition function. An attribute dependency induces a vertical decomposition if the dependency is satisfied by every subrelation. In general, a dependency induces many vertical decompositions, even if we are interested in *minimal* decompositions (that are not finer partitions than other decompositions induced by the same dependency). Choosing a meaningful vertical decomposition requires domain knowledge, and is done in INDEX with the help of an oracle.

Both horizontal and vertical decompositions introduce new extensional relations, with new predicates. The composition function then serves as an intensional definition of the original relation. Thus, decompositions 'intensionalise' existing relations in terms of new, extensional relations. This will be illustrated below.

We will first consider horizontal decompositions, induced by non-violated dependencies. E.g., the mvd `[stop1]->->[hour]` says that we can remove the attribute `hour` from the train relation, and store it in a separate relation with the `stop1` attribute. This results in the decomposition in fig. 3. The composition function is a join over the attribute `stop1`. This can be expressed as a logical formula:

```
train(A,B,C,D):-train1(A,C,D),stop1hour(D,B)
```

Given an extensional relation and a dependency satisfied by it, INDEX automatically constructs the horizontal decomposition and the clause expressing the join (querying the user to name the new predicates).

A vertical decomposition is induced if the dependency is violated by the relation. Formally, given a relation R and a logical formula F representing a dependency, a pair of facts $\langle f_1, f_2 \rangle$ is called *F-conflicting* if it satisfies the body of F by means of a substitution θ, while $H\theta$ is false, where H denotes the head of F. F-conflicting tuples are separated in the

`regulartrain1(maastricht,10,weert).`	(F)
`regulartrain1(utrecht,13,eindhoven-bkln).` `regulartrain1(utrecht,43,eindhoven-bkln).`	(S)
`regulartrain1(utrecht,8,den-bosch).` `regulartrain1(utrecht,25,den-bosch).`	(F)
`regulartrain1(tilburg,17,eindhoven-bkln).` `regulartrain1(tilburg,47,eindhoven-bkln).`	(S)
`regulartrain1(tilburg,10,tilburg).`	(F)

Fig. 4. A non-minimal decomposition.

first two steps of the following procedure. In the third step, blocks are combined to form a minimal decomposition.

1. Partition R into subsets with equal values for the antecedent attributes; call this the *antecedent partition*.
2. In each block B, define $f_1 \approx_B f_2$ if $\langle f_1 f_2 \rangle$ is not F-conflicting; $\approx_B$ is an equivalence relation. Refine each block of the antecedent partition into $\approx_B$-equivalence classes. Let m be the maximum number of equivalence classes constructed for a class.
3. We have now constructed a non-minimal vertical decomposition. To obtain a minimal decomposition, we combine as many blocks with different antecedent values as possible. This can be done in many ways, and we assume an oracle to guide this process. Note that this minimal decomposition consists of m relations.

We will illustrate this procedure by two examples. First, we consider the mvd `[]->->[hour]`, that is almost satisfied by the train relation. Since this dependency doesn't have antecedent attributes, the first step of the procedure is superfluous. The second step of the algorithm will result in two blocks (m=2), one containing the exceptional fact `train(utrecht,8,31,utrecht)`, and the other containing the rest. This is the unique minimal decomposition, and no user interaction is required (apart from naming the new relations). The composition rule is the disjunctive clause

`train(A,B,C,D):-regulartrain(A,B,C,D) ; irregulartrain(A,B,C,D)`

which can be written as two separate clauses, if preferred. Since `regulartrain` now satisfies the constraint `[]->->[hour]`, we can horizontally decompose it and obtain the rule `regulartrain(A,B,C,D):-hour(B),regulartrain1(A,C,D)` (note

that both composition rules and new predicates are fully determined by the decomposition).

We now consider a more elaborate example of vertical decomposition of `regulartrain1`. Consider the fd `[direction]-->[stop1]`, which is not satisfied by `regulartrain1`. INDEX will now construct the non-minimal decomposition in fig. 4. In this figure, double lines represent the antecedent partition, while single lines represent the division into non-conflicting subsets, constructed in the second step. Again, we have m=2, and any minimal decomposition will consist of two subrelations.

Currently, INDEX does not provide any help in putting blocks together in a meaningful way. In general, this seems something that can't be done without user interaction. It might be possible hower to formulate some useful heuristics. For instance, in fig. 4 all slow trains (marked (S)) have the value `eindhoven-bkln` for the consequent attribute, while fast trains have different values.

We end this section with a brief analysis of the complexity of the decomposition algorithm. Step 1 is accomplished by sorting the facts on the values of the antecedent attributes, requiring $O(af \log f)$ comparisons (a is the number of antecedent attributes, f is the number of facts in the relation). Likewise, step 2 is of complexity $O(f_B \log f_B)$, where f_B is the number of facts in block B. Finally, the number of queries to the user is $n_B{*}m$ in the worst case (n_B is the number of blocks in the antecedent partition).

5. Heuristics

Satisfied dependencies induce horizontal decompositions, while dependencies that are not satisfied induce vertical decompositions. Many dependencies however induce uninteresting decompositions. Thus, we need heuristics for predicting whether a dependency induces a meaningful decomposition.

In INDEX, two heuristics are used: satisfaction and confirmation. *Satisfaction* estimates the extent to which a dependency is satisfied (1 means no contradiction). It is abstractly calculated as follows:

$$Sat = 1 - \textit{weighted fraction of deviating facts} \qquad (1)$$

In order to estimate the fraction of deviating facts, the two-step partitioning procedure of the previous section is executed. For each block of the antecedent partition, the largest block resulting from the second step is taken to represent non-deviating facts. If the sum of the sizes of these largest blocks is N_n, then the fraction of deviating facts is $(N_R - N_n)/N_R$, where N_R is the total number of facts. This fraction is weighted with $m-1$ (recall that m is the maximum number of blocks constructed in the second step for a block of the antecedent partition; if m=1, the dependency is satisfied). This weight is added because minimal decompositions consist of m blocks, and decompositions with fewer blocks are preferred.

This gives the following formula:

$$Sat = 1 - (m-1) * \frac{N_R - N_n}{N_R} \qquad (2)$$

For instance, for the dependency `[]->->[hour]` on the relation `train`, we have N_R=17, N_n=16, and m=2, which gives *Sat*=0.94. This value indicates that the dependency is almost satisfied. The fd `[direction]-->[stop1]` on the relation `regulartrain1` gets the value 0.63 (N_R=8, N_n=5, m=2). This indicates a vertical decomposition into two subrelations of similar size. Thus, *Sat* can be used in two ways: with a lower bound (e.g. 0.8), one selects dependencies that are almost satisfied. With an interval around 0.5, one selects dependencies that are likely to result in an evenly-sized decomposition.

In order to avoid generating very specific dependencies (with many antecedent attributes), a confirmation measure is used. If a dependency is very specific, then the blocks in the antecedent partition will be small. *Confirmation* is defined as the average block size in the antecedent partition:

$$Conf = \frac{N_R}{n_B} \qquad (3)$$

where n_B is the number of blocks in the antecedent partition. Putting a lower bound on *Conf* (typically 2.5) avoids too specific dependencies.

In practice, putting $Sat \geq 0.8$ for tracing exceptions, $0.3 \leq Sat \leq 0.7$ for vertical decompositions, and $Conf \geq 2.5$ worked well in the train example. For the `train` relations, the following dependencies were found:

`[]-->[hour]`	*Sat*=0.53	*Conf*=8.5
`[]->->[hour]`	*Sat*=0.94	*Conf*=8.5

For the `regulartrain1` relation, the dependencies found by INDEX were

`[direction]-->[stop1]`	*Sat*=0.63	*Conf*=2.7
`[direction]->->[stop1]`	*Sat*=0.63	*Conf*=2.7

As was to be expected, user interaction is still required to choose the preferred dependency for vertical decomposition. For instance, choosing `[]-->[hour]` means splitting the relation according to the hour (an even partition), while choosing the mvd `[]->->[hour]` means splitting the relation into general cases and exceptions. In the second case, there is not a real choice involved since both the fd and the mvd lead to the same decomposition (as is already suggested by the equal *Sat* values).

It should be noted that these heuristics are computationally expensive, since they require almost the same amount of work involved in constructing a vertical decomposition. Currently, the applicability of static data analysis (for instance, correlation between attribute values) is investigated.

6. Related Work

We presented INDEX as a tool for inductive data engineering. However, since INDEX operates in the framework of Deductive Databases, it can also be viewed as an ILP system. If we use E to denote the ground facts of the initial relation, B to denote the ground facts in the new relations obtained by decomposition, and H to denote the corresponding composition rules, then we have $B \wedge H \models E$. ILP systems typically aim at constructing H from B and E. INDEX extends this by constructing B and H from E[1].

To illustrate the relation between INDEX and other ILP systems, we reformulated the train problem as an ILP problem, using the ground facts in fig. 2 as background knowledge B, and the ground facts in fig. 1 as examples E. We then applied mFOIL (Dzeroski & Bratko, 1992), a descendant of FOIL (Quinlan, 1990), to the problem[2]. mFOIL induced the following set of rules:

```
train(A,B,C,D):-fast_stop1(A,D),fasttrain1(A,C).
train(A,B,C,D):-slow_stop1(A,D),slowtrain1(A,C).
train(A,B,C,D):-irregulartrain(D,B,C,A).
```

There are two minor differences with the rules induced by INDEX. Firstly, the `hour` literal is missing in the body of the first two clauses. Since mFOIL requires variables to be typed, and 8 and 9 are the known hours, this literal is redundant. Secondly, the first and fourth argument of `irregulartrain` are swapped in the third clause, which is explained by the fact that the only example for this clause has the same value for these arguments.

The most salient feature of INDEX as an ILP system is the invention of new predicates, a capability shared with CIGOL (Muggleton & Buntine, 1988), LFP2 (Wirth, 1989) and BLIP (Wrobel, 1989). The main difference with these systems is that in INDEX introduces new predicates indirectly, as a result of constructing integrity constraints.

INDEX is able to identify exceptions to dependencies that are 'almost' satisfied. Thus, it is related to the Closed World Specialisation technique of (Bain & Muggleton, 1991). This is demonstrated clearly by the composition rule that distinguishes between regular and irregular trains. Given the extensional definitions of `train`, `regulartrain` and `irregulartrain`, the implication in this rule is in fact an equivalence, and we may also write

```
regulartrain(A,B,C,D):-train(A,B,C,D),¬irregulartrain(A,B,C,D)
```

Interpreting ¬ as negation as failure, this represents a default rule with exceptions.

INDEX is also related to De Raedt's Clausal Discovery Engine (De Raedt 1992),

---

[1]This change of perspective prohibits a more extensive evaluation of INDEX relative to other ILP systems.

[2]We also applied GOLEM (Muggleton & Feng, 1990) to the problem, but the result was a set of specific rules that didn't cover all the examples.

which induces a clausal integrity theory from a Datalog database. Both systems apply refinement operators to search the space of possible integrity constraints in a top-down fashion. However, in De Raedt's system inducing constraints (such as 'nobody can be both a father and a mother') is an end in itself, while in our framework, it is a means to achieve knowledge base reformulation. Another method to induce functional dependencies is described in (Ziarko, 1991). Ziarko's method does not extend to multivalued dependencies. Finally, we note that there is a strong relation between attribute dependencies and determinations (Russell, 1989).

The increased power of INDEX as an ILP system comes at a price. First of all, some user interaction is required to choose the most meaningful dependencies used for decomposition. In the present context, we think this is inevitable: invented predicates require semantic and pragmatic justification, which seems beyond the capabilities of an inductive system. Secondly, the language for composition rules employed by INDEX is limited in expressive power: it disallows existentially quantified variables in the body of clauses.

7. Conclusion

Inductive data engineering aims at automating part of the database design process by means of inductive methods. INDEX is a system for inductive data engineering, that achieves relation decomposition through the induction of attribute dependencies. As such, it is related to other approaches to the induction of integrity constraints (Ziarko, 1991; De Raedt, 1992), and to the general problem of knowledge discovery in databases (Piatetsky-Shapiro & Frawley, 1991). The search for meaningful decompositions is guided by heuristics, that are able to locate exceptions to dependencies and to find decompositions into two subrelations of approximately equal size, while avoiding the generation of dependencies that are too specific. Since INDEX operates in the framework of Deductive Databases, it can be seen as an ILP system with predicate invention capabilities.

INDEX is a research prototype, implemented in some 1000 lines of Quintus Prolog code. We are currently working on a reimplementation that can handle more substantial decomposition problems. Also, we are working on heuristics that are easier to compute, by employing static analysis of the attribute values occurring in the given tuples. Future work includes methods for constraining the search by domain knowledge, thereby alleviating the amount of user interaction needed. Furthermore, the restriction that composition rules exclude existentially quantified variables should be relaxed. A possible approach is to search for dependencies between attributes of different relations. Another approach would be the introduction of derived attributes.

References

M. BAIN & S. MUGGLETON (1991), 'Non-monotonic learning'. In *Machine Intelligence 12*, J.E. Hayes, D. Michie & E. Tyugu (eds.), pp. 105-119, Oxford University Press, Oxford.

L. DE RAEDT (1992), 'A clausal discovery engine'. In *Proc. ECAI Workshop on Logical approaches to Machine Learning*, C. Rouveirol (ed.).

S. DZEROSKI & I. BRATKO (1992), 'Handling noise in inductive logic programming'. In Proc. Second International Workshop on Inductive Logic Programming, ICOT TM-1182, Tokyo.

P.A. FLACH (1990), 'Inductive characterisation of database relations'. In *Proc. International Symposium on Methodologies for Intelligent Systems*, Z.W. Ras, M. Zemankowa & M.L. Emrich (eds.), pp. 371-378, North-Holland, Amsterdam. Full version appeared as ITK Research Report no. 23.

J.W. LLOYD (1987), *Foundations of Logic Programming*, second edition, Springer-Verlag, Berlin.

D. MAIER (1983), *The theory of relational databases*, Computer Science Press, Rockville.

J. MINKER (1988), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos.

S. MUGGLETON & W. BUNTINE (1988), 'Machine invention of first-order predicates by inverting resolution'. In *Proc. Fifth International Conference on Machine Learning*, J. Laird (ed.), pp. 339-352, Morgan Kaufmann, San Mateo.

S. MUGGLETON & C. FENG (1990), 'Efficient induction of logic programs'. In *Proc. First Conference on Algorithmic Learning Theory*, Ohmsha, Tokyo.

S. MUGGLETON, ed. (1992), *Inductive Logic Programming*, Academic Press.

G. PIATETSKY-SHAPIRO & W.J. FRAWLEY, eds. (1991), *Knowledge discovery in databases*, MIT Press.

J.R. QUINLAN (1990), 'Learning logical definitions from relations', *Machine Learning* 5:3, 239-266.

S. RUSSELL (1989), *The use of knowledge in analogy and induction*, Pitman, London.

E.Y. SHAPIRO (1981), *Inductive inference of theories from facts*, Techn. rep. 192, Comp. Sc. Dep., Yale University.

R. WIRTH (1989), 'Completing logic programs by inverse resolution'. In *Proc. Fourth European Working Session on Learning*, K. Morik (ed.), pp. 239-250, Pitman, London.

S. WROBEL (1989), 'Demand-driven concept formation'. In *Knowledge representation and organization in Machine Learning*, K. Morik (ed.), pp. 289-319, LNAI 347, Springer-Verlag, Berlin.

W. ZIARKO (1991), 'The discovery, analysis, and representation of data dependencies in databases'. In (Piatetsky-Shapiro & Frawley, 1991).