Inductive Learning in Deductive Databases

Sašo Džeroski and Nada Lavrač

Abstract—**Most current applications of inductive learning in databases take place in the context of a single extensional relation. This paper puts inductive learning in the context of a set of relations defined either extensionally or intentionally in the framework of deductive databases. It presents LINUS, an inductive logic programming system that induces virtual relations from example positive and negative tuples and already defined relations in a deductive database. Based on the idea of transforming the problem of learning relations to attribute-value form, it incorporates several attribute-value learning systems. As the latter handle noisy data successfully, LINUS is able to learn relations from real life noisy databases. The paper illustrates the use of LINUS for learning virtual relations and then presents a study of its performance on noisy data.**

Keywords—**Deductive databases, inductive logic programming, machine learning.**

I. Introduction

FRAWLEY *et al.* [10] recognize that machine learning can be applied to knowledge discovery in databases. They list the conflicting viewpoints between database management and machine learning. Among the main problems, they emphasize the fact that real world databases are often incomplete and noisy, as well as much larger than typical machine learning data sets.

Attribute-value learning systems, such as C4.5 [31], ASSISTANT [3] and CN2 [4], already include techniques for handling incomplete and noisy data. They are also fairly efficient, but even more efficient approaches have been recently developed [2] that can handle very large training sets. Machine learning techniques of this kind can be used to discover patterns in an isolated file of database records, i.e., a single relation in a relational database, where records (tuples) can be viewed as training instances.

The patterns discovered by the above learning systems are expressed in attribute-value languages which have the expressive power of propositional logic. These languages are limited and do not allow for representing complex structured objects and relations among objects or their components. The domain (background) knowledge that can be used in the learning process is also of a very restricted form and other relations from the database cannot be used in the learning process.

Manuscript received October 1, 1992; revised June 15, 1993. This research was supported in part by the Slovenian Ministry of Science and Technology and is part of the ESPRIT BRA Project 6020 Inductive Logic Programming.

The authors are with the Jožef Stefan Institute, Ljubljana, Slovenia.

IEEE Log Number 9212794.

Recent developments in inductive learning focus on the problem of constructing a logical (intentional) definition of a relation [32] from example tuples known to belong or not to belong to it, where other relations (background knowledge) may be used in the induced definition. In this way, new relations can be specified by a small number of example tuples, which are then generalized to induce a logical definition. Alternatively, existing relations can be compressed into their corresponding logical definitions.

As the intentional definitions induced can be recursive [32], we may say that they are expressed in the formalism of *deductive databases* [35]. The logic programming school in deductive databases [19] argues that deductive databases can be effectively represented and implemented using logic and logic programming. Within this framework, the induction of logical definitions of relations can be considered logic program synthesis and has been recently named *Inductive Logic Programming* (*ILP*) [6], [15], [24].

Early ILP systems, such as MIS [34] and CIGOL [25], did not address the problem of handling noisy data. FOIL [32], however, has a noise handling mechanism based on an encoding length heuristic. Based on the idea of extending attribute-value learning approaches to the deductive database (logic programming) framework, it can induce logical definitions of relations from possibly noisy data.

Our ILP system LINUS [16], described in the paper, is also based on the idea of extending attribute-value approaches. Unlike FOIL, which directly uses ideas from propositional approaches, LINUS explicitly transforms an ILP problem to propositional form and then uses attribute-value learners to solve it. The solution of the propositional problem is then transformed back into the deductive database formalism.

The transformation process can only be used for a restricted class of ILP problems. The hypothesis language (i.e., the class of logic programs that can be induced) is the language of deductive hierarchical database (DHDB) clauses [19], with the additional restriction that no new variables may be introduced in the body of a clause. Strictly speaking, this class of programs is a subset of the nonrecursive views (virtual relations) that can be defined in the relational algebra. Nevertheless, this language is still more expressive than attribute-value languages, as some relations can be concisely expressed in terms of other relations, but not in an attribute-value language. Furthermore, we have extended the transformation approach to a broader class of problems, namely to those

expressible in the language of deductive database clauses [13].

LINUS is a descendent of the learning module of QuMAS (Qualitative Model Acquisition System [23]), which was used to learn functions of components of a qualitative model of the heart in the system KARDIO [1]. LINUS contributes to attribute-value learners the additional expressiveness of the DHDB formalism, which enables learning of relations in the presence of background knowledge. To ILP, LINUS brings the techniques for handling imperfect (noisy) data and therefore the potential of practical applications. In practice, LINUS was used to induce medical diagnostic rules [17], [18], and to generate rules that determine the resolution of finite element meshes in computer aided design [7]. Furthermore, it was used to learn relational descriptions in several domains known from the machine learning literature [16].

In this paper, we explore the use of LINUS for learning relations, both from non-noisy and noisy data. Section II first introduces the basic deductive databases and logic programming terminology and then defines the ILP learning task. In Section III, the LINUS environment is described. Section IV presents the results of experiments in learning relations with LINUS on two domains from the machine learning literature. One of these domains, the problem of learning illegal positions in a chess endgame, is used in Section V to study the ability of LINUS to handle noisy data. Both sections include a comparison with FOIL.

II. Inductive Logic Programming

A. Deductive Databases and Logic Programming

A n-ary relation p is a set of tuples, i.e., a subset of the Cartesian product of n domains $D_1 \times D_2 \times \cdots \times D_n$, where a *domain* (or a *type*) is a set of values. We assume that a relation is finite unless stated otherwise. A set of relations forms a *relational database* (RDB) [35].

A deductive database (DDB) consists of a set of *database* clauses. A database clause is a *typed program clause* of the form:

$$p(X_1, \cdots, X_n) \leftarrow L_1, \cdots, L_m.$$

where the body of a clause is a conjunction of positive literals $q_i(Y_1, \cdots, Y_{n_i})$ and/or negative literals *not* $q_j(Y_1, \cdots, Y_{n_j})$. The basic difference between program clauses and database clauses is in the use of types. In typed clauses, a type is associated with each variable appearing in a clause. The type of a variable specifies the range of values which the variable can take. For example, in the relation *lives_in*(X, Y), we may want to specify that X is of type *person* and Y is of type *city*.

A set of program clauses with the same predicate symbol p in the head forms a *predicate definition*. A predicate can be defined *extensionally* as a set of ground facts or *intentionally* as a set of database clauses [35]. It is assumed that each predicate is either defined extensionally or intentionally. A relation in a database is essentially the same as a predicate definition in a logic program. Table I relates database [35] and logic programming [19] terms that will be used throughout the paper.

TABLE I
Relating Database and Logic Programming Terms

DB Terminology	LP Terminology
relation name p	predicate symbol p
attribute of relation p	argument of predicate p
tuple $<a_1, \cdots, a_n>$	ground fact $p(a_1, \cdots, a_n)$
relation p—a set of tuples	definition of predicate p—a set of ground facts

Database clauses use variables and function symbols in predicate arguments. Recursive types and recursive predicate definitions are allowed. *Deductive hierarchical databases* (DHDB) [19] are deductive databases restricted to nonrecursive predicate definitions and to nonrecursive types. The latter determine finite sets of values which are constants or structured terms with constant arguments.

B. Empirical Inductive Logic Programming

Some recent inductive learning systems [16], [26], [32], construct logical definitions of relations from examples and background knowledge (other relations). They use restricted forms of program clauses [19] to represent training examples, background knowledge, and induced concept descriptions. In this case, learning can be considered logic program synthesis and has been recently named *Inductive Logic Programming* (*ILP*) [6], [15], [24].

In ILP, one can distinguish between interactive and empirical ILP systems, which learn definitions of single and multiple predicates, respectively. *Empirical ILP* systems require all training examples at the start of the learning process, while *interactive ILP* systems process the examples one by one and possibly generate examples in the learning process. Examples of empirical ILP systems are GOLEM [26], FOIL [32] and LINUS [16]. Interactive ILP systems representatives are MIS [34], CLINT [6] and CIGOL [25]. Other ILP systems include MOBAL [21] and FOCL [29].

Since empirical ILP systems deal with large sets of examples, they are more likely to be applied in practice. The task of empirical single predicate learning in ILP can be formulated as follows:

Given:

- a set of training examples $\mathcal{E}$, consisting of true $\mathcal{E}^+$ and false $\mathcal{E}^-$ ground facts of an unknown predicate p,
- a concept description language $\mathcal{L}$, specifying syntactic restrictions on the definition of predicate p,
- background knowledge $\mathcal{B}$, defining predicates q_i (other than p) which may be used in the definition of p

Find:

- a definition $\mathcal{H}$ for p, expressed in $\mathcal{L}$, such that $\mathcal{H}$ is complete, i.e., $\forall e \in \mathcal{E}^+: \mathcal{B} \wedge \mathcal{H} \neq e$, and consistent

TABLE II
A SIMPLE ILP PROBLEM: LEARNING THE *daughter* RELATIONSHIP

Training Examples		Background Knowledge			
daughter(sue, eve).	⊕	*mother(eve, sue).*	*parent(X, Y)* ←	*female(ann).*	*male(pat).*
daughter(ann, pat).	⊕	*mother(ann, tom).*	*mother(X, Y).*	*female(sue).*	*male(tom).*
daughter(tom, ann).	⊖	*father(pat, ann).*	*parent(X, Y)* ←	*female(eve).*	
daughter(eve, ann).	⊖	*father(tom, sue).*	*father(X, Y).*		

with respect to $\mathcal{E}$ and $\mathcal{B}$, i.e., $\forall e \in \mathcal{E}^-: \mathcal{B} \wedge \mathcal{H} \nvdash e$.

One usually refers to the true facts $\mathcal{E}^+$ as *positive* (⊕) *examples*, the false facts $\mathcal{E}^-$ as *negative* (⊖) *examples* and the definition of p as definition of the *target* relation. Positive examples are tuples known to belong to the target relation, while negative examples are tuples known not to belong to the target relation. When learning from noisy examples, the completeness and consistency criteria need to be relaxed in order to avoid overly specific hypotheses.

The language of concept descriptions $\mathcal{L}$ is usually called the *hypothesis language*. It is typically some subset of the language of program clauses. The complexity of learning grows with the expressiveness of the hypothesis language $\mathcal{L}$. Therefore, to reduce the complexity, additional restrictions can be applied to clauses expressed in $\mathcal{L}$. In LINUS, the concept description language is the language of DHDB clauses, with the additional restriction that clauses may not introduce new variables.

To illustrate the above definition, consider the simple problem of learning family relationships. The task is to define the target relation *daughter(X, Y)*, which states that person X is daughter of person Y, in terms of the background knowledge relations *female(X)*, *male(X)* and *parent(X, Y)*. The relations *female* and *male* are defined extensionally, while *parent* is defined intentionally in terms of *mother* and *father*. These relations are given in Table II, where all attributes are of type *person*. There are two positive and two negative examples of the target relation.

III. LINUS: AN ENVIRONMENT FOR INDUCTIVE LEARNING

The main idea in LINUS is to transform the task of learning relational DHDB descriptions into an attribute-value learning task. This is achieved by what is called a *DHDB interface*, consisting of over 2000 lines of Prolog code, which allows for incorporating attribute-value learners into the logic programming environment [16]. At present, LINUS incorporates three attribute-value learning programs: ASSISTANT [3], NEWGEM [22] and CN2 [4]. It is easy to incorporate other attribute-value learners, from which we can choose the ones best suited to the problem at hand.

The interface transforms the training examples from the DHDB form into the form of attribute-value tuples. The most important feature of this interface is that, by taking into account the types of the arguments of the target predicate, applications of background knowledge predicates are considered as attributes for attribute-value learning. Existing attribute-value algorithms can then be used to induce *if-then* rules, which are in turn transformed into the form of DHDB clauses by the DHDB interface. An overview of LINUS is given in Fig. 1.

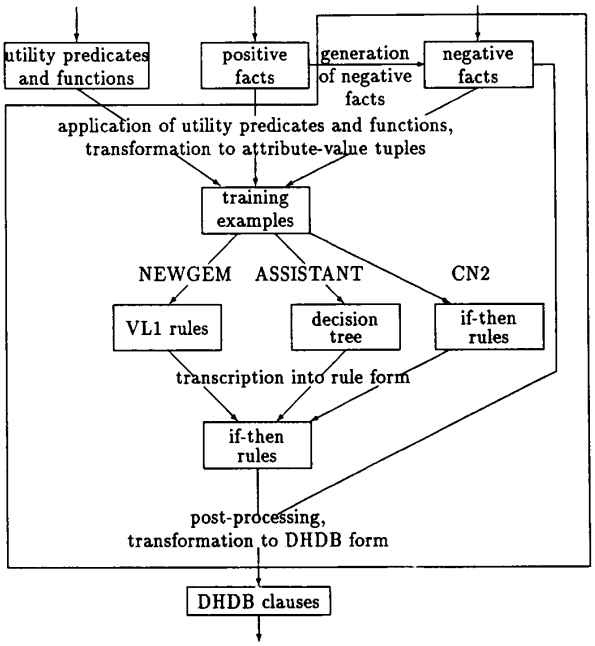


Fig. 1. An overview of LINUS.

A. Training Examples and Background Knowledge

The training examples in LINUS are given as a set of ground facts about the target predicate. Positive examples are always provided explicitly, while negative examples may be either given explicitly or generated automatically by the DHDB interface. The negative examples may be generated under the *closed world assumption* or in a number of other ways [12], [16]. The background knowledge, a set of deductive database clauses, consists of utility functions and utility predicates.

Utility functions are annotated predicates: mode declarations specify the *input* and *output* arguments, similar to mode declarations in GOLEM [26] and FOIL2 [33]. When applied to ground input arguments from the training examples, utility functions compute the unique ground

values of their output arguments. Utility predicates have only *input* arguments and can be regarded as Boolean utility functions having values *true* or *false* only.

The predicate argument types specified in the background predicate definitions reduce the number of attributes generated. Similar reduction can be achieved by exploiting the fact that some utility predicates are symmetric. For example, a binary predicate $q_i(X, Y)$ is symmetric in $\{X, Y\}$ if X and Y are of the same type, and $q_i(X, Y) = q_i(Y, X)$ for every value of X and Y. The equality predicate $(X = Y)$ is symmetric as $X = Y$ if and only if $Y = X$. This predicate is predefined in LINUS to work on arguments of the same type, and can be specified as background knowledge predicate for any learning problem.

B. Hypothesis Language

In the current implementation of LINUS, the selected hypothesis language $\mathcal{L}$ is restricted to deductive hierarchical database (DHDB) clauses. In DHDB, variables are typed and recursive predicate definitions are not allowed. In LINUS, all variables that appear in the body of an induced clause have to appear in the head as well, i.e., only constrained clauses are induced.

More specifically, the body of an induced clause in LINUS is a conjunction of literals, each having one of the following four forms:

1) an instantiation of a variable, e.g., *Income* = *high*;
2) an equality of a pair of variables, e.g., *Income* = *Expenses*;
3) an atom with a utility predicate symbol and input arguments chosen among the arguments of the target predicate, e.g., *greater*(*Income*, *Expenses*); and
4) an atom with a utility function symbol, having as input arguments some of the target predicate arguments and output arguments instantiated to constants, e.g., *net*(*Income*, *Expenses*, *NetIncome*), *NetIncome* = *high*.

In the above, *Income* and *Expenses* are variables from the head of the clause, i.e., arguments of the target predicate and *high* is a constant of the appropriate type. Literals of form (2) and (3) can be either positive or negative. Literals under items (1) and (4) may also have the form *Income* > *amount* and/or *Expenses* < *amount*, where *amount* is a real valued constant.

The attributes given to propositional learners are (1) the arguments of the target predicate, (2)–(3) binary valued attributes resulting from applications of utility predicates and (4) output arguments of utility functions. Attributes under (1) and (4) may be either discrete or real valued. To guide induction, LINUS can use metalevel knowledge specified by the user, which can exclude any of the above four cases, thus reducing the search space. For example, if only case (1) is retained, the hypothesis language is restricted to an attribute-value language.

C. The LINUS Algorithm

At the outermost level, the LINUS learning algorithm works as follows:

1) establish the training set of positive and negative facts;
2) using background knowledge, transform facts from the DHDB form into attribute-value tuples;
3) induce an attribute-value concept description by an attribute-value learning program;
4) transform the induced attribute-value concept description into the form of DHDB clauses;
5) postprocessing of DHDB clauses.

Let us illustrate the work of LINUS on the simple example given in Table II. The task is to define the target relation *daughter*(*Daughter*, *Parent*) in terms of the background knowledge relations (utility predicates) *female*, *male* and *parent*. All attributes are of type *person* with values {*ann*, *eve*, *pat*, *sue*, *tom*}. The built-in symmetric predicate equality (=), which works on arguments of the same type, may also be used in the induced clauses. In the following, we describe in detail the individual steps of the algorithm working on the *daughter* example.

1. First, the sets of positive and negative facts are established. In our domain, given are two positive examples (labeled ⊕) and two negative examples (labeled ⊖):

% *daughter*(*Daughter*, *Parent*).

daughter(*sue*, *eve*). ⊕

daughter(*ann*, *pat*). ⊕

daughter(*tom*, *ann*). ⊖

daughter(*eve*, *ann*). ⊖

2. The facts are then transformed into attribute-value tuples. The algorithm first determines the possible applications of the background predicates on the arguments of the target relation, taking into account argument types. Each such application is considered as an attribute. Considering the above mentioned background knowledge predicates, the set of attributes determining the form of the tuples is the following:

$$<D = P, f(D), f(P), m(D), m(P), p(D, D), p(D, P), p(P, D), p(P, P)>$$

where D, P, f, m and p stand for *Daughter*, *Parent*, *female*, *male* and *parent*, respectively.

The tuples, i.e., the values of the attributes, are generated by calling the corresponding predicates with argument values from the ground facts of predicate *daughter*. In this case, the attributes can take values *true* or *false*. For the given examples, the following tuples are generated. For instance, for the first example *daughter*(*sue*, *eve*), the value of attribute $D = P$ is *false* (since *sue* ≠ *eve*), the value of attribute $f(D)$ is *true* (*sue* is female),

TABLE III
ACCURACIES ACHIEVED IN THE KRK ENDGAME EXPERIMENT

System	100 Training Instances	1000 Training Instances
CIGOL	77.2%	N/A
DUCE	33.7%	37.7%
FOIL on different sets	92.5% sd 3.6%	99.4% sd 0.1%
FOIL	90.8% sd 1.7%	99.7% sd 0.1%
LINUS using ASSISTANT	98.1% sd 1.1%	99.7% sd 0.1%
LINUS using NEWGEM	88.4% sd 4.0%	99.7% sd 0.1%
LINUS using CN2	92.9% sd 1.3%	99.3% sd 0.3%

etc.

$<D = P$	$f(D)$	$f(P)$	$m(D)$	$m(P)$	$p(D, D)$	$p(D, P)$	$p(P, D)$	$p(P, P)>$	
$<false$	*true*	*true*	*false*	*false*	*false*	*false*	*true*	*false*$>$	$\oplus$
$<false$	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*$>$	$\oplus$
$<false$	*false*	*true*	*true*	*false*	*false*	*false*	*true*	*false*$>$	$\ominus$
$<false$	*true*	*true*	*false*	*false*	*false*	*false*	*false*	*false*$>$	$\ominus$

These tuples are generalizations (relative to the given background knowledge) of the individual facts about the target relation.

3. Next, an attribute-value learning program is used to induce a set of *if-then* rules. When ASSISTANT is used, the induced decision trees are transcribed into *if-then* rules. NEWGEM induces the following *if-then* rule from the above tuples:

if *female*(*Daughter*) = *true* $\wedge$ *parent*(*Parent*, *Daughter*) = *true* then *Class* = $\oplus$

4. Finally, the induced *if-then* rules for *Class* = $\oplus$ are transformed into DHDB clauses. In our example, we get the following clause:

daughter(*Daughter*, *Parent*)
$\leftarrow$ *female*(*Daughter*), *parent*(*Parent*, *Daughter*).

5. The DHDB clauses may be further postprocessed and made more compact by eliminating irrelevant literals. In exact non-noisy domains, a literal in a clause is *irrelevant* if it can be removed from the clause body without causing the clause to cover new negative training examples. Postprocessing in noisy domains is slightly different and is described in Section V-A.

In summary, the learning problem is transformed from a relational to an attribute-value form and solved by an attribute-value learner. The induced hypothesis is then transformed back into relational form, which may be further postprocessed.

D. Learning Modes in LINUS

LINUS can basically be used in two different modes: relation learning mode and class learning mode. Each specifies a different language bias (hypothesis language). The user can set the language bias to either mode, depending on the learning task at hand.

In *relation learning mode*, there are two classes, $\oplus$ and $\ominus$, and LINUS induces constrained DHDB clauses of the form

$$p(X_1, \cdots, X_n) \leftarrow L_1, \cdots, L_m.$$

where p is the name of the target predicate and literals L_i have any of the forms (1)–(4) from Section III-B. In the experiments described in this paper, LINUS was used to learn function free clauses; thus, only literals of form (2) and (3) were actually used.

When the problem at hand has more than two classes, LINUS can be used in the *class learning mode*. The classes can be different from $\oplus$ and $\ominus$ and are determined by the values of a selected argument (or a set of arguments) of the target relation. The induced clauses have the form

$$class(Class) \leftarrow L_1, \cdots, L_m.$$

where *Class* is a class name and L_i can take any of the four forms outlined in Section III-B.

IV. EXPERIMENTS WITH NON-NOISY DATA

This section discusses the performance of LINUS on two relation learning tasks taken from the machine learning literature. More experiments with LINUS are described in [16]. The domain descriptions are taken from [32] and the LINUS results are compared to the results obtained by FOIL [32]. An early version of FOIL (FOIL0) was used in the experiments.

LINUS was used in the *relation learning mode*. No literals of the form (1) from Section III-B, instantiating a variable to a constant, were allowed in the induced predicate definitions, in order to facilitate the comparison with FOIL. The arguments of the background predicates used in FOIL were not typed and the same predicates could be used for different types of arguments. In LINUS, each such predicate was replaced by several predicates, one for each combination of predicate argument types (see Section IV-B).

In the experiments, ASSISTANT, NEWGEM and CN2 were used within LINUS. The results obtained with ASSISTANT, NEWGEM and CN2 on each of the domains were typically slightly different; they were all comparable to the results obtained by FOIL. In the chess endgame domain we compared the classification accuracy of LINUS (using ASSISTANT, NEWGEM and CN2) and FOIL, as well as DUCE and CIGOL [27].

A. Learning Family Relationships

The family relationships learning task is described in Quinlan [32]. Given are two stylized families of twelve members each, as shown in Fig. 2.

We used LINUS to learn the relation *mother*(X, Y) from examples of this relation and the background relations *father*(X, Y), *wife*(X, Y), *son*(X, Y) and *daughter*(X, Y). All predicate arguments are of type *person*. Negative examples were generated under the closed world assumption, both for LINUS and FOIL. To illustrate the postprocessing feature of LINUS, we also present the intermediate Prolog rules induced by LINUS using ASSISTANT:

mother(A, B) ← *daughter*(B, A), *not father*(A, B).

mother(A, B) ← *not daughter*(B, A), *son*(B, A),
not father(A, B).

The second clause contains the irrelevant literal *not daughter*(B, A) which was removed by postprocessing. The resulting definition is equal to the definitions induced by LINUS using NEWGEM and LINUS using CN2, which contained no irrelevant literals. The induced description of the relation *mother*(A, B) can be paraphrased as follows: "A is the mother of B if B is a child of A and A is not the father of B."

FOIL, on the other hand, induced the following definition

mother(A, B) ← *daughter*(B, A), *not father*(A, C).

mother(A, B) ← *son*(B, A), *not father*(A, C).

which can be summarized as: "A is the mother of B if B is a child of A and A is not the father of anybody (C)." The definition contains a new variable (C), which stands for any person. While not necessary in this case, new variables are necessary when learning definitions of relations such as *grandmother*(X, Y) in terms of the relations *mother*(X, Y) and *father*(X, Y). Thus, the hypothesis language of FOIL is more expressive than the one of LINUS, which does not allow for the introduction of new variables.

B. Learning Illegal Positions in a Chess Endgame

In the chess endgame domain White King and Rook versus Black King, described in [27] and [32], the target relation *illegal*(*WKf*, *WKr*, *WRf*, *WRr*, *BKf*, *BKr*) states whether the position where the White King is at (*WKf*, *WKr*), the White Rook at (*WRf*, *WRr*) and the Black King at (*BKf*, *BKr*) is an illegal White-to-move position. Fig. 3 depicts the example illegal position *illegal*(g, 6, c, 7, c, 8).

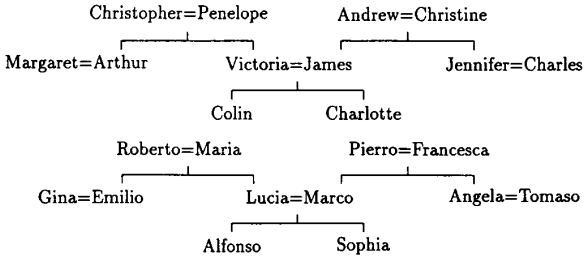


Fig. 2. Two family trees, where = means "married to."

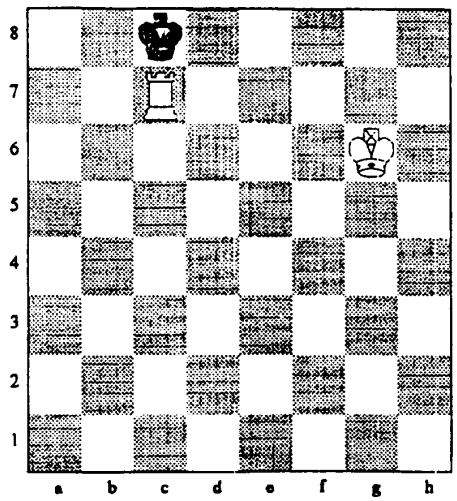


Fig. 3. An example illegal white-to-move position.

In FOIL, the background knowledge for this task is represented by two relations, *adjacent*(X, Y) and *less_than*(X, Y), indicating that rank/file X is adjacent to rank/file Y and rank/file X is less than rank/file Y, respectively. The arguments of these background predicates are not typed and the same predicates are used for both types of arguments. In LINUS, each of these predicates is replaced by two predicates, one for each type of arguments. Thus, LINUS uses the following relations: *adjacent_file*(X, Y) and *less_file*(X, Y) with arguments of type *file* (with values *a* to *h*), *adjacent_rank*(X, Y) and *less_rank*(X, Y) with arguments of type *rank* (with values 1 to 8), and equality $X = Y$, used for both types of arguments.

The experiments with LINUS and FOIL were performed on the same training and testing sets as used in [27]. There were five small sets of 100 examples each and five large sets of 1000 examples each. Each of the sets was used as a training set for FOIL and LINUS (using ASSISTANT, NEWGEM and CN2). The induced sets of clauses were then tested as described in [27]: the clauses obtained from a small set were tested on the 5000 exam-

ples from the large sets and the clauses obtained from each large set were tested on the remaining 4500 examples.

The classification procedures used in LINUS were the classification procedures of the corresponding attribute-value learners. For the rules obtained by ASSISTANT (by transcribing the decision trees into rules) and NEWGEM, the classification procedure was as follows: if an example is covered by the Prolog clauses of the definition (rules for class positive), it is classified as positive, otherwise as negative. In CN2, where unordered rules [4] were induced, the classification procedure was different. Definitions are built both for the positive and the negative class. All rules that cover an example (both positive and negative class rules), are taken into account when classifying the example [4].

Table III gives the accuracies achieved in the chess endgame experiment. The classification accuracy is given by the percentage of correctly classified testing instances and by the standard deviation (sd), averaged over five experiments. The first two rows are taken from [27], the third is from [32] and the last four rows present the results of our experiments. Note that the results reported in [32] were not obtained on the same training and testing sets.

In brief, on the small training sets LINUS using ASSISTANT (with postprocessing), outperformed FOIL. According to the t-test for dependent samples, this result is significant at the 99.5% level. LINUS using CN2 also achieved slightly better accuracy than FOIL. Although LINUS using NEWGEM performed slightly worse than FOIL, this result is not significant (even at the 80% level). The clauses obtained with LINUS are as short and understandable (transparent) as FOIL's. On the large training sets both systems performed equally well. Both LINUS and FOIL performed much better than DUCE and CIGOL [27].

For illustration, the clauses induced by LINUS using ASSISTANT (with postprocessing) from one of the sets of 100 examples are given below:

$$illegal(A, B, C, E, F) \leftarrow C = E.$$

$$illegal(A, B, C, E, F) \leftarrow D = F.$$

$$illegal(A, B, C, E, F) \leftarrow adjacent\_file(A, E), B = F.$$

$$illegal(A, B, C, E, F) \leftarrow adjacent\_file(A, E), adjacent\_rank(B, F).$$

$$illegal(A, B, C, E, F) \leftarrow A = E, adjacent\_rank(B, F).$$

$$illegal(A, B, C, E, F) \leftarrow A = E, B = F.$$

These clauses may be paraphrased as: "a position is illegal if the Black King is on the same rank or file as (i.e., is attacked by) the Rook, or the White King and the Black King are next to each other, or the White King and the Black King are on the same square." Although these clauses are neither consistent nor complete, they correctly classify 98.5% of the unseen cases.

V. Experiments with Noisy Data

In real life domains, learning systems often have to deal with imperfect data. When learning definitions of relations, the following kinds of data imperfection are usually met:

- noise in the training examples (caused by erroneous argument values and/or erroneous classification of facts as true or false), as well as noise in the background knowledge;
- insufficiently covered example space, i.e., too sparse training examples from which it is difficult to reliably detect correlations;
- inexactness, i.e., inappropriate (some predicates may not be relevant) or insufficient (important predicates may be missing) background knowledge; and
- missing argument values in the training examples.

Learning systems usually have a single mechanism for dealing with the first three kinds of imperfect data, i.e., with noisy, incomplete and inexact data. Such mechanisms, often called *noise handling mechanisms*, are typically designed to prevent the induced hypothesis from *overfitting* the data set, i.e., to avoid overly specific concept descriptions. Missing values are usually handled by a separate mechanism.

Until recently, relation learning systems did not address the issue of data imperfection. However, both FOIL and LINUS include sophisticated noise handling mechanisms. In this section, we first describe the noise handling mechanisms used in LINUS and FOIL. We then present the experimental setup and the results of our investigation of the effects of noise in inductive logic programming. The domain under study was the domain of learning position illegality in the KRK chess endgame (see Section IV-B) from training examples with artificially added noise.

The ultimate performance test for ILP algorithms should be their application to practical domains involving imperfect data. However, it is difficult to measure the level of noise (imperfection) in such domains. Furthermore, while several standard data sets obtained from practical settings are now available for attribute-value systems [4], not many datasets of this kind are available for the relational (ILP) case. The problem of learning illegal positions in the KRK chess endgame, on the other hand, has been used as a testbed for most empirical ILP systems. Having chosen to introduce a controlled amount of noise in the training examples artificially, it was possible to assess the dependence of learning performance on the noise level more accurately.

A. Noise Handling in LINUS and FOIL

As LINUS incorporates different attribute-value learners, the noise handling mechanisms used in it are the ones of the underlying attribute-value learners. This allows for a wide variety of noise handling mechanisms to be used within LINUS. Furthermore, the latest advances in noise

handling, such as the use of Bayesian probability estimates in rule induction [4], [9], can be easily incorporated into LINUS. In the following, we first outline the noise handling mechanisms of ASSISTANT, NEWGEM (AQ15, in fact) and CN2, then describe the encoding length restriction used to handle noise in FOIL and finally touch upon related work on noise handling in ILP.

The noise handling mechanisms in LINUS using ASSISTANT are *tree pruning* [3] and *postprocessing* of rules derived from decision trees [31]. There are two types of tree pruning:

- *prepruning*, performed during the construction of a decision tree, which decides whether to stop or to continue tree construction at a certain node; and
- *postpruning*, used after the decision tree is constructed, which estimates the classification errors in the nodes and then decides whether to prune certain subtrees or not.

The *postprocessing* of *if-then* rules derived from decision trees is a form of reduced error pruning [31]. This mechanism checks whether a set of clauses (rules) can be made more compact by eliminating irrelevant literals. A literal in a clause is *irrelevant* if it can be removed from the body of the clause without decreasing its expected classification accuracy, which is estimated from the training set.

The effects of prepruning and postpruning on the classification accuracy of decision trees are very similar [3]; thus, only prepruning was used in our experiments. The combination of prepruning and postprocessing was used in the experiments on the same domain described in [14], and was also applied to a medical domain [17].

The NEWGEM rule induction system [22] is a member of the AQ family of learning programs. It was developed further into AQ15 [20], which has a noise handling mechanism based on *flexible matching* and *rule truncation*. However, NEWGEM itself has no noise handling mechanisms.

The original version of CN2 [5] induces an ordered list of rules. It uses an entropy based search heuristic, which favors specific, apparently very accurate rules. To prevent overfitting, it uses a statistical significance test which distinguishes between true regularities and regularities that are likely to have occurred by chance.

A recent improved variant of CN2 [4], can induce both ordered and unordered rules. When unordered rules are induced (which have proved to perform better), the classification procedure takes into account the decisions of all rules that cover the example being classified. This variant of CN2 is incorporated in LINUS. To avoid overly specific rules, CN2 uses the Laplace estimate [4] to assess the accuracy of a rule from its coverage on the training set. Using the Laplace estimate as a search heuristic, CN2 favors more general, and thus more reliable rules, which may still cover some noisy examples.

In FOIL, the noise handling mechanism is the *encoding length restriction* [32]. This heuristic restricts the total length of an induced clause to the number of bits needed to explicitly enumerate the positive training examples it covers. If a clause covers p positive examples out of the n examples in the training set, the length of the clause should not exceed $L(n, p) = \log_2(n) + \log_2(\binom{n}{p})$. If there are no bits available for adding another literal, but the clause is still at least 85% accurate, it is retained in the induced set of clauses; otherwise, it is discarded.

The encoding length restriction has two deficiencies. In nonnoisy exact domains, it sometimes prevents FOIL from building complete descriptions [16]. In noisy domains, it allows very specific clauses, which is not desirable. Suppose we have a training set with one positive and 1023 negative examples. Knowing that the domain is noisy, most systems that can handle noise would regard the single positive example as erroneous and would not even attempt to build a clause to cover it. Nevertheless, FOIL will have 20 ($20 = \log_2(1024) + \log_2(\binom{1024}{1})$) bits available to build a clause covering the single positive example.

The ILP system GOLEM [26] also shows awareness of the need to handle imperfect data. It allows a generated clause to cover a predetermined number of negative examples. This is a rudimentary way of handling noise. Suppose clauses are allowed to cover at most two negative examples. If a clause covers 1000 positive and two negative examples, it probably reflects a genuine correlation in the training examples. However, if it covers two positive and two negative examples, the correlation it represents is very likely due to chance. Thus, GOLEM should take into account at least both the number of positive and negative examples covered by the clause and allow, for example, clauses to be built which are at least 85% accurate. New successful mechanisms for handling noise in GOLEM were lately proposed in [28].

B. Experimental Setup

To examine the effects of noise on the induced relational concept descriptions, various amounts of noise were added to the examples from the chess endgame domain in Section IV-B. The small sets of 100 examples were used as training sets, while the large sets were merged into one testing set of 5000 examples.

In the following, x% of noise in argument A means that in x% of the examples, the values of argument A were replaced by random values [30]. For example, 5% of noise in argument A means that its value was changed to a random value in 5 out of 100 examples, independently of noise in other arguments. The class variable was treated as an additional argument [5] when introducing noise. The percentage of introduced noise varied from 5% to 80% as follows: 5%, 10%, 15%, 20%, 30%, 50%, and 80%. No noise was introduced into the background knowledge.

Three series of experiments were conducted, introducing noise in the training examples in three different ways. Noise was first added in the values of the arguments of the target relation, then in the values of the class variable, and finally, in both the argument and the class variable

values. Each experiment was performed using one of the three ways of introducing noise and a chosen noise level. Noise was first introduced in the training sets. A set of clauses was then induced from each of the training sets using LINUS and FOIL. Finally, the classification accuracy of the sets of clauses was tested on the testing set, and the average of the five results was computed.

In the experiments, ASSISTANT, NEWGEM and CN2 were used as attribute-value learning algorithms in LINUS. Within LINUS, ASSISTANT was run with its default parameters, which set the noise handling mechanism to tree prepruning only. The NEWGEM parameters were set to minimize the number of literals in a clause and to maximize the number of examples uniquely covered by a clause. CN2 was used to induce unordered rules with the Laplace estimate as a search heuristic and no significance test used. FOIL0, an early version of FOIL, was used in the experiments.

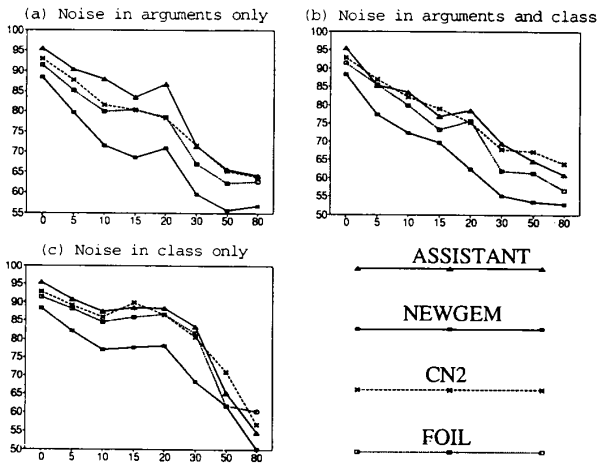


Fig. 4. Results of experiments with noise in the training examples only.

C. Experimental Results

Fig. 4(a) gives the results of the experiments with noise introduced only in the arguments, Fig. 4(b) with noise in the class variable only, and Fig. 4(c) with noise introduced in the arguments and the class variable at the same time.

Noise affected adversely the classification accuracy of both systems. The classification accuracy decreased as the percentage of noise increased: the most when it was introduced in both the arguments and the class variable, and the least when introduced in the class variable only. LINUS using ASSISTANT and CN2 achieved better classification accuracy than FOIL. As NEWGEM used no mechanism for handling imperfect data, LINUS using NEWGEM performed worse than FOIL.

Analysis of the clauses induced by LINUS using ASSISTANT and FOIL shows that with the increase of the percentage of noise in the training examples, the following can be observed:

- The average clause length increases in FOIL and decreases in LINUS using ASSISTANT.
- The average number of examples covered by a clause slowly decreases in FOIL and increases in LINUS using ASSISTANT.
- The number of positive examples not covered by the induced predicate definitions rises for both systems, faster for LINUS using ASSISTANT than for FOIL.

All these facts indicate that FOIL slightly overfits the training examples, which results in a lower performance on unseen cases. This is mainly due to the deficiency of the encoding length restriction discussed in Section V-A.

Two further remarks about the results have to be made. First, there is an increase in the classification accuracy when the noise level increases from 15% to 20%. This is due to the nonincremental manner of introducing noise in the training examples. This phenomenon disappeared in the experiments where noise was added incrementally [7]. Second, in the training data, 1/3 of examples are positive (illegal positions) and 2/3 are negative (legal positions). A learning system with a good noise handling mechanism should produce rules at least as accurate as the classification rule that classifies each example into the majority class (in our case class *legal*).

As shown in Fig. 4, the classification accuracy of LINUS using NEWGEM drops under the 66% majority class accuracy already at the 30% noise level. This is acceptable when noise is introduced in the class variable (Fig. 4(b)) and both in the arguments and the class variable (Fig. 4(c)), since the noise changes the class distribution, but not in the case when only arguments are corrupted (Fig. 4(a)). This indicates that the training examples are being overfitted. To explain this phenomenon, we adapt an argument from [4]. At 100% noise in the arguments, the arguments and the class are completely independent. The maximally overfitted rules (one per example) would then be correct with probability 1/3 if the class is *illegal* and with probability 2/3 if the class is *legal* (the default probabilities of positive and negative examples). The probability of a correct answer would then be $2/3 \times 2/3 + 1/3 \times 1/3 = 5/9$, or 55%, which is lower than the 66% default accuracy. The overfitting observed in our experiments reflects behavior between these two extremes.

The experiments with FOIL and LINUS using ASSISTANT and NEWGEM were repeated with noisy testing sets. Noise was introduced in the testing set in exactly the same way as in the training sets (same amounts of noise in the arguments, the class variable or both). The classification accuracy of both LINUS and FOIL dropped significantly, but their relative performance remained unchanged, i.e., LINUS using ASSISTANT achieved better, and LINUS using NEWGEM, achieved worse classification accuracy than FOIL.

VI. Conclusion

The paper presented LINUS, an inductive logic programming system that can be used to induce intentional

definitions of relations, i.e., virtual relations, in a deductive database setting. It is based on the idea that a relation learning problem can be transformed into an attribute-value learning problem. In the learning process, it uses example tuples that are known to belong (or not to belong) to the target relation and the definitions of other relations from the given deductive database.

This transformation approach only works for a limited class of relation learning problems. The virtual relations that can be induced by LINUS are a subset of those expressible in relational algebra. More specifically, the hypothesis language in LINUS is the language of deductive hierarchical database clauses (nonrecursive by definition) that do not introduce new variables. We have recently shown that the transformation approach can be extended to induce determinate deductive database clauses [13], which introduce new variables with uniquely determined values, and may be recursive.

LINUS can use nonground, possibly recursive, background knowledge and can thus be applied in a deductive database setting. Other empirical ILP systems, such as FOIL and GOLEM, can only use ground background knowledge in the form of extensional definitions of relations. Consequently, they can only be used in a relational database setting.

Using attribute-value learners in a logic programming environment, LINUS has the ability to learn intentional relations from noisy data. This ability is of paramount importance for knowledge discovery in real life databases containing imperfect data. For this reason, LINUS was carefully evaluated and compared to FOIL on the problem of learning relations from noisy examples. Using ASSISTANT and CN2, LINUS performed better than FOIL.

In the expert system community, significant effort has been devoted to the development and use of toolkits from which to select appropriate tools according to the nature of the domain and type of knowledge under investigation [11]. This philosophy was also adopted in the Machine Learning Toolbox (MLT) Project [21], aimed at developing a workbench of machine learning tools, from which one or more can be selected in order to find the best solution to a specific problem. LINUS can be considered such a workbench, as it incorporates several attribute-value learners.

To conclude, LINUS is able to learn virtual relations in a deductive database context, while most of the work on knowledge discovery in databases takes place in the context of a single extensional relation. It is a workbench of attribute-value learners integrated in an ILP framework. Having the ability to handle noisy data, it is a good starting point for knowledge discovery in deductive databases.

Acknowledgment

We are grateful to Ross Quinlan for making available his system FOIL and the data used in the experiments, and Michael Bain for the chess endgame data. We wish to thank Igor Mozetič and Marko Grobelnik for their contribution to the development of LINUS, Dunja Mladenič for the implementation of ASSISTANT in the VAX/VMS environment, and Ivan Bratko, Bojan Cestnik, Peter Flach, Igor Kononenko and Stephen Muggleton for their comments on the individual parts of the paper.

References

[1] I. Bratko, I. Mozetič, and N. Lavrač, *KARDIO: A study in deep and qualitative knowledge for expert systems*, MIT Press, Cambridge, MA, 1989.

[2] J. Catlett, "Peepholing: Choosing attributes efficiently for megainduction," in *Proc. 9th Int. Conf. Mach. Learn.*, Morgan Kaufmann, San Mateo, CA, 1992, pp. 49-54.

[3] B. Cestnik, I. Kononenko, and I. Bratko, "ASSISTANT 86: A knowledge elicitation tool for sophisticated users," in *Progress in Machine Learning*, I. Bratko and N. Lavrač, eds. Wilmslow, Sigma, 1987, pp. 31-45.

[4] P. Clark and R. Boswell, "Rule induction with CN2: Some recent improvements," in *Proc. 5th European Working Session on Learning*, Berlin: Springer, 1991, pp. 151-163.

[5] P. Clark and T. Niblett, "The CN2 induction algorithm," *Machine Learning*, vol. 3, no. 4, 1989, pp. 261-283.

[6] *Interactive Theory Revision: An Inductive Logic Programming Approach*. London: Academic, 1992.

[7] S. Džeroski, "Handling noise in inductive logic programming," M.Sc. thesis, Faculty of Elec. Eng. and Comp. Sci., University of Ljubljana, Slovenia, 1991.

[8] S. Džeroski and N. Lavrač, "Learning relations from noisy examples: An empirical comparison of LINUS and FOIL," in *Proc. 8th Int. Workshop on Machine Learning*, Morgan Kaufmann, San Mateo, CA, 1991, pp. 399-402.

[9] S. Džeroski, B. Cestnik, and I. Petrovski, "The use of Bayesian probability estimates in rule induction," in *Proc. 1st Elect. Comp. Sci. Conf.*, Volume B, Slovenia Section IEEE, Ljubljana, 1992, pp. 155-158.

[10] W. Frawley, G. Piatetsky-Shapiro, and C. Matheus, "Knowledge discovery in databases: An overview," *AI Magazine*, Fall, 1992, pp. 57-70.

[11] P. Harmon, R. Maus, and W. Morrisey, *Expert Systems: Tools and Applications*. New York: Wiley, 1988.

[12] N. Lavrač, "Principles of knowledge acquisition in expert systems," Ph.D. Thesis, Faculty of Tech. Sci., University of Maribor, Slovenia, 1990.

[13] N. Lavrač and S. Džeroski, "Background knowledge and declarative bias in inductive concept learning," in *Proc. 3rd Int. Workshop on Analogical and Inductive Inference*, K. Jantke, ed. Berlin: Springer, 1992, pp. 51-71.

[14] N. Lavrač and S. Džeroski, "Inductive learning of relations from noisy examples," in *Inductive Logic Programming*, S. H. Muggleton, ed. London: Academic, 1992, pp. 495-516.

[15] N. Lavrač and S. Džeroski, *Inductive Logic Programming: Techniques and Applications*. Chichester: Ellis Horwood, 1993. To appear.

[16] N. Lavrač, S. Džeroski, and M. Grobelnik, "Learning nonrecursive definitions of relations with LINUS," in *Proc. 5th European Working Session on Learning*. Berlin: Springer, 1991, pp. 265-281.

[17] N. Lavrač, S. Džeroski, V. Pirnat, and V. Križman, "Learning rules for early diagnosis of rheumatic diseases," in *Proc. 3rd Scandinavian Conf. Art. Intell.* Amsterdam: IOS Press, 1991, pp. 138-149.

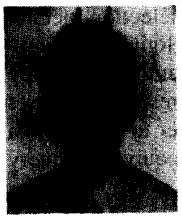
[18] N. Lavrač, S. Džeroski, V. Pirnat, and V. Križman, "The use of background knowledge in learning medical diagnostic rules," *Appl. Art. Intell.*, 7, 1993. To appear.

[19] J. W. Lloyd, *Foundations of Logic Programming* (2nd ed.) Berlin: Springer, 1987.

[20] R. S. Michalski, I. Mozetič, J. Hong, and N. Lavrač, "The multipurpose incremental learning system AQ15 and its testing application on three medical domains," in *Proc. National Conf. Art. Intell.* San Mateo, CA: Morgan Kaufmann, 1986, pp. 1041-1045.

[21] K. Morik, K. Causse, and R. Boswell, "A common knowledge representation integrating learning tools," in *Proc. 1st Int. Workshop on Multistrategy Learning*. Fairfax, VA: George Mason University, 1991, pp. 81-96.

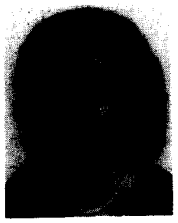
[22] I. Mozetič, "NEWGEM: Program for learning from examples, technical documentation and user's guide," *Reports of Intelligent Systems Group*, No. UIUCDCS-F-85-949, Dep. Comp. Sci., Univ. Illinois, Urbana-Champaign, 1985. (Also *Technical Report IJS-DP-4390*, Jožef Stefan Institute, Ljubljana, Slovenia.)
[23] ——, "Learning of qualitative models," in *Progress in Machine Learning*, I. Bratko and N. Lavrač, eds. Wilmslow: Sigma, 1987, pp. 201–217.
[24] S. H. Muggleton, ed. *Inductive Logic Programming*. London: Academic, 1992.
[25] S. H. Muggleton and W. Buntine, "Machine invention of first-order predicates by inverting resolution," in *Proc. 5th Int. Conf. Mach. Learn.* San Mateo, CA: Morgan Kaufmann, 1988, pp. 339–352.
[26] S. H. Muggleton and C. Feng, "Efficient induction of logic programs," in *Proc. First Conf. Algorithmic Learning Theory*. Tokyo: Ohmsha, 1990, pp. 368–381.
[27] S. H. Muggleton, M. Bain, J. Hayes-Michie, and D. Michie, "An experimental comparison of human and machine learning formalisms," in *Proc. 6th Int. Workshop Mach. Learn.* San Mateo, CA: Morgan Kaufmann, 1989, pp. 113–118.
[28] S. H. Muggleton, A. Srinivasan, and M. Bain, "Compression, significance, and accuracy," in *Proc 9th Int. Conf. Mach. Learn.* San Mateo, CA: Morgan Kaufmann, 1992, pp. 338–347.
[29] M. Pazzani and D. Kibler, "The utility of knowledge in inductive learning," *Machine Learning*, vol. 9, no. 1, pp. 57–94, 1992.
[30] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.
[31] ——, "Simplifying decision trees," *Int. J. Man-Machine Studies*, vol. 27, no. 3, pp. 221–234, 1987.
[32] ——, "Learning logical definitions from relations," *Machine Learning*, vol. 5, no. 3, pp. 239–266, 1990.
[33] ——, "Knowledge acquisition from structured data—Using determinate literals to assist search," *IEEE Expert*, vol. 6, no. 6, pp. 32–37, 1991.
[34] E. Y. Shapiro, *Algorithmic Program Debugging*. Cambridge, MA: MIT, 1983.
[35] J. D. Ullman, *Principles of Database and Knowledge Base Systems (Volume I)*. Rockville, MA: Computer Press, 1988.



Sašo Džeroski was born in Ohrid, Macedonia, on May 31, 1968. He received the B.Sc. and M.Sc. degrees in computer science, from the Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Slovenia, in 1989 and 1991, respectively. He is currently working toward the Ph.D. degree in the area of discovering system dynamics by machine learning.

Since 1989, he has been a Research Assistant at the Artificial Intelligence Laboratory, Computer Science Department, Jozef Stefan Institute, Ljubljana, Slovenia. His research interests have been in the area of machine learning, including theoretical and practical topics in inductive logic programming. His recent work focuses on the use of machine learning methods for modelling and controlling dynamic systems.

Mr. Džeroski was awarded several prizes at the Macedonian National Competition in Mathematics and Physics. In 1985, he won a third prize at the International Mathematical Olympiad in Helsinki, Finland.



Nada Lavrač was born in Ljubljana, Slovenia, in 1953. After graduating in Technical Mathematics, she received the M.Sc. degree in computer science and the Ph.D. degree in technical sciences in 1990.

She was a Visiting Researcher at the University of Illinois, Urbana-Champaign, at George Mason University, Fairfax, VA, and at Catholic University, Leuven, Belgium. She has taught graduate courses in machine learning as a Visiting Professor at the University of Stockholm, Linköping, Sweden, and at the University of São Paulo, Brazil. Currently, she is a Senior Researcher in the Artificial Intelligence Laboratory, Computer Science Department, Joszef Stefan Institute, Ljubljana, Slovenia. Her research interests have been in machine learning, knowledge acquisition, qualitative modelling, and logic programming. Since 1988, her main interest has been in inductive logic programming. She has co-authored four books, including *Inductive Logic Programming: Techniques and Applications*.

In 1982, she received the Yugoslav Cybernetics Association Award (with a research team) for the development of a medical expert system. In 1986, she received the Slovenian B. Kidrič Award for research in knowledge synthesis using qualitative modeling techniques. In 1991, she received the Danish Association for Artificial Intelligence (SCAI-91) Award.