Applications of a logical discovery engine

Wim Van Laer, Luc Dehaspe and Luc De Raedt

April 25, 1994

Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
email :{Wim.VanLaer,Luc.Dehaspe,Luc.DeRaedt}@cs.kuleuven.ac.be
fax : ++ 32 16 20 53 08; telephone : ++ 32 16 20 10 15

Abstract

The clausal discovery engine CLAUDIEN is presented. CLAUDIEN discovers regularities in data and is a representative of the inductive logic programming paradigm. As such, it represents data and regularities by means of first order clausal theories. Because the search space of clausal theories is larger than that of attribute value representation, CLAUDIEN also accepts as input a declarative specification of the language bias, which determines the set of syntactically well-formed regularities.

Whereas other papers on CLAUDIEN focuss on the semantics or logical problem specification of CLAUDIEN, on the discovery algorithm, or the PAC-learning aspects, this paper wants to illustrate the power of the resulting technique. In order to achieve this aim, we show how CLAUDIEN can be used to learn 1) integrity constraints in databases, 2) functional dependencies and determinations, 3) properties of sequences, 4) mixed quantitative and qualitative laws, 5) reverse engineering, and 6) classification rules.

Keywords: inductive logic programming, knowledge discovery in databases, deductive databases, first order logic, machine learning.

1 Introduction

In the literature, a wide range of discovery systems are described (see e.g. [23, 24]). Although many of these systems are based on the same search principles (i.e. general to specific, possibly guided by heuristics), they often focuss on finding particular forms of regularities expressible in an attribute value representation. When analysing these discovery systems and the new trend of inductive logic programming (cf. [20, 21]), two important questions arise:

1. Can (some of) these techniques be abstracted into a more general technique? or can we build a generic discovery algorithm?
2. Can the representation of these discovery systems be upgraded towards the use of the more expressive first order logic framework (as in inductive logic programming)?

It is our firm belief that the answer to both questions is 'yes'. Throughout the paper, we will provide evidence to support this claim. The argumentation will start with a specification of discovery in a first order logical setting and the presentation of a generic discovery algorithm (implemented in the CLAUDIEN system) operating under this setting. In our setting, knowledge (represented in the formalism of logic programming) can be incorporated very easily in the discovery process. We will then continue to show the generality of our approach by demonstrating it on a wide variety of different discovery tasks. This will include the discovery of 1) integrity constraints in databases, 2) functional dependencies and determinations, 3) properties of sequences, 4) mixed quantitative and qualitative laws, 5) reverse engineering, and 6) classification rules. It will turn out that the language bias (which will be used to determine the syntax of the regularities of interest) will be crucial to achieve our aim. Throughout the paper, we will focuss on the mentioned applications, as the problem setting and the CLAUDIEN system were already presented elsewhere (see [5, 6, 7, 21]).

The paper is structured as follows: in section 2, we introduce the CLAUDIEN setting and algorithm, and in section 3, we focuss on the mentioned applications. Finally, in section 4, we conclude.

2 The discovery framework

2.1 Some logic programming concepts

We briefly review some standard logic programming concepts (see [19] for more details). A clause is a formula of the form $A_1, ..., A_m \leftarrow B_1, ..., B_n$ where the A_i and B_i are positive literals (atomic formulae). The above clause can be read as A_1 or ... or A_m if B_1 and ... and B_n. All variables in clauses are universally quantified, although this is not explicitly written. Extending the usual convention for *definite clauses* (where $m = 1$), we call $A_1, ..., A_m$ the *head* of the clause and $B_1, ..., B_n$ the *body* of the clause. A *fact* is a definite clause with empty body, ($m = 1$, $n = 0$). Throughout the paper, we shall assume that all clauses are *range restricted*, which means that all variables occurring in the head of a clause also occur in its body. A knowledge base KB is a set of definite clauses.

The least Herbrand model $M(KD)$ of a knowledge base KB is the set of all ground facts (constructed using the predicate, constant and functor symbols in KB) that are logically entailed by KB. A clause c is true in a model M if and only if for all substitutions θ for which $body(c)\theta \subset M$, we have that $head(c)\theta \cap M \neq \emptyset$. Roughly speaking, the truth of a clause c in knowledge base KB can be determined by running the query

$$? - body(c), not\ head(c)$$

on KB using a theorem prover (such as PROLOG). If the query succeeds, the clause is false in M. If it finitely fails, it is true.

Let us illustrate this on a small example. Suppose the KB consists of

$human(X) \leftarrow male(X)$ $\qquad male(luc) \leftarrow$
$human(X) \leftarrow female(X)$ $\qquad female(soetkin) \leftarrow$

In this knowledge base, the least model would be

$$\{male(luc), female(soetkin), human(luc), human(soetkin)\}$$

The clause $\leftarrow female(X), male(X)$ is true in this model, and the clause $male(X) \leftarrow female(X)$ is false.

2.2 Formalizing discovery in logic

As we will often be interested in regularities of a specific type, we introduce the notion of language bias. The language bias $\mathcal{L}$ will contain the set of all syntactically well-formed clauses (or regularities). The hypotheses space contains all subsets of $\mathcal{L}$. By now, we are able to formally define our notion of discovery:

Given

- a knowledge base KB
- a language $\mathcal{L}$

Find a maximal hypothesis $H \subset \mathcal{L}$ such that $H \subset \{c \in \mathcal{L} \mid c \text{ is true in } M(KB)\}$ and H does not contain logically redundant clauses, i.e. there is no $c \in H$ such that $H - \{c\} \models H$.

Often additional restrictions are imposed on clauses in H, clauses should be maximally general, the body of the clauses in H should cover at least a prespecified number of substitutions, etc. When desired, these can be used as further restrictions on H.

Our problem setting differs from the normal inductive logic programming paradigm, see [21], where one learns concepts or predicates from positive and negative examples. The differences between the two settings are elaborated in [7, 21]. From a computational point of view, the most important difference is that in our framework all clauses may be considered independent of each other, which is not true in the normal setting of inductive logic programming. Two important consequences of this are that the PAC-learning results for our setting are much better than those for the normal setting (see [6, 11, 16]) and that there are problems in the normal setting when learning multiple predicates (see for instance [8]).

One of the main contributions of this paper will be to show that a variety of different discovery tasks fit in this logical paradigm. In particular, we will show how apparently different discovery tasks can be obtained by varying the set of well-formed clauses in $\mathcal{L}$. As our aim is to design a general algorithm to solve these different discovery tasks, we need an elegant mechanism to specify the language bias.

2.3 Specifying well-formed formulae

Several formalisms to specify the bias exist in inductive logic programming, see for instance [1, 2, 3, 4, 15, 21]. It is generally agreed that among these competing formalisms that of Cohen is the most powerful but also the least declarative. On the other hand, the formalisms by Kietz and Wrobel and by Bergadano are very declarative and also complementary in the sense that languages that are easy to represent in one formalism are hard to represent in the other formalism. This motivated our group [1] to integrate these two formalisms in a straightforward manner. The resulting formalism approaches the expressive power of Cohen's formalism while retaining the same declarative spirit of the Bergadano and Kietz and Wrobel representations. We briefly present our language bias formalism here.

A language is specified as a set of clausemodels. A clausemodel is an expression of the form $HeadSet, Head \leftarrow Body, BodySet$ where

- $HeadSet$ and $BodySet$ denote sets of the form $\{A_1, ..., A_n\}$, where the A_i are logical atoms;
- $Head$ and $Body$ are of the form $A_1, ..., A_n$ where the A_i are either logical atoms or variabilized atoms;
- a logical atom is of the form $p(t_1, ..., t_n)$ where p is a predicate and the t_i are terms;
- a variabilized atom is of the form $P(t_1, ..., t_n)$ where P is a predicate variable and the t_i are terms;

The language $\mathcal{L}$ specified by a clausemodel $HeadSet, Head \leftarrow Body, BodySet$ is

$\mathcal{L} = \{Head\Theta \cup H \leftarrow Body\Theta \cup B \mid \Theta$ is a second order substitution that substitutes all predicate variables in $Head \leftarrow Body$ with predicate names; $H \subset HeadSet$ and $B \subset BodySet\}$

The notation using sets is inspired on Bergadano's work whereas the variabilized atoms are according to Kietz and Wrobel. If a language is defined by several clausemodels, the global language is the union of the local languages (consisting of the language for each individual clausemodel).

We illustrate the use of clausemodels on a simple example. Suppose the aim is to discover whether the first argument of any predicate of arity 4 is functionally dependent on its other arguments. Then an adequate clausemodel would be (with P a predicate variable):

$$X = Y \leftarrow P(X, A, B, C), P(Y, D, E, F), \{A = D, B = E, C = F\}$$

If *train* is the only predicate of arity 4, the resulting language is

$\mathcal{L} = \{X = Y \leftarrow train(X, A, B, C), train(Y, D, E, F);$
$X = Y \leftarrow train(X, A, B, C), train(Y, D, E, F), A = D;$
$X = Y \leftarrow train(X, A, B, C), train(Y, D, E, F), B = E;$
$X = Y \leftarrow train(X, A, B, C), train(Y, D, E, F), C = F;$
$X = Y \leftarrow train(X, A, B, C), train(Y, D, E, F), A = D, B = E;$
$X = Y \leftarrow train(X, A, B, C), train(Y, D, E, F), A = D, C = F;$
$X = Y \leftarrow train(X, A, B, C), train(Y, D, E, F), B = E, C = F;$
$X = Y \leftarrow train(X, A, B, C), train(Y, D, E, F), A = D, B = E, C = F\}$

Following Bergadano, further syntactic sugar can be added to this language including term-sets, lists of alternatives, etc. A full discussion of these further extensions is outside the scope of this paper.

2.4 The CLAUDIEN algorithm

We briefly sketch the CLAUDIEN algorithm that efficiently implements the above discovery paradigm. For a full discussion of CLAUDIEN and its several optimizations, we refer to [5].

A key observation underlying CLAUDIEN is that clauses c that are false in the least model model of the knowledge base KB are overly general, i.e. that there exist substitutions θ for which $body(c)\theta$ is true and $head(c)\theta$ is false in the model. As they are overly general they should be specialized. Applying standard ILP principles, we can use a refinement operator ρ (under θ-subsumption) for this (cf.[21, 28]). Combining this with artificial intelligence search techniques results in the following basic algorithm:

```
$Q := \{false\}$; $H := \emptyset$;
while $Q \neq \emptyset$ do
    delete $c$ from $Q$
    if $c$ is true in the minimal model of $KB$
    then add $c$ to $H$
    else add all refinements $\rho(c)$ of $c$ to $Q$
    endif
endwhile
```

Figure 1: The simplified CLAUDIEN algorithm

First, we want to stress that we have made several important optimizations of this algorithm, in particular, we employ an optimal refinement operator (which generates all clauses in $\mathcal{L}$ at most once), we use advanced search strategies, we test whether clauses c are logically redundant with regard to H, and we apply the balance principle to prune away useless clauses, see [5] for more information on this. Secondly, it is important to regard CLAUDIEN as an *any-time* algorithm. By this we mean that the algorithm can be interupted at any time. The longer CLAUDIEN will run, the more clauses the hypothesis will contain, and the more interesting the results will be. Using partial results has proven to be sufficient for many interesting tasks. Moreover, searching the whole space of solutions may not be possible because the space of possible clauses could be infinite. This any-time approach to discovery contrasts with the classical covering and classification oriented approach, where the aim is to distinguish between positive and negative examples of given concept. Here, we are more interested in finding interesting regularities of a certain form, and this without a priori restricting the use of the hypotheses to classification or prediction. Whereas an any-time algorithm is acceptable for discovery tasks, it probably is not for classification (but cf. also our experiments).

3 Experiments

3.1 Databases

The first experiments shows CLAUDIEN at work in a database setting containing facts about family relations, including $human, male, female$. Upon running CLAUDIEN with the clausemodel
$\{human(X), male(X), female(X)\} \leftarrow \{human(X), male(X), female(X)\}$
CLAUDIEN discovers the following non-redundant hypothesis:

$\leftarrow female(X), male(X)$	$human(X) \leftarrow female(X)$
$human(X) \leftarrow male(X)$	$male(X), female(X) \leftarrow human(X)$

This simple example shows that CLAUDIEN could contribute to databasedesign, where one would start with a set of facts (a model) and where CLAUDIEN could help to derive the definitions of view predicates and integrity constraints. In this example, the first and last clauses would typically be integrity constraints and the second and third one would define the view predicate *human*.

3.2 Functional dependencies and determinations

One of the important topics in knowledge discovery in databases addresses how to efficiently discover specific types of regularities, such as functional and multivalued dependencies (see e.g. [13, 14, 26]) and determinations (see [27, 29]). We ran CLAUDIEN on the following data from Flach (the term *train(From,Hour,Min,To)* denotes that there is a train from $From$ to To at time $Hour, Min$):

train(utrecht,8,8,den-bosch)	*train(tilburg,8,10,tilburg)*
train(maastricht,8,10,weert)	*train(utrecht,8,25,den-bosch)*
train(utrecht,9,8,den-bosch)	*train(tilburg,9,10,tilburg)*
train(maastricht,9,10,weert)	*train(utrecht,9,25,den-bosch)*
train(utrecht,8,13,eindhoven-bkln)	*train(tilburg,8,17,eindhoven-bkln)*
train(utrecht,8,43,eindhoven-bkln)	*train(tilburg,8,47,eindhoven-bkln)*
train(utrecht,9,13,eindhoven-bkln)	*train(tilburg,9,17,eindhoven-bkln)*
train(utrecht,9,43,eindhoven-bkln)	*train(tilburg,9,47,eindhoven-bkln)*
train(utrecht,8,31,utrecht)	

using the following clausemodels:
$X = Y \leftarrow P(X,A,B,C), P(Y,D,E,F), \{A = D, B = E, C = F\}$
$X = Y \leftarrow P(A,X,B,C), P(D,Y,E,F), \{A = D, B = E, C = F\}$
$X = Y \leftarrow P(A,B,X,C), P(D,E,Y,F), \{A = D, B = E, C = F\}$
$X = Y \leftarrow P(A,B,C,X), P(D,E,F,Y), \{A = D, B = E, C = F\}$

CLAUDIEN found (as Flach's INDEX) the following two dependencies:
$X = Y \leftarrow train(X,A,B,C), train(Y,D,E,F), C = F, B = E.$
$X = Y \leftarrow train(A,B,C,X), train(D,E,F,Y), C = F, A = D.$

It is easy to write clausemodels that would find determinations $P(X,Y) \leftarrow Q(X,Z), R(Z,Y)$ (as [29]), determinations as [27] and multivalued dependencies as [13].

3.3 Non deterministic sequence prediction

Dietterich and Michalski [9] describe an approach to non deterministic sequence prediction. The problem of non deterministic sequence prediction is that of determining constraints on the k-th event in a sequence given the $k-1$ previous events. They illustrate their system SPARC/E on the card game of Eleusis, which involves two players, of which one has to guess the secret non deterministic sequence prediction rule the other player has in mind. Since then, the game of Eleusis has been employed in the context of inductive logic programming by Quinlan [25] and Lavrac and Dzeroski [18]. We show how the task was addressed by CLAUDIEN.

Given were the following sequences of cards (taken from [25]):

1. $J\clubsuit, 4\clubsuit, Q\heartsuit, 3\spadesuit, Q\diamondsuit, 9\heartsuit, Q\clubsuit, 7\heartsuit, Q\diamondsuit, 9\diamondsuit, Q\clubsuit, 3\heartsuit, K\heartsuit, 4\clubsuit, K\diamondsuit, 6\clubsuit, J\diamondsuit, 8\diamondsuit, J\heartsuit, 7\clubsuit,$ $J\diamondsuit, 7\heartsuit, J\heartsuit, 6\heartsuit, K\diamondsuit$
2. $4\heartsuit, 5\diamondsuit, 8\clubsuit, J\spadesuit, 2\clubsuit, 5\spadesuit, A\clubsuit, 5\spadesuit, 10\heartsuit$

These sequences were translated into facts of the form:
$canfollow(4, \clubsuit, J, \clubsuit)$, $canfollow(Q, \heartsuit, 4, \clubsuit)$, ...

Notice that in contrast to the other approaches, CLAUDIEN only uses the positive examples. The backgroundknowledge in these experiments contained the definitions of *red, black, samecolor, number, face, precedesrank, lowerrank, precedessuit.* The bias consisted of the following models:

$P(R2) \leftarrow canfollow(R2, S2, R1, S1), \{red(S1), black(S1), number(R1), face(R1),$
$samecolor(S1, S2), precedesuit(S1, S2), precedesuit(S2, S1), lowerrank(R1, R2),$
$lowerrank(R2, R1), precedesrank(R1, R2), precedesrank(R2, R1)\}$

and similar ones where the variabilized atom in the head was replaced by $P(S2)$, $P(R2, R1)$, $P(R1, R2)$, $P(S2, S1)$, $P(S1, S2)$.

The results for the first experiment were (where CLAUDIEN was run till depth 4 and rules whose condition part did not cover 3 substitutions were pruned):
$number(R2) \leftarrow canfollow(R2, S2, R1, S1), face(R1)$
$number(R2) \leftarrow canfollow(R2, S2, R1, S1), lowerrank(R2, R1)$
$face(R2) \leftarrow canfollow(R2, S2, R1, S1), number(R1)$
$face(R2) \leftarrow canfollow(R2, S2, R1, S1), lowerrank(R1, R2)$

The first and third rule correspond to the target concept as in the other experiments. CLAUDIEN however also discovered two other regularities, which indeed also hold on the data.

The results for the second one were (where CLAUDIEN was run till depth 4, and rules whose condition part did not cover 2 substitutions were pruned):
$lowerrank(R1, R2) \leftarrow canfollow(R2, S2, R1, S1), precedessuit(S1, S2)$
$precedessuit(S1, S2) \leftarrow canfollow(R2, S2, R1, S1), lowerrank(R1, R2)$

An important difference between CLAUDIEN and both SPARC/E and the other inductive logic programming techniques, is that CLAUDIEN learns from positive examples only. Furthermore, a comparison with the other inductive logic programming clearly shows that our representation (i.e. that of the learned rules) is more natural and corresponds more closely to the original representation of SPARC/E.

3.4 Merging quantitative and qualitative discovery

To illustrate the potential of merging a first order logic framework and abilities to handle numbers, we will provide a simple example in analysing card sequences. First however, we need to explain how numbers are handled by CLAUDIEN (see also [30]). To handle numbers the refinement operator has to be adapted. The reason is that a standard refinement operator can only enumerate the refinements of a given clause. Since numeric data handling requires constants, and since the number of possible constants is always too large (if not infinite), an alternative mechanism is needed. In CLAUDIEN, the alternative refinement operator not only employs the given clause but also the substitutions that are and are not covered by the clause. The covered substitions θ are those for which both the body and the head are true in the least model. The uncovered subsitutions are those for which the body is true but the head is not. Based on these substitutions one can easily determine more relevant refinements. The refinement procedure employed in the example below, takes 2 covered substitions and 2 numeric variables X, Y. Then it determines the coefficients a and b such that $aX + Y = b$ for the two substitutions. If the resulting coefficients also hold for all covered substitutions and for none of the uncovered substitutions, the refinement is passed on to the queue of candidate clauses in the algorithm. Otherwise, it is discarded. Although the procedure illustrated is simple, it is quite general and could be extended towards more interesting forms of regularities (e.g. employing statistical regression techniques as in Dzeroski's LAGRANGE [12]), and towards more advanced techniques (e.g. the BACON strategy [17]).

We illustrate the quantitative technique on discovering non deterministic sequence prediction in Eleusis. The sequence employed was:

$$5\heartsuit, 7\heartsuit, 9\heartsuit, 8\spadesuit, 10\clubsuit, 2\diamondsuit, 4\diamondsuit, 6\heartsuit, 8\diamondsuit, 7\clubsuit, 6\heartsuit, 5\clubsuit, 7\spadesuit, 9\spadesuit, 3\diamondsuit, 5\diamondsuit$$

The induced rules were:
$number(R2) \leftarrow canfollow(R2, S2, R1, S1)$
$lowerrank(R1, R2) \leftarrow canfollow(R2, S2, R1, S1), samecolor(S2, S1)$
$samecolor(S1, S2) \leftarrow canfollow(R2, S2, R1, S1), lowerrank(R1, R2)$
$R2 = R1 + 2 \leftarrow canfollow(R2, S2, R1, S1), samecolor(S1, S2)$
$R2 = R1 - 1 \leftarrow canfollow(R2, S2, R1, S1), precedesrank(R2, R1)$

3.5 Reverse Engineering

The Minesweeper learning problem is based on a computer board game that comes with MICROSOFT WINDOWS © Version 3.1.

When playing Minesweeper you are presented with a mine field, simulated by a grid of covered squares, and your objective is to locate all the mines. To do this, you uncover the squares that do not contain mines, and you mark the squares that do contain mines. If you uncover a square that contains a mine, you lose the game. If the square is not a mine, a number appears that represents the number of mines in the surrounding squares. With this contextual information you can cautiously proceed, marking the squares that must be mines and uncovering those that cannot be mines. Thus for instance, an uncovered "0" will allow you to clear all the surrounding squares. The learning task presently addressed is finding also the less trivial rules of this kind. Only one-row game boards will be considered.

In this experiment the learner is given a prolog program that contains a definite clause grammar with context sensitive rules. The grammar produces legal game boards up to a certain length, set to 9. CLAUDIEN was run with a language model allowing one of the following two literals in the head of the clause: $mine(Square)$ for learning when to mark a square, and $no\_mine(Square)$ for finding situations where it is safe to uncover a square. The resulting rules are ("$\star$" is a mine, "$\surd$" is a safe square) :

Rules for *mine*(*Square*)

		★	2		←			?	2	
	2	★			←		2	?		
		★	1	√	←			?	1	√
√	1	★			←	√	1	?		
	★	1		0	←		?	1		0
0		1	★		←	0		1	?	
★	1		1	★	←	?	1		1	★
★	1		1	★	←	★	1		1	?

Rules for *no_mine*(*Square*)

		√	0		←			?	0	
	0	√			←		0	?		
		√	1	★	←			?	1	★
★	1	√			←	★	1	?		
√	1		1	√	←	?	1		1	√
√	1		1	√	←	√	1		1	?
	√	1		2	←		?	1		2
2		1	√		←	2		1	?	

This application indicates that CLAUDIEN can address a reverse engineering task. Indeed, in the Minesweeper task, CLAUDIEN starts from any program that generates legal sequences. Such programs contain all information about the legal sequences in an *implicit* form. CLAUDIEN is able to discover some relevant properties of interest in *explicit* symbolic form. Analogously, CLAUDIEN could be run on programs such as for instance *quicksort* and discover properties as $sorted(Y) \leftarrow quicksort(X,Y)$.

3.6 Classification

One standard benchmark for inductive logic programming systems operating under the normal setting (i.e. that where positive as well as negative examples are supplied of a target predicate), is that of learning finite element mesh-design (see e.g. [10, 18]). Here we will address the same learning task. However, whereas the other approaches require positive as well as negative examples, CLAUDIEN needs only the positive. Secondly, the other approaches employ Michalski's covering algorithm, where the aim is to find hypotheses that cover each positive example once. CLAUDIEN follows an alternative approach, as it merely looks for valid rules. There is therefore no guarantee that hypotheses found by CLAUDIEN will cover all positives and also a hypothesis may cover a positive example several times. We believe – and our experiments in mesh-design show – that when the data are sparse, the CLAUDIEN is to be preferred.

The original mesh-application contains data about 5 different structures (a-e), with the number of edges per structure varying between 28 and 96. There are 278 positive examples (and 2840 negative ones) and the original backgroundtheory contains 1872 facts. The original backgroundtheory was made determinate (because the GOLEM system of [22] cannot work with indeterminate clauses). As CLAUDIEN does not suffer from this restriction, we could compact the database to 639 (equivalent facts). An example of a positive example is $mesh(b11, 6)$ meaning that edge 11 of structure b should be divided in 6 subedges. Backgroundknowledge contains information about edge types, boundary conditions, loading, and the geometry of the structure. Some of the facts are shown below:

Edge types: $long(b19), short(b10), notimportant(b2), shortforhole(b28), halfcircuit(b3),$

Structure	incorrect	correct	novalue	percentage correct
A	17	31	7	56
B	30	9	3	21
C	16	5	7	18
D	37	19	1	33
E	37	15	44	16
Totals	137	79	62	28

Table 1: Results of CLAUDIEN on the mesh-data.

Structure	FOIL	MFOIL	GOLEM	CLAUDIEN
A	17	22	17	31
B	5	12	9	9
C	7	9	5	5
D	0	6	11	19
E	5	10	10	15
Total	34	59	52	79
Percentage	12	21	19	28

Table 2: Comparing CLAUDIEN to FOIL, MFOIL and GOLEM.

$halfcircuithole(b1)$
Boundary conditions: $fixed(b1)$, $twosidefixed(b6)$
Loading: $notloaded(b1)$, $contloaded(b22)$
Geometry: $neighbor(b1, b2)$, $opposite(b1, b3)$, $same(b1, b3)$

We ran CLAUDIEN on this data-set using a slightly different but equivalent representation for examples, using the leave-one-out strategy, using (complete unpruned) best-first search, with a time-limit of 1000 cpu-seconds on a SPARC. The heuristic employed was to prefer those clauses c which maximized the number of substitutions θ for which $body(c)\theta$ and $head(c)\theta$ hold. The discovered rules were then tested against the structure left out. The result are summarized in table 1.

The results of CLAUDIEN are compared with those of GOLEM, FOIL and MFOIL in table 2, these results were taken from [18].

We believe the results of these tests are very encouraging because the rules learned by CLAUDIEN have by far the best classification accuracy and also because the cpu-requirements of CLAUDIEN are of the same order as those by the other systems. The high classification accuracy can be explained by the sparseness of the data and the non-covering approach. About the time requirements, GOLEM ran for 1 hour on this data, FOIL for 5 minutes, and MFOIL for 2 hours. FOIL and GOLEM are implemented in C, MFOIL and CLAUDIEN in Prolog. The experiment clearly shows that an anytime algorithm (implemented in Prolog) is not necessarily slower than a covering approach. (Part of) a possible explanation for this may be that CLAUDIEN is the only system that does not need to employ the (large number) of negative examples.

4 Conclusions

We have presented a general and generic discovery algorithm operating in the inductive logic programming paradigm. We have shown it at work on a number of seemingly disparate discovery tasks, thus showing the power and the potential of the technique. Very crucial in this respect was the use of a flexible and declarative bias specification mechanism that allowed us to specify the syntax of the target regularities. We want to stress here that the system is also efficient, demonstrated by the fact that the experiments on the mesh-data ran in time comparable to that of the two fastest inductive logic programming system implemented in C. In conclusion, we have provided important evidence to the belief that the two questions raised in the introduction may be answered positively.

References

[1] H. Adé, L. De Raedt, and M. Bruynooghe. Declarative Bias for Bottom Up ILP Learning Systems, 1994. Submitted to Machine Learning.

[2] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1044–1049. Morgan Kaufmann, 1993.

[3] W. Cohen. Grammatically biased learning: learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 1994. To appear.

[4] L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach.* Academic Press, 1992.

[5] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1058–1063. Morgan Kaufmann, 1993.

[6] L. De Raedt and S. Džeroski. First order *jk* clausal theories are PAC-learnable. Technical Report KUL-CW-, Department of Computer Science, Katholieke Universiteit Leuven, 1993. submitted to Artificial Intelligence.

[7] L. De Raedt and N. Lavrač. The many faces of inductive logic programming. In J. Komorowski, editor, *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1993. invited paper.

[8] L. De Raedt, N. Lavrač, and S. Džeroski. Multiple predicate learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1037–1042. Morgan Kaufmann, 1993.

[9] T.G. Dietterich and R.S. Michalski. Discovering patterns in sequences of events. *Artificial Intelligence*, 25:257–294, 1985.

[10] B. Dolsak and S. Muggleton. The application of inductive logic programming to finite element mesh design. In S. Muggleton, editor, *Inductive logic programming*, pages 453–472. Academic Press, 1992.

[11] S. Džeroski, S. Muggleton, and S. Russel. PAC-learnability of determinate logic programs. In *Proceedings of the 5th ACM workshop on Computational Learning Theory*, pages 128–135, 1992.

[12] S. Džeroski and L. Todorovski. Discovering dynamics: from inductive logic programming to machine discovery. In *Proceedings of the AAAI'93 Workshop on Knowledge Discovery in Databases*, pages 125–137. AAAI Press, 1993. Washington DC.

[13] P. Flach. Predicate invention in inductive data engineering. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence, pages 83–94. Springer-Verlag, 1993.

[14] M. Kantola, H. Mannila, K.J. Raiha, and H. Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 7(7), 1992.

[15] J-U. Kietz and S. Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton, editor, *Inductive logic programming*, pages 335–359. Academic Press, 1992.

[16] J.U. Kietz. Some lower bounds for the computational complexity of inductive logic programming. In *Proceedings of the 6th European Conference on Machine Learning*, volume 667, pages 115–124. Lecture Notes in Artificial Intelligence, 1993.

[17] P. Langley, G.L. Bradshaw, and H.A. Simon. Rediscovering chemistry with the BACON system. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: an artificial intelligence approach*, volume 1, pages 307–330. Morgan Kaufmann, 1983.

[18] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1993.

[19] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, 2nd edition, 1987.

[20] S. Muggleton, editor. *Inductive Logic Programming.* Academic Press, 1992.

[21] S. Muggleton and L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 1994. to appear.

[22] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st conference on algorithmic learning theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.

[23] G. Piatetsky-Shapiro and W. Frawley, editors. *Knowledge discovery in databases.* The MIT Press, 1991.

[24] G. (Ed.) Piatetsky-Shapiro. Special issue on knowledge discovery in databases. *International Journal of Intelligent Systems*, 7(7), 1992.

[25] J.R. Quinlan. Learning logical definition from relations. *Machine Learning*, 5:239–266, 1990.

[26] I. Savnik and P.A. Flach. Bottom-up induction of functional dependencies from relations. In *Proceedings of the AAAI'93 Workshop on Knowledge Discovery in Databases*, pages 174–185. AAAI Press, 1993. Washington DC.

[27] J. Schlimmer. Learning determinations and checking databases. In *Proceedings of the AAAI'91 Workshop on Knowledge Discovery in Databases*, pages 64–76, 1991. Washington DC.

[28] E.Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1983.

[29] W.M Shen. Discovering regularities from knowledge bases. *International Journal of Intelligent Systems*, 7(7), 1992.

[30] W. Van Laer and L. De Raedt. Discovering quantitative laws in inductive logic programming. In *Proceedings of the Familiarization Workshop of the ESPRIT Network of Excellence on Machine Learning*, pages 8–11, 1993. Extended Abstract, Blanes, Spain.